pyoomph

FINITE ELEMENT SIMULATIONS IN PYTHON

Pyoomph Tutorial

Release 0.1.8

Christian Diddens and Duarte Rocha

Feb 27, 2026

CONTENTS

1 Preface **1**
1.1 License and acknowledgements . . . 1
1.1.1 Acknowledgements . . . 4
1.2 How to cite . . . 4
1.3 Motivation . . . 4
1.4 General Information . . . 5
1.5 When to use pyoomph, and when better use something else . . . 6

2 Installation **7**
2.1 Installing the precompiled wheels . . . 7
2.1.1 Windows . . . 7
2.1.2 Mac . . . 8
2.1.3 Linux . . . 9
2.1.4 Trying whether pyoomph works . . . 9
2.1.5 Updating pyoomph . . . 9
2.2 Compilation from source . . . 9
2.2.1 On Mac . . . 9
2.2.2 On Linux . . . 11
2.3 Further practical software . . . 11
2.3.1 VSCode as Python IDE . . . 11
2.3.2 Paraview to visualize the output . . . 12
2.4 Optional installation of PETSc/SLEPc . . . 12
2.5 Typical problems with the installation . . . 13

3 Temporal Ordinary Differential Equations **15**
3.1 Nondimensional harmonic oscillator . . . 15
3.1.1 Mathematical Formulation . . . 15
3.1.2 Python code using the predefined harmonic oscillator equation class . . . 16
3.2 Defining your own harmonic oscillator equations . . . 18
3.3 Test functions and the residual form . . . 20
3.4 Coupled harmonic oscillators . . . 20
3.5 A nonlinear oscillator . . . 23
3.6 Time stepping . . . 25
3.6.1 Testing different time stepping method . . . 25
3.6.2 Trapezoidal rule and the implicit midpoint rule . . . 27
3.6.3 Easily selecting time stepping methods for first order time derivatives . . . 29
3.6.4 Time derivatives of third or higher order . . . 32
3.6.5 Temporal adaptivity . . . 32
3.7 Enforcing constraints by Lagrange multipliers . . . 33
3.8 Using physical units and nondimensionalization . . . 38

3.9 Modifying simulation parameters . . . 41
3.9.1 Inside Python . . . 41
3.9.2 Via the command line . . . 42
3.9.3 Parameter scans via Python . . . 42
3.10 Adding custom math functions in Python . . . 43
3.10.1 A harmonic oscillator driven by a trapezoidal forcing . . . 43
3.10.2 Playing tennis in pyoomph - custom expression with dimensions . . . 45
3.10.3 Custom functions with multiple return values . . . 49
3.11 Stability analysis . . . 49
3.11.1 Global parameters . . . 49
3.11.2 Stationary solutions . . . 51
3.11.3 Calculating eigenvalues and eigenfunctions . . . 52
3.11.4 Stability of second order ODEs or in presence of constraints . . . 53
3.11.5 Following a stationary solution along a parameter by pseudo-arc length continuation . . . 56
3.11.6 Jumping on bifurcations . . . 60
3.11.7 Bifurcation tracking . . . 61
3.11.8 Deflated solving and deflated continuation . . . 64
3.11.9 Lyapunov exponents . . . 66
3.12 Periodic orbits . . . 68
3.12.1 From Hopf bifurcations to periodic orbits . . . 69
3.12.2 Constructing orbits manually . . . 72
3.12.3 Stability of orbits . . . 74

4 Spatial Differential Equations 79
4.1 The Poisson equation . . . 79
4.1.1 Weak formulation . . . 79
4.1.2 One-dimensional Poisson equation with Dirichlet boundary conditions . . . 80
4.1.3 Coupled one-dimensional Poisson equations with Dirichlet boundary conditions . . . 82
4.1.4 Poisson equation with a Neumann boundary condition . . . 83
4.1.5 Pure Neumann boundary conditions for the Poisson equation - Using a Lagrange multiplier to remove the nullspace . . . 85
4.1.6 Robin boundary conditions . . . 87
4.1.7 Cauchy boundary condition . . . 90
4.1.8 Two-dimensional Poisson equation . . . 90
4.1.9 Spatial adaptivity . . . 91
4.1.10 Changing the coordinate system . . . 92
4.2 Mathematical details on the solution procedure . . . 94
4.2.1 Continuous basis functions, spatial discretization and solution procedure . . . 94
4.2.2 Internals: What is happening in pyoomph . . . 97
4.3 Creating custom meshes . . . 98
4.3.1 Defining nodes and elements by hand . . . 99
4.3.2 A helical line mesh & differential operators on manifolds . . . 101
4.3.3 Mesh with metric dimensions a curved boundaries . . . 104
4.3.4 Generating meshes from points and lines via Gmsh . . . 107
4.3.5 Splines, adding holes and locally controlling the mesh resolution . . . 110
4.4 The Stokes equations . . . 110
4.4.1 Strong and weak formulation . . . 111
4.4.2 Implementation of the Stokes equations and the inf-sup condition . . . 112
4.4.3 Non-Newtoninan fluids . . . 114
4.4.4 A case with pure Dirichlet boundary conditions . . . 115
4.4.5 Using physical dimensions and imposing a traction . . . 115
4.4.6 Enforcing zero normal flow . . . 118
4.4.7 Stokes' law - Obtaining forces by traction integrals and using global Lagrange multipliers . . . 122
4.4.8 Using a discontinuous pressure - Crouzeix-Raviart elements . . . 126

4.5 The Helmholtz equation with PML . . . 128

5 Spatio-Temporal Differential Equations 133
5.1 The wave equation . . . 133
5.1.1 Simple wave equation in one dimension . . . 133
5.1.2 Playing drums - Wave equation on a circular domain . . . 135
5.1.3 Double-slit . . . 136
5.2 Convection-diffusion equation . . . 140
5.2.1 Naive implementation . . . 140
5.2.2 SUPG implementation . . . 142
5.3 Navier-Stokes equation . . . 145
5.3.1 Womersley flow . . . 146
5.3.2 Transient and nonlinear generalization of Stokes' law . . . 147
5.3.3 Rayleigh-Taylor instability . . . 148
5.3.4 Marangoni instability . . . 149
5.4 Lubrication equation . . . 152
5.4.1 Relaxation of a perturbation . . . 153
5.4.2 Spreading of a droplet . . . 154
5.4.3 Coalescence of droplets . . . 155
5.5 Continuing a stopped simulation . . . 156
5.6 Pattern formation, stability analysis and bifurcation tracking . . . 157
5.6.1 Damped Kuramoto-Sivashisky equation with periodic boundaries . . . 157
5.6.2 Stability via eigenvalues . . . 159
5.6.3 Stability via bifurcation tracking . . . 162

6 Moving Mesh (ALE) Methods 165
6.1 Lagrangian coordinates . . . 165
6.2 Laplace smoothed mesh . . . 166
6.3 Time derivatives of fields on moving meshes . . . 167
6.4 Mesh reconstruction upon large deformations . . . 170
6.5 Free surface Navier-Stokes equation . . . 171
6.6 Droplet spreading with equilibrium contact angle . . . 175
6.6.1 With free slip at the substrate . . . 175
6.6.2 With Navier-slip at the substrate . . . 176
6.6.3 Full three dimensional implementation with wetting gradients on the substrate . . . 179
6.6.4 Stationary solutions of a dimensional droplet with Marangoni flow and gravity . . . 181
6.6.5 Hyperelastic meshes and tangentially consistent mesh motion . . . 186
6.7 Nonlinear solid mechanics . . . 187
6.7.1 Bending of a cantilever beam . . . 188
6.7.2 Compression of 2D circular disk . . . 191
6.7.3 Oscillations of a released torsion . . . 193

7 Multi-Domain Methods 195
7.1 Temperature conduction through two bodies of different conductivity . . . 195
7.2 Propagation of an ice front . . . 199
7.3 Melting of an ice cylinder with natural convection . . . 202
7.4 Connecting two fluid domains . . . 207
7.5 Stokes' law for a falling droplet with insoluble surfactants . . . 210
7.6 Evaporation of a sessile droplet . . . 212
7.7 Fluid-Structure Interaction . . . 216

8 Multi-Component Flow 219
8.1 Bulk equations . . . 219
8.2 The material library and defining bulk properties . . . 220
8.2.1 Definition of a pure gaseous substance . . . 220

8.2.2 Properties as function of temperature and pressure . . . 221
8.2.3 Definition of gaseous mixtures . . . 223
8.2.4 Pure liquids . . . 227
8.2.5 Liquid mixtures . . . 228
8.2.6 Solids . . . 229
8.3 Example: Rayleigh-Taylor instability . . . 229
8.4 Free surfaces and evaporation . . . 231
8.4.1 What the MultiComponentNavierStokesInterface does . . . 233
8.4.2 Evaporation model . . . 235
8.5 Example: Marangoni instability in a Hele-Shaw cell . . . 236
8.6 Contact line models . . . 238
8.6.1 PinnedContactLine . . . 240
8.6.2 UnpinnedContactLine . . . 240
8.6.3 StickSlipContactLine . . . 241
8.6.4 YoungDupreContactLine . . . 242
8.6.5 KwokNeumannContactLine . . . 242
8.7 Surfactants . . . 242
8.7.1 Insoluble surfactant transport equation in presence of mass transfer . . . 242
8.7.2 Defining insoluble surfactants . . . 243
8.7.3 Soluble surfactants and surfactant isotherms . . . 245
8.8 UNIFAC models . . . 248

9 Discontinuous Galerkin methods **249**
9.1 Advection-diffusion equation with an upwind scheme . . . 249
9.2 Weakly imposing Dirichlet boundary conditions . . . 254

10 Advanced stability analysis **257**
10.1 Linear response to periodic driving . . . 257
10.1.1 Damped harmonic oscillator . . . 257
10.1.2 Drums getting exited by a guitar . . . 260
10.2 Stability analysis involving the shape, i.e. on a moving mesh . . . 262
10.2.1 Droplet detaching by gravity . . . 262
10.3 Azimuthal stability analysis . . . 266
10.3.1 Rayleigh-Benard convection in a cylindrical container . . . 267
10.3.2 Path instability of a rising bubble . . . 269
10.4 Cartesian normal mode stability analysis . . . 273
10.4.1 Numerically obtaining the dispersion relation of a Turing instability . . . 274
10.4.2 Rayleigh-Plateau instability in presence of a substrate . . . 276
10.5 Continuation of eigenbranches . . . 280

11 Plotting interface **285**
11.1 Two-dimensional plotting example . . . 285
11.2 Replotting of existing data . . . 288
11.3 Plotting of eigenfunctions . . . 288
11.4 Generating a movie of eigendynamics . . . 290

12 Coupling multiple simulations with preCICE **293**
12.1 Solving the heat equation on a domain by two simulations . . . 293
12.2 Non-matching meshes and passing vectorial quantities . . . 297

13 Miscellaneous settings **299**
13.1 Controlling the spatial integration order . . . 299

14 Mathematical expressions **301**
14.1 Elementary functions . . . 301

14.2 Keyword variables 301

15 References **303**

Bibliography **305**

CHAPTER

ONE

PREFACE

This tutorial is not intended to be a full reference sheet of all functions pyoomph can offer. However, reading it will gradually guide you through all the important features pyoomph has to offer. Since these capabilities will be introduced where necessary, it is best to read this tutorial completely and trying out all the explained capabilities along the course.

Alternatively, you could just try to start somewhere in a section of interest and search this tutorial for any occuring function or approach. If you are interested e.g. in solving multi-phase & multi-component flow dynamics with mass transfer, just start your journey in Section 8 and backtrack by searching this tutorial whenever something is not clear.

1.1 License and acknowledgements

For pyoomph, the conditions of the GNU General Public License 3 apply:


```
pyoomph - a multi-physics finite element framework based on oomph-lib and GiNaC
Copyright (C) 2021-2025  Christian Diddens & Duarte Rocha

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```


A copy of the license can be found in distribution. Mind also the following licenses

1. pyoomph contains code taken from other authors/projects:

   - In src/thirdparty/oomph-lib/include, you find the necessary main files of oomph-lib, [LGPL v2.1 or later license]. Minor modifications as mentioned in src/thirdparty/INFO_oomph-lib had to be made. Furthermore, code parts of these oomph-lib files had been copied to corresponding derived classes of pyoomph.
   - A copy of the header-only library nanoflann is located in src/thirdparty/nanoflann.hpp, [BSD license]:


```
Software License Agreement (BSD License)

Copyright 2008-2009  Marius Muja (mariusm@cs.ubc.ca). All rights reserved.
Copyright 2008-2009  David G. Lowe (lowe@cs.ubc.ca). All rights reserved.
```


(continues on next page)

(continued from previous page)

```
Copyright 2011 Jose L. Blanco (joseluisblancoc@gmail.com). All rights␣
↪reserved.

THE BSD LICENSE

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

- A copy of the header-only library delaunator-cpp is located in src/thirdparty/delaunator.hpp, [MIT license]:

```
MIT License

Copyright (c) 2018 Volodymyr Bilonenko

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

- The file src/pyginacstruct.hpp is strongly based on the file structure.h of GiNaC, [GPL v2 or later license].
- A copy of the library Project Nayuki/smallest enslosing circle, [LGPL v3 or later license] is added (after adding type specifications) to pyoomph/utils/smallest_circle.py.
- Also, when using materials or the thermodynamic activity models AIOMFAC, original UNIFAC or modified UNIFAC (Dortmund), *please cite the relevant publications*.

The third-party licenses/acknowledgement files can be also found in src/thirdparty.

2. During compilation, pyoomph includes/links against or makes use of the following libraries:
   - GiNaC, [GPL v2 or later license], also statically linked in the distribution as python wheels
   - CLN, [GPL v2 or later license], also statically linked in the distribution as python wheels
   - MPI, depending on the system e.g. OpenMPI [3-clause BSD license], MPICH [MPICH license], Microsoft MPI [MIT license], note that MPI support is experimental and deactivated in the python wheels
   - python3.8+, [PSF license], also dynamically linked in the distribution as python wheels
   - pybind11, [BSD-style license], also statically linked in the distribution as python wheels
   - pybind11-stubgen, [BSD-style license]
   - pip, [MIT license]
3. Beyond that, pyoomph makes use of the following libraries at runtime. During installation with pip, many (but not all) of these libraries are automatically fetched as requirements.
   - python core libraries, [PSF license]
   - numpy, [BSD license]
   - pygmsh, [GPL v3 license]
   - gmsh, [GPL v2 or later license]
   - meshio, [MIT license]
   - mpi4py, [BSD 2-Clause “Simplified” license]
   - scipy, [BSD-3-Clause license]
   - matplotlib, [PSF-based license]
   - mkl, [Intel Simplified Software license]
   - petsc and petsc4py, [BSD 2-Clause license]
   - slepc and slepc4py, [BSD 2-Clause license]
   - vtk, [BSD 3-clause license]
   - paraview, [BSD 3-clause license]
   - setuptools, [MIT license]
   - pybind11-stubgen, [BSD 3-Clause license] is used to generate python stubs from the C++ core
   - cibuildwheel, [BSD 2-Clause license] is used to compile the provided wheels
   - tccbox used to invoke the TinyC compiler

   Be aware that some of these libraries can have further dependencies.

1.1.1 Acknowledgements

The authors gratefully acknowledge financial support by the Industrial Partnership Programme Fundamental Fluid Dynamics Challenges in Inkjet Printing of the Netherlands Organisation for Scientific Research (NWO) & High Tech Systems and Materials (HTSM), co-financed by Canon Production Printing Netherlands B.V., IamFluidics B.V., TNO Holst Centre, University of Twente, Eindhoven University of Technology and Utrecht University. This work was supported by an Industrial Partnership Programme, High Tech Systems and Materials (HTSM), of the Netherlands Organisation for Scientific Research (NWO); a funding for public-private partnerships (PPS) of the Netherlands Enterprise Agency (RVO) and the Ministry of Economic Affairs (EZ); Canon Production Printing Netherlands B.V.; and the University of Twente.

1.2 How to cite

At the moment, just cite the following paper for *pyoomph*:

- Christian Diddens and Duarte Rocha, *Bifurcation tracking on moving meshes and with consideration of azimuthal symmetry breaking instabilities*, J. Comput. Phys. **518**, 113306, (2024), doi:10.1016/j.jcp.2024.113306.

Please mention that *pyoomph* is based on *oomph-lib* and *GiNaC*, i.e. **also cite at least**:

- M. Heil, A. L. Hazel, *oomph-lib - An Object-oriented multi-physics finite-element library*, Lect. Notes Comput. Sci. Eng. **53**, 19-49, (2006), doi:10.1007/3-540-34596-5_2.
- C. Bauer, A. Frink, R. Kreckel, *Introduction to the GiNaC framework for symbolic computation within the C++ programming language*, J. Symb. Comput. **33** (1), 1-12, (2002), doi:10.1006/jsco.2001.0494.

Citing of material properties and activity models

When using UNIFAC-like group contribution methods (cf. Section 8.8), you please cite the following:

- The published parameters of the original and modified (Dortmund) UNIFAC model were taken with kind permission from the *DDBST* website https://www.ddbst.com. Please cite the papers listed at https://www.ddbst.com/published-parameters-unifac.html for original UNIFAC and https://www.ddbst.com/PublishedParametersUNIFACDO.html for modified (Dortmund) UNIFAC.
- Also note that the *UNIFAC Consortium* provides *updated and revised parameters*, which will increase the accuracy of the predicted activity coefficients. Please refer to the https://unifac.ddbst.com/unifac_.html for more information. These updated parameters are not included in pyoomph.
- When using the AIOMFAC model, please cite the papers listed here https://aiomfac.lab.mcgill.ca/citation.html.
- When using the material properties from `pyoomph.materials.default_materials`, please have a look at the comments in this file to cite the correct papers.

1.3 Motivation

I would like to start this tutorial for pyoomph with a little history to shed some light into the decision to develop pyoomph. Feel free to skip this section!

In 2015, I started my postdoc at the TU Eindhoven with Professor Hans Kuerten where I generalized an existing Fortran code for the evaporation of sessile droplets to account for multi-component mixture droplets [12, 42]. Since this code was based on lubrication theory, a reviewer of my first publication on this topic [12] asked for a validation at higher contact angles, i.e. where the lubrication theory might fall short. Inspired by the work of Hu & Larson on pure droplets [30, 31], who used a finite element method for this validation step, I found FEniCS (fenicsproject.org) [1, 38] as a quick and simple solution to easily use the finite element method, but only after I noticed that writing a finite element library myself is quite

demanding. It is in fact reinventing the wheel given all the nice existing open source libraries. With FEniCS, things are so simple: just type a few lines in Python and you directly get your solution - perfect!

This nice experience gave rise to move the previous lubrication theory Fortran code to a full Navier-Stokes simulation of evaporating multi-component droplets in FEniCS [13, 17, 35, 37, 43, 44]. However, while FEniCS is very powerful in quickly developing simple simulations, it has serious downsides for free surface problems and multi-physics scenarios, where individual equations have to be solved on different domains. Also the arbitrary Lagrangian-Eulerian method (ALE) cannot, to my best knowledge, be well implemented in FEniCS, at least not in a monolithic solution approach. This method, however, is very powerful for the problems I was working on that times. By accident, I came across the finite element library oomph-lib (oomph-lib.maths.man.ac.uk) [28]. This toolbox offered everything I was looking for: free surfaces, multi-physics, ALE methods and the possibility to add surfactants were already ready to be used more or less out of the box. Even more, spatial and temporal adaptivity, arc length continuation and bifurcation tracking are already part of oomph-lib. So I again rewrote the code, now with the help of the very flexible and powerful library oomph-lib. In direct comparison with FEniCS, however, the definition of custom equations in oomph-lib is a demanding and time-consuming task, since the entire assembly of the spatially discretized weak forms of your equations have to be written by hand in C++. In fact, for best speed and convergence behavior, it is even required to also fill the Jacobian matrix of the weak form by hand, which easily ends up in writing 1000 lines of code for a rather simple equation.

While my work with oomph-lib resulted in several publications [7, 10, 14, 24, 27, 34, 36], the limitations became more and more obvious: Each new equation you want to introduce in your system requires a lot of code writing and compilation, as well as substantial code overhead to couple all equations at the end. I was longing to return to the quick development approach I had with FEniCS, but getting all the benefits of oomph-lib.

I therefore decided to aim for the best of both worlds - and pyoomph was born. . .

In 2022, Duarte Rocha joined our group as PhD student and started to contribute to the development of pyoomph.

1.4 General Information

pyoomph is a library for Python which internally consists of two parts: The first part is a compiled dynamic library which contains a stripped version of the finite element toolbox oomph-lib and the required connection to expose it to Python via the library pybind11. The second part is a surrounding framework written in Python, where custom equations, a material library and further tools, like e.g. the calculation of activity coefficients via group contribution methods, are implemented. However, doing heavy computations (like the assembly of large residual vectors or Jacobian matrices) directly in Python is not desirable, since Python is a script language which is substantially slower than optimized compiled code. To that end, pyoomph also contains a just-in-time compilation step: All equations are defined and mutually linked across different physical domains in Python. The equations required in the physical system of interested are converted to C code via the library GiNaC. These C codes are compiled in the background and subsequently loaded back into pyoomph. This step happens automatically whenever a simulation is started. It might result in some seconds of waiting time for the compilation, but this pays off very quickly when multiple time steps or a large number of degrees of freedom are considered. In that sense, pyoomph has quite much in common with FEniCS, however with more focus on monolithic solving for coupled multi-physics domains and less focus on a comprehensive zoo of all kind of possible finite element spaces.

1.5 When to use pyoomph, and when better use something else

As every numerical framework, pyoomph has some strong points but of course also plenty of limitations.

You can consider using pyoomph

- when you want to have a simple python interface, but still want to have high computational speed
- you want to quickly setup a multi-physics problem
- you are too lazy to nondimensionalize your equations by hand before implementation
- you want to write equations only once and reuse them in different coordinate systems, potentially in combination with other equations
- for problems involving multi-component & multi-phase flow, including Marangoni flow, mass transfer and surfactants
- when you don't want to code a lot of matrix filling routines by hand
- when you want to track (azimuthal) bifurcations
- when you want to use a monolithic sharp-interface moving mesh method

You should consider using something else

- when you want to use all features of oomph-lib.
- when you want to operate on a lower level for more flexibility
- when you need highly parallelize computational power
- for computationally expensive three-dimensional problems
- for high Reynolds numbers (go for e.g. advanced finite differences as in AFiD)
- for topological changes (the sharp-interface method is not well suited for this, go for VoF instead, e.g. Basilisk)
- if you need more fancy finite-element spaces (go for FEniCS or NGSolve)
- if you need spline basis functions (go for nutils)

CHAPTER

TWO

INSTALLATION

Depending on your operation system (Windows, Mac and Linux are supported), you have to perform different steps to install pyoomph. On all operation systems, you should have a python version 3.9 to 3.13 already installed on your system.

2.1 Installing the precompiled wheels

The easiest method to get pyoomph installed on your system is using `pip`, i.e. just enter the following in a terminal:

```
python -m pip install pyoomph
```

This will install pyoomph on your system. However, please also read the system-specific steps below.

If you get errors, let us know (c.diddens@utwente.nl), and we see whether we can provide a suitable wheel for your system.

> **Warning:** If you are using a recent Mac with the Apple silicon (arm64 architecture) processor, you must execute this command in a Rosetta terminal. At https://www.courier.com/blog/tips-and-tricks-to-setup-your-apple-m1-for-development/ you can find instructions how to create such a Rosetta terminal (**note**: recent systems must be handled differently, see e.g. here: https://developer.apple.com/forums/thread/718666). Also, please see below regarding the *mkl* module.
>
> You can install pyoomph from source directly on arm64, but unfortunately without support for the fast MKL Pardiso solver. See Section 2.2.1 for details. Also, you might have to use a less recent version of python. See https://github.com/pyoomph/pyoomph/blob/main/Mac_arm64_with_Pardiso.md for details.

Depending on your system, you have to do additional steps to obtain the full performance:

2.1.1 Windows

pyoomph can use a minimal C compiler, namely the TinyC compiler (TCC), wrapped by the `tccbox` package. This compiler generates machine code very quickly, but the generated code is usually not well optimized so that the execution is slow compared to more sophisticated compilers. On Windows, Microsoft offers the MS Build Tools, a free compiler suite, which can be utilized by pyoomph to generate faster machine code. It is best to download the Build Tools for Visual Studio 2019, since this is the version Python is usually referring to. The installer can be found at https://aka.ms/vs/16/release/vs_buildtools.exe. It is important to install at least the following components:

If you do not want to install MS Build Tools for any reason, you always can use the internal TinyC compiler. To do so, call the method `set_c_compiler("tcc")` of the `Problem` class so select the internal compiler. This has to be done for each problem and before any calls of the methods `initialise()`, `output()`, `solve()` or `run()`. Alternatively, you can add the command line arguments *–tcc*, e.g. run a your simulation script `my_simulation.py` as follows:

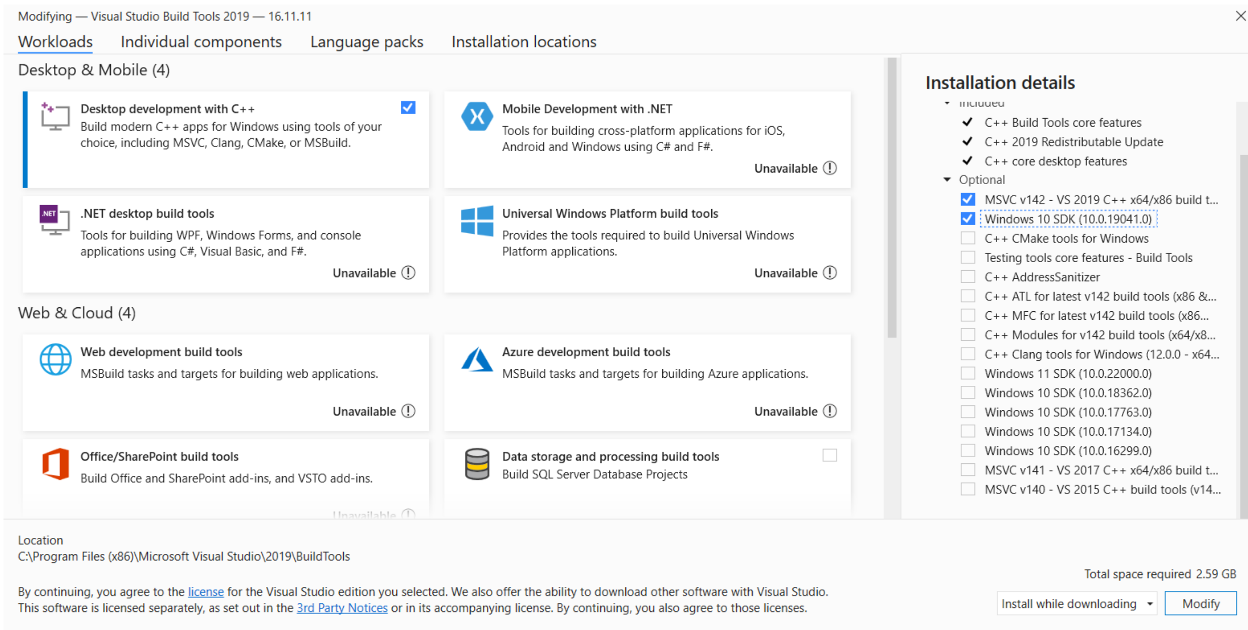

Fig. 2.1: Required packages to install from MS Build Tools

```
python my_simulation.py --tcc
```

Note: If you encounter segmentation faults during solving, you likely have a bugged version of the MKL package installed. In that case, please downgrade to an older version, e.g. via *python -m pip install mkl==2024.1.0*.

2.1.2 Mac

On Mac, `clang` will be used as high performance compiler. To get `clang`, install the developer tools via

```
xcode-select --install
```

Warning: If you are using a recent Mac with an Apple silicon processor (arm64 architecture), make sure to not upgrade the package `mkl`. Also on Macs with an Intel processor, more recent versions can cause a crash. If you by accident upgrade your mkl package, reset it by entering (in a Rosetta 2 terminal for arm64 chips):

```
python -m pip install mkl==2021.4.0
```

2.1.3 Linux

On Linux, make sure that you have the `gcc` compiler installed to get optimal performance, e.g. on Ubuntu by

```
sudo apt install gcc
```

Other Linux distributions, you might have to use `yum`, `pacman`, etc., instead.

Note: If you encounter segmentation faults during solving, you likely have a bugged version of the MKL package installed. In that case, please downgrade to an older version, e.g. via *python -m pip install mkl==2024.1.0*.

2.1.4 Trying whether pyoomph works

To check whether pyoomph has been installed and the compilers and solvers can be detected, try it with

```
python -m pyoomph check all
```

2.1.5 Updating pyoomph

Pyoomph is under continuous development and the wheels are regularly updated. To update pyoomph to the recent version, just do a

```
python -m pip install --upgrade pyoomph
```

2.2 Compilation from source

The following sections discuss the possiblity of building pyoomph from source. This is only covered for Mac and Linux, since it is quite complicated to compile all dependencies on Windows.

2.2.1 On Mac

Warning: If you are using a recent Mac with an Apple silicon (arm64 architecture) processor, you might encounter some problems, since not all required python packages are present in the pip repository yet. Therefore, you must use Rosetta 2 to emulate the x86 64 architecture. You must execute the following commands in a Rosetta terminal. At https://www.courier.com/blog/tips-and-tricks-to-setup-your-apple-m1-for-development/ you can find instructions how to create such a Rosetta terminal. On more recent systems, please refer to https://developer.apple.com/forums/thread/718666 to setup a corresponding terminal.

Note: If you do not want to use the *Rosetta 2 terminal* detour, you can also install it directly on arm64 systems. Vatsal Sanjay from CoMPhy Lab has contributed an installation script

```
bash installOnArm.sh
```

to perform this installation. The only downside is that the fast *MKL Pardiso* solver is not supported on native arm64 systems. For optimal performance, in particular for larger problems, one therefore should consider the *Rosetta 2 terminal* way for the time being. Also, you might have to use a less recent version of python. See https://github.com/pyoomph/pyoomph/blob/main/Mac_arm64_with_Pardiso.md for details.

To clone the git repository, you require git, but this comes along with the Xcode developer tools, which is required anyhow. The latter can be installed via

```
xcode-select --install
```

for a terminal. After that, you should have git so that you can clone the repository:

```
git clone https://www.github.com/pyoomph/pyoomph.git
```

Before building it, a bunch of additional software has to be installed. For Mac, there is e.g. homebrew (https://brew.sh), which easily manage these additional packages. Hence, install homebrew by pasting the installation command from https://brew.sh.

Afterwards, you can install some required tools, by

```
brew install openmpi ccache ginac
```

You might have to close and reopen the (Rosetta) terminal now. Afterwards, you first have to build the stripped and slightly modified version of oomph-lib which is shipped along with pyoomph:

```
cd <PYOOMPH_DIR>
bash ./prebuild.sh
```

where `<PYOOMPH_DIR>` is the directory of your local pyoomph repository.

Before building pyoomph, we first have to make sure that additional python packages are installed. This can be done e.g. by

```
python3 -m pip install pybind11 gmsh mpi4py matplotlib numpy pygmsh scipy meshio␣
↪pybind11-stubgen setuptools
```

The `python3` command might be also `python`, depending on the system. Be sure to use the more recent version of python.

On Mac, make sure you have not upgraded your mkl package to the recent version, which actually crashes on Mac:

```
python3 -m pip install mkl==2021.4.0
```

For older Macs with Intel processor, you can install the recent version

```
python3 -m pip install mkl
```

Afterwards, you should be able to build pyoomph by:

```
cd <PYOOMPH_DIR>
bash ./build_for_develop.sh
```

In the worst case, try to execute the last command up to three times. If it still does not work, please contact me at c.diddens@utwente.nl. Finally, check whether everything works well via:

```
python3 -m pyoomph check all
```

2.2.2 On Linux

To obtain the code, clone the GitHub repository

```
git clone https://www.github.com/pyoomph/pyoomph.git
```

Once you have cloned the repository with git, you first have to install a few packages. On a Debian/Ubuntu distribution, you habe to do e.g.

```
sudo apt-get install libopenmpi-dev pybind11-dev libginac-dev libcln-dev libgmp-dev␣
↪ccache libmkl-rt
```

There are additional python packages required. You can either install these with `python3 -m pip install ...` (note that you might have to use `python` or `python3` as command) or find the corresponding Linux packages. If you do not install them now, they should be installed during the first build of pyoomph via `pip`. The required `python` libraries are

```
gmsh mkl mpi4py matplotlib numpy petsc4py pybind11 pygmsh scipy meshio pybind11-
↪stubgen setuptools wheel
```

Make sure you have recent versions, e.g. when using `pip`, you could do

```
python -m pip install --upgrade gmsh mkl mpi4py matplotlib numpy petsc4py pybind11␣
↪pygmsh scipy meshio pybind11-stubgen setuptools wheel
```

Afterwards, you first have to build the stripped and slightly modified version of oomph-lib which is shipped along with pyoomph.

```
cd <PYOOMPH_DIR>
bash ./prebuild.sh
```

where `<PYOOMPH_DIR>` is the directory of your local pyoomph repository.

Finally, build pyoomph and install it, and check whether it works:

```
cd <PYOOMPH_DIR>
bash ./build_for_develop.sh
python -m pyoomph check all
```

Note: If you encounter segmentation faults during solving, you likely have a bugged version of the MKL package installed. In that case, please downgrade to an older version, e.g. via *python -m pip install mkl==2024.1.0*.

2.3 Further practical software

2.3.1 VSCode as Python IDE

As an editor and IDE for the Python scripts, we highly recommend downloading and installing Visual Studio Code from code.visualstudio.com. After downloading and installing, make sure to install the Python extensions. If you have installed pyoomph in a *Conda Environment*, make sure to use this environment for the Python interpreter. One rather important setting is to activate the option *Python > Terminal: Execute In File Dir*. Thereby, the output of each simulation will be written in a subdirectory with the same name as the simulation script, without the .py extension. To easily find it, search for *python.terminal.executeInFileDir* in the settings search bar.

You also might want to install the extension PyLance, which allows for type checking and error highlighting during development.

2.3.2 Paraview to visualize the output

While pyoomph can be used without any visualization tools, e.g. by plotting the resulting data by hand or using the plotting framework of pyoomph, it is beneficial to install a viewer for VTU/PVD files, which are the default output files. The typical free software to visualize these files is Paraview, which can be downloaded for free at www.paraview.org. You can open all .vtu and .pvd files you find in the output folder of a pyoomph simulation.

2.4 Optional installation of PETSc/SLEPc

If you want to solve for eigenvalue problems, pyoomph by default will invoke scipy's eigensolver based on ARPACK. However, for unsymmetric matrices which usually arise in complicated problems, SLEPc provides a much more stable alternative. So whenever you want to investigate linear stability, you should consider performing the following steps. The are all optional, but usually give better and more stable eigenvalue results. In any case, it is advised to occasionally check your eigenvalues by adding `report_accuracy=True` to calls of `solve_eigenproblem()`. We unfortunately do not really know how to install SLEPc on Windows, so you have to find your own way of installing it (and let us know the steps. A good start can be found here).

SLEPc depends on PETSc, so it is advisable to install both of these packages together. Also, since we often will obtain matrices with a zero on a diagonal (mainly due to Lagrange multipliers, incompressibilty constraints, etc.) we need a suitable linear solver backend in PETSc which can perform pivoting. We usually use MUMPS for that. Also, if you want to solve normal mode eigenvalue problems (cf. Section 10.3 and Section 10.4), we must have support for complex-valued eigenvalue problems, which requires to compile PETSc/SLEPc and MUMPS with complex values.

All three packages can be downloaded an installed together. On Mac with an M1 (arm64) chip, this must be again done in a Rosetta 2 terminal, at least if you want to use MKL Pardiso as linear solver (see previous pages).

We start by downloading PETSc in a folder of our choice (replace `A_FOLDER_OF_YOUR_CHOICE` in the following accordingly). If you have installed pyoomph in a python environment, it is advisable to also activate this environment now.

```
cd A_FOLDER_OF_YOUR_CHOICE
git clone -b release https://gitlab.com/petsc/petsc.git petsc
cd petsc
```

We know have to export some environment variables:

```
export PETSC_DIR=$(pwd)
export PETSC_ARCH=pyoomph_petsc_arch
```

Note that the choice of the name `pyoomph_petsc_arch` can be changed arbitrarily.

We then have to make sure that we have flex and Bison. On Ubuntu (and other Linux types analogously), you can install it system-wide via

```
sudo apt install flex bison
```

Alternatively, you can let PETSc download it as well by adding `--download-bison` at the end of the following configuration command. Note that we download and install further solver pacakges here, which are usually not needed, but likely will be used in future.

```
./configure --with-mpi  --with-petsc4py --download-mumps=yes --download-hypre=yes --
↪download-parmetis=yes --download-ptscotch=yes --download-slepc=yes --download-
↪superlu=yes --download-superlu_dist=yes --download-suitesparse=yes --download-
↪metis=yes --download-scalapack --with-scalar-type=complex
```

You can also add optimization or OpenMP support, e.g. `--with-debugging=0`, `-with-openmp`, `--with-openmp-kernels`. For all details, please call `./configure --help`.

Note:

If you should have issues with *cmake* on Ubuntu (and potentially other distros), try

1. Install cmake (updated version, see https://askubuntu.com/questions/355565/how-do-i-install-the-latest-version-of-cmake-from-the-command-line)
2. add flag `--download-fblasapack=1` when configuring

At the end of the configuration process, a `make` command will be written, which you have to execute as a next step.

Afterwards, PETSc/SLEPc is installed to the folder `A_FOLDER_OF_YOUR_CHOICE/petsc/pyoomph_petsc_arch`. At the end, it will also show a test command, by what you can test the basic functionality of of your installation.

To use it within pyoomph, you have to make sure that you always

```
export PETSC_DIR=A_FOLDER_OF_YOUR_CHOICE/petsc
export PETSC_ARCH=pyoomph_petsc_arch
export PYTHONPATH=$PYTHONPATH:$PETSC_DIR/$PETSC_ARCH/lib
```

It is advised to copy these statements into your `.bashrc` or `.zshrc` (depending on the terminal you use). Alternatively, if you use a python environment for pyoomph, you can also put these in the `activate` script of the environment. Note, however, that these won't be unset automatically if you deactivate the environment then, only if you close the terminal.

To use SLEPc with MUMPS as eigensolver, either set it in python during your driver code, e.g.

```
problem.set_eigensolver("slepc").use_mumps()
```

or supply the flag `--slepc_mumps` when calling your driver code:

```
python my_eigenvalue_simulation.py --slepc_mumps
```

2.5 Typical problems with the installation

I the following, there are some typical errors listed, which may occur after installing and trying pyoomph:

distutils.errors.DistutilsPlatformError: Unable to find vcvarsall.bat

This happens on Windows if you have not installed MS Build Tools (cf. Section 2.1.1). Either install it or set the compiler to TinyC, which can be done by calling the method `set_c_compiler("tcc")` of your problem instance or by passing the command line argument *–tcc*.

CHAPTER

THREE

TEMPORAL ORDINARY DIFFERENTIAL EQUATIONS

We start our tutorial by the simplest form of equations pyoomph can handle, namely (systems of) ordinary differential equations (ODEs) as function of time. When reading this section, one might ask: *can't this be done easier, e.g. with MATLAB?* - And the answer is probably *yes*. And for simple ODEs, MATLAB will be even faster than pyoomph. However, all steps performed and explained in this section are relevant for the understanding of the assembly of complicated coupled multi-physics problems later on. In fact, when a system of spatio-temporal differential equations is spatially discretized, one ends up in a large system of ODEs. Furthermore, ODEs offer the possibility to discuss a few steps analytically and thereby show what pyoomph actually does.

This section is devoted to (a set of) ordinary differential equations in the sense of an initial value problem. In physical problems, the independent variable is usually the time. If you want to solve a boundary value problem for (a set of) ordinary differential equations, please refer to Section 4. For such purposes, the independent variable is sampled on a one-dimensional mesh, i.e. you will use a spatial coordinate instead of the time as independent variable.

3.1 Nondimensional harmonic oscillator

A simple but yet illustrative example is a harmonic oscillator (without any physical units).

3.1.1 Mathematical Formulation

Let us start with the harmonic oscillator equation for a non-dimensional unknown $y(t)$ as function of a non-dimensional time t, i.e.

$$\partial_t^2 y + \omega^2 y = 0\,.$$

with $\omega = 1$ and with the initial condition $y(0) = 1$ and $(\partial_t y)(0) = \dot{y}(0) = -1$.

Note: Note that we use ∂_t here instead of $\mathrm{d}/\mathrm{d}t$ as notation for the time derivative, since we will later turn to spatio-temporal equations and in both cases, the time derivative in pyoomph is obtained by the function `partial_t()`.

3.1.2 Python code using the predefined harmonic oscillator equation class

For the Python code, we first have to import pyoomph:

```
# Import pyoomph
from pyoomph import *
# Also import the predefined harmonic oscillator equation
from pyoomph.equations.harmonic_oscillator import HarmonicOscillator
```

Every problem you want to solve in pyoomph has to be defined in a class inherited from the generic `Problem` class. In the constructor of this class, i.e. the method `__init__()`, one should set the default parameters, e.g. here $\omega = 1$. Later on, we can change this parameter before running the simulation, but some default value has to be set. However, before doing so, we first have to call the constructor of the generic problem class, which can be done by a `super()` call:

```
# Create a specific problem class to solve your problem. It is inherited from the␣
↪generic problem class 'Problem'
class HarmonicOscillatorProblem(Problem):

    # In the constructor of the problem, we can set some default values. here omega
    def __init__(self):
            super(HarmonicOscillatorProblem,self).__init__() #we have to call the␣
↪constructor of the parent class
            self.omega=1 #Set the default value of omega to 1
```

The next important step is the definition of the problem. In our case, we have three ingredients:

1. We want to solve a harmonic oscillator,
2. with $y(0) = 1$ and $\dot{y}(0) = -1$ as initial condition
3. and finally, we also want to get some output, namely the curve of $y(t)$.

pyoomph takes here the approach, that all these elements can be combined by the operator `+`. We therefore overload the method `define_problem()` of the `Problem` class as follows:

```
# The method define_problem will define the entire problem you want to solve. Here,␣
↪it is quite simple...
def define_problem(self):
        eqs=HarmonicOscillator(omega=self.omega,name="y") #Create the equation,␣
↪passing omega
        eqs+=InitialCondition(y=1-var("time")) #We can set both initial conditions␣
↪for y and y' simultaneously
        eqs+=ODEFileOutput() #Add an output of the ODE to a file
        self.add_equations(eqs@"harmonic_oscillator") #And finally, add this combined␣
↪set to the problem with the name "harmonic_oscillator"
```

Let us go through these steps once more: The function `define_problem()` will be called, whenever we try to run the simulation with our problem class `HarmonicOscillatorProblem`. Within this method, we first create an instance of the `HarmonicOscillator` equations (which will be re-implemented by hand in the next section). There, we pass `omega` ($=\omega$) from the problem to the equation class and inform the equation class that the unknown variable of the harmonic oscillator shall be called y.

In the next step, this equation is augmented with an instance of `InitialCondition`, setting the initial condition of $y(t)$ to $y(t) = 1 - t$. Here, we use the function `var()` with the argument `"time"` to obtain the time variable t and combine it into an expression `1-var("time")`, i.e. $1 - t$. However, above we stated that the initial condition shall be $y(0) = 1$ and $\dot{y}(0) = -1$. But in fact, if one evaluates $1 - t$ and its temporal derivative at $t = 0$, one indeed recovers these initial conditions. Hence, this trick allows to express the required two initial conditions by a single statement. Even more, one can e.g. set the initial condition from a known analytical solution, e.g. via `InitialCondition(y=A*cos(self.omega*var("time")+phi))` for some amplitude `A` and phase `phi`.

Finally, all simulations are meaningless if there is no output. Therefore, an instance of the output class `ODEFileOutput` is added to the combined object of `HarmonicOscillator+InitialCondition`. This object will make sure, output of the ODE is written to a file.

At this stage, the combined object of the harmonic oscillator ODE, the initial condition and the output object are just stored in the local variable `eqs`, i.e. the problem class does not know yet that we want to solve this. This is done by adding the combined equation object to the problem by the method `add_equations()`. However, before doing so, the equations have to be restricted to a particular named domain. For ODEs, this domain name is quite arbitrary, but for multi-physics, the domain name identifies the spatial domain where the equations shall be solved, e.g. in either the gas or the liquid phase in multi-phase problems. To restrict equations to a domain, the operator `@` is used, followed by the desired domain name. For the case of the ODE here, the domain name (`"harmonic_oscillator"` here), will just influence the name of the output file.

The only step remaining at this stage is running the simulation:

```
if __name__=="__main__":
    with HarmonicOscillatorProblem() as problem:
            problem.run(endtime=100,numouts=1000)
```

Here, it is first check whether the current script is the main script, a typical step in Python scripts to allow for the import of definitions in external scripts without invoking any code execution. In the next step, an instance of the problem is created and finally, the simulation is started and solved until $t = 100$ using the method `run()`. The number of (here equidistant) output steps is set to 1000, i.e. an output interval of $\Delta t_{\text{out}} = 0.1$.

To run the script, just invoke Python on it, either via a command line or via an Python IDE.

By default, the output will be written to a sub-directory of the current directory with the name of the executed script, but without the extension `.py`. If the script is called `nondim_harmonic_osci.py`, the output can be found in the sub-directory `nondim_harmonic_osci/`. Here, multiple files can be found. In the sub-folder `_ccode`, we can find the generated C code of the equations. The sub-directory `_states` contains state file, which allow to continue a previously interrupted simulation (cf. Section 5.5 later on). Since we have added an `ODEFileOutput` object to our equations (defined on the domain `"harmonic_oscillator"`), we will also find a text file `harmonic_oscillator.txt` containing the curve $y(t)$ as raw text file. A plot of this output file is depicted in Fig. 3.1.

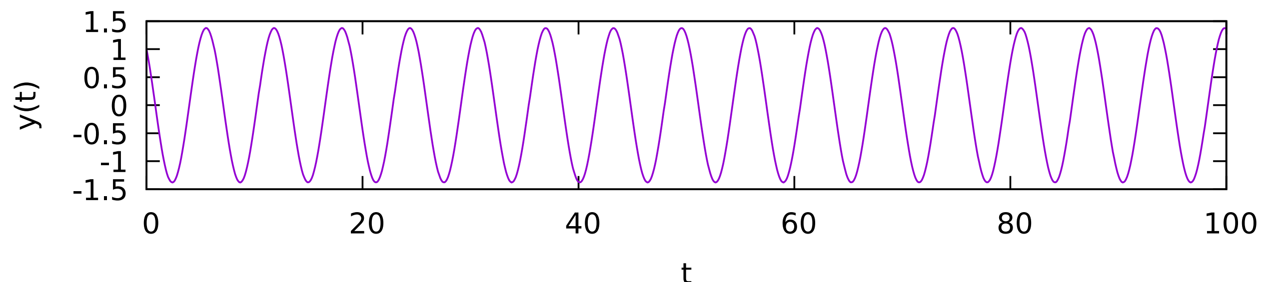

Fig. 3.1: Output for the simple harmonic oscillatior problem

3.2 Defining your own harmonic oscillator equations

Up to now, we have loaded the predefined harmonic oscillator equations with the line

```
from pyoomph.equations.harmonic_oscillator import HarmonicOscillator
```

However, one of the main features of pyoomph is actually the definition of arbitrary equations. Instead of including the predefined equation class, we will write our own class, inheriting from the generic `ODEEquations`. This will serve as an example how to express arbitrary ODEs within pyoomph. At the same time, let us generalize the equation to include damping δ and driving $f(t)$ as follows

$$\partial_t^2 y + 2\delta\partial_t y + \omega^2 y = f$$

The corresponding code reads as follows:

```
from pyoomph import * # Import pyoomph
from pyoomph.expressions import * # Import some additional things to express e.g.␣
↪partial_t

# We define a new class called HarmonicOscillator, which is inherited from the␣
↪generic ODEEquations
class HarmonicOscillator(ODEEquations):
    # Constructor, allow to set some parameters like the name of the variable, omega,
↪ damping and driving
    def __init__(self,*,name="y",omega=1,damping=0,driving=0):
        super(HarmonicOscillator,self).__init__()
        self.name=name #Store these as members of the equation object
        self.omega=omega
        self.damping=damping
        self.driving=driving

    # This function is called to define all fields in this ODE (system)
    def define_fields(self):
        self.define_ode_variable(self.name)

    # This function will finally define the equations
    def define_residuals(self):
        y=var(self.name) # bind the local variable y to the ODE variable
        # Write the equation in residual form, i.e. lhs-rhs=0
        residual=partial_t(y,2)+2*self.damping*partial_t(y)+self.omega**2 *y -␣
↪self.driving
        # And add the residual to the equation. Here, we have to project it on␣
↪the test function.
        self.add_residual(residual*testfunction(y))
```

In the `__init__()` method, we take optional keyword arguments which have default values. These are then stored as members of the class. Of course, we have to call again the constructor of the super-class `ODEEquations` using the Python builtin `super`.

In the next step, the method `define_fields()` is overloaded. Inside this functions, all unknowns of the ODE system have to be defined. Here, it is just the single unknown y.

Finally, the nitty-gritty, the equation has to be defined. This happens in the method `define_residuals()`. To that end, the equation first has to be cast to a residual form, which can be done by putting all terms on one side of the equation:

$$\partial_t^2 y + 2\delta\partial_t y + \omega^2 y - f = 0$$

Of course, only the lhs of this equation is of relevance. Before adding it to the system of equations with the `add_residual()` method, it must be multiplied by a so-called *test function*, which can be obtained by the function

`testfunction()`. This all happens in the line `self.add_residual(residual*testfunction(y))`. Here, we only have one equation to be solved, i.e. the harmonic oscillator. In general, one will have a system of ODEs, i.e. multiple equations for multiple unknowns. This is where test functions become important, which will be explained in the next section.

The remainder of the code is very similar as before, but now also damping and driving will be considered:

```
# The remainder is almost the same is in the example nondim_harmonic_osci.py
class HarmonicOscillatorProblem(Problem):

    def __init__(self):
        super(HarmonicOscillatorProblem,self).__init__()
        self.omega=1
        self.damping=0.1 #But we add some default damping here
        t=var("time")
        self.driving=0.2*cos(0.2*t) #and some driving


    def define_problem(self):
        eqs=HarmonicOscillator(omega=self.omega,damping=self.damping,
↪driving=self.driving,name="y") #We also pass the damping and driving here
        eqs+=InitialCondition(y=1-var("time"))
        eqs+=ODEFileOutput()
        self.add_equations(eqs@"harmonic_oscillator")


if __name__=="__main__":
    with HarmonicOscillatorProblem() as problem:
        problem.run(endtime=100,numouts=1000)
```

The output is plotted in Fig. 3.2.

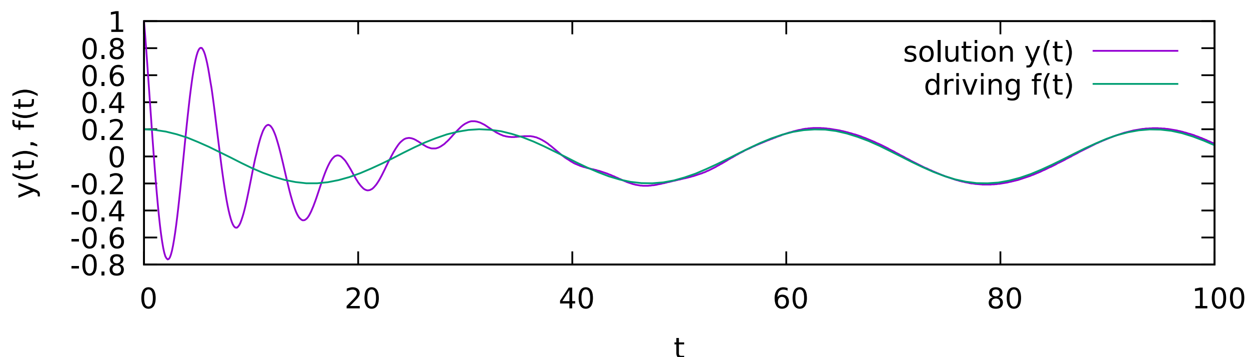

Fig. 3.2: Output for the user-defined harmonic oscillatior equation with damping and driving.

3.3 Test functions and the residual form

Suppose we have a system of N equations for N degrees of freedom. Let us assume the degrees of freedom are stored in the N-dimensional vector of unknowns $\vec{U}$. If all equations are written into residual form, we can assemble a vectorial residual vector $\vec{\mathcal{R}}$, which gives the solution of the system once

$$\vec{\mathcal{R}}(\vec{U}) = \vec{0}\,. \tag{3.1}$$

In the previous example of the harmonic oscillator, the system is one-dimensional, i.e. $N = 1$, $\vec{U} = [y(t)]$ and $\vec{\mathcal{R}} = \left[\partial_t^2 y + 2\delta\partial_t y + \omega^2 y - f\right]$.

One can easily separate (3.1) into distinct equations again, if one takes the dot product with an arbitrary vector $\vec{V}$, i.e.

$$\vec{V}\cdot\vec{\mathcal{R}}(\vec{U}) = \sum_{i=0}^{N}\mathcal{R}_i(\vec{U})V_i = 0\,. \tag{3.2}$$

Of course, in order to ensure that both formulations are equivalent, it is fundamental to demand that $\vec{V}$ is arbitrary, i.e. (3.2) has to be fulfilled for all choices of $\vec{V}$. However, due to linearity of the projection of $\vec{R}$ on $\vec{V}$, it is sufficient to require that (3.2) has to be fulfilled for N linearly independent choices of $\vec{V}$, i.e. $\vec{V}^{(j)}$ for $j = 1,\ldots,N$. The trivial choice is of course to have $\vec{V}_i^{(j)} = \delta_{ij}$ using the Kronecker-δ, i.e. the ith component of the jth vector $\vec{V}^{(j)}$ is 1, all other entries are 0. This is exactly how it is internally handled by pyoomph when solving ODEs. We hence have

$$\sum_{i=0}^{N}\mathcal{R}_i(\vec{U})V_i^{(j)} = 0 \quad \text{for} \quad j = 1,\ldots,N\,. \tag{3.3}$$

Obviously, (3.3) with the particular choice of $V^{(j)}$ being the unit vectors in the jth direction, is exactly the same as (3.1). Later on, when it comes to spatial differential equations, it is beneficial to express it as in (3.3), i.e. this section serves as a precursor for the weak formulation later on.

When it comes to usability, it is not very handy to assign an index i or j to each degree of freedom by hand. However, since we now have introduced the vectors $V^{(j)}$, which are indeed the so-called *test functions*, pyoomph allows you so select the test function corresponding to a degree of freedom by the unknown itself. In the example above, we have assigned `y` to the degree of freedom $y(t)$ by using the statement `y=var("y")`. We get the corresponding test function $\vec{V}^{(j)}$ not by the index j, but instead by the variable itself, i.e. by `testfunction(y)`.

Finally, the statement `self.add_residual(residual*testfunction(y))` will add the residual part, i.e. the harmonic oscillator in residual form, multiplied by the corresponding test function to the total residual of the equation. This will become more clear in the next example.

3.4 Coupled harmonic oscillators

In the following, we will discuss three different routes to the very same result, namely to couple two harmonic oscillators, i.e.

$$\begin{aligned}\partial_t^2 y_1 + K_{11}y_1 + K_{12}y_2 &= 0\\ \partial_t^2 y_2 + K_{21}y_1 + K_{22}y_2 &= 0\end{aligned} \tag{3.4}$$

Method I

We start by the naive way of implementing this, i.e. by defining a specific equation class, inherited from the generic `ODEEquations` class:

```
from pyoomph import * # Import pyoomph
from pyoomph.expressions import * # Import some additional things to express e.g.␣
↪partial_t

class TwoCoupledHarmonicOscillator(ODEEquations):
     def __init__(self,Kmatrix): #Required to pass the coupling matrix here
            super(TwoCoupledHarmonicOscillator,self).__init__()
            self.Kmatrix=Kmatrix #Store the matrix in the equations object

     def define_fields(self):
            self.define_ode_variable("y1") #define both unknowns
            self.define_ode_variable("y2")

     def define_residuals(self):
            y1,y2=var(["y1","y2"]) #Bind both unknowns to variables
            # Calculate the residuals
            residual1=partial_t(y1,2)+self.Kmatrix[0][0]*y1+self.Kmatrix[0][1]*y2
            residual2=partial_t(y2,2)+self.Kmatrix[1][0]*y1+self.Kmatrix[1][1]*y2

            # Add both residuals
            self.add_residual(residual1*testfunction(y1)+residual2*testfunction(y2))


# The remaining part is analogous to the normal oscillator
class TwoCoupledHarmonicOscillatorProblem(Problem):

     def __init__(self):
            super(TwoCoupledHarmonicOscillatorProblem,self).__init__()
            self.Kmatrix=[[1,-0.5],[-0.2,0.4]] # Some coefficient matrix

     def define_problem(self):
            eqs=TwoCoupledHarmonicOscillator(self.Kmatrix)
            eqs+=InitialCondition(y1=1,y2=0) #Setting initial condition
            eqs+=ODEFileOutput()
            self.add_equations(eqs@"coupled_oscillator")

if __name__=="__main__":
     with TwoCoupledHarmonicOscillatorProblem() as problem:
            problem.run(endtime=100,numouts=1000)
```

Here, in particular the line

```
self.add_residual(residual1*testfunction(y1)+residual2*testfunction(y2))
```

is of interest. It is exactly the formulation as required in (3.3), with $\vec{U} = [y_1(t), y_2(t)]$ and $\vec{\mathcal{R}}$ the vector containing the lhs of (3.4), i.e. `residual1` and `residual2` in the code. The test functions $V_i^{(j)}$ are just obtained by the calls of `testfunction()`. We actually solve now the residual of the first equation in (3.4) by adjusting y_1 and the second equation by adjusting y_2 until both equations are fulfilled.

Method II

The previous method is straightforward, but would require to rewrite a lot of code if e.g. a third oscillator should be coupled to the system. Of course, one could use `len` to obtain the dimension of the coupling matrix `Kmatrix` and perform the calls for `define_ode_variable()` and the residual calculation in a loop. However, pyoomph also offers another way of coupling multiple equations. We have already seen that we can augment an equation e.g. with an

`InitialCondition` object to set an initial condition or an `ODEFileOutput` object to make sure that output is written. By the very same way, also multiple equations can be coupled. To that end, we write an equation class for a general second order equation of the form

$$\partial_t^2 y + T = 0\,,$$

where T is a place holder for an arbitrary mathematical expression. The corresponding equation class looks like this:

```
from pyoomph import * # Import pyoomph
from pyoomph.expressions import * # Import some additional things to express e.g.␣
↪partial_t

class SingleHarmonicOscillator(ODEEquations):
     def __init__(self,name,terms): #Pass the name of the unknown and the terms T
           super(SingleHarmonicOscillator,self).__init__()
           self.name=name #Store the name of the unknown
           self.terms=terms #and the terms to consider

     def define_fields(self):
           self.define_ode_variable(self.name)

     def define_residuals(self):
           y=var(self.name) #Bind the single variable
           # Calculate the residuals
           residual=partial_t(y,2)+self.terms #Just add the passed terms here
           self.add_residual(residual*testfunction(y))
```

In the definition of the problem, we can now combine two instances of the `SingleHarmonicOscillator` class and by passing the correct names and additional terms, we can recreate the system (3.4). This is achieved by changing the method `define_problem()` of the `TwoCoupledHarmonicOscillatorProblem` as follows:

```
def define_problem(self):
       y1,y2=var(["y1","y2"]) #bind the variables here
       K=self.Kmatrix # shorthand
       #Adding two single oscillators, but passing the coupling
       eqs=SingleHarmonicOscillator("y1",K[0][0]*y1+K[0][1]*y2)
       eqs+=SingleHarmonicOscillator("y2",K[1][0]*y1+K[1][1]*y2)
       eqs+=InitialCondition(y1=1,y2=0) #Setting initial condition
       eqs+=ODEFileOutput()
       self.add_equations(eqs@"coupled_oscillator")
```

This is a common way to couple multiple equations in multi-physics, e.g. a Navier-Stokes equation and a temperature equation for a Rayleigh-Bénard setting, later on.

Method III

The final method is very similar to the second method, but the equations are not combined to a single equation on the domain `"coupled_oscillator"`, but we use two different domains:

```
def define_problem(self):
       #bind the variables. Both are called just "y", but on different domains
       y1=var("y",domain="oscillator1") #note the important keyword argument domain!
       y2=var("y",domain="oscillator2")
       K=self.Kmatrix # shorthand
```

(continues on next page)

(continued from previous page)

```
        #Define the first harmonic oscillator
        eqs1=SingleHarmonicOscillator("y",K[0][0]*y1+K[0][1]*y2)
        eqs1+=InitialCondition(y=1)
        eqs1+=ODEFileOutput()

        #Define the second harmonic oscillator
        eqs2=SingleHarmonicOscillator("y",K[1][0]*y1+K[1][1]*y2)
        eqs2+=InitialCondition(y=0)
        eqs2+=ODEFileOutput()

        #Add both to different domains
        self.add_equations(eqs1@"oscillator1"+eqs2@"oscillator2")
```

Note that the unknowns of both oscillators have the same name, namely `"y"`. However, since we are creating two distinct equations on different domains (`"oscillator1"` and `"oscillator2"`), the same names do not interfere. However, we have to be careful, since the unknown obtained by `var("y")` will have different meanings in both domains, namely the unknown `"y"` defined in the particular domain. When coupling unknowns of different domains, it is therefore required to pass the keyword `domain` in the `var()` function to remove this ambiguity.

Also, the initial conditions are now separated, since initial conditions can only be set on the domain where the corresponding unknown is defined. Since there are also two instances of `ODEFileOutput`, each domain will write its own output file, containing just the single degree of freedom within this domain. The calculated solution is depicted in Fig. 3.3.

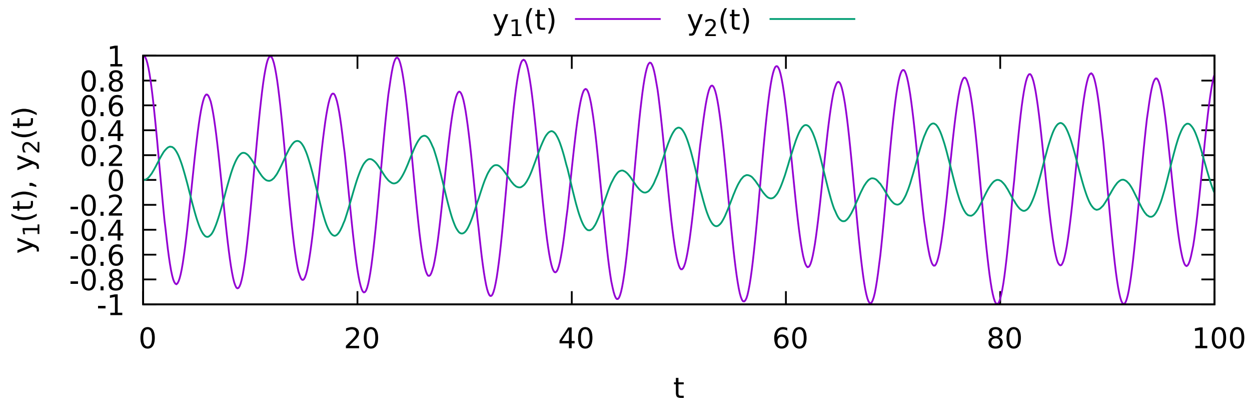

Fig. 3.3: Output for the user-defined harmonic oscillatior equation with damping and driving.

3.5 A nonlinear oscillator

So far, all considered equations were linear ODEs or linear ODE systems. However, unlike e.g. FEniCS, there is no difference in defining linear and nonlinear equations in pyoomph, a feature which is directly inherited from oomph-lib. One famous nonlinear oscillator is the *Van der Pol oscillator*

$$\partial_t^2 y - \mu\left(1-y^2\right)\partial_t y + y = 0\,. \tag{3.5}$$

Here, the parameter μ controls the nonlinearity. It is obvious that a positive value of μ will damp the oscillation whenever $y^2 > 1$ and enhance the amplitude whenever $y^2 < 1$. The straightforward way of implementing this equation would be again to write an equation class:

```
class VanDerPolOscillator(ODEEquations):
    def __init__(self,mu): #Requires the parameter mu
        super(VanDerPolOscillator,self).__init__()
        self.mu=mu #Store the value of mu

    def define_fields(self):
        self.define_ode_variable("y") #same as usual

    def define_residuals(self):
        y=var("y")
        residual=partial_t(y,2)-self.mu*(1-y**2)*partial_t(y)+y
        self.add_residual(residual*testfunction(y))
```

It is not a problem to add the nonlinear damping term to the residuals - a step where FEniCS would complain unless explictly implemented as nonlinear problem.

There is also another way to implement exactly the same equation by using already implemented equations. In the following, we make use of the predefined harmonic oscillator that comes with pyoomph:

```
# import the predefined harmonic oscillator equation
from pyoomph.equations.harmonic_oscillator import HarmonicOscillator

#Inherit from the HarmonicOscillator class
class VanDerPolOscillator(HarmonicOscillator):
    def __init__(self,mu):
        damping=-mu/2*(1-var("y")**2)
        super(VanDerPolOscillator,self).__init__(name="y",damping=damping,
↪omega=1)
```

Here, we use inheritance, i.e. the methods `define_fields()` and `define_residuals()` will be inherited from the super-class `HarmonicOscillator`, which will calculate

$$\partial_t^2 y + 2\delta\partial_t y + \omega^2 y = 0\,.$$

If we replace $\delta = -1/2\mu(1-y)^2$, we get in fact the Van der Pol oscillator (3.5). This fact is used here: when calling the constructor `__init__` of the super-class `HarmonicOscillator`, we pass exactly this expression as damping parameter and thereby recover the Van der Pol oscillator. Obviously, there are always multiple ways in pyoomph to achieve the same goal. A plot of the numerical solution is shown in Fig. 3.4.

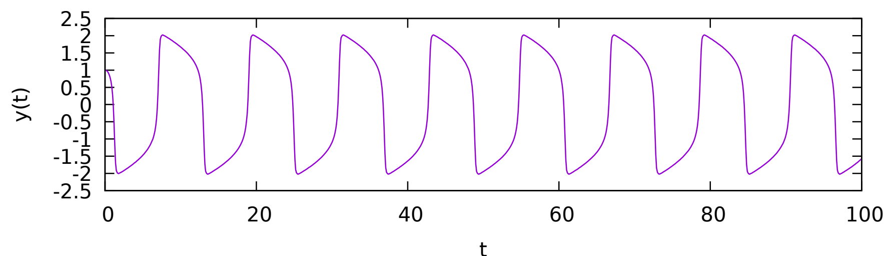

Fig. 3.4: Output for the nonlinear Van der Pol oscillator.

3.6 Time stepping

Until now, we just used `partial_t()` to express time derivatives. However, the particular time stepping method in numerical simulations is of utmost importance for accurate results, which is even more the case if e.g. energy should be conserved. Therefore, this section is dedicated to the time stepping within pyoomph. Second order time derivatives are always calculated with the *Newmark-beta method* of second order (called `"Newmark2"` in the following), which is exactly the same time stepping which is used in oomph-lib. First order derivatives can be either calculated with the *implicit (or backward) Euler method*, which is first order accurate. Since this method coincides with the *backward differentiation formula* (BDF) of first order, it will be called `"BDF1"` in the following. The BDF of second order for first derivatives is also implemented, which is called `"BDF2"`. In fact `"BDF2"` is the default method for first order time derivatives in pyoomph.

3.6.1 Testing different time stepping method

A good way to test whether the chosen time stepping method is appropriate is to compare the numerical results of a simple test problem with the corresponding analytical solution. Furthermore, one can test for conserved quantities, e.g. the total energy of an undamped harmonic oscillator. This will be done for all time stepping methods in the following.

To do so, the harmonic oscillator equation class will be modified to be able to calculate with all time stepping methods implemented in pyoomph. For the `"Newmark2"` method, we use the second order ODE

$$\partial_t^2 y + \omega^2 y = 0$$

whereas for the other methods, i.e. the ones which evaluate only first order time derivatives, `"BDF1"` and `"BDF2"`, this equation is separated into a system of two first order equations

$$\begin{aligned} \partial_t z + \omega^2 y &= 0 \\ \partial_t y - z &= 0 \end{aligned}$$

i.e. where $z = \partial_t y$. The code for the oscillator equation that allows to select the time stepping scheme is the following:

```
class HarmonicOscillator(ODEEquations):
    def __init__(self,*,omega=1,scheme="Newmark2"): #Passing a time stepping scheme
        super(HarmonicOscillator,self).__init__()
        allowed_schemes={"Newmark2","BDF1","BDF2"} #Possible values
        if not (scheme in allowed_schemes): #Test for valid input
            raise ValueError("Unknown time stepping scheme: "+str(scheme)+".␣
↪Allowed: "+str(allowed_schemes))
        self.scheme=scheme
        self.omega=omega

    def define_fields(self):
        self.define_ode_variable("y")
        if self.scheme!="Newmark2":
            self.define_ode_variable("dot_y") #Additional variable for first␣
↪order ODE system

    def define_residuals(self):
        y=var("y")
        if self.scheme=="Newmark2": #One second order ODE
            residual=(partial_t(y,2)+self.omega**2 *y)*testfunction(y)
        else:   #Two first order ODEs
            dot_y=var("dot_y")
            residual=(partial_t(dot_y,scheme=self.scheme)+self.
↪omega**2*y)*testfunction(dot_y)
```

(continues on next page)

(continued from previous page)

```
                residual+=(partial_t(y,scheme=self.scheme)-dot_y)*testfunction(y)
        self.add_residual(residual)
```

A string `scheme` is passed to the constructor to select the time stepping scheme. First, the validity of the passed argument is checked. In the method `define_fields()`, we define the variable y and, if necessary, i.e. if `"BDF1"` or `"BDF2"` are selected, the variable z, which is called `"dot_y"` in the code.

In `define_residuals()`, we add the single second order ODE if `scheme=="Newmark2"` was selected, otherwise the two first order ODEs are added to the residual. Note that it is arbitrary here, which of the two equations is projected on which test function, i.e. we could also project the first equation on `testfunction(y)` and the second one on `testfunction(dot_y)` instead.

Important is the keyword argument `scheme` in the `partial_t()` function: a second order time derivative is calculated by `partial_t(...,2)` (always with the `"Newmark2"` method) and for first order derivatives, we can pass the optional keyword argument `scheme` which can either take the value `"BDF2"` (default), `"BDF1"` or `"Newmark2"`. The latter calculates the first order derivative which is internally used for the second derivative calculation in `"Newmark2"`.

In the problem class, also some modifications are necessary:

```
class HarmonicOscillatorProblem(Problem):
    def __init__(self,scheme="Newmark2"): # Passing scheme here
        super(HarmonicOscillatorProblem,self).__init__()
        self.omega=1
        self.scheme=scheme

    def define_problem(self):
        eqs=HarmonicOscillator(omega=self.omega,scheme=self.scheme)

        t=var("time") # Time variable
        Ampl, phi=1, 0 #Amplitude and phase
        y0=Ampl*cos(self.omega*t+phi) #Initial condition with full time␣
↪depencency
        dot_y0 = -self.omega*Ampl * sin(self.omega * t + phi) #derivative of it
        eqs+=InitialCondition(y=y0) #Set initial condition for y(t) at t=0
        if self.scheme!="Newmark2":
            eqs += InitialCondition(dot_y=dot_y0)  # And if required also␣
↪for dot_y

        #Calculate the total energy
        y=var("y")
        total_energy=1/2*partial_t(y,scheme=self.scheme)**2+1/2*(self.omega*y)**2
        eqs+=ODEObservables(Etot=total_energy) # Add the total energy as␣
↪observable

        eqs+=ODEFileOutput()
        self.add_equations(eqs@"harmonic_oscillator")
```

First of all, we take an argument `scheme` already in the constructor of the problem class. This is stored as member of the problem and passed later on to the equation class, namely in the method `define_problem()`. We make sure that the initial condition is perfectly accurate, i.e. that the derivatives at the beginning are approximated correctly by passing the full time-dependent analytical solution $y(t) = A\cos(\omega t + \phi)$ as initial condition. This condition and its temporal derivatives will be evaluated at $t = 0$ and the corresponding discretized history values are set appropriately. If `"BDF1"` or `"BDF2"` are selected, we also have to explicitly add an initial condition for z, i.e. for `dot_y`, here.

In a next step, we want to monitor the total energy $E = 1/2\,(\partial_t y)^2 + 1/2\,\omega^2 y^2$. This is an observable which depends on the unknowns, but it is not an unknown itself. Therefore, we use the class `ODEObservables`, which allows to add exactly these kind of observables to the output. After running, there will be an additional column in the output file

containing the values of "Etot".

Finally, we let our script successively create a problem for each of the time stepping methods, set an individual output directory with set_output_directory() to prevent overwriting of the previous results and run the simulations:

```
if __name__=="__main__":
    for scheme in {"Newmark2","BDF1","BDF2"}:
        with HarmonicOscillatorProblem(scheme=scheme) as problem:
            problem.set_output_directory("osci_timestepping_"+scheme)
            problem.run(endtime=100,numouts=1000)
```

3.6.2 Trapezoidal rule and the implicit midpoint rule

So far, we have used different time stepping method, but there are obviously more possibilities. Let us focus on systems of first order ODEs only, which can be written in general as

$$\partial_t \vec{U} = \vec{F}(\vec{U}, t)$$

with the vector of unknowns $\vec{U}$ and the rhs $\vec{F}(\vec{U}, t)$. Of course, as usual in numerical simulations, $\vec{U}$ is discretized in time, i.e. we have the time steps $\vec{U}^{(0)}$, $\vec{U}^{(1)}$, ..., $\vec{U}^{(n-1)}$ already computed and we want to compute the step $\vec{U}^{(n)}$. With this notation, an approximation of the time derivative can be written as

$$\partial_t \vec{U} \approx \sum_{i=0}^{k} w_i \vec{U}^{(n-k)} . \tag{3.6}$$

Here, k is the order of the time stepper, e.g. $k = 1$ for "BDF1" and $k = 2$ for "BDF2", and w_i are the time stepper weights. For example, in "BDF1", the weights are $w_0 = -w_1 = 1/\Delta t^{(n)}$, which gives

$$\partial_t \vec{U} \approx \sum_{i=0}^{1} w_i \vec{U}^{(n-k)} = \frac{1}{\Delta t^{(n)}} \left(\vec{U}^{(n)} - \vec{U}^{(n-1)} \right) ,$$

where $\Delta t^{(n)}$ is the time step we want to take to obtain $\vec{U}^{(n)}$.

Until now, we have not addressed the right hand side, i.e. $\vec{F}(\vec{U})$. In fact, if we use any variable bound by var() in the code, it always means the value corresponding to the time step n, i.e. the right hand side up to now was always fully implicit. This means e.g. for "BDF1" time stepping that the discretized equations read

$$\frac{1}{\Delta t^{(n)}} \left(\vec{U}^{(n)} - \vec{U}^{(n-1)} \right) = \vec{F} \left(\vec{U}^{(n)}, t^{(n)} \right) ,$$

where $t^{(n)} = t^{(n-1)} + \Delta t^{(n)}$ is the time we are currently solving for. Indeed, this is the implicit Euler integration.

However, there are time integration methods, where also previous values of $\vec{U}$, i.e. $\vec{U}^{(n-1)}$ etc., are used in the rhs. One particular example is the *trapezoidal rule* (sometimes also called *Crank-Nicolson method*). This one reads

$$\frac{1}{\Delta t^{(n)}} \left(\vec{U}^{(n)} - \vec{U}^{(n-1)} \right) = \frac{1}{2} \left[\vec{F} \left(\vec{U}^{(n)}, t^{(n)} \right) + \vec{F} \left(\vec{U}^{(n-1)}, t^{(n-1)} \right) \right] , \tag{3.7}$$

In pyoomph, the lhs can easily be obtained by writing partial_t(...,scheme="BDF1"), but how to evaluate the right hand side appropriately? As mentioned before, var() statements are always evaluated at the current time step n we are currently solving for, so how to access the values of $\vec{U}$ at previous time steps?

The answer is the function evaluate_in_past(), which takes an expression as argument and evaluate it previous time steps. Without an argument, evaluate_in_past(expr) evaluates expr at the time step $n-1$. A second optional argument gives the offset, e.g. evaluate_in_past(expr,2) evaluates the expression expr

at the time step $n-2$ and, beyond that, e.g. `evaluate_in_past(expr,0.5)` is equal to `0.5*expr+0.5*evaluate_in_past(expr,1)`, i.e. with fractional values as second argument, one can linearly interpolate between two time steps.

This means that the trapezoidal rule (3.7) in residual form in pyoomph can be written as `partial_t(..., scheme="BDF1")-evaluate_in_past(...,0.5)`, where we have cast the equation already in residual form, i.e. all terms are on the lhs and the rhs is zero.

As an example, let us - once again - consider the harmonic oscillator, but this time integrated with the trapezoidal rule (`"TPZ"`). The code for the equation class now reads

```
class HarmonicOscillator(ODEEquations):
     def __init__(self,*,omega=1):
             super(HarmonicOscillator,self).__init__()
             self.omega=omega

     def define_fields(self):
             self.define_ode_variable("y")
             self.define_ode_variable("dot_y")

     def define_residuals(self):
             y=var("y")
             dot_y=var("dot_y")
             residual=(partial_t(dot_y,scheme="BDF1")+evaluate_in_past(self.
↪omega**2*y,0.5))*testfunction(dot_y)
             residual += (partial_t(y,scheme="BDF1")-evaluate_in_past(dot_y,0.5)) *␣
↪testfunction(y)
             self.add_residual(residual)
```

It can be seen that indeed (3.7) is recovered by using `scheme="BDF1"` for the lhs and `evaluate_in_past(...,0.5)` for the rhs.

When using the trapezoidal rule for advancing in time, one also has to be consistent with the definition of the total energy, i.e. the argument which is passed to the `ODEObservables` class (as in the previous section) now has to take the time derivative of y in the kinetic energy via the `"BDF1"` scheme, whereas the value of y in the potential energy has to be evaluated at the average between the time steps n and $n-1$, i.e. via `evaluate_in_past(...,0.5)`:

```
#Calculate the total energy. Important to also stick to the convention: BDF1␣
↪derivative and evaluate_in_past(...,0.5)
y=var("y")
total_energy=1/2*partial_t(y,scheme="BDF1")**2+1/2*evaluate_in_past(self.omega*y,0.
↪5)**2
eqs+=ODEObservables(Etot=total_energy) # Add the total energy as observable
```

Of course, this time stepping can be generalized, which is the so-called *θ-method*:

$$\frac{1}{\Delta t^{(n)}}\left(\vec{U}^{(n)}-\vec{U}^{(n-1)}\right)=(1-\theta)\vec{F}\left(\vec{U}^{(n)},t^{(n)}\right)+\theta\vec{F}\left(\vec{U}^{(n-1)},t^{(n-1)}\right)$$

Obviously, $\theta=0$ coincides with `"BDF1"` and $\theta=1/2$ with the trapezoidal rule. To blend between these methods or any $0\leq\theta<1$, one can just adjust the second argument of `evaluate_in_past(...,theta)`. The case $\theta=1$ corresponds to the *explicit (forward) Euler method*, which is highly unstable and inaccurate.

Another method is the *midpoint rule* (`"MPT"`), which reads

$$\frac{1}{\Delta t^{(n)}}\left(\vec{U}^{(n)}-\vec{U}^{(n-1)}\right)=\vec{F}\left(\frac{\vec{U}^{(n)}+\vec{U}^{(n-1)}}{2},\frac{t^{(n)}+t^{(n-1)}}{2}\right).$$

If $\vec{F}$ is linear, the midpoint rule and the trapezoidal rule are identical, but for nonlinear $\vec{F}$ it is not. The midpoint rule can be used well for *symplectic integration*, i.e. it is expected that the total energy of the harmonic oscillator is quite well

conserved with this method. As explained above, the function `evaluate_in_past(...,0.5)` gives the trapezoidal rule, not the midpoint rule. To obtain the midpoint rule, one has to use the function `evaluate_at_midpoint()` instead.

Warning: If you use any history values, e.g. by `evaluate_in_past()` or `evaluate_at_midpoint()`, the compilation of the generated code is slower, i.e. you might experience some additional waiting time at the beginning. The reason is that one additional routine has to be generated and compiled to solve for steady solutions in that case, where all history values are identical to the current value. More on steady solutions can be found in Section 3.11.2.

3.6.3 Easily selecting time stepping methods for first order time derivatives

The easiest method to switch between all implemented time stepping schemes is the usage of the function `time_scheme()`. This function can be applied to any mathematical expression and will expand the expression to the time scheme selected via the string argument `scheme`. This is done in the following with the anharmonic oscillator

$$\partial_t y + y^3 = 0$$
$$\partial_t z - y = 0\,.$$

So let us implement an equation class `AnharmonicOscillator` for this as follows

```
# Anharmonic oscillator by first order system with different time stepping schemes
class AnharmonicOscillator(ODEEquations):
    def __init__(self, scheme):
        super(AnharmonicOscillator, self).__init__()
        self.scheme = scheme

    def define_fields(self):
        self.define_ode_variable("y")
        self.define_ode_variable("dot_y")

    def define_residuals(self):
        y = var("y")
        dot_y = var("dot_y")
        residual = (partial_t(dot_y) + y ** 3) * testfunction(dot_y)
        residual += (partial_t(y) - dot_y) * testfunction(y)
        # Here, we evaluate the chosen scheme just by applying time_scheme(scheme,...)
        self.add_residual(time_scheme(self.scheme, residual))
```

The constructor takes and argument `scheme`, which is - as usual - stored in a member variable. At the end of `define_residuals()`, just before adding the residuals to the problem, `time_scheme()` with the passed `scheme` is applied on the residual. The problem class and the corresponding code to run all simulations in different output directories reads

```
class AnharmonicOscillatorProblem(Problem):
    def __init__(self, scheme):  # Passing scheme here
        super(AnharmonicOscillatorProblem, self).__init__()
        self.scheme = scheme

    def define_problem(self):
        eqs = AnharmonicOscillator(scheme=self.scheme)

        t = var("time")  # Time variable
        eqs += InitialCondition(y=1,dot_y=0)
```

(continues on next page)

(continued from previous page)

```
        # Calculate the total energy. We also use time_scheme here, e.g. the energy␣
↪is evaluated by the same scheme as the time stepping
        y = var("y")
        total_energy = time_scheme(self.scheme, 1/2 * partial_t(y) ** 2 + 1/4 * y **␣
↪4)
        eqs += ODEObservables(Etot=total_energy)  # Add the total energy as observable

        eqs += ODEFileOutput()
        self.add_equations(eqs @ "anharmonic_oscillator")


if __name__ == "__main__":
    for scheme in {"BDF1", "BDF2", "Newmark2", "MPT", "TPZ", "Simpson", "Boole"}:
        with AnharmonicOscillatorProblem(scheme) as problem:
            problem.set_output_directory("osci_timestepping_scheme_" + scheme)
            problem.run(endtime=100, numouts=200)
```

Before comparing all time stepping schemes here, let us briefly discuss what the function `time_scheme()` actually does. Let us consider any generic expression $G(\partial_t\vec{U},\vec{U},t)$, or in Python some expression `G` which may contain `partial_t(var("U"))`, `var("U")` and `var("time")`. To understand what the application of `time_scheme()` on `G`, i.e. `time_scheme(scheme,G)`, actually does, let us first introduce the shorthand notation

$$G^{(n-k)}=G\left(\partial_t\vec{U},\vec{U}^{(n-k)},t^{(n-k)}\right)$$

i.e. the evaluation of the expression `G` at the kth history time step, with $k=0$ meaning the step we are currently solving for and $k=1$ meaning the values at the last successfully taken step. For a fractional k, the arguments are interpolated linearly, e.g. for $k=\frac{1}{4}$ we get

$$G^{(n-\frac{1}{4})}=G\left(\partial_t\vec{U},\frac{3}{4}\vec{U}^{(n)}+\frac{1}{4}\vec{U}^{(n-1)},\frac{3}{4}t^{(n)}+\frac{1}{4}t^{(n-1)}\right). \tag{3.8}$$

The function `time_scheme()` does two things: depending on the selected `scheme`, it replaces all occurrences of $\partial_t\vec{U}$, i.e. all `partial_t()` calls, by an approximation (cf. (3.6)) suitable for the particular `scheme`. Then, the expression `G` is expanded by a linear combination of the current and previous values of $\vec{U}$ and t, i.e.

$$\text{time_scheme(scheme},G)=\sum_i g_i G^{(n-k_i)}$$

where g_i are the weights of the contributions and k_i are the corresponding history offsets, which might be also fractional. In the latter case, (3.8) is used. The possible time stepping methods with their approximation of $\partial_t\vec{U}$ and the used pairs (g_i,k_i) are listed in Table 3.1.

Table 3.1: Time stepping methods for systems of first order ODEs that can selected via the call of `time_scheme()`. (*) For non-equidistant Δt, this approximation is more complicated. (**) The Newmark2 scheme has additional history fields which are not elaborated here.

scheme	$\partial_t\vec{U}$ replacement	(g_i,k_i)
"BDF1"	$\frac{1}{\Delta t^{(n)}}\left(\vec{U}^{(n)}-\vec{U}^{(n-1)}\right)$	$(1,0)$
"BDF2"	$\frac{1}{\Delta t^{(n)}}\left(\frac{3}{2}\vec{U}^{(n)}-2\vec{U}^{(n-1)}+\frac{1}{2}\vec{U}^{(n-2)}\right)$ (*)	$(1,0)$
"Newmark2"	(**)	$(1,0)$
"TPZ"	cf. "BDF1"	$(\frac{1}{2},0)$, $(\frac{1}{2},1)$
"MPT"	cf. "BDF1"	$(1,\frac{1}{2})$
"Simpson"	cf. "BDF1"	$(\frac{1}{6},0)$, $(\frac{2}{3},\frac{1}{2})$, $(\frac{1}{6},1)$
"Boole"	cf. "BDF1"	$(\frac{7}{90},0)$, $(\frac{16}{45},\frac{1}{4})$, $(\frac{2}{15},\frac{1}{2})$, $(\frac{16}{45},\frac{3}{4})$, $(\frac{7}{90},1)$

Note that the scheme for `partial_t(...,2)`-terms within `G` will not be adjusted by the application of `time_scheme()`. These are always calculated via the Newmark-beta method.

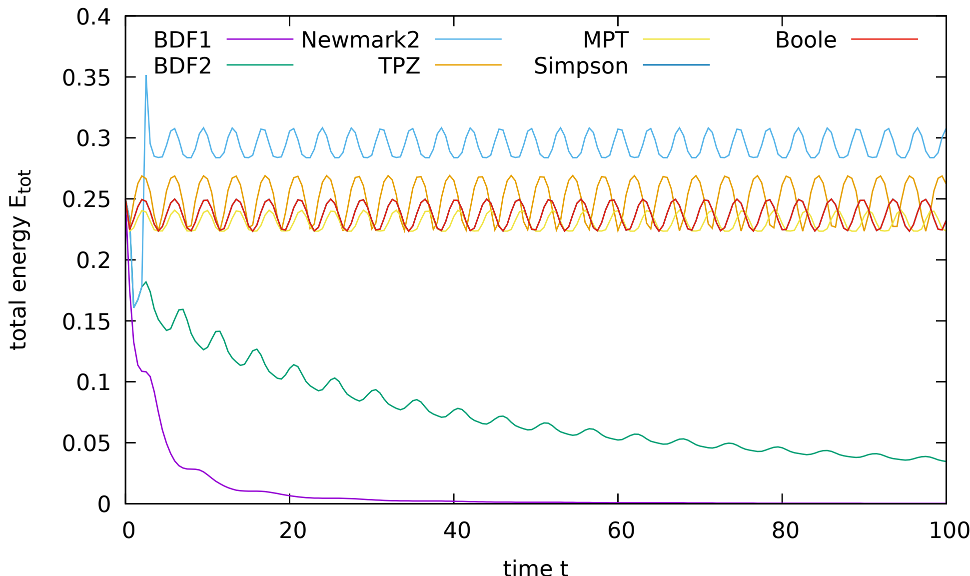

Fig. 3.5: Total energy conservation of an anharmonic oscillator with different time stepping schemes. To visualize the impact a rather larger time step of $\Delta t = 0.5$ was taken.

In Fig. 3.5, the total energy of the simulations of the anharmonic oscillator with the different time steps is depicted. The time stepping is quite coarse, so that the differences are easily visible: First of all `"BDF1"` fails to conserve the energy dramatically and also `"BDF2"` is not suitable for this coarse time step. `"Newmark2"` conserves the energy quite acceptable over long time, however, it has considerable problems the first time steps. The reason is that `"Newmark2"` (and also `"BDF2"`) require two history values to be set. However, the initial condition specified with `InitialCondition` in the code is independent of the time, i.e. there is no variable t (=`var("time")`) occurring in the expressions we set for the initial condition. In this case, the first time step of `"Newmark2"` and `"BDF2"` is evaluated by `"BDF1"`, which does not require any history values except of the initial values at $t = 0$. Alternatively, one can also can supplement the `InitialCondition` object with the keyword argument `degraded_start=False` to fill all history values with the passed values, i.e. with `y=1` and `dot_y=0` here. In that case, or if the initial condition explicitly depends on the time, the first time step is not degraded to `"BDF1"`. One can best circumvent this problem if the analytical solution is known: In that case, one can simply set the initial condition based on the analytical solution, as it was done in Section 3.6.1.

All other methods do not have these problems: they (as also `"BDF1"`) have no requirements of further history values. Furthermore, by explicitly considering the evaluation of $\vec{U}$ at the last successful time step, these methods are quite accurate and energy-conserving, given the large time step. However, in particular the method `"Boole"` is quite expensive since it involves a lot of evaluations at different sub-steps. If one reduces the time step, all methods increase in accuracy, but for e.g. `"BDF1"` it has to be reduced drastically to yield acceptable results. Moreover, if there is substantial dissipation in the system, i.e. conservation is not required, `"BDF2"` can already give quite good results. However, it is always best to check the time stepping scheme for your particular problem with an analytical solution if possible. If an analytical solution is not at hand, one should at least test whether a halved and a doubled time step influences the results. If so, one should take a smaller time step and repeat this procedure.

As a final note, other well-established methods, as e.g. the *Runge-Kutta method* of fourth order, is currently not possible in pyoomph. The reason is that it requires the storage of the results of the sub-stages and multiple solves for each time step. This is not implemented yet in pyoomph.

3.6.4 Time derivatives of third or higher order

As we have seen so far, we can easily calculate first and second order time derivatives by `partial_t()` and - if wrapped in `time_scheme()` - we can select a bunch of time stepping schemes for the first order derivatives. However, if you try to calculate a third or higher time derivatives, e.g. by `partial_t(...,3)` or `partial_t(partial_t(partial_t(...)))`, pyoomph will not be able to handle this and raises an error. This can only be circumvented by reducing it to a system with time derivatives of first or second order as we have done it in the case of the anharmonic oscillator in the previous section, where the second order ODE was split into two first order ODEs. This procedure can be generalized to an arbitrary degree. E.g. the *jerky dynamics* of a system $\dddot{x} = J(x, \dot{x}, \ddot{x})$ can be conveniently written as $\dot{z} = J(x, y, z)$ with $\dot{y} = z$ and $\dot{x} = y$.

3.6.5 Temporal adaptivity

It is not always easy to find a good time step that is sufficiently accurate. Of course, you can make a guess and rerun the simulation with reduced time step to see whether the dynamics is affected by the time step. If not, you might have found a good time step. However, some systems have unpredictable dynamics, as e.g. chaotic systems, which might show very strongly fluctuating dynamics.

pyoomph allows to flexibly adjust the time step during the simulation depending on an estimate of the error the current time step might have. To that end, the next time step is first predicted based on the previous steps, then a step is taken and the result is compared with the prediction. If the difference is too large, pyoomph will discard this step and retries with a smaller time step. At the same time, if the prediction and the computed solution match very well, pyoomph gradually tries to increase the time step.

Here, we will use the chaotic Lorenz system, known for its chaotic *strange attractor*, to illustrate this feature. The Lorenz system consists of three coupled non-linear ODEs of first order

$$\begin{aligned}
\partial_t x &= \sigma(y - x) \\
\partial_t y &= x(\rho - z) - y \\
\partial_t z &= xy - \beta z
\end{aligned}$$

with three parameters (σ, ρ, β). Implementing this in pyoomph is trivial:

```
class LorenzSystem(ODEEquations):
    def __init__(self,*,sigma=10,rho=28,beta=8/3,scheme="BDF2"): # Default␣
↪parameters used by Lorenz
        super(LorenzSystem,self).__init__()
        self.sigma=sigma
        self.rho=rho
        self.beta=beta
        self.scheme=scheme

    def define_fields(self):
        self.define_ode_variable("x","y","z")

    def define_residuals(self):
        x,y,z=var(["x","y","z"])
        residual=(partial_t(x)-self.sigma*(y-x))*testfunction(x)
        residual+=(partial_t(y)-x*(self.rho-z)+y)*testfunction(y)
        residual+=(partial_t(z)-x*y+self.beta*z)*testfunction(z)
        self.add_residual(time_scheme(self.scheme,residual))


class LorenzProblem(Problem):
```

To add the feature of temporal adaptivity to the system, it is sufficient to combine the equations with a `TemporalErrorEstimator`

```
class LorenzProblem(Problem):

    def define_problem(self):
        eqs=LorenzSystem(scheme="BDF2") # Temporal adaptivity works best with␣
↪BDF2
        eqs+=InitialCondition(x=0.01)  # Some non-trivial initial position
        eqs+=TemporalErrorEstimator(x=1,y=1,z=1) # Weight all temporal error␣
↪with unity
        eqs+=ODEFileOutput()
        self.add_equations(eqs@"lorenz_attractor")
```

`TemporalErrorEstimator` takes keyword arguments of the form `variable_name=error_weight`, i.e. we can set to each of the ODE variables a different weighting for the computation of the temporal error between prediction and actual solution. In the code here, all weights have been set to unity, i.e. all variables x, y, z are weighted equally in the determination of the temporal error.

Finally, the `run()` statement takes a keyword argument `temporal_error`, which defined the error we are ready to accept. The smaller the temporal error, the finer the dynamics time steps, but the longer the computation takes.

```
if __name__=="__main__":
    with LorenzProblem() as problem:
        # outstep=True means output every step
        # startstep is the first time step
        # temporal_error controls the maximum difference between prediction and␣
↪actual result
        problem.run(endtime=100,outstep=True,startstep=0.0001,temporal_error=0.
↪005)
```

Note the also `outstep=True` was passed instead of `numouts`. It will just output each step. Of course, also a `startstep` should be set so that the problem has a good guess how to start. If the value of `temporal_error` is set too large, the system might not show the correct dynamics, see Fig. 3.6. The accepted time steps are displayed in Fig. 3.7.

3.7 Enforcing constraints by Lagrange multipliers

As an example, let us consider the pendulum equation. A mass m is located at (x, y), but its movement is constrained by the length L of the pendulum, i.e. by the constraint function

$$g(x, y) = \sqrt{x^2 + y^2} - L = 0 \,.$$

The easiest way to solve this is of course, the conventional way: instead of expressing the system by the two coordinates $x(t)$ and $y(t)$, we introduce the angle ϕ, which is the *generalized coordinate*. This angle is measured with respect to the equilibrium at $(x, y) = (0, -L)$. Thereby, one arrives at the pendulum equation

$$\partial_t^2 \phi + \frac{g}{L} \sin(\phi) = 0 \,.$$

If you have read this tutorial until here, implementing this equation should be a trivial task:

```
from pyoomph import * # Import pyoomph
from pyoomph.expressions import * # Import some additional things to express e.g.␣
↪partial_t

class PendulumEquations(ODEEquations):
    def __init__(self,*,g=1,L=1):
        super(PendulumEquations,self).__init__()
```

(continues on next page)

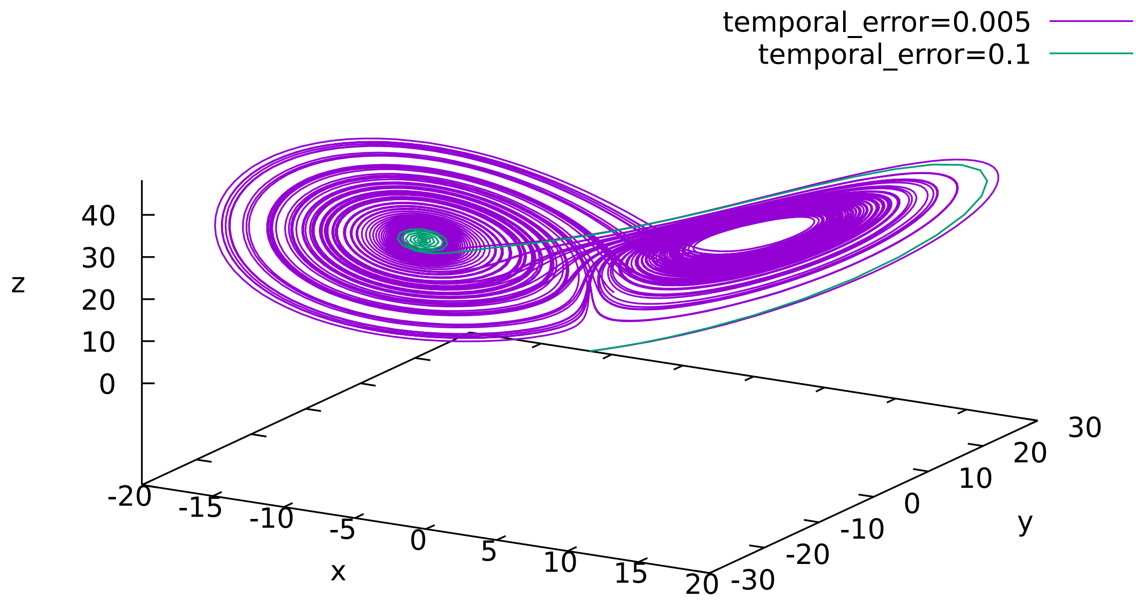

Fig. 3.6: Dynamics of the Lorenz system with adaptive time stepping. Allowing too large errors will give the wrong dynamics.

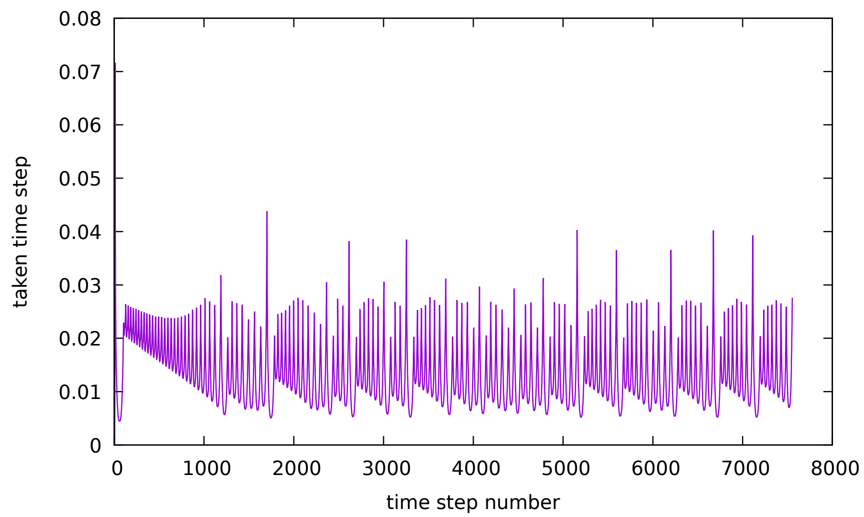

Fig. 3.7: Accepted time steps in the case of `temporal_error=0.005`

(continued from previous page)

```
            self.g=g
            self.L=L

    def define_fields(self):
            self.define_ode_variable("phi") #Angle

    def define_residuals(self):
            phi=var("phi")
            residual=partial_t(phi,2)+self.g/self.L*sin(phi)
            self.add_residual(residual*testfunction(phi))



class PendulumProblem(Problem):

    def __init__(self):
            super(PendulumProblem,self).__init__()
            self.g=1 #Gravity
            self.L=1 #Length

    def define_problem(self):
            eqs=PendulumEquations(g=self.g,L=self.L)
            eqs+=InitialCondition(phi=0.9*pi) #High initial position
            eqs+=ODEFileOutput()
            self.add_equations(eqs@"pendulum")

if __name__=="__main__":
    with PendulumProblem() as problem:
            problem.run(endtime=100,numouts=1000)
```

However, in general, it is not always easy to find the generalized coordinate(s) for which the system automatically fulfills all imposed constraints. In that case, one still can enforce the constraints via Lagrange multipliers. In the given example of the pendulum, let us assume we were unable to find the generalized coordinate ϕ from the constraint $g(x, y)$. We therefore would have to solve the full system, i.e. the equations of motion

$$\begin{aligned} m\partial_t^2 x &= 0 \\ m\partial_t^2 y &= -mg \end{aligned}$$

subject to the *scleronomic* and *holonomic* constraint $g(x, y) = \sqrt{x^2 + y^2} - L = 0$. From the *analytical mechanics*, i.e. *Lagrangian mechanics*, we know how implement the constraint, namely by introducing a Lagrange multiplier λ and minimizing the action functional, which eventually lead to Lagrange's equations of first kind, i.e.

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial x} - \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial \mathcal{L}}{\partial \dot{x}} + \lambda \frac{\partial g}{\partial x} &= 0 \\ \frac{\partial \mathcal{L}}{\partial y} - \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial \mathcal{L}}{\partial \dot{y}} + \lambda \frac{\partial g}{\partial y} &= 0 \end{aligned}$$

with the Lagrangian

$$\mathcal{L} = T - V = \frac{1}{2} m \left(\dot{x}^2 + \dot{y}^2\right) - mgy$$

This gives rise to the equations of motion augmented by the additional force stemming from the tension of the rod of the pendulum

$$\begin{aligned} m\partial_t^2 x &= -\lambda \frac{x}{\sqrt{x^2 + y^2}} \\ m\partial_t^2 y &= -mg - \lambda \frac{y}{\sqrt{x^2 + y^2}} \end{aligned} \tag{3.9}$$

As third equation to determine the three unknowns x, y and λ (in fact, usually it is treated as five unknowns x, y, $\dot{x}$, $\dot{y}$ and λ), one has $g(x,y)=0$.

We solve these equations by splitting the system into the unconstrained motion in an equation class `NewtonsLaw2d`:

```
class NewtonsLaw2d(ODEEquations):
	def __init__(self,*,mass=1,force_vector=vector([0,-1])):
		super(NewtonsLaw2d,self).__init__()
		self.force_vector=force_vector
		self.mass=mass

	# Here, we use BDF2 time stepping, i.e. we split the system into a 4d system of␣
↪first order ODEs
	def define_fields(self):
		self.define_ode_variable("x")
		self.define_ode_variable("y")
		self.define_ode_variable("xdot") #partial_t x
		self.define_ode_variable("ydot") #partial_t y

	def define_residuals(self):
		x,y=var(["x","y"])
		xdot,ydot=var(["xdot","ydot"])
		# Motion equations
		self.add_residual( (self.mass*partial_t(xdot)-self.force_
↪vector[0])*testfunction(x))
		self.add_residual( (self.mass*partial_t(ydot)-self.force_
↪vector[1])*testfunction(y))
		# Definition of xdot and ydot
		self.add_residual( (partial_t(x)-xdot)*testfunction(xdot))
		self.add_residual( (partial_t(y)-ydot)*testfunction(ydot))
```

and the constraint itself, which adds the additional terms stemming from the constraint to the equation of motion and solves for the unknown Lagrange multiplier λ by solving the constraint equation $g(x,y)=0$:

```
#Pendulum constraint: Enforcing sqrt(x**2+y**2)=L via a Lagrange multiplier
class PendulumConstraint(ODEEquations):
	def __init__(self,*,L=1):
		super(PendulumConstraint,self).__init__()
		self.L=L

	def define_fields(self):
		self.define_ode_variable("lambda_pendulum") #Lagrange multiplier

	def define_residuals(self):
		x,y,lambda_pendulum=var(["x","y","lambda_pendulum"])
		currentL=square_root(x**2+y**2) #Current length
		currentL=subexpression(currentL) #Wrap it into a subexpression, since it␣
↪occurs multiple times in the equations
		self.add_residual(lambda_pendulum*x/currentL*testfunction(x))
↪#additional forces
		self.add_residual(lambda_pendulum*y/currentL*testfunction(y))
		self.add_residual((currentL-self.L)*testfunction(lambda_pendulum))
↪#constraint equation to solve for the Lagrange multiplier
```

In the `define_fields()`, we introduce the Lagrange multiplier λ as ODE variable. In the function `define_residuals()`, we add the corresponding forces to the residual form of (3.9). Since the residual form requires to put all terms on one side, note that the sign of the additional terms proportional to λ has changed. By using `testfunction(x)` and `testfunction(y)`, it is ensured that this additional forcing is indeed added to the correct equation of motion. Finally, there one still has the constraint equation $g(x,y)=0$ and the degree of freedom λ. This is accounted

for in the last line where the constraint equation is solved in the residual term for the Lagrange multiplier λ.

Besides `square_root()`, which is just the mathematical square root for pyoomph expressions, there is one additional function occuring in this snippet have not been discussed yet, namely `subexpression()`. Wrapping an expression in a `subexpression()` does not change the results at all, however, we note that the term $\sqrt{x^2+y^2}$ occurs multiple times in the residuals. By wrapping it in a `subexpression()`, pyoomph will internally make sure to evaluate this term only once, store it in a local variable and reuse this local variable for all further occurrences of the wrapped expression. Depending on the complexity of the expression wrapped in a `subexpression()`, this can lead to a huge performance gain.

At a last step, the problem definition reads like this:

```
    def __init__(self):
        super(PendulumProblem,self).__init__()
        self.gvector=vector([0,-1]) #Default gravity direction, g is assumed to␣
↪be 1
        self.L=1 #pendulum length
        self.mass=1

    def define_problem(self):
        eqs=NewtonsLaw2d(force_vector=self.mass*self.gvector,mass=self.mass)
        eqs+=PendulumConstraint(L=self.L)
        phi0=0.9*pi #Initial phi
        x0=self.L*sin(phi0) #Initial position
        y0=-self.L*cos(phi0)
        eqs+=InitialCondition(x=x0,y=y0)  #Set the initial position
        eqs+=ODEFileOutput()  #Output
        eqs+=ODEObservables(phi=atan2(var("x"),-var("y"))) #Calculate phi from x␣
↪and y
        self.add_equations(eqs@"pendulum")

if __name__=="__main__":
    with PendulumProblem() as problem:
        # We need many outputs, i.e. a small dt for the time stepping scheme to␣
↪be nearly energy-conserving
        problem.run(endtime=100,numouts=10000)
```

Here, both equations, `NewtonsLaw2d` and `PendulumConstraint` get combined. While `NewtonsLaw2d` can be solve without the constraint (which would just result in a free fall of the mass), `PendulumConstraint` is only valid when combined with and equation that defines the variables x and y, since these are required for the constraint.

Again, we make use of the `ODEObservables` class to define further output quantities, which depend on the degrees of freedom. Here, we are interested e.g. on the angle ϕ, which can be calculated by $\phi = \arctan(-x/y)$, or in order to accurately treat the special case $y = 0$, `phi=atan2(var("x"),-var("y"))`. By adding this to the system, the output file will contain one additional column for ϕ, which is again automatically calculated at each output step. As before in Section 3.6, one can e.g. also add further arguments to the constructor of `ODEObservables`, e.g. `Ekin=0.5*self.mass*(partial_t(var("x"))**2+partial_t(var("y"))**2)` for the kinetic energy. A plot of $\phi(t)$ and $\lambda(t)$ is shown in Fig. 3.8.

Finally, note that we need quite some time steps to get a stable and conserving scheme (using `"BDF2"`) here. One can play around with the `time_scheme()` function as explained in Section 3.6.3, but one should not apply `time_scheme()` on the residuals added in the `PendulumConstraint`, since the pendulum constraint should be always fulfilled, in particular exactly in the time step n we are solving for, whereas time schemes like `"TPZ"`, `"MPT"` or similar would include values from previous time steps.

Here, we have seen how to enforce a constraint in analytical mechanics. However, the method of Lagrange multipliers is even more powerful since it can be applied nearly anywhere to enforce constraints in a system. Let us first see what happens if we remove (or comment out) the line `self.add_residual(lambda_pendulum*y/currentL*testfunction(y))`. In this case, there will be no additional force added to the y-direction, i.e. the

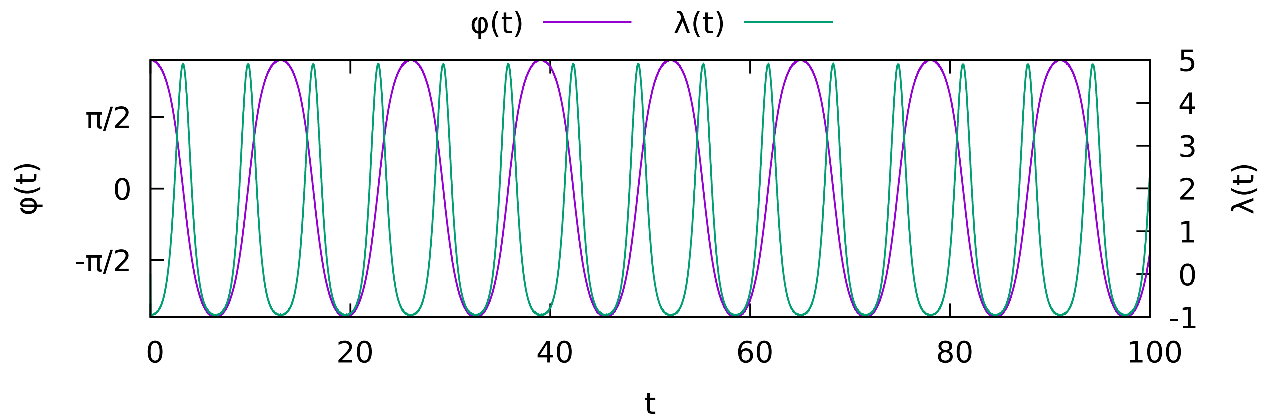

Fig. 3.8: Pendulum equation with the help of a Lagrange multiplier λ enforcing the rod constraint. Note how the oscillation $\phi(t)$ shows an anharmonic curve. Furthermore, it is apparent that λ is negative whenever $|\phi|$ is large. In this case, the rod of the pendulum is pushing the mass outwards. Close to the points where the velocity is highest, i.e. the slope of ϕ is steepest, maxima in λ can be found. This corresponds to the high centripetal force required for the high velocity.

particle will exhibit a free fall in y-direction. However, there is now still a force acting on the x-direction to keep the constraint fulfilled. The particle will hence fall down and experience exactly that force in x-direction, which is necessary to keep it on the circle with radius L. Of course, since it does not obey Lagrange's equations anymore, the energy will not be conserved. Furthermore, the simulation will crash the moment the particle is reaching the south pole at $(x, y) = (0, -L)$. It is still falling, but the constraint cannot be satisfied by any force acting just in x-direction. However, this discussion shows that you can have a constraint in a system that depends on multiple variables, here x and y, but the dynamics is only changed in a particular degree of freedom, which is x here.

This gives rise to the general recipe how to use Lagrange multipliers to enforce arbitrary constraints in a system: Suppose you have a vector of unknowns $\vec{U} = (U_1, \ldots, U_N)$ and M constraints, which are expressed by implicit equations $g_i(\vec{U}) = 0$ for $i = 1, \ldots, M$. We can enforce these constraints by adding the M Lagrange multipliers λ_i $(i = 1, \ldots, M)$ to the system. We then add the constraints $g_i(\vec{U})$ times the test function of λ_i for all $i = 1, \ldots, M$ to residuals of the system. Finally, for each degree of freedom U_j which shall be adjusted to ensure the ith constraint to hold, we add $\lambda_i \partial g_i(\vec{U})/\partial_{U_j}$ to the residual of U_j, i.e. by projecting it to the corresponding test function of U_j.

3.8 Using physical units and nondimensionalization

As every numerical software in its internal core, pyoomph is calculating just with floating-point arithmetic, i.e. with numbers of limited precision and in particular without physical units. Due to the limited precision and numerical bounds, it is beneficial to nondimensionalize the equations in a way that the processor never has to deal with very tiny or huge numbers during the calculation. Therefore, one should avoid to nondimensionalize the equations by just omitting the units, i.e. just substituting 1000 for a mass of $1000\,\mathrm{kg}$, which is even more important for e.g. calculations on extreme scales, e.g. in quantuum mechanics or astrophysics.

Nondimensionalization furthermore comes with the benefit to identify nondimensional numbers that entirely characterize the system dynamics. However, sometimes, in terms of usability of a numerical code for real applications, it is easier to just input the dimensional quantities directly instead of reading through the particular required nondimensional characteristic numbers required for the numerical code. pyoomph offers therefore the possibility input quantities with units and automatically nondimensionalize the equations.

Let us discuss this feature once more on the simple basis of the harmonic oscillator. In a mass spring system, one has the mass m (in $\,\mathrm{kg}$), the spring constant k (in $\,\mathrm{N/m}$), the dimensional time t (in $\,\mathrm{s}$) and the displacement x in (in $\,\mathrm{m}$), i.e. the

dimensional equation reads

$$\partial_t^2 x + \frac{k}{m} x = 0\,.$$

However, in pyoomph, we write these equations as a product with a test function χ corresponding to x, i.e.

$$\left(\partial_t^2 x + \frac{k}{m} x\right)\chi = 0\,. \tag{3.10}$$

To nondimensionalize this equation, one introduces characteristic scales, namely a typical displacement X and a time scale T, defines the nondimensional quantities via $x = X\tilde{x}$ and $t = T\tilde{t}$. Additionally, we allow the test function χ to be associated with a dimensional unit, i.e. $\chi = C\tilde{\chi}$. This gives the equation

$$\left(\frac{X}{T^2}\partial_{\tilde{t}}^2\tilde{x} + X\frac{k}{m}\tilde{x}\right)C\tilde{\chi} = \left(\frac{CX}{T^2}\partial_{\tilde{t}}^2\tilde{x} + CX\frac{k}{m}\tilde{x}\right)\tilde{\chi} = 0\,,$$

The scale C of the test function is arbitrary at the moment, but we can choose it that way that the coefficient in front of the term $\partial_{\tilde{t}}^2\tilde{x}$ becomes units by setting $C = T^2/X$. When using this test function scale, we arrive at

$$\left(\partial_{\tilde{t}}^2\tilde{x} + \frac{kT^2}{m}\tilde{x}\right)\tilde{\chi} = 0\,, \tag{3.11}$$

It is important that the residual form may not have any dimensional quantities left. When the dimensions of k and m and the time scale T are correctly chosen, this will be the case also for the second term in the brackets. Obviously, since we have a homogeneous linear equation, no suitable scale for the displacement x, i.e. the scale factor X, can be found. However, we can identify a reasonable time scale by e.g. $T = \sqrt{\frac{m}{k}}$, which simplifies the equation to having only coefficients of unity. The displacement scale X can e.g. be set by the initial displacement $X = x(t = 0)$.

In pyoomph, we can do the same steps as follows within the definition of our equation class:

```
from pyoomph import *
from pyoomph.expressions import * #Import the basic expressions
from pyoomph.expressions.units import * #Import units like meter and so on


class DimensionalOscillator(ODEEquations):
     def __init__(self,*,m=1,k=1): #Default values can be nondimensional
             super(DimensionalOscillator,self).__init__()
             self.m=m
             self.k=k

     def define_fields(self):
             # bind the scaling of time
             T=scale_factor("temporal")
             X=scale_factor("x") # and of the variable x
             self.define_ode_variable("x",testscale=T**2/X) #set the test function␣
↪scale here

     def define_residuals(self):
             x=var("x") # dimensional x
             # write the equation as before with dimensions
             eq=partial_t(x,2)+self.k/self.m*x
             self.add_residual(eq*testfunction(x)) # testfunction(x) will expand to␣
↪T**2/X * ~chi
```

We have implemented the equation as `partial_t(x,2)+m/k*x`, i.e. it will have the units of `x` divided by the unit of the time squared. To get rid of this scales, we have passed the argument `testscale` to the `define_ode_variable()` method. Thereby, we set the scale factor C of the test function to be T^2/X, where

T and X can be obtained by the function `scale_factor()`, i.e. by `scale_factor("temporal")` and `scale_factor("x")`, respectively. pyoomph will automatically expand all parts that are added to the residuals into scales times the nondimensional equivalent, i.e. internally exactly the same steps are done from (3.10) to (3.11).

We have not yet specified the scales X and T. Both will be done at problem level:

```
class DimensionalOscillatorProblem(Problem):

    def __init__(self):
        super(DimensionalOscillatorProblem,self).__init__()
        self.mass=100*kilogram  #Specifying dimensional parameters
        self.spring_constant=1000*newton/meter
        self.initial_displacement=2*centi*meter #and a dimensional initial␣
↪condition

    def define_problem(self):
        eqs=DimensionalOscillator(m=self.mass,k=self.spring_constant) #Setting␣
↪dimensional parameters
        eqs+=InitialCondition(x=self.initial_displacement) #Setting a␣
↪dimensional displacement
        eqs+=ODEFileOutput()

        #Important step: Introduce a good scaling
        T=square_root(self.mass/self.spring_constant)
        self.set_scaling(temporal=T,x=self.initial_displacement) #and set it to␣
↪the problem

        self.add_equations(eqs@"harmonic_oscillator")


if __name__=="__main__":
    with DimensionalOscillatorProblem() as problem:
        problem.run(endtime=10*second,numouts=1000) #endime is now also in␣
↪seconds!
```

In the constructor, we see how we can simply define dimensional units, i.e. just by multiplying or dividing with units like `meter`, `newton` or whatever. In the `define_problem()` method, we pass this dimensional parameters to the equation and also the initial condition for `x` gets a dimensional value. However, in order to make this work, we have to introduce the typical scalings for the time and the displacement. The time scale can be set via the argument `temporal` in the method `set_scaling()`, whereas the scaling of any other variable, i.e. here `x`, can also be set with this method. As discussed before, a good time scale is $\sqrt{m/k}$ and the initial displacement is used as scale for x. The rest remains the same as before, except that we have to use a dimensional time for the `endtime` keyword argument in `run()`, since our time is now dimensional. Also, after running the simulation, the output file will have units in the header. This is beneficial, since it is not required to redimensionalize the output to compare it e.g. against experimental data. It is important to note that `scale_factor()` calls in the equation class gives unity if there is no scaling set in the problem class. This allows to use the same equation class for dimensional and nondimensional calculations. For the latter case, of course, the variables `mass`, `spring_constant` and `initial_displacement` may not contain units and also the `run()` statement needs a nondimensional numeric value for the `endtime`. Unfortunately, one still has to set good values for the scaling by hand via the `set_scaling()` method. While for the harmonic oscillator, it would be possible to guess a good scaling just from the equation (as we have done with the time scale), it is in general, in particular for highly coupled systems with multiple driving mechanisms, not feasible.

Let us discuss once more what is happening internally in pyoomph here: After `define_problem()` is processed, the C code generation is invoked, which will call among others the functions `define_fields()` and `define_residuals()` of the equation class. Here, the `scale_factor()` quantities will be expanded to the quantities set by the `set_scaling()` method of the problem class. Furthermore, when evaluating the added residuals, all occurrences of `var()` will be replaced by the scale factor times the nondimensional variable, which is accessible

in pyoomph via `nondim()`. Here, `var("x")` will be replaced by `scale_factor("x")*nondim("x")`, what is exactly the step we performed analytically before, i.e. $x = X\tilde{x}$. In a similar manner, `partial_t(...,2)` will be nondimensionalized to a nondimensional variant of `partial_t(...,2)` divided by the square of the temporal scale. When the scaling and the equation is set up correctly, all units, e.g. `meter` etc., will cancel out and one arrives at an nondimensional equation. In fact, due to the chosen scaling in this particular problem, always the same nondimensional equation will be solved, namely $\partial_{\tilde{t}}^2\tilde{x} + x = 0$ with $\tilde{x}(\tilde{t}{=}0) = 1$, no matter what values are set for `mass`, `spring_constant` and `initial_displacement`.

The usage of units is also helpful to check for consistency: Whenever pyoomph cannot cancel the units in the residual, there is either a scaling not set appropriately, the equation is inconsistent or a parameter has a wrong unit. In this case, pyoomph will report an error.

Finally, instead of setting a scale at problem level, it is also possible to set a scale only for a particular domain, i.e. here for the domain `"harmonic_oscillator"`. Instead of passing `x=self.initial_displacement` as argument for the `set_scaling()` method at problem level, one can also augment the equation `eqs` with a `Scaling` object, i.e. via `eqs+=Scaling(x=self.initial_displacement)` instead. This allows to introduce different scalings for variables with the same name on different domains. Of course, it does not make any sense to have individual temporal scales, as the time is a global variable.

3.9 Modifying simulation parameters

Up to now, we always have used the predefined parameter values which where set in the constructor of the problem class. As an example, we have set values for `mass`, `spring_constant` and `initial_displacement` in the problem class of the harmonic oscillator in the previous section. These were then passed to the equation class and used as initial condition, respectively. Each change of the problem parameters would naively require to edit these values in the constructor of this class. The idea of a complete problem class is, however, that only reasonable default parameters are set and all parameters can be modified easily. In fact for complicated problems, it is a good practice to even put the problem class in another Python file than the script actually used to start the simulation.

3.9.1 Inside Python

The direct way of modifying parameters before running the simulation is directly in Python. As described, we can just import the previously discussed Python file where the problem is defined.

```
#Import the script where the problem is defined (here, the file dimensional_
↪oscillator_with_units.py is in the same directory)
from dimensional_oscillator_with_units import *

if __name__=="__main__":
    with DimensionalOscillatorProblem() as problem:

            #Modify the parameters
            problem.initial_displacement=-10*centi*meter
            problem.mass=1*kilogram
            problem.spring_constant=1*newton/meter

            problem.run(endtime=10*second,numouts=1000)
```

Since the default output directory will have the same name as the script name without the `.py` extension, i.e. `dim_osci_run` here, the output directories are normally different by default. You can create multiple run scripts of this type with different parameters and each will write to its own output folder. To change the folder, you still can use `set_output_directory()`.

Note that after the first call of `run()` (or also several other methods like `output()` and so on), one cannot change the output directory or any parameters that easily. The reason is that all these settings are required for the code generation, e.g. for the nondimensionalization and so on. If it is really necessary, one can invoke a recompilation by `redefine_problem()`, but this is not discussed here. Therefore, it is important to set all parameters before the first `run()` statement.

Sometimes, it is not beneficial to have an individual run script for each simulation indented to be performed. If you want to run e.g. hundreds or thousands of simulations to perform a parameter scan, it would require to write a lot of script file. Of course, you could loop through it in python, e.g.

```
from dimensional_oscillator_with_units import *

if __name__=="__main__":
    for k_in_N_per_m in [0.1,0.2,0.5,1,2,5]: #Scan the spring constant
        with DimensionalOscillatorProblem() as problem:

            #Modify the parameters
            problem.initial_displacement=-10*centi*meter
            problem.mass=1*kilogram
            problem.spring_constant=k_in_N_per_m*newton/meter

            problem.set_output_directory("dim_osci_seq_run_k_"+str(k_in_N_
↪per_m))

            problem.run(endtime=10*second,numouts=1000)
```

However, this would run all simulations sequentially. For this simple problem, it is not an issue, but if each simulation would take several hours or days, it is not an option.

3.9.2 Via the command line

You can always override simulation parameters from the command line. For this, you can add the command line arguments *–outdir* so change the output directory and *-P* to modify parameters, e.g.

```
python dimensional_oscillator_with_units.py --outdir dim_osci_run_modified_params -P␣
↪spring_constant='1.5*newton/meter' initial_displacement='0.25*meter'
```

invokes the script `dimensional_oscillator_with_units.py`, output will be written to the directory `dim_osci_run_modified_params` and we set the spring constant $k = 1.5\,\mathrm{N/m}$ and the initial displacement to $x_0 = 0.25\,\mathrm{m}$. Note that the parameters passed via the command line will be set after the parameters set in `dimensional_oscillator_with_units.py` before the `run()` statement. Hence, the command line will override the parameters set in the script.

This allows to use e.g. bash in Linux or batch script in Windows to call multiple simulations in a loop.

3.9.3 Parameter scans via Python

Finally, you can also scan through parameters in parallel in Python. Again, we want to run the script `dimensional_oscillator_with_units.py` with multiple parameter settings. pyoomph comes with a class that can take care of looping over parameters and invoke a call of the simulation script with each particular parameter combination as follows:

```
#Import the parallel parameter scanner
from pyoomph.utils.paramscan import *
from pyoomph.expressions.units import * #Import the units (meter etc)
```

(continues on next page)

(continued from previous page)

```
if __name__=="__main__":

    #Create a parameter scanner, give the script to run and the max number of␣
↪processes to run simultaneously
    scanner=ParallelParameterScan("dimensional_oscillator_with_units.py",max_procs=4)

    for k_in_N_per_m in [0.1,0.2,0.5,1,2,5]: #Scan the spring constant
        sim=scanner.new_sim("dim_osci_seq_run_k_"+str(k_in_N_per_m))

        #Modify the parameters
        sim.initial_displacement=-10*centi*meter
        sim.mass=1*kilogram
        sim.spring_constant=k_in_N_per_m*newton/meter


    #Run all (and rerun also already finished sims)
    scanner.run_all(skip_done=False)
```

First, a `ParallelParameterScan` object is created, passing the script that should be started in parallel and an optional argument of how many processes to be used. If the latter is omitted, it will default to the number of CPUs on the system. The script to run must be either in the same directory or the corresponding full or relative directory has to be passed. Then, for each parameter combination, we add a new simulation to the list using `new_sim()` with the output directory of this particular script. Note that these output directories will be sub-directories of the directory of the parameter scan, which defaults to `dim_osci_para_run` here, but can be set via the keyword argument `output_dir` in the constructor of the `ParallelParameterScan` object. We can then modify any parameter we like to adjust for the simulation by setting them to the object obtained by `new_sim()`.

Finally, if all simulations to be started are added, we can invoke the `run_all()` method to start them in parallel. The moment one simulation finishes, the next one is started, but the maximum number of parallel processes is never exceeded. The optional argument `skip_done` let you control whether you want to rerun already completed simulations or not. If you make e.g. relevant changes in the script `dimensional_oscillator_with_units.py`, it obviously has to be set to `False`, since the finished results may differ. Otherwise, you can easily continue a previously interrupted scanning process by setting `skip_done=True`, which will skip all simulations that were already completed previously.

3.10 Adding custom math functions in Python

pyoomph comes with the basic math functions implemented, i.e. `exp()`, `sin()`, `cos()`, etc. are implemented in the package `pyoomph.expressions`. However, sometimes, one need additional custom mathematical functions. This can be done with the class `CustomMathExpression`. In the following, we will discuss one non-dimensional case and one case with physical dimensions.

3.10.1 A harmonic oscillator driven by a trapezoidal forcing

Let us again consider a nondimensional harmonic oscillator, but now with a custom driving function, which resembles a trapezoidal pulse. This custom pulse function can be implemented in pyoomph via the `CustomMathExpression` class from the `pyoomph.expressions.cb` as follows:

```
from pyoomph.expressions.cb import * # Import custom math callback expressions

class TrapezoidalFunction(CustomMathExpression):
    def __init__(self,*,wait_time=5, flank_time=0.25,high_time=10,amplitude=1):
```

(continues on next page)

(continued from previous page)

```
            super(TrapezoidalFunction, self).__init__()
            self.wait_time=wait_time # Pass some parameters to the function already␣
↪in the constructor
            self.flank_time=flank_time
            self.high_time=high_time
            self.amplitude=amplitude

    # This method will be called whenever the function must be evaluated
    def eval(self,arg_array):
            t=arg_array[0] # Bind local t to the first passed argument
            if t<self.wait_time:
                    return 0.0 # Before the pulse
            elif t<self.wait_time+self.flank_time:
                    return self.amplitude*(t-self.wait_time)/self.flank_time # flank␣
↪up
            elif t<self.wait_time+self.flank_time+self.high_time:
                    return self.amplitude # at the plateau
            elif t<self.wait_time+2*self.flank_time+self.high_time:
                    return self.amplitude*(1-(t-self.wait_time-self.flank_time-self.
↪high_time)/self.flank_time) # flank down
            else:
                    return 0.0 # after the plateau
```

In the constructor, we take the parameters that are fixed during the simulation, namely the quantities describing the shape of the pulse. Then, the method `eval()` has to be implemented, which takes a list object `arg_array` as parameters. In this list object, the current numerical values of the passed parameters (here, it will be the time t later on) are stored. Based on this value, we return the current value of the pulse.

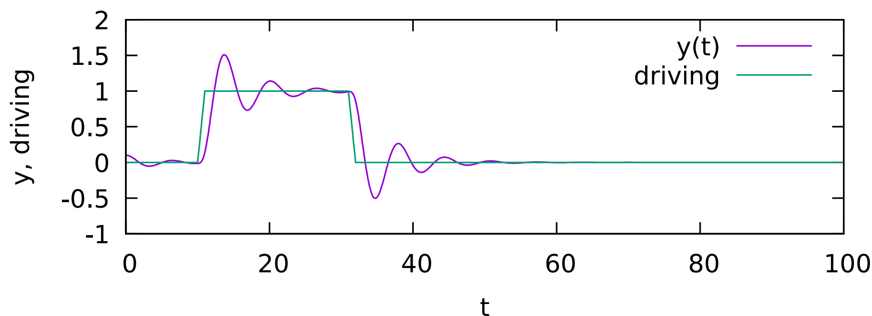

Fig. 3.9: Using a `CustomMathExpression`, we can implement custom functions, here the trapezoidal driving.

> **Warning:** All custom functions must be deterministic on their input arguments, i.e. evaluating the function `eval()` multiple times for the same input must yield the same result. This rules out any contribution of random numbers or any dependence on the degrees of freedom or parameters which are not passed via the argument list `arg_array`.

The problem class looks like this, where we reuse the predefined `HarmonicOscillator` equation class:

```
class TrapezoidallyDrivenOscillatorProblem(Problem):

    def define_problem(self):
            t = var("time")
            # Create a trapezoidal driving
            driving = TrapezoidalFunction(wait_time=10, high_time=20, flank_time=1)
```

(continues on next page)

(continued from previous page)

```
            # Evaluate at t (which is the current time) and wrap it in a␣
↪subexpression (optional, but recommended)
            driving = subexpression(driving(t))
            # pass the driving function evaluated at t here
            eqs=HarmonicOscillator(omega=1,damping=0.2,driving=driving,name="y")
            eqs+=InitialCondition(y=0.1)
            eqs+=ODEFileOutput()
            eqs+=ODEObservables(driving=driving) # Also output the driving to the␣
↪file
            self.add_equations(eqs@"harmonic_oscillator")

if __name__=="__main__":
     with TrapezoidallyDrivenOscillatorProblem() as problem:
            problem.run(endtime=100,numouts=1000)
```

The result is depicted in Fig. 3.9.

3.10.2 Playing tennis in pyoomph - custom expression with dimensions

The next problem will be a bit more comprehensive and it will involve several techniques we have learned so far, namely using physical dimensions, custom math expressions and temporal adaptivity. We want to simulate a simple tennis match. As in all physics exams, there is no air friction acting on the ball, the problem is one-dimensional and the tennis rackets behave as a Hookian spring, but only if the position of the ball exceeds the position of the racket. Mathematically, we solve

$$m\ddot{x} = F_1(x) + F_2(x)$$

where F_1 and F_2 are the forces of the rackets, which take action whenever the ball reaches the positions X_1 and X_2 of the rackets:

$$F_1(x) = \begin{cases} -k_1(x - X_1) & \text{if} \quad x > X_1 \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad F_2(x) = \begin{cases} -k_2(x - X_2) & \text{if} \quad x < X_2 \\ 0 & \text{otherwise} \end{cases}$$

For these forces, we write our `CustomMathExpression` in python, however, now taking the positions in meters and resulting in a dimensional force with the unit N:

```
from pyoomph import *
from pyoomph.expressions import *
from pyoomph.expressions.units import *

# Force of a tennis racket as function of the ball position and the racket position
# This will create a function Force(ball_position,racket_position)
# Both positions must have the unit of meter
# And the result is a force measured in Newton
class TennisRacket(CustomMathExpression):
    def __init__(self,*,direction=1,spring_constant=1000*newton/meter):
        super(TennisRacket, self).__init__()
        self.direction=direction # sign of the force
        # Store a non-dimensional value of the spring constant
        self.k_in_N_per_m=float(spring_constant/(newton/meter))

    # Input arguments are converted to numerical values by treating the input as␣
↪meter (for both arguments, ball and racket position)
    def get_argument_unit(self,index:int):
        return meter # return meter, no matter whether index==0 or index==1
```

(continues on next page)

(continued from previous page)

```
    # The result is obtained by multiplying the result of 'eval' by newton
    def get_result_unit(self):
        return newton

    # This routine is now entirely nondimensional
    def eval(self,arg_array):
        # get the input values (numerical float values!)
        ball_pos_in_m=arg_array[0] # measured int meter
        racket_pos_in_m=arg_array[1] # measured int meter
        # calculate the distance (also in meter)
        distance_in_m=self.direction*(ball_pos_in_m-racket_pos_in_m)
        if distance_in_m>=0: # in front of the racket
            return 0.0 # numerical float value of the force [in newton]
        else:
            # Force of the racket on the ball
            force_in_newton=-self.direction*self.k_in_N_per_m*distance_in_m
            return force_in_newton # result is treated in newton
```

To allow for arguments with physical units as input arguments, we must implement the function `get_argument_unit()`, which returns the unit used for non-dimensionalization of the input arguments. The argument `index` here can be used to identify which argument to the custom math function is meant. Furthermore, the result of our calculation, i.e. of the force F_1 and F_2 shall be measured in N, which we tell pyoomph by implementing the method `get_result_unit()`. Everything else, i.e. the entire calculation of the force will now be done by `float` numbers in Python. The values stored in `arg_array` are `float` numbers giving the position of the ball and the position of the racket in meters. The `return` value of eval must also be a `float`, which is eventually multiplied by the result of `get_result_unit()`, i.e. by N, to give a dimensional force.

The class for the equation for the ball position x is again trivial:

```
class NewtonsLaw1d(ODEEquations):
    def __init__(self,mass,force):
        super(NewtonsLaw1d, self).__init__()
        self.mass=mass
        self.force=force

    def define_fields(self):
        # bind the scale factors (defined on problem level)
        T=scale_factor("temporal")
        X=scale_factor("spatial")
        # we set the scales as well as the test function scales here locally in the␣
↪equation class
        self.define_ode_variable("x",scale=X,testscale=T**2/X) # same test scale as␣
↪in the dimensional harmonic oscillator before
        self.define_ode_variable("xdot",scale=X/T,testscale=T/X) # velocity scales as␣
↪X/T, test scale T/X will cancel this out

    def define_residuals(self):
        x,xdot=var(["x","xdot"])
        residual=(partial_t(xdot)-self.force/self.mass)*testfunction(x)
        residual+=(partial_t(x)-xdot)*testfunction(xdot)
        self.add_residual(residual)
```

Different as before, we define not only the test function scales in the `define_fields()` method, but also the scales itself by adding the argument `scale` to the `define_ode_variable()`. The position x will be nondimensionalized by a scale `"spatial"`, which will be set later. The velocity $\dot{x}$, i.e. `xdot`, is nondimensionalized by `"spatial"/"temporal"`, which is a reasonable choice for a velocity. The test scales are again chosen that way that all

physical units cancel out in the added residual. Both missing scales "spatial" and "temporal" are set at problem level using set_scaling().

```
class TennisProblem(Problem):
    def __init__(self):
        super(TennisProblem, self).__init__()
        self.top_racket_force=TennisRacket(direction=-1,spring_constant=5*newton/
↪meter)
        self.bottom_racket_force=TennisRacket(direction=1,spring_constant=20*newton/
↪meter)
        self.top_position=10*meter
        self.bottom_position=-10*meter
        self.ball_mass=60*gram
        self.ball_pos0=0*meter
        self.ball_velo0=10*meter/second

    def define_problem(self):
        self.set_scaling(spatial=1*meter,temporal=1*second)
        ball_pos=var("x")
        racket_force=self.top_racket_force(ball_pos,self.top_position)
        racket_force+=self.bottom_racket_force(ball_pos,self.bottom_position)
        racket_force=subexpression(racket_force)

        ball_eq=NewtonsLaw1d(mass=self.ball_mass,force=racket_force)
        ball_eq+=InitialCondition(x=self.ball_pos0,xdot=self.ball_velo0)
        ball_eq += ODEObservables(top_position_in_m=self.top_position/meter,bottom_
↪position_in_m=self.bottom_position/meter)
        ball_eq+=ODEFileOutput()
        ball_eq+=TemporalErrorEstimator(x=1,xdot=1)

        self.add_equations(ball_eq@"ball")

if __name__=="__main__":
    with TennisProblem() as problem:
        problem.run(endtime=20*second,outstep=True,temporal_error=0.0025,startstep=0.
↪01*second)
```

In the constructor, we initialize two rackets, one at the top and one at the bottom with different values for the spring constant k. Thereby, we provide the two functions that calculate the force of the racket as function of the ball position and the racket position. In the define_problem() method, first the scalings "spatial" and "temporal" are set at problem level. The former is then used for the scale of "x" and the quotient of both is used for "xdot" at equation level. Then, the equation of motion, NewtonsLaw1d is constructed, with a force consisting the sum of both racket forces. Again, it is encapsulated in a subexpression() as recommended for additional computation speed. The remainder is trivial, but note that again a TemporalErrorEstimator is added to monitor the error made by the adaptive time stepping.

Finally, we run the problem, again with an adjustable accepted temporal_error value for dynamic time stepping. The effect of the dynamic time stepping is visible in Fig. 3.10, where clearly the steps are smaller whenever the ball is subject to the force of a racket.

As a last note, we also can let the players move easily, since the positions of the rackets, stored in the problem class in the members top_position and bottom_position, is passed to the custom expressions TennisRacket as second argument. Hence, a slight modification before running allows for motion of the players, see Fig. 3.11 This feature, i.e. changing the problem by modifying the expressions, is later on helpful, when e.g. modifying the mass density or dynamic viscosity of a liquid mixture.

```
if __name__=="__main__":
```

(continues on next page)

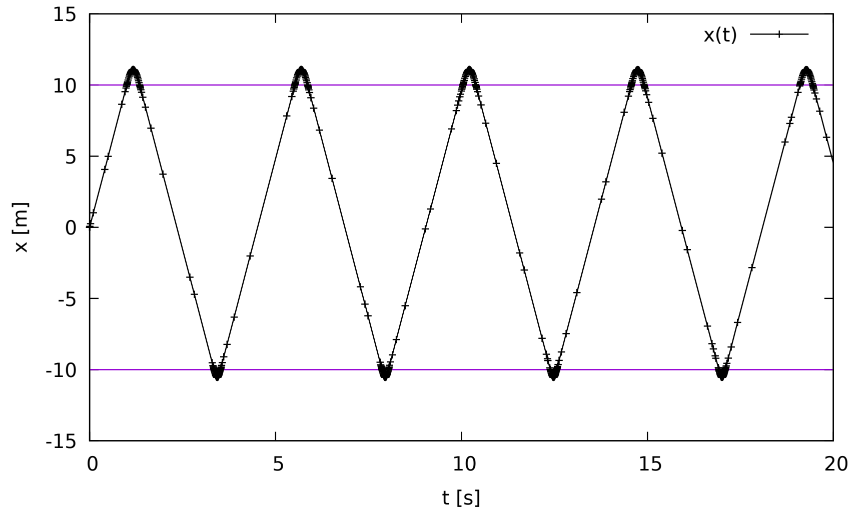

Fig. 3.10: Tennis with pyoomph. One clearly sees how the dynamic time stepping kicks in when the forces of the rackets are acting.

(continued from previous page)

```
    with TennisProblem() as problem:
        t=var("time")
        # Let the players move up and down
        problem.bottom_position=-10*meter+4*meter*sin(2*pi * 0.25*hertz*t)
        problem.top_position = 10 * meter + 6 * meter * cos(2*pi * 0.1*hertz*t)
        problem.run(endtime=20*second,outstep=True,temporal_error=0.0025,startstep=0.
↪01*second)
```

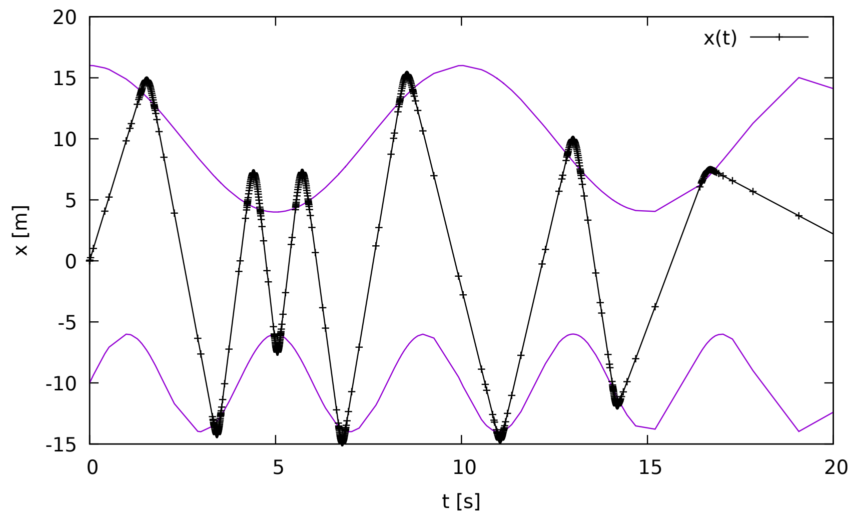

Fig. 3.11: The tennis players are moving. Note that the motion of the players is not well resolved, since the dynamic time stepping is not affected by their positions at all when the ball is in mid air. Also, the velocity of the ball is reduced when a player is moving backward during striking and enhanced when moving forward.

Warning: The usage of `CustomMathExpression` should be considered as last resort, since the call of a python function is quite expensive compared to the execution on the generated `C` code. In particular, here one could have used `heaviside((var("x")-self.top_position)/meter)` to kick in the force of the racket instead. The division by `meter` is required in the argument of `heaviside()`, since functions like `sin()`, `cos()`, but also `heaviside()` require an argument without any dimension.

3.10.3 Custom functions with multiple return values

The aforementioned `CustomMathExpression` can only return a single scalar value at each time. Sometimes, however, you want to obtain more than one return value. Of course, one could implement one `CustomMathExpression` for each of these values, but a lot of computations might be the same for each of these values. In this case, it is beneficial to use another class to obtain multiple values at once, namely the `CustomMultiReturnExpression`. This is not explained in detail here, but we refer to the code in `pyoomph.expressions.tensor_funcs`.

3.11 Stability analysis

So far, we have only considered time-dependent ODEs. However, it is also of interest how ODEs, in particular nonlinear, behave by performing a stability analysis. This means that we are not interested in the total time behavior, but just whether a stationary solution of a system of ODEs will be stable under a small perturbation around this stationary solution or not. In particular, the stability of a stationary solution usually depends on a parameter, i.e. a stationary solution may switch from unstable to stable or vice versa at a critical value of this parameter, which is called a *bifurcation*. Different from the previous section, we do not want to start a bunch of simulations to scan this parameter, but ideally, one want to find the critical value of the parameter automatically.

Obviously, in order to investigate the stability of stationary solutions of ODE systems, we first need to address two things, namely how to find a stationary solution and how to introduce parameters that can be changed at run time.

3.11.1 Global parameters

Let us discuss the procedure on the basis of the *transcritical normal form*, which reads

$$\partial_t x = rx - x^2\,, \tag{3.12}$$

where r is the parameter. As usual, the stationary solutions can be obtained by setting $\partial_t x = 0$, which yields the two stationary solutions $x_0 = 0$ and $x_0 = r$. The stability of these solutions can also be assessed easily by hand. One just introduces a perturbed solution $x(t) = x_0 + \delta x(t)$ and linearized the dynamics of δx:

$$\begin{aligned} \partial_t\left(\delta x\right) &= + r\delta x \qquad \text{for} \quad x_0 = 0 \\ \partial_t\left(\delta x\right) &= - r\delta x \qquad \text{for} \quad x_0 = r \end{aligned}$$

Obviously, when r passes 0, the dynamics of the system changes. For $r < 0$, the stationary solution $x_0 = 0$ is stable and $x_0 = r$ is unstable, whereas it is vice versa for $r > 0$.

In pyoomph, we again formulate the very simple equation class for this, passing some value (or expression) as parameter `r`:

```
class TranscriticalNormalForm(ODEEquations):
    def __init__(self,r):
        super(TranscriticalNormalForm, self).__init__()
        self.r=r
```

(continues on next page)

(continued from previous page)

```
    def define_fields(self):
        self.define_ode_variable("x")

    def define_residuals(self):
        x,x_test=var_and_test("x") #Shortcut to get var("x") and testfunction("x")
        self.add_residual((partial_t(x)-self.r*x+x**2)*x_test)
```

In the problem class, we use `define_global_parameter()` to define a global parameter object with an initial value. This parameter is defined on the problem level. When combined in an expression, automatically its symbolical (i.e. adjustable value) will be used. However, unlike `var()`, parameters are not considered to be unknowns of the system. To adjust the value of a parameter, we can access the current value by the property `value`. If we apply `float()` on an expression containing a parameter, the current `value` will be substituted.

```
class TranscriticalProblem(Problem):
    def __init__(self):
        super(TranscriticalProblem, self).__init__()
        # Bifuraction parameter with default value
        self.r=self.define_global_parameter(r=1)
        self.x0=1

    def define_problem(self):
        eq=TranscriticalNormalForm(r=self.r) #Pass the symbolic 'r' here
        eq+=InitialCondition(x=self.x0)
        eq+=ODEFileOutput()
        self.add_equations(eq@"transcritical")
```

Let us first run the problem as before, namely by using the `run()` method of the `Problem` class:

```
if __name__=="__main__":
    with TranscriticalProblem() as problem:

        problem.r.value=1 # You can set the value here
        problem.x0=0.001 # Start slightly above the unstable solution x=0
        problem.run(endtime=20,numouts=100) # And let it evolve towards the stable␣
↪branch x=r

        problem.r.value=-1 # Change the parameter value, x=1 is now not a stationary␣
↪solution anymore
        problem.run(endtime=40, numouts=100)  # And let it evolve towards the now␣
↪stable branch x=0
```

We first set $r = 1$, but start closely to the unstable branch $x = 0$. In the output, we will see an initial exponential growth of $x(t)$, followed by a convergence into the stable branch at $x = r = 1$. However, after $t = 20$, we change the parameter value. This feature, i.e. modifying a system parameter is only possible with parameters as constructed here. Changes in other properties, e.g. the initial condition x_0 or the harmonic oscillator frequency ω in the first examples within this book have no effect after the problem has been initialized, since the code is generated based on the values at the beginning. However, global parameters are still variables within the generated code and they are re-evaluated every single time step. Therefore, it is possible to change the value of r in between.

After setting $r = -1$, the dynamics of the system entirely changes. In particular, the branch $x_0 = 1$, which the solution was just trying to attain, has now moved to $x_0 = -1 = r$. Furthermore, as discussed before, $x = 0$ has now become the stable branch. Therefore, the output for $t > 20$, generated by the second `run()` statement, will show how x approaches 0 instead, see Fig. 3.12.

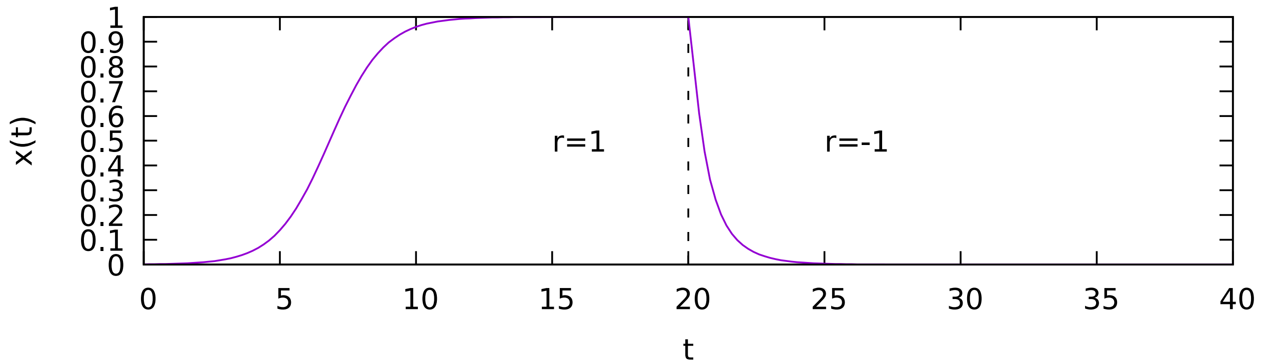

Fig. 3.12: Temporal integration of the transcritical normal form with a switch of the parameter r at $t = 20$. Depending on r, the stationary solution at $x = 0$ is either unstable or stable.

3.11.2 Stationary solutions

By simple time integration, we have seen how the transcritical normal form avoids unstable stationary solutions and tries to approach stable stationary solutions. While time integration can help to approach a stable stationary solution, one is unable to approach unstable stationary solutions that way. Also, it takes a lot of time steps to approach the stable ones. Instead, one can just jump to a stationary solution, both stable and unstable ones, by solving the stationary problem:

$$0 = rx - x^2 \,,$$

Of course, one could implement this equation by hand again, but it is better to reuse the full equation with $\partial_t x$-term, i.e. (3.12), and just tell pyoomph to search for a stationary solution instead of integrating the temporal evolution. To that end, we first import the transcritical bifurcation equation and problem from the previous example `bifurcation_transient_transcritical.py`. Afterwards, we use the `solve()` method of the problem, which exactly looks for a stationary solution.:

```
from bifurcation_transient_transcritical import * #Import the equation and problem␣
↪from the previous script

if __name__=="__main__":
    with TranscriticalProblem() as problem:
        problem.quiet() # Do not pollute our output with all the messages

        problem.r.value=1 # Parameter value
        problem.x0=0.001 # Start slightly above the unstable solution x=0

        ode = problem.get_ode("transcritical")  # Get access the ODE (note: it will␣
↪initialize the problem!)

        xvalue = ode.get_value("x") #Get the current value of x
        print(f"We are starting at x={xvalue}")

        problem.solve(timestep=None) # Solve without a timestep means stationary solve
        xvalue = ode.get_value("x") # Get the current value of x
        print(f"Currently, we are at the stationary solution x={xvalue} with r=
↪{problem.r.value}")

        ode.set_value(x=0.8) # Set the current value of x
        problem.solve() # we can omit timestep=None, since it is default
        xvalue = ode.get_value("x")
        print(F"Currently, we are at the stationary solution x={xvalue} with r=
↪{problem.r.value}")
```

First, we tell the problem class to be `quiet()`, so that there is no output from the code generation and solution. Thereby, the `print` statements can be better found in the output. We can get access to the ODE by using the `get_ode()` method. Note that this will initialize the problem, i.e. generate the code and so on, since before that the problem does not know about the existence of an ODE domain called `"transcritical"`. This domain is added in `define_problem()`. With the reference to the ODE, we can use `set_value()` and `get_value()` to access the current value of x. Initially, we are close to the unstable solution $x = 0$, so the `solve()` statement will likely run into this solution, which is indeed the case. In the second solve, we are close to the stable solution at $x = r = 1$, so we run into this with the second invocation of `solve()`.

Obviously, one can easily find a stationary solution by the `solve()` function. However, this only works if one is close to that solution. If one tries to e.g. start at $x = 0.5$, i.e. by `ode.set_value(x=0.5)` followed by a `problem.solve()` statement, it will not find any solution. The reason lies in the internal *Newton method*, which requires the *Jacobian*, which is singular at $x = 0.5$. In fact, we are exacly between both solutions, so the system cannot decide which way to go. In that case, one can solve a few steps by temporal integration, either via the `run()` command or via `solve(timestep=...)` to move a bit away from the singularity. After that, the normal `solve()` command will converge to a stationary solution again. Note that the numerical stationary solutions are not exactly 0 and 1. Deviations come from the fact that (i) we have a nonlinear problem, i.e. the solution procedure is stopped after reaching sufficient accuracy and (ii) numerical calculations on computers always have a limited accuracy due to the finite accuracy of floating point numbers. The first issue can be improved by passing `newton_solver_tolerance=1e-20` or even smaller numbers to the `solve()` command. However, in larger systems the latter, i.e. the finite accuracy of float arithmetic, might hamper to reach tiny accuracy thresholds.

3.11.3 Calculating eigenvalues and eigenfunctions

In the previous example, we have found stationary solutions, but we could not obtain the stability of these solutions. To explicitly calculate the eigenvalues, we can use the method `solve_eigenproblem()` of the problem class, which requires the number of eigenvalues we want to calculate. Here, there is just one degree of freedom, namely x, so we should only aim for a single eigenvalue. The method will return a list of eigenvalues (complex numbers) and a list of the corresponding eigenvectors. Of course, using `solve_eigenproblem()` is only meaningful if the problem is currently on a stationary solution.

The rest is more or less the same as the previous code, i.e. first importing the problem and equation from the previous example `bifurcation_transient_transcritical.py` and then use `solve_eigenproblem()` after finding the stationary solutions via the `solve()` call.

```
from bifurcation_transient_transcritical import * #Import the equation and problem␣
↪from the previous script

if __name__=="__main__":
    with TranscriticalProblem() as problem:
        problem.quiet() # Do not pollute our output with all the messages

        problem.r.value=1 # Parameter value

        ode = problem.get_ode("transcritical")  # Get access the ODE (note: it will␣
↪initialize the problem!)

        for startpoint in [0.001,0.8]: #Take different start points
            ode.set_value(x=startpoint) #Start there
            problem.solve() # Solve for the stationary solution
            xvalue = ode.get_value("x") # Get the current value of x
            print(f"Starting at {startpoint} gives the stationary solution x={xvalue}␣
↪with r={problem.r.value}")
            eigen_vals,eigen_vects=problem.solve_eigenproblem(1)
            print("Eigenvalues are "+str(eigen_vals))
```

(continues on next page)

(continued from previous page)

```
            print("Thus, this solution is "+("unstable" if eigen_vals[0].real>0 else
↪"stable"))
```

3.11.4 Stability of second order ODEs or in presence of constraints

Let us revert back to the pendulum equation of Section 3.7. There, the dynamics has been implemented in two ways, namely (i) by solving the second order ODE

$$\partial_t^2\phi + \frac{g}{L}\sin(\phi) = 0\,,$$

or alternatively the first order ODE system with a Lagrange multiplier λ enforcing the pendulum constraint

$$\begin{aligned} m\partial_t w &= -\lambda\frac{x}{\sqrt{x^2+y^2}} \\ m\partial_t z &= -mg-\lambda\frac{y}{\sqrt{x^2+y^2}} \\ \partial_t x &= w \\ \partial_t y &= z \\ 0 &= \sqrt{x^2+y^2}-L\,. \end{aligned}$$

Both variants are somewhat special regarding the calculation of the stability, i.e. of the eigenvalues. The first one is a second order ODE, for which pyoomph cannot calculate the eigenvalues directly. Hence, we have to cast it to a first order system as usual:

```
from pyoomph import * # Import pyoomph
from pyoomph.expressions import * # Import some additional things to express e.g.␣
↪partial_t

class PendulumEquations(ODEEquations):
    # Lets assume g=L=1
    def define_fields(self):
        self.define_ode_variable("phi","psi") # angle and angular velocity

    def define_residuals(self):
        phi,phi_test=var_and_test("phi")
        psi, psi_test = var_and_test("psi")
        self.add_residual((partial_t(psi)+sin(phi))*phi_test) # psi'=phi''=-sin(phi)
        self.add_residual((partial_t(phi)-psi) * psi_test) # psi=dot(phi)


class PendulumProblem(Problem):
    def define_problem(self):
        eqs=PendulumEquations() #No output or initial condition required
        self.add_equations(eqs@"pendulum")

    # A function to investigate the stability of solutions
    def investigate_stability_close_to(self,phi_guess):
        ode=self.get_ode("pendulum")
        ode.set_value(phi=phi_guess,psi=0) # set the guess
        self.solve() # stationary solve
        eigvals,eigvects=self.solve_eigenproblem(2) # get eigenvectors
        phi_in_terms_of_pi=(ode.get_value("phi")/pi).evalf() # output phi as multiple␣
↪as pi
```

(continues on next page)

(continued from previous page)

```
        print(f"Eigenvalues at phi={phi_in_terms_of_pi}*pi are {eigvals[0]} and
↪{eigvals[1]}")

if __name__=="__main__":
    with PendulumProblem() as problem:
        problem.quiet()
        problem.investigate_stability_close_to(phi_guess=0.01) # pendulum is almost␣
↪hanging straight down
        problem.investigate_stability_close_to(phi_guess=0.9*pi)  # pendulum is␣
↪almost at the apex
```

Indeed, the result is expected: A marginally stable solution at $\phi = 0$ with imaginary eigenvalues $\pm i$, i.e. an undamped oscillatory motion and an unstable solution at $\phi = \pi$. Since there are two degrees of freedom, we solve for 2 eigenvalues. Furthermore, not the method `evalf` applied on ϕ/π to express π in terms of π. Without `evalf`, π is treated as a constant. Also, we have added some custom functionality to our problem by providing the method `investigate_stability_close_to`.

Now, let's turn towards the system with the Lagrange multiplier. Naively, one might anticipate that we can find 5 eigenvalues, since we have 5 degrees of freedom. Let's see:

```
#We can reuse the old class, since it is already a system of first order ODEs
from pendulum_lagrange_multiplier import *


if __name__=="__main__":
     with PendulumProblem() as problem:
             problem.quiet()
             ode = problem.get_ode("pendulum")

             problem.solve(timestep=0.001) #Make one little time step to find a good␣
↪guess for lambda
             problem.solve()
             x,y,lambd,xdot,ydot=ode.get_value(["x","y","lambda_pendulum","xdot","ydot
↪"])
             print(f"Solution: x={x}, y={y}, lambda={lambd}, xdot={xdot}, ydot={ydot}
↪")
             eig_vals,eig_vects=problem.solve_eigenproblem(5)
             print(eig_vals)


             # Just flip the solution upside down
             ode.set_value(x=0.0,y=-y,lambda_pendulum=-lambd) # We also need to flip␣
↪the rod tension lambda
             problem.solve() # We can still solve
             x, y, lambd, xdot, ydot = ode.get_value(["x", "y", "lambda_pendulum",
↪"xdot", "ydot"])
             print(f"Solution: x={x}, y={y}, lambda={lambd}, xdot={xdot}, ydot={ydot}
↪")
             eig_vals, eig_vects = problem.solve_eigenproblem(5)
             print(eig_vals)
```

First, we use the fact that our system in Section 3.7 was already converted in first order form. Hence, we can just import the previous classes from `pendulum_lagrange_multiplier.py` and reuse them directly. However, the initial guess of λ is not good and the problem might not converge with a simple `solve()` command. This can be done by performing a single short time step. Within a time step, the ∂_t-terms ensure that the degrees x and y do not change to much within the small time interval. Thereby, λ can relax better to a value which is close to the stationary solution near the top. Then, we solve for the stationary solution, get up to 5 eigenvalues and finally we flip the pendulum upside down

and repeat this. Note that λ has been also flipped, since it represents the normal force stemming from the rod of the pendulum. If it is at the top, we need a pushing force, if it is at the bottom, we need the same pulling force.

While everything runs smoothly, instead of 5 eigenvalues, only 2 are returned, which are exactly the same as the ones in the simple system for the angle ϕ before. To see why this is the case and what is actually going on in pyoomph, we can go through the calculation of the eigenvalues analytically. First, the system is written as

$$\mathbf{M}\partial_t\vec{U} = \vec{F}\left(\vec{U}\right)$$

where $\vec{U} = (w, z, x, y, \lambda)$ is the vector of unknowns and

$$\vec{F}(\vec{U}) = \begin{pmatrix} -\lambda\frac{x}{\sqrt{x^2+y^2}} \\ -mg - \lambda\frac{y}{\sqrt{x^2+y^2}} \\ w \\ z \\ \sqrt{x^2+y^2} - L \end{pmatrix}$$

is the right hand side. The matrix $\mathbf{M}$ is called *mass matrix* and it reads

$$\mathbf{M} = \begin{pmatrix} m & 0 & 0 & 0 & 0 \\ 0 & m & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

This matrix reflects the fact that in particular for the equation of the constraint no time derivatives is present. The stationary solutions $\vec{U}_0$ for the parameters $m = L = g = 1$ read

$$\vec{U}_0^+ = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ -1 \end{pmatrix} \quad \text{and} \quad \vec{U}_0^- = \begin{pmatrix} 0 \\ 0 \\ 0 \\ -1 \\ 1 \end{pmatrix}$$

To determine the eigenvalues, the right hand side $\vec{F}(\vec{U})$ is linearized around $\vec{U}_0^\pm$, which gives the *Jacobian*

$$\mathbf{J}^\pm = \left.\frac{\partial\vec{F}}{\partial\vec{U}}\right|_{\vec{U}_0^\pm} = \begin{pmatrix} 0 & 0 & \pm1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mp1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \pm1 & 0 \end{pmatrix}$$

As usual in linear stability analysis, we consider the a perturbed solution $\vec{U}^\pm = \vec{U}_0^\pm + \delta\vec{U}^\pm$. The small perturbation $\delta\vec{U}^\pm$ hence evolves according to

$$\mathbf{M}\partial_t\delta\vec{U}^\pm = \mathbf{J}^\pm\delta\vec{U}^\pm$$

And finally, given the linearity of this system, the ansatz $\vec{U}^\pm = \vec{v}\exp(\mu t)$ is chosen, where μ and $\vec{v}$ is the eigenvalue-eigenvector pair we want to solve for. This gives rise to the *generalized eigenvalue problem*

$$\mu\mathbf{M}\vec{v} = \mathbf{J}\vec{v}\,,$$

where the indices $\pm$ where omitted for brevity. This equation is exactly the one which is solved within pyoomph, whenever `solve_eigenproblem()` is called. The mass matrix $\mathbf{M}$ is calculated based on the occurrences of the `partial_t()`-expressions in the added residuals and the Jacobian $\mathbf{J}$ is analytically calculated and numerically evaluated at the stationary solution.

Warning: When solving for the generalized eigenproblem, most solvers internally used require that the mass matrix **M** is positive (semi-)definite or at least that the diagonal entries are positive. This means that you always should implement your residuals that way, that the sign of the `partial_t()` terms is positive. Therefore, one should prefer

```
add_residual((partial_t(u)-rhs))*testfunction(u))
```

over

```
add_residual((rhs-partial_t(u)))*testfunction(u))
```

Analogous to the conventional eigenvalue problem, which arises for the case of $\mathbf{M}=\mathbf{1}$, we demand that

$$\det|\mathbf{J}-\mu\mathbf{M}|=0$$

In fact, if we calculate this determinant, the characteristic polynomial is indeed only of second order, not of order 5. We get

$$\det\left|\mathbf{J}^{\pm}-\mu\mathbf{M}\right|=-\mu^2\pm 1$$

which have exactly the pairs of solutions for the eigenvalues μ we got from the numerical calculation via pyoomph, namely $\mu=\pm 1$ for $\vec{U}_0^+$, i.e. the pendulum at the apex and $\mu=\pm i$ for $\vec{U}_0^-$, i.e. the pendulum at the equilibrium position at the bottom.

Obviously, in generalized eigenvalues problems, it is not true that the sum of the *algebraic multiplicity* of each eigenvalue yields the dimension of the matrix (here 5). Instead, we can get less, so that the dynamics in the system with the constraint enforced by the Lagrange multiplier exactly resembles the dynamics of the simple system expressed in the generalized coordinate ϕ.

In pyoomph, both situations are well treated by the method `solve_eigenproblem()`.

3.11.5 Following a stationary solution along a parameter by pseudo-arc length continuation

In the light of knowing now how to consider parameters, solve for stationary solutions and calculate the stability, it is time to move on to another feature, which is helpful. The question is simple: *How does the stationary solution change if we gradually increase or decrease a parameter?* To answer this question, *pseudo-arc length continuation* is a powerful tool.

The simplest problem where arc length continuation becomes handy is the normal form of another kind of bifurcation, namely the *fold bifurcation*, which reads

$$\partial_t x=r-x^2\,. \tag{3.13}$$

Obviously, there is no stationary solution for $r<0$, whereas we have a stable stationary solution at $x_0=\sqrt{r}$ and an unstable stationary solution at $x_0=-\sqrt{r}$ for $r>0$. Let us implement this equation and gradually reduce from a positive r and follow the stationary solution:

```
from pyoomph import *
from pyoomph.expressions import *

class FoldNormalForm(ODEEquations):
    def __init__(self,r):
        super(FoldNormalForm, self).__init__()
        self.r=r
```

(continues on next page)

(continued from previous page)

```
    def define_fields(self):
        self.define_ode_variable("x")

    def define_residuals(self):
        x,x_test=var_and_test("x")
        self.add_residual((partial_t(x)-self.r+x**2)*x_test)


class FoldProblem(Problem):
    def __init__(self):
        super(FoldProblem, self).__init__()
        # bifurcation parameter
        self.r=self.define_global_parameter(r=1)
        self.x0=1

    def define_problem(self):
        eq=FoldNormalForm(r=self.r) #Pass the paramter 'r' here
        eq+=InitialCondition(x=self.x0)
        # Instead of having a file with time as first column, we want to have the␣
↪parameter value r
        eq+=ODEFileOutput(first_column=self.r)

        self.add_equations(eq@"fold")

if __name__=="__main__":
    with FoldProblem() as problem:
        while True:
            problem.solve()
            problem.output_at_increased_time()
            problem.r.value-=0.02
```

Of course, this code will crash the moment we decrease $r < 0$, since there is no stationary solution and hence `solve()` will fail. Since we know that the fold bifurcation takes place at $r = 0$, one could try to increase the parameter again after reaching $r = 0$, but then - which branch of the solution will be taken? The stable one at $x = \sqrt{r}$ or the unstable one at $x = -\sqrt{r}$.

Obviously, in this situation, the parameter r is not the best quantity to vary in order to obtain the entire solution curve as function of the parameter. One could prescribe a value of x_0 and determine r so that this value of $x_0(r)$ is indeed a stationary solution. However, there is an even better way: The fundamental idea is to neither solve for solutions x_0 for varying r, nor solve for the parameter r for which a varying x_0 is the stationary solution, but instead vary both at the same time. However, what to choose as independent variable in that case? A good choice is obviously the arc length s along the curve $(x_0(r), r)$. This means that r becomes part of the unknowns and we are solving the system

$$
\begin{aligned}
F(x,r) = r - x^2 = 0 \\
(x - x^*)\partial_x F + (r - r^*)\partial_r F = \Delta s
\end{aligned}
\tag{3.14}
$$

where (x^*, r^*) is a starting point for which $F(x^*, r^*) = 0$ holds. The second equation now prescribes a step Δs in tangent direction, i.e. along the tangent $(\partial_x F, \partial_r F)$ along the curve $F(x, r) = 0$. In pyoomph, this can be done with the method `arclength_continuation()`:

```
from bifurcation_fold_param_change import *
if __name__=="__main__":
    with FoldProblem() as problem:

        # Find start solution
        problem.r.value=1
```

(continues on next page)

(continued from previous page)

```
        problem.get_ode("fold").set_value(x=1)
        problem.solve()
        problem.output()

        # Initialize ds (the first step is in direction of the parameter r, i.e. we␣
↪decrease r first)
        ds=-0.02

        # Scan as long as x>-1
        while problem.get_ode("fold").get_value("x",as_float=True)>-1:
            # adjust r, solve for x along the tangent and return a good new ds
            ds=problem.arclength_continuation(problem.r,ds,max_ds=0.025)
            problem.output()
```

First, reuse the classes from the beginning of this page, i.e. from `bifurcation_fold_param_change.py`. We get a start solution (x^*, r^*). Then, we use `arclength_continuation()` to solve the system above for a step Δs. In the first step the sign of Δs (`ds` in python here) gives the initial direction for the parameter, i.e. `ds<0` means we want to initially decrease the parameter r. `arclength_continuation()` will now adjust r and solve for the stationary solution at the same time. Furthermore, it returns a new guess for `ds` for the next step. In order to capture the curve well, we can add the optional argument `max_ds` to limit the maximum step along the tangential direction. After the first call of `arclength_continuation()`, the direction of the tangent and further parameters required for the next steps are implicitly stored in the `Problem` class. These are reset whenever one performs an `arclength_continuation()` with respect to another parameter or explicitly calls `reset_arc_length_parameters()`.

We can combine the `arclength_continuation()` with the calculation of eigenvalues to get a bifurcation diagram of the fold bifurcation:

```
from bifurcation_fold_param_change import *

if __name__=="__main__":
    with FoldProblem() as problem:

        # Find start solution
        problem.r.value=1
        problem.get_ode("fold").set_value(x=1)
        problem.solve()
        problem.output()

        # Initialize ds (the first step is in direction of the parameter r, i.e. we␣
↪decrease r first)
        ds=-0.02

        # File to write the parameter r, the value of x and the eigenvalue
        fold_with_eigen_file=open(os.path.join(problem.get_output_directory(),"fold_
↪with_eigen.txt"),"w")
        # Function to write the current state into the file
        def write_to_eigen_file():
            eigenvals,eigenvects=problem.solve_eigenproblem(1)
            line=[problem.r.value,problem.get_ode("fold").get_value("x",as_
↪float=True),eigenvals[0].real,eigenvals[0].imag]
            fold_with_eigen_file.write("\t".join(map(str,line))+"\n")
            fold_with_eigen_file.flush()

        write_to_eigen_file() # write the first state

        while problem.get_ode("fold").get_value("x",as_float=True)>-1:
```

(continues on next page)

(continued from previous page)

```
        ds=problem.arclength_continuation(problem.r,ds,max_ds=0.005)
        problem.output()
        write_to_eigen_file() # write the updated state
```

Here, we use the classes from `bifurcation_fold_param_change.py` to continue along the branch and solve the for the eigenvalues. We write the eigenvalue with the largest real part (by default stored at index 0) to a file. A plot of this can be found in Fig. 3.13, where we marked the stability of the branches.

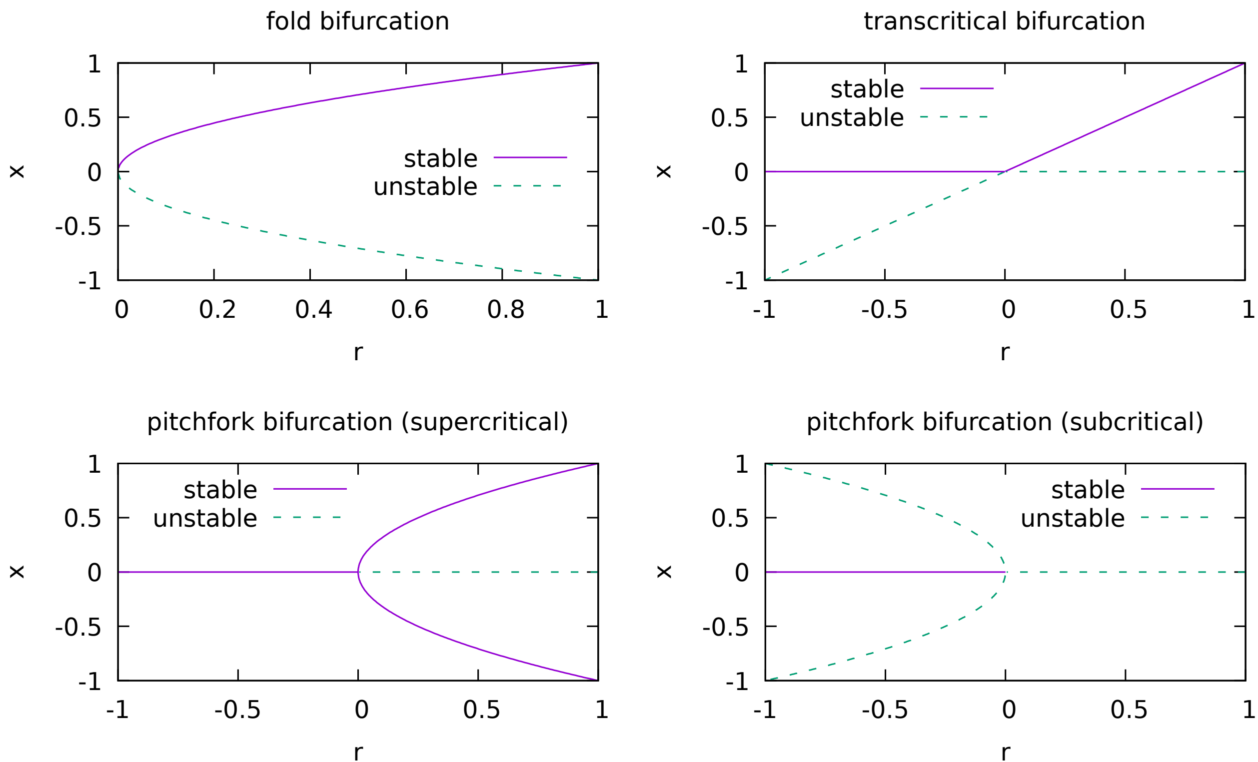

Fig. 3.13: Bifurcation diagrams of the fold, transcritical and pitchfork (super- and subcritical) bifurcations determined by arc length continuation and eigenvalues.

With the same approach, we can find the bifurcation diagram of the transcritical normal form (3.12) and the pitchfork normal form

$$\partial_t x = rx \pm x^3 \,, \tag{3.15}$$

where a minus sign gives the supercritical and a plus sign the subcritical version of the pitchfork bifurcation. All bifurcations are plotted in Fig. 3.13. The corresponding codes are not discussed here, since they all are similar to the code of the fold bifurcation above. However, they are shipped along with this tutorial and can be found in the files `bifurcation_transcritital_arclength_eigen.py` and `bifurcation_pitchfork_arclength_eigen.py`. For the pitchfork bifurcation, we had to set `set_arc_length_parameter(scale_arc_length=False)` to stay on the non-trivial branch. If `scale_arc_length` is `True`, the taken arc length step Δs will be scaled with the magnitudes of $\partial_x F$ and $\partial_r F$ in (3.14). However, at the pitchfork bifurcation, both will approach zero when approaching the bifurcation at $r = 0$.

The approach presented here cannot only be applied on the simple normal forms but on arbitrary systems, also on discretized spatio-temporal partial differential equations, which will be done later in Section 5.6. This provides an easy way to investigate the stability of complicated highly nonlinear systems.

3.11.6 Jumping on bifurcations

If one is just interested at the critical parameter value where the bifurcation happens, e.g. $r = 0$ in all previously discussed cases, there is another possibility to spot the bifurcation than performing arclength continuation and the calculation of eigenvalues. In complicated systems, one neither knows the critical stationary solution $\vec{x}_\text{c}$ (here a vector comprising e.g. multiple coupled ODEs) nor the critical parameter r_c. However, one knows that the following has to hold at e.g. a fold bifurcation:

$$\begin{aligned} \vec{F}(\vec{x}_\text{c}, r_\text{c}) = 0 &\quad \text{i.e. } \vec{x}_\text{c} \text{ is a stationary solution at the parameter } r_\text{c} \\ \mathbf{J}(\vec{x}_\text{c}, r_\text{c})\vec{v} = 0 &\quad \text{i.e. there is an eigenvector } \vec{v} \text{ with eigenvalue zero} \end{aligned}$$

For a system with N degrees of freedom ($N = \dim(\vec{x})$), we have now $2N + 1$ unknowns, namely the N unknowns of $\vec{x}_\text{c}$, the N components of the unknown eigenvector $\vec{v}$ and the critical parameter r_c, for which an eigenvalue becomes zero. There is obviously one equations missing, which related with the magnitude of $\vec{v}$, since without any further equation all parameters r with corresponding stationary solutions will solve it for the trivial choice $\vec{v} = 0$. One could demand that e.g. $\|\vec{v}\| = 1$, but since a reasonable initial guess $\vec{v}_\text{g}$ for the eigenvector $\vec{v}$ is usually required anyhow, we just demand $\vec{v} \cdot \vec{v}_\text{g} = 1$. Thereby, this $(2N + 1)^\text{th}$ equation is linear.

For the fold normal form (3.13), one gets the system

$$r_\text{c} - x_\text{c}^2 = 0, \qquad -2x_\text{c}v = 0, \qquad vv_\text{g} = 1$$

which yields for all $v_\text{g} \neq 0$ the known solution $r_\text{c} = 0$ and $x_\text{c} = 0$. For more complicated systems, however, this has to be solved numerically, which can be done in pyoomph by the `activate_bifurcation_tracking()` method (again after importing the problem and equation classes from `bifurcation_fold_param_change.py`):

```
from bifurcation_fold_param_change import *

if __name__=="__main__":
    with FoldProblem() as problem:

        # Find any start solution, which must be close to the bifurcation
        problem.r.value=1
        problem.get_ode("fold").set_value(x=1)
        problem.solve()

        # Find a guess for the normalization constraint
        problem.solve_eigenproblem(0)
        vguess=problem.get_last_eigenvectors()[0] # use the eigenvector as guess

        # Activate fold bifurcation tracking in parameter r and solve the augmented␣
↪system
        problem.activate_bifurcation_tracking(problem.r,"fold",eigenvector=vguess)
        problem.solve()

        print(f"Critical at r_c={problem.r.value} and x_c={problem.get_ode('fold').
↪get_value('x')}")
```

The same works also for a pitchfork bifurcation, but these are subject to a symmetry, which is broken by the bifurcation. If we apply the fold tracking method to the pitchfork normal form (3.15), we would get the augmented system

$$r_\text{c}x_\text{c} \pm x_\text{c}^3 = 0, \qquad \left(r \pm 3x_\text{c}^2\right) v = 0, \qquad vv_\text{g} = 1\,.$$

While it indeed gives the correct solution $x_\text{c} = 0$ at $r_\text{c} = 0$, the root has a multiplicity of three. Numerically, this hampers the Newton solver to converge.

To reduce the multiplicity and account for the symmetry in the pitchfork bifurcation, a symmetry vector $\vec{\psi}$ is considered and following the $2N+2$-system is solved:

$$\begin{aligned} \vec{F}(\vec{x}_\mathrm{c},r_\mathrm{c})+\sigma\vec{\psi}&=0 && \text{i.e. } \vec{x}_\mathrm{c} \text{ is a stationary solution at the parameter } r_\mathrm{c} \\ \mathbf{J}(\vec{x}_\mathrm{c},r_\mathrm{c})\vec{v}&=0 && \text{i.e. there is an eigenvector } \vec{v} \text{ with eigenvalue zero} \\ \vec{v}_\mathrm{g}\cdot\vec{v}&=1 && \text{i.e. the eigenvector } \vec{v} \text{ in non-trivial (for reasonables guesses) } \vec{v}_\mathrm{g} \\ \vec{\psi}\cdot\vec{x}_\mathrm{c}&=0 && \text{i.e. the solution is symmetric with respect to the symmetry vector } \vec{\psi} \end{aligned}$$

Note that the slack variable σ enforcing the symmetry will be zero at the bifurcation. For the pitchfork normal form with the simple scalar equivalents $\psi = v_\mathrm{g} = 1$, we obtain indeed $x_\mathrm{c} = 0$, $r_\mathrm{c} = 0$, $v = 1$, $\sigma = 0$ and the root of the system has a multiplicity of one, i.e. the Jacobian of the augmented system at the solution is invertible. Thereby, the Newton solver converges well. To find a pitchfork bifurcation, one just has to pass `"pitchfork"` instead of `"fold"` as second argument for the `activate_bifurcation_tracking()` method. Again, one can pass an `eigenvector` argument which will be used as eigenvector normalization vector $\vec{v}_\mathrm{g}$ and as symmetry vector ψ. Please refer to the supplied code `bifurcation_pitchfork_tracking.py` (dependent on `bifurcation_pitchfork_arclength_eigen.py`) for an example.

Warning: When solving with `activate_bifurcation_tracking()`, you must deactivate it via `deactivate_bifurcation_tracking()` after the solve to solve the normal system again (i.e. the system without the augmentation). Also the calculation of eigenvalues and -vectors does not work as usual while bifurcation tracking is active. See next section how to obtain the critical eigenvector.

Warning: Tracking pitchfork bifurcations in spatio-temporal problems (cf. Section 5) requires that the mesh is conforming with the symmetry vector, i.e. the mesh should be also symmetric along the symmetry that it broken by the bifurcation.

Note: Pyoomph can improve the convergence of bifurcation tracking by calling the method `setup_for_stability_analysis()`. This will generate symbolical C code for the required Hessian terms in the augmented systems for bifurcation tracking. Without calling this method, finite differences are used to calculate the Hessian terms, which can be less accurate and slower. For more details, we refer to Section 10.2.1 and our article [16].

3.11.7 Bifurcation tracking

Finally, when a bifurcation point is a function of multiple parameters, one can jump on the bifurcation by adjusting one parameter and find the position of the bifurcation as function of another parameter by arc length continuation. To that end, let us consider the Lorenz system (3.9) from Section 3.6.5. It is known that the Lorenz system has one trivial fix point for $\rho < 1$, after what a supercritical pitchfork bifurcation can be found at $\rho = 1$, which eventually loses the stability in a subcritical *Hopf bifurcation*, which location and presence is dependent on the parameters σ and β. Beyond the Hopf bifurcation, chaotic behavior can be expected.

First of all, let us find the pitchfork bifurcation at $\rho = 1$ with pyoomph. To that end, the equations are loaded from the code from Section 3.6.5 and a problem class is define where all three parameters can be adjusted at run time, i.e. are bound by `define_global_parameter()` and can hence be changed after the C code generation and can be used for arc length continuation and bifurcation tracking. We reuse the code of the file `adaptive_lorenz_attractor.py` from Section 3.6.5 here:

```
from adaptive_lorenz_attractor import * # import the Lorenz problem

# Simple Lorenz system where all parameters can be changed at runtime
class LorenzBifurcationProblem(Problem):
    def __init__(self):
        super(LorenzBifurcationProblem, self).__init__()
        self.rho=self.define_global_parameter(rho=0)
        self.sigma = self.define_global_parameter(sigma=0)
        self.beta = self.define_global_parameter(beta=0)

    def define_problem(self):
        ode=LorenzSystem(sigma=self.sigma,beta=self.beta,rho=self.rho)
        self.add_equations(ode@"lorenz")
```

First, find the pitchfork bifurcation as described in the previous section, including the analytically derived Hessian terms using `setup_for_stability_analysis()`. We must be somewhat close to pitchfork bifurcation, so we start at $\rho = 0.5$, solve for the stationary solution and find the critical eigenvector as guess for $\vec{v}_\mathrm{g}$:

```
if __name__=="__main__":
    with LorenzBifurcationProblem() as problem:
        problem.quiet() # shut up and use the symbolical Hessian terms
        problem.setup_for_stability_analysis(analytic_hessian=True)

        # Start near the pitchfork at rho=1
        problem.rho.value=0.5
        problem.sigma.value=10
        problem.beta.value=8/3
        problem.solve(timestep=0.1) # Get the initial solution (trivial solution here)
        problem.solve()
        problem.solve_eigenproblem(1) # get an eigenvector as guess
```

Now we find the pitchfork and indeed confirm that it is at $\rho = 1$:

```
# Find the pitchfork in terms of rho
problem.activate_bifurcation_tracking(problem.rho,"pitchfork",eigenvector=problem.get_
↪last_eigenvectors()[0])
problem.solve()
x,y,z=problem.get_ode("lorenz").get_value(["x","y","z"])
print(f"Pitchfork starts at rho={problem.rho.value}, x,y,z={x,y,z}")
```

At a pitchfork bifurcation, we cannot easily continue in ρ since it is not clear which branch to take. Therefore, we obtain the critical eigenvector by `get_last_eigenvectors()`. As long as bifurcation tracking is active and it has been solved, it is not necessary (and not possible) to use `solve_eigenproblem()` for that. Instead, `get_last_eigenvectors()` gives the critical eigenvector at the bifurcation. We therefore first store this eigenvector and then deactivate the bifurcation tracking to be ready to solve the normal Lorenz system (i.e. without the augmentation for the bifurcation tracking):

```
# this will be now the critical eigenvector at the bifurcation
perturb=numpy.real(problem.get_last_eigenvectors()[0])
# deactivate bifurcation tracking: Solve again the normal Lorenz system
problem.deactivate_bifurcation_tracking()
```

To jump on the stable branch of the pitchfork bifurcation, we can add this eigenvector to the degrees of freedom using the `perturb_dofs()` method, increase ρ a bit beyond $\rho > 1$ and perform a few transient solves to move towards the stable branch, before the stationary solve jumps on it:

```
problem.perturb_dofs(perturb) # Go in the direction of the critical eigenvector
problem.rho.value+=0.1 # and go a bit higher with the rho value
problem.solve(timestep=[0.1,1,2,None]) # do a few time steps and then a stationary␣
↪solve (timestep=None)
eigvals, eigvects=problem.solve_eigenproblem(1) # get the initial eigenvalues
```

Then, we gradually increase ρ by arc length continuation, solve the eigenvalues and monitor whether the largest real part of the eigenvalues crosses zero:

```
# Scan rho to the Hopf bifurcation
ds=0.001
while eigvals[0].real<-0.001:
    ds=problem.arclength_continuation(problem.rho,ds)
    x, y, z = problem.get_ode("lorenz").get_value(["x", "y", "z"])
    eigvals, eigvects = problem.solve_eigenproblem(1)
    print(f"On pitchfork branch rho={problem.rho.value}, x,y,z={x, y, z}, eigenvalue=
↪{eigvals[0]}")
```

The eigenvalue will have a non-zero imaginary value, which indicates a Hopf bifurcation. This means the critical eigenvalue will not be zero, but in fact a pair of imaginary values $\pm i\omega$. For the same reason, the eigenvector $\vec{v}$ will be complex (and a complex conjugate counter-pair), i.e. $\vec{v}=\vec{\phi}+i\vec{\psi}$ with real valued $\vec{\phi}$ and $\vec{\psi}$. The bifurcation tracking of a Hopf bifurcation with respect to parameter r ($=\rho$ here) is internally again handled by augmenting the system as follows:

$$
\begin{aligned}
\vec{F}(\vec{x}_\mathrm{c},r_\mathrm{c})&=0 && \text{,i.e. } \vec{x}_\mathrm{c} \text{ is a stationary solution at the parameter } r_\mathrm{c}\\
\mathbf{J}(\vec{x}_\mathrm{c},r_\mathrm{c})\vec{\phi}+\mathbf{M}(\vec{x}_\mathrm{c},r_\mathrm{c})\vec{\psi}&=0 && \text{,i.e. the generalized eigenproblem is solved}\\
\mathbf{J}(\vec{x}_\mathrm{c},r_\mathrm{c})\vec{\psi}-\mathbf{M}(\vec{x}_\mathrm{c},r_\mathrm{c})\vec{\phi}&=0 && \text{for a pure imaginary eigenvalue } i\omega\\
\vec{v}_\mathrm{g}\cdot\vec{\phi}&=1 && \text{,i.e. the eigenvector } \vec{v} \text{ is non-trivial}\\
\vec{v}_\mathrm{g}\cdot\vec{\psi}&=0 && \text{and } \vec{\psi} \text{ does not contribute to the real part of } \vec{v}_\mathrm{g}
\end{aligned}
$$

Besides the complex eigenvector $\vec{v}=\vec{\phi}+i\vec{\psi}$, which can be obtained after bifurcation tracking by the two eigenvectors returned from `get_last_eigenvectors()`, one also gets the critical parameter, where the bifurcation happens, and the frequency ω, which can be obtained by the imaginary part of `get_last_eigenvalues[0]`.

Now, this is utilized to find the critical ρ_c where the Hopf bifurcation is located:

```
# Jump on the Hopf bifurcation
problem.activate_bifurcation_tracking(problem.rho,"hopf",eigenvector=problem.get_last_
↪eigenvectors()[0],omega=numpy.imag(problem.get_last_eigenvalues()[0]))
problem.solve()
x, y, z = problem.get_ode("lorenz").get_value(["x", "y", "z"])
print(f"On Hopf branch rho={problem.rho.value}, x,y,z={x, y, z}, omega={numpy.
↪imag(problem.get_last_eigenvalues()[0])}")
```

Since we do not `deactivate_bifurcation_tracking()`, it is still active. We can now perform an arc length continuation in another parameter, e.g. in σ, to obtain the curve $\rho_\mathrm{c}(\sigma)$:

```
# Go down with sigma but staying on the Hopf bifurcation (i.e. do not call deactivate_
↪bifurcation_tracking)
ds=-0.001
while problem.sigma.value>2+problem.beta.value:
    ds=problem.arclength_continuation(problem.sigma,ds,max_ds=0.1)
    x, y, z = problem.get_ode("lorenz").get_value(["x", "y", "z"])
    print(f"On Hopf branch rho,sigma={problem.rho.value,problem.sigma.value}, x,y,z=
↪{x, y, z}, omega={numpy.imag(problem.get_last_eigenvalues()[0])}")
```

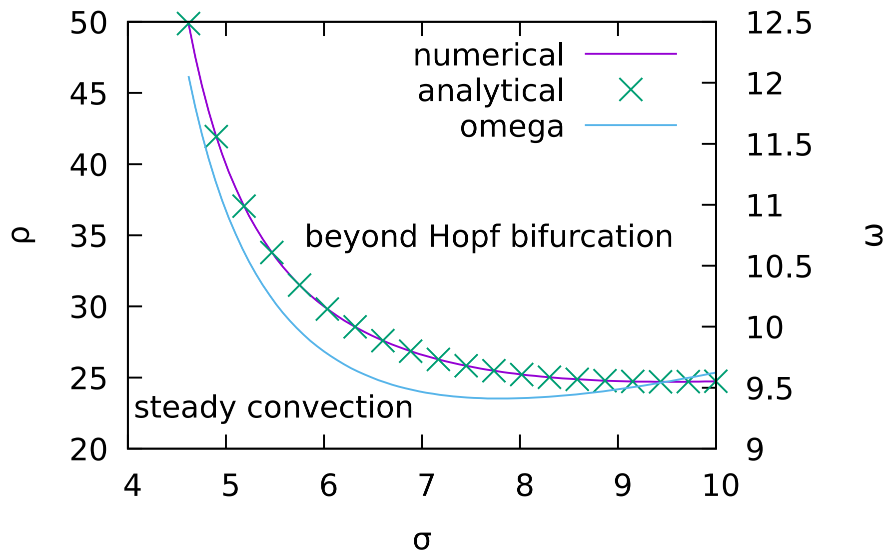

Fig. 3.14: Position of the Hopf bifurcation of the Lorenz system as function of the parameters ρ and σ at $\beta = 8/3$. The analytical solution $\rho_\mathrm{c} = \sigma(\sigma+\beta+3)/(\sigma-\beta-1)$ agrees perfectly.

Thereby, one can directly generate phase diagrams as e.g. depicted in Fig. 3.14.

The very same methods also work for spatio-temporal differential equation. An example will be discussed in Section 5.6.

3.11.8 Deflated solving and deflated continuation

Nonlinear problems generically have multiple solutions for fixed parameter values. While for the pitchfork normal form (3.15), all solutions can be found analytically (depending on the parameter either 1 or 3 solutions), this is in general not possible, e.g. when having a sufficiently complicated system of nonlinear equations. Even numerically, it is generically not possible to be sure that you have found all solutions, in particular for automatic methods. Without any knowledge of the residuals of a system with only a single unknown, you would have to sample all potential starting guesses, from $-\infty$ to ∞, and perform Newton's method with these initial guesses and record all found solutions.

This is of course not feasible and we therefore can only hope that any numerical algorithm can find a subset of solutions. Since a single initial guess will always run into the same solution during Newton's method (convergence provided), we either can use multiple starting guesses (as described above), or somehow prevent that Newton's method can run into the same solution again. This is obviously possible if we ensure that (i) all found solutions are penalized so that Newton's method cannot converge into it and (ii) that all not yet found solutions of the original problem are still solutions of the penalized problem. These observations suggest a product of a penalizer (called deflation operator) and the original residuals. Here, we follow the approach of Ref. [21]. If a single solution $\vec{U}_1$ is known, we define the deflation operator

$$\mathbf{W}_{\vec{U}_1}(\vec{U}) = \left(\frac{1}{\|\vec{U}-\vec{U}_1\|^p} + \alpha\right)\mathbf{1}$$

If multiple solutions $\vec{U}_1, \vec{U}_2, \ldots, \vec{U}_n$ are known, we can just use the product of the deflation operators. This defines a new residuals

$$\vec{R}_\mathrm{defl}(\vec{U}) = \mathbf{W}(\vec{U})\vec{R}(\vec{U})$$

which obviously fulfill the required properties. The shift α prevents that numerically diverging unknowns $\vec{U}$ will be considered as new solution, since the inverse norm can easily fall below the accuracy threshold of the Newton solver applied on $\vec{R}_\mathrm{defl}$.

pyoomph can apply the method of Ref. [21] automatically and iterate over multiple found solutions at a given parameter value. As an example, we will calculate the solutions of a pitchfork bifurcation at a specific parameter, where three solutions exists. The considered pitchfork normal form is the same as in (3.15), so we do not reiterate it here. Also, we do not define a problem class, but assemble our problem directly in the run script:

```
# Simple problems can be assembled without a specific class
problem=Problem()
problem+=PitchForkNormalForm(r=1,sign=-1)@"pitchfork"
# Find the solutions by deflation
solutions=[]
for sol in problem.iterate_over_multiple_solutions_by_deflation(deflation_alpha=0.1,
↪deflation_p=2,perturbation_amplitude=0.1,num_random_tries=2):
        solutions.append(sol)
print("Found solutions at r=1 are x = ",solutions)
```

When running, the method `iterate_over_multiple_solutions_by_deflation()` will first solve the problem normally, without any deflation. The `for`-loop will receive the corresponding degrees of freedom. After that, the first solution is removed by the deflation operator specified above. Here, you can select the exponent p and the shift α by the keyword arguments `deflation_p` and `deflation_alpha`. However, we must perturb the first solution before trying to find the next one. Otherwise, we would divide by zero. This is done by a random perturbation of the solution with an amplitude given by `perturbation_amplitude`. A single random try can be too less, so we allow also to specify the number of attempted solves with different random perturbations of the previous solution, which can be selected by `num_random_tries`. Optionally, you can also perform a perturbation by the dominant eigenvector when adding the argument `use_eigenperturbation=True`. An attempted solve with the perturbation in eigendirection will then be done additionally to the random perturbation(s). Whenever a new solution is found, also this solution is considered in the deflation operator. Afterwards, all found solutions are perturbed again and new attempts are started to find even more solutions.

Depending on the generated random numbers, one either find all three solutions $x = 0$, $x = \pm 1$, or only two of them. Hence, deflation provides no guarantee that indeed all solutions are found, but it is at least a promising approach to find further solutions.

Deflation can furthermore by combined with parameter scanning, in a sort of continuation. Opposed to arclength continuation, we do not solve along the arclength of a single solution branch, but just scan over the parameter branch once. But at each scanned parameter value, we try to find new solutions by deflation and try to connect the solutions at the previous parameter value to the new solutions. This algorithm has been proposed in Ref. [20], which can be invoked using the method `deflated_continuation()`:

```
problem=Problem()
r=problem.define_global_parameter(r=-1)
problem+=PitchForkNormalForm(r=r,sign=-1)@"pitchfork"

# Storage for the output files: Branch index -> output file
output_files={}

# Scan r from -1 to 1, apply deflated continuation
for branch_index,rvalue,sol in problem.deflated_continuation(r=numpy.linspace(-1,1,
↪50)):
        # we get the branch_index (increasing), the value of the parameter and the␣
↪degrees of freedom
        if branch_index not in output_files:
                # Create an output file for the new branch
                output_files[branch_index]=problem.create_text_file_output("branch_
↪{:02d}.txt".format(branch_index))
        # We can e.g. solve eigenproblems, or output solutions here
        problem.solve_eigenproblem(1)
        Re_ev=numpy.real(problem.get_last_eigenvalues()[0])
```

(continues on next page)

(continued from previous page)

```
        # Write the output
        output_files[branch_index].add_row(rvalue,sol[0],Re_ev)
```

A call of `deflated_continuation()` expect a parameter sampling range and has similar additional optional arguments as `iterate_over_multiple_solutions_by_deflation()`. At each solution, the `for`-loop receives and increasing branch index, the current parameter value and the degrees of freedom of the solution. Feel free to calculate e.g. eigenvalues or call e.g. `output()` inside the loop to process the current solution. You could also consider adding a `write_state()` whenever a new branch index starts. With another script, you can load these states via `load_state()` and e.g. finalize the bifurcation diagram by arclength continuation of all found solutions.

We can indeed recover the diagram of the pitchfork normal form (cf. Fig. 3.15), however, deflated continuation cannot connect the branching points nicely. It also does not note that branch 1 and branch 2 actually belong to the same branch, if we define a branch in a result of an arclength continuation:

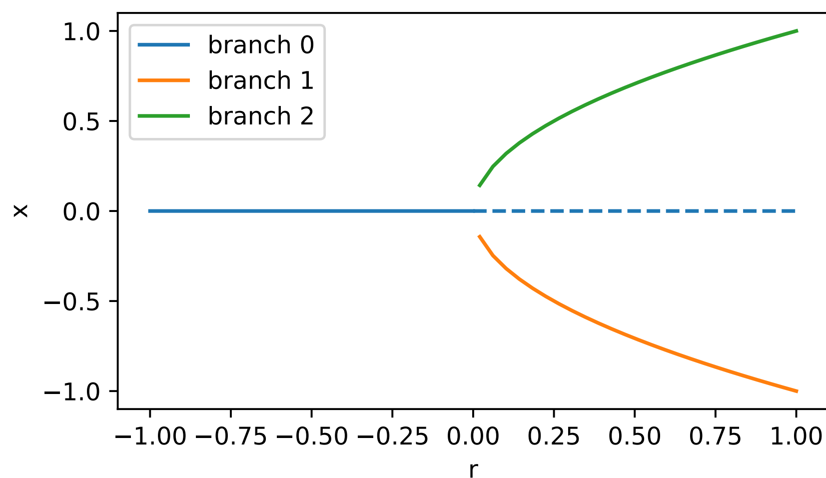

Fig. 3.15: Pitchfork solutions by deflated continuation.

3.11.9 Lyapunov exponents

After a stationary state becomes unstable due to bifurcations, it can either converge to another stationary solution, diverge or show nontrivial temporal dynamics. In the latter case, the dynamics can be either periodic, quasi-periodic or chaotic. A prominent example of chaotic dynamics is given by the Lorenz system, which was already discussed in Section 3.6.5. However, just from the trajectory of the system alone, it is not directly apparent that it is indeed a chaotic system.

A necessary requirement for deterministic chaos is the sensitivity to small changes of the initial conditions. If a system is chaotic, a small perturbation in the initial condition will grow over time and the long term dynamics of the unperturbed and the perturbed system will be separated by a growing distance in phase space. However, usually, dissipative systems are considered, i.e. systems that relax on a *strange attractor*, which is bounded in phase space. Therefore, eventually both the unperturbed and perturbed trajectory, can converge to the same *strange attractor* and the distance between both trajectories in time will be bounded as well.

Therefore, instead of measuring the divergence of perturbtions in the initial conditions (which is ultimately bounded) it is beneficial to investiage the growth of the a tiny perturbation along the trajectory. Let $\vec{x}(t)$ be a trajectory, then we are interested in how a perturbation $\vec{x}(t)+\delta\vec{x}(t)$ will develop. Here, we choose a tiny initial pertrubation vector $\delta\vec{x}(0)$ and make sure that it remains tiny during its evolution, so that the linear dynamics around $\vec{x}(t)$ alone govern the choice of $\delta\vec{x}(0)$. This can be achieved by either choosing the magnitude of the initial perturbation $\delta\vec{x}(0)$ sufficiently small that it does not grow within the considered simulation time to a magnitude where nonlinear contributions become relevant.

Alternatively, one can renormalize $\delta\vec{x}(t)$, e.g. after each nth time step, to a tiny magnitude, but monitoring its exponential growth.

More precisely, for a system $\partial_t\vec{x} = \vec{F}(\vec{x})$, it is sufficient to calculate the linearized evolution of

$$\partial_t\delta\vec{x} = \mathbf{J}(\vec{x}(t))\delta\vec{x} \tag{3.16}$$

where $\mathbf{J}$ is the linearization of $\vec{F}$, i.e. the Jacobian along the trajectory. Due to nonlinearities, the Jacobian is not constant along the trajectory, but the long term dynamics still follow an exponential growth or decay in the long-term limit, i.e. $\delta\vec{x}(t) \sim \exp(\lambda t)$. For an N-dimensional system, we generically have N different solutions for the exponent λ, which are called Lyapunov exponents. If at least one λ is positive, a random perturbation will generically grow over time. If the sum of all Lyapunov exponents is additionally negative, we converge to a strange attractor, i.e. an indicator for chaos. Note that one Lyapunov exponent is usually zero (if the solution is not attaining a stationary solution), corresponding to the initial perturbation $\delta\vec{x}(0) \propto \partial_t\vec{x}(0)$, i.e. just a shift in the direction of the trajectory $\vec{x}(0)$.

Conventional implementations to numerically calculate Lyapunov exponents rely on the explicit form (3.16). However, not all systems can be written in this form, e.g. the pendulum constraint in Section 3.7, which has no time derivative. Since pyoomph allows to formulate equations in the implicit residual formulation $\vec{R}(\partial_t\vec{x}, \vec{x}) = 0$, a general form of (3.16) reads

$$\mathbf{M}\partial_t\delta\vec{x} + \mathbf{J}\delta\vec{x} = 0 \tag{3.17}$$

with the mass matrix $\mathbf{M}$ and the Jacobian (without any time-derivatives) $\mathbf{J}$.

> **Warning:** This form is of course only valid it the maximum time derivative order is one, i.e. for second order time derivatives, again the usual substitution has to be done, cf. Section 3.6.4.

To calculate Lyapunov exponents, pyoomph comes the class `LyapunovExponentCalculator` from the `pyoomph.utils.lyapunov` module. It starts with one or multiple initially random guesses of $\delta\vec{x}(0)$ and integrates (3.17) along the trajectory. If more than one guess is taken, a Gram-Schmidt orthogonalization is performed after each step. Thereby, components of the faster growing perturbations (i.e. corresponding to higher Lyapunov exponents λ) are removed for the slower growing or even decaying perturbation components. Automatically, the class `LyapunovExponentCalculator` writes the desired number of Lyapunov exponents λ to a text file. Due to the random initial guess, usually the largest Lyapunov exponents will be calculated, but these are normally also the most interesting ones. Moreover, the class `LyapunovExponentCalculator` allows to select two times, T_{wait} and T_{relax}, by the keyword arguments `waiting_time` and `prerelaxation_time`. The former is the time delay when the actual calculation of the perturbation vector sets in, the latter is the time offset to start the actual calculation of the Lyapunov exponents, i.e. allowing some time to let the pertubation vectors adjust according to the dynamics. The Lyapunov exponents will then be estimated by

$$\lambda(t) = \frac{1}{t - T_{\text{init}}} \ln \frac{\|\delta\vec{x}(t)\|}{\|\delta\vec{x}(T_0)\|} \tag{3.18}$$

where $T_0 = T_{\text{wait}} + T_{\text{relax}}$.

As an example, we will check the Lorenz system (with the default parameters $\sigma = 10$, $\rho = 28$ and $\beta = 8/3$) from Section 3.6.5 for chaos in the following. When modifying the run code of section Section 3.6.5 to

```
# Import the LyapunovExponentCalculator from the utils module
from pyoomph.utils.lyapunov import LyapunovExponentCalculator

with LorenzProblem() as problem:
# We want to save memory, since we have a fine temporal discretization.
# So we do not write state files for continue simulations
problem.write_states=False
```

(continues on next page)

(continued from previous page)

```
# Add the LyapunovExponentCalculator to the problem.
# Calculating k=3 Lyapunov exponents. Starting after t=10, then relaxing perturbation␣
↪vectors until t=10+5
# Then start the actual Lyapunov exponent calculation
problem+=LyapunovExponentCalculator(k=3,waiting_time=10,prerelaxation_time=10,store_
↪as_eigenvectors=False,use_crank_nicholson_integration=False)
# Run it with a rather fine time step
problem.run(endtime=200,outstep=0.001)
```

we get a file called `lyapunov.txt` in the output directory. The delay times are chosen to $T_{\text{wait}} = T_{\text{relax}} = 10$ The resulting plot is the following, where we also added the long-time limit literature values by dotted lines. The sum of all Lyapunov exponents corresponds to the phase space divergence, i.e. the trace of the Jacobian, which can be obtained analyically by $\sum_{i=1}^{3} \lambda_i = -\sigma - 1 - \beta \approx -13.666$. Finer time steps give even better agreement.

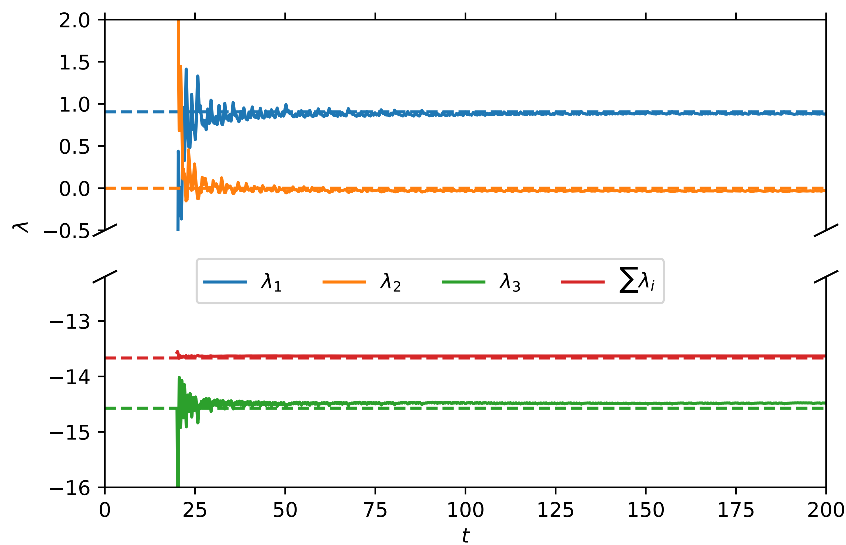

Fig. 3.16: Lyapunov spectrum of the Lorenz system with $\sigma = 10$, $\rho = 28$ and $\beta = 8/3$. Dotted lines are the long-time limit literature values.

As expected, we have one positive Lyapunov exponent (indicating chaos), one is around zero (in the direction of the trajectory) and another one is negative, so that the sum of all is in total negative (requirement for phase volume contraction, i.e. for a strange attractor).

The method described here can also be applied to spatio-temporal PDEs, which are discussed later in Section 5.

3.12 Periodic orbits

Frequently, a dynamical system (or even a PDE system with constraints) shows periodic orbits. Such orbits e.g. arise at Hopf bifurcations. A periodic orbit of a solution $\vec{x}(t)$ is defined by

$$\begin{aligned} \vec{R}(\partial_t \vec{x}(t), \vec{x}(t)) &= 0 \\ \vec{x}(t+T) &= \vec{x}(t) \end{aligned} \tag{3.19}$$

Here, the orbit period T is chosen uniquely so that $T > 0$ is the smallest value fulfilling the periodicty equation (obviously, multiples of T will also fulfill it).

We furthermore demand in the following, that the time derivatives are of first order (higher order time derivatives can be again realized by auxiliary degrees of freedom) and that we can write our implicit residual formulation $\vec{R}(\partial_t\vec{x}(t),\vec{x}(t))=0$ as

$$\begin{aligned}\mathbf{M}(\vec{x})\partial_t\vec{x}(t)+\vec{R}_0(\vec{x})&=0\\ \vec{x}(t+T)&=\vec{x}(t)\end{aligned}\tag{3.20}$$

Here, $\mathbf{M}(\vec{x})$ is the mass matrix, which might depend on the unknows, but - as a restriction here - not on the time derivatives. The time-independent residual $\vec{R}_0$ is just the part of the residual which is recovered when subsituting $\partial_t\vec{x}=0$, i.e. the same one used for stationary solutions. Usually, i.e. in almost all cases, (3.19) can be rewritten in the form (3.20), potentially by the usage of auxiliary degrees of freedom for e.g. terms nonlinear in $\partial_t\vec{x}$.

If we have a good initial guess for the orbit, we can solve for the orbit $\vec{x}(t)$ for $t\in[0,T)$. In pyoomph, this can be done in a monolithic, fully implicit manner. To that end the time-dependency of $\vec{x}(t)$ is discretized in time by state vectors, $\vec{x}_l$ with $l=1,2,\ldots,N_T$, where N_T is the number of considered discrete representations in time. The interpolation for arbitrary times in between can be done in different ways. Pyoomph e.g. offers periodic B-splines or conventional Lagrange polynomials to perform the interpolation in time. Subsequently, the system (3.20) can be solved. However, note that we have an additional unknown, namely the exact period T. Luckily, we also have an additional equation stemming from the invariance of $\vec{x}(t)$ by a shift in time, i.e. $t\to t+c$ for any constant c. This invariance can be removed by posing a constraint, which simultaneously serves as equation for the unknown period T. Pyoomph either can use a plane constraint, which demands that $\vec{x}(0)$ is located on a plane in phase space. This plane is chosen by the initial guess of $\vec{x}_0(0)$ (or its value from the previous step in a arc-length continuation of an orbit in a parameter). The normal of the plane is then given by $\partial_t\vec{x}_0(0)$. Alternatively, we can pose a phase constraint. Here, we use the initial guess $\vec{x}_0$ (or the previous continuation step) to enforce that

$$\int_0^T \partial_t\vec{x}_0\cdot\vec{x}\,\mathrm{d}t=0\tag{3.21}$$

holds. For $\vec{x}=\vec{x}_0$, this is trivially fulfilled and therefore, it automatically constitutes a good guess for actual solving or continuation.

3.12.1 From Hopf bifurcations to periodic orbits

Let us consider again the Lorenz system. From the example Section 3.11.7, we know that this system exhibits Hopf bifurcations. We therefore know what periodic orbits exist, at least in the vicinity of these bifurcations. However, a linear stability analysis cannot reveal whether the Hopf bifurcation is super- or subcritical, since this requires the knowledge of the so-called first Lyapunov coefficient (don't mess it up with the Lyapunov exponent of the previous page). The first Lyapunov coefficient c_1 appears in the normal form of the Hopf bifurcation, which reads

$$\dot{z}=(p+i+c_1|z|^2)z\tag{3.22}$$

Here, z is a single complex unknown, p is a real parameter and c_1 is, as mentioned before, the first Lyapunov coefficient. If $\mathrm{Re}(c_1)<0$, stable periodic orbits appear for $p>0$ with an amplitude $\sqrt{-p/\mathrm{Re}(c_1)}$. The Hopf bifurcation is supercritical. Otherwise, if $\mathrm{Re}(c_1)>0$, unstable orbits are present for $p<0$ and the bifurcation is subcritical.

Close to the Hopf bifurcation, the normal form (3.22) and the actually considered system are equivalent. This can be utilized so calculate c_1 also for Hopf bifurcations in general systems and thereby classify the type of the Hopf bifurcation. The calculation is quite intricate, but straightforward. Details can be found here [32], but note that pyoomph generalizes the approach presented in the book [32] by the presence of an arbitrary mass matrix.

Additionally, we can calculate the amplitude of the orbit and thereby construct an initial guess for the orbit which can subsequently be solved and followed along the bifurcation parameter. The definition of the Lorenz system and the problem class is analogously to Section 3.11.7, so we just focus on the orbit tracking here. The code is actually really short:

```
with LorenzProblem() as problem:
        # To calculate c_1, we need the Hessian, so the symbolical code must be␣
↪generated and compiled
        problem.setup_for_stability_analysis(analytic_hessian=True)
        # Add a non-trivial initial condition
        problem+=InitialCondition(x=1,z=24)@"lorenz"
        problem.rho.value=24 # Start close to the Hopf

        problem.solve() # Find a stationary solution (will be on one of the pitchfork␣
↪branches)
        problem.solve_eigenproblem(n=1) # And get some eigenvalue for the Hopf tracker

        problem.activate_bifurcation_tracking("rho","hopf") # Activate the Hopf␣
↪tracking
        problem.solve() # Find the Hopf bifurcation by adjusting rho

        # Since we are on the Hopf bifurcation, we can switch to the orbit
        # We chose NT=100 time points for the orbit
        # The initial period T and the initial guess of the orbit will be calculated␣
↪automatically
        with problem.switch_to_hopf_orbit(NT=100) as orbit:
            print("Bifurcation is supercritical: "+str(orbit.starts_
↪supercritically()))
            print("Period at rho=",problem.rho.value, " is ",orbit.get_T())
            # This function will write the output along the orbit to a subdirectory␣
↪in the output directory
            orbit.output_orbit("orbit_at_rho_{:.4f}".format(problem.rho.value))
            # Perform continuation in rho
            # We do not know in which direction we have to go (depends on the nature␣
↪of the Hopf)
            # But a good guess including direction can be obtained from the Lyapunov␣
↪coefficient
            ds=orbit.get_init_ds()
            while problem.rho.value>16:
                ds=problem.arclength_continuation("rho",ds)
                print("Period at rho=",problem.rho.value, " is ",orbit.get_T())
                orbit.output_orbit("orbit_at_rho_{:.4f}".format(problem.rho.value))
```

As a first step, we must the code generator to derive the Hessian and generate C code to fill the Hessian. We need it later, when we want to calculate the first Lyapunov coefficient c_1. As detailed in Ref. [32], the calculation of c_1 requires directional derivatives of $\vec{R}_0$ of second and third order around the Hopf bifurcation point. With the Hessian, we can calculate the second order directional derivatives fully symbolically, whereas the third order directional derivative is calculated by first order finite differences of the symbolically calculated second order derivatives.

We then find the Hopf bifurcation by first starting nontrivially near the Hopf and solve the problem. We will end up on one of the pitchfork branches. Then, solving the eigenproblem gives a good guess for the subsequently invoked Hopf bifurcation tracking. We will thereby locate the Hopf bifurcation and the critical parameter ρ.

Once this is done, we can activate the orbit tracking by `switch_to_hopf_orbit()`. As arguments, we can pass e.g. the number of points to use, the time interpolation mode, whether a phase or a plane constraint should be used to remove the shift invariance in the time and simultaneously constitute an equation for the unknown period T. The main interpolation modes are `"collocation"` (default), which performs the orthogonal collocation method as e.g. used in AUTO [19], and `"bspline"`, which samples the orbit by periodic B-splines. As long as we stay in the `with`-statement, orbit tracking is activated. Once we leave it, it will be deactivated with suitable history conditions to perform conventional time integration via the `run()` command afterwards. The returned `orbit` object provides several methods to inspect the orbit. In particular, we can ask whether the Hopf bifurcation is supercritical (i.e. $c_1 < 0$) by the method `starts_supercritically()`. Here, it is not, meaning that the orbits - at least close to the Hopf bifurcation -

are unstable and therefore cannot be found by conventional time integration. We can obtain the period T by the method `get_T()`. Likewise, we can output the orbit (which will be written in a subdirectory of the output directory) by the `output_orbit()` method. Continuation in the parameter ρ works as before, but we do not really know any good initial step for the arclength continuation. In particular, super- and subcritical Hopf bifurcations must continue in a different direction, since orbits only exists in one direction. However, `switch_to_hopf_orbit()` already calculates a reasonable step, which is available via `get_init_ds()`.

Eventually, a plot as shown in Fig. 3.17 can be obtained, visualizing the orbits as function of the parameter ρ. With time integration, these orbits cannot be found due to their unstable nature.

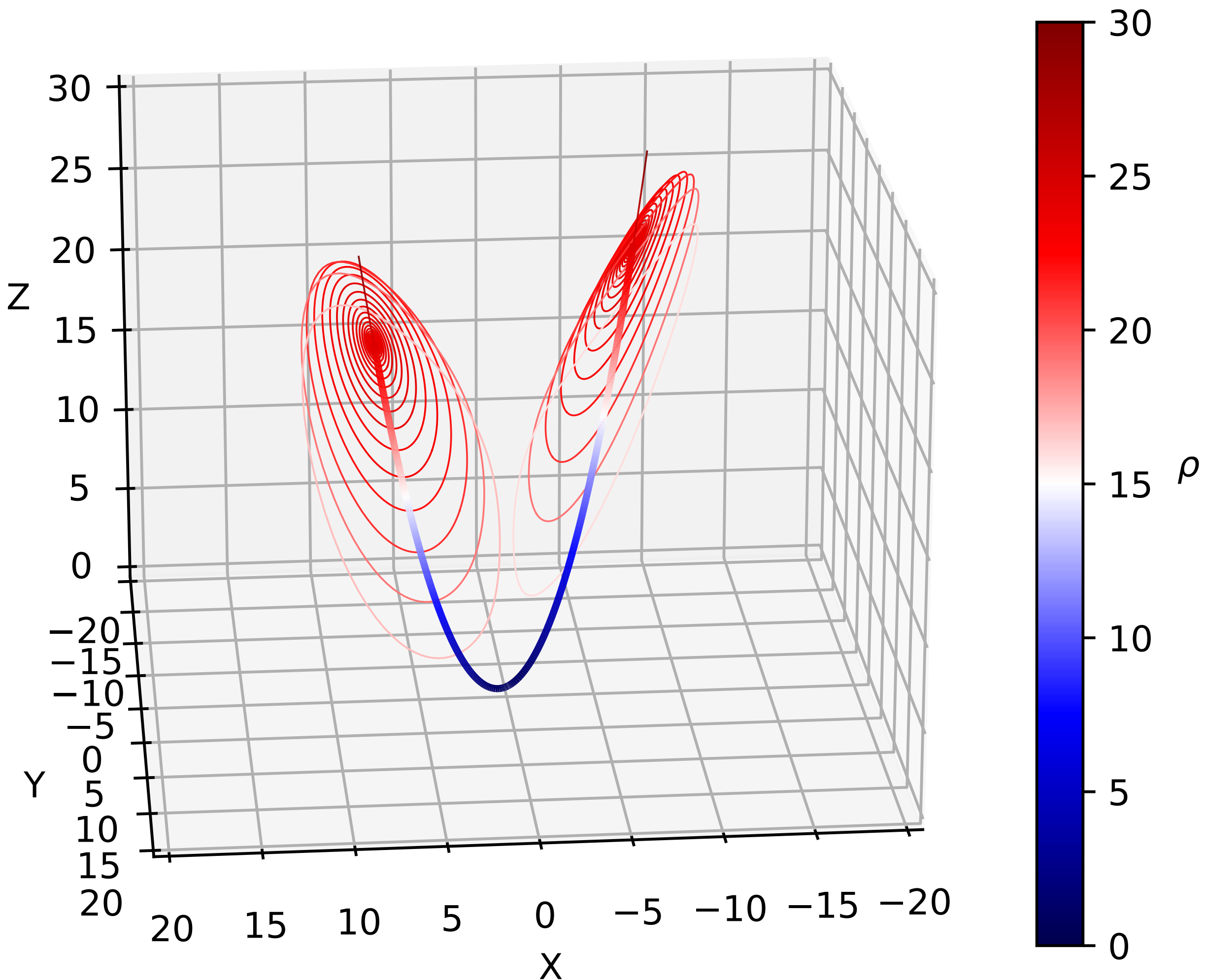

Fig. 3.17: Orbits originating from the Hopf bifurcations in the Lorenz system. The color-code indicates the value of ρ. Thick lines represent the stable pitchfork branches, which become unstable in a subcritical Hopf bifurcation. By the orbit tracking, we can calculate the unstable orbits which cannot be done by time integration.

3.12.2 Constructing orbits manually

So far, we have constructed orbits in the Lorenz system by the emergence at the Hopf bifurcation. While this is convenient way, you sometimes note dynamics which might be periodic (or at least quasi-periodic/chaotic), but you do not know any Hopf bifurcation in the vicinity. This in particular happens for the Lorenz system beyond the Hopf bifurcations, i.e. for values of ρ higher than the Hopf bifurcation point. In the last example, we got unstable orbits for ρ below the Hopf bifurcation, since it is a subcritical bifurcation. Beyond the bifurcation, such orbits do not exists. Instead, as we have seen in Section 3.11.9, we have chaotic dynamics there. However, inside this chaos, there are actually unstable orbits. Since they are unstable, they are never approached by the dynamics, but yet they exist. However, we cannot reach them from the Hopf bifurcation.

For such cases, one has to manually construct a guess for the periodic orbit. We can do so easily by first performing time integration in the chaotic region. After some transient time integration $[0, t_0)$, we will consider a representative time interval $[t_0, t_0 + T)$. Obviously, since we are on a chaotic attractor, the periodicity condition $\vec{x}(t_0) = \vec{x}(t_0 + T)$ will not be fulfilled. Due to the violated periodicity condition, huge jumps would occur if we just use this representative time dynamics as initial guess for the orbit solver. This will lead to severe convergence issues when solving for the actual orbit.

Therefore, we will apply a low pass filter to match the start and end point nicely. Huge jumps correspond to high frequencies which then will be filtered out. The evolution after the application of the low pass filter will not fulfil the Lorenz system anymore, but it still constitutes a reasonable guess for the Newton solver.

Then, we manually start orbit tracking to solve for an initial orbit and subsequently continue it in ρ. The corresponding code just reads:

```
from hopf_switch import * # Load the previous code

if __name__=="__main__":
    with LorenzProblem() as problem:
        problem.setup_for_stability_analysis(analytic_hessian=True)

        # Go above the Hopf bifurcation into the chaos
        problem.rho.value=28
        problem+=InitialCondition(x=1,y=1,z=28)@"lorenz"
        # Solve for the unstable pitchfork branch
        problem.solve()
        # Perturb it by the eigenfunction to leave the pitchfork by time integration
        problem.solve_eigenproblem(0)
        problem.perturb_dofs(100*problem.get_last_eigenvectors()[0])
        # First run without any outputs
        problem.run(endtime=10,maxstep=0.0125,outstep=False,startstep=0.0001,temporal_
↪error=0.005,do_not_set_IC=True)
        # Then write the dynamics from t=10 to t=13 to the output
        problem.run(endtime=13,startstep=0.0125,outstep=True,do_not_set_IC=True)

        # Load the output
        data=numpy.loadtxt(problem.get_output_directory("lorenz.txt"))
        Tguess=data[-1,0]-data[0,0] # Get a guess for the period (will be T=3)
        # Apply a low pass filter to make the guess periodic
        fft=numpy.fft.rfft(data[:,1:],axis=0)
        freqs=numpy.fft.rfftfreq(data.shape[0],d=Tguess/data.shape[0])
        fft[fft.shape[0]//16:,:]=0.0    # Just let only 1/16 of the frequencies pass
        smoothed=numpy.fft.irfft(fft,axis=0)
        numpy.savetxt(problem.get_output_directory("lorenz_smoothed.txt"),numpy.
↪column_stack([data[:,0],smoothed]),header="t x y z")
        # Start with the smoothed data
        problem.set_current_dofs(smoothed[0])
        # And give the smoothed recorded time history as orbit guess
        orbit=problem.activate_periodic_orbit_handler(Tguess,history_
```

(continues on next page)

(continued from previous page)

```
↪dofs=smoothed[1:],mode="bspline",order=3,GL_order=3)
        # Solve for the real orbit
        problem.solve()

        # Continue the orbit in rho
        def output_orbit_rho():
            orbit.output_orbit("orbit_at_rho_{:.3f}.txt".format(problem.rho.value))

        output_orbit_rho()
        ds=-0.01
        while problem.rho.value>20:
            ds=problem.arclength_continuation("rho",ds,max_ds=0.1)
            output_orbit_rho()
```

As described above, we first start in chaos, then run some initial steps followed by a representative period where we write output. This output is loaded, smoothed by a low-pass filter and subsequently used as guess for the orbit. To that end, we first use `set_current_dofs()` to set the starting point of the orbit guess and ship the remaining history values to `activate_periodic_orbit_handler()`. Afterwards, it the continuation is analogous to the previous example.

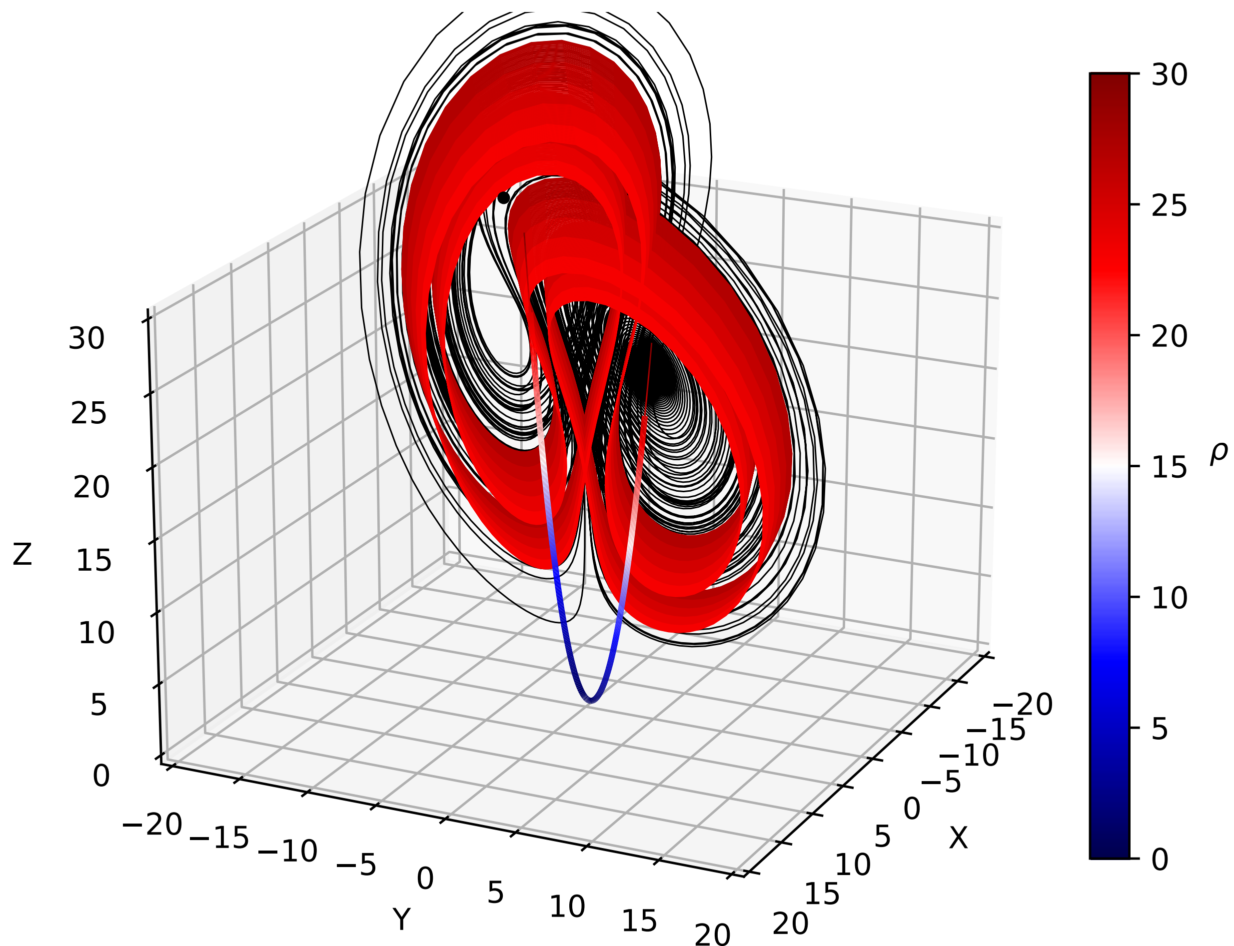

Fig. 3.18: Unstable orbits in the chaotic region of the Lorenz system. The color-code indicates the value of ρ. Visualized is the pitchfork branch, time integrated chaos in black and the found periodic orbit branch.

> **Warning:** Here, we just could load the text file and calculate the smoothed history dofs from there. For e.g. PDEs, it might be more involved, since the order of the degrees of freedom is not necessarily the same as in the written output. Therefore, instead of loading the output from file, it is more suitable to make a loop over `solve()` command with a suitable `timestep` argument and afterwards append the current degrees of freedom to an array. The current degrees of freedom can be obtained by the first return value of `get_current_dofs()`. After this loop, you have an array of history dofs, exactly as required here, which then can be filtered and used as orbit guess as above.

3.12.3 Stability of orbits

As stationary solutions, periodic orbits can exhibit different kinds of stabilities. In the discussed Lorenz system, we only found unstable orbits, but how to prove that they are indeed unstable? As usual, stability can be defined by the linearized dynamics around the state of interest, here the orbit.

Recalling (3.20), a linearization around the orbit $\vec{x}(t)$ corresponds to the solution

$$\begin{aligned}\mathbf{M}(\vec{x})\partial_t\vec{v}(t)+\mathbf{J}_0(\vec{x})\vec{v}&=0\\ \vec{v}(t+T)&=\lambda\vec{v}(t)\end{aligned}\tag{3.23}$$

Here, $\mathbf{J}_0(\vec{x})$ is the Jacobian of the time-independent residual $\vec{R}_0(\vec{x})$ and $\vec{v}(t)$ is the linear perturbation of the orbit. After one period, we do not necessarily end up at the very same point, i.e. in general $\vec{v}(t+T)\neq\vec{v}(t)$. Instead, we generically end up at another point, which is expressed by the so-called *Floquet multiplier* λ. *Floquet theory* tells us that such solutions $\vec{v}(t)$ with corresponding values of λ exist, which resemble the conventional eigenvectors and -values used for the stability of stationary solutions, although there are some differences as detailed in the following. Due to linearity, a second turn along the orbit will end up at $\vec{v}(t+2T)=\lambda^2\vec{v}(t)$ and (as long as the perturbation remains small) a multiplicator of λ^n after n periods. Obviously, an orbit is stable if all Floquet multipliers (which are in general complex-valued) satisfy $|\lambda|<1$. However, there is always a Floquet multiplier $\lambda=1$ present in the system, which corresponds to the shift invariance in time. The corresponding perturbation is just $\vec{v}=\partial_t\vec{x}$, i.e. if we just start our perturbation somewhere else in time, we will end up shifted as well after a period of T.

Pyoomph can calculate Floquet multipliers, which is demonstrated in the following on the basis of the Langford ODE system. This one reads with suitable parameters [25]:

$$\begin{aligned}\partial_t x&=(\mu-3)x-\frac{1}{4}y+x\left(z+\frac{1}{5}(1-z^2)\right)\\ \partial_t y&=(\mu-3)y+\frac{1}{4}x+y\left(z+\frac{1}{5}(1-z^2)\right)\\ \partial_t z&=\mu z-\left(x^2+y^2+z^2\right)\end{aligned}\tag{3.24}$$

For $\mu>1.683$, perfectly circular orbits can be found which change the stability from stable to unstable at $\mu=2$ [25].

Implementing the equation and setting up the problem is again trivial:

```
#See https://arxiv.org/abs/2407.18230v1
class LangfordSystem(ODEEquations):
	def __init__(self,mu):
		super(LangfordSystem,self).__init__()
		self.mu=mu

	def define_fields(self):
		self.define_ode_variable("x","y","z")

	def define_residuals(self):
		x,y,z=var(["x","y","z"])
```

(continues on next page)

(continued from previous page)

```
            xrhs=(self.mu-3)*x-0.25*y+x*(z+0.2*(1-z**2))
            yrhs=(self.mu-3)*y+0.25*x+y*(z+0.2*(1-z**2))
            zrhs=self.mu*z-(x**2+y**2+z**2)
            residual=(partial_t(x)-xrhs)*testfunction(x)
            residual+=(partial_t(y)-yrhs)*testfunction(y)
            residual+=(partial_t(z)-zrhs)*testfunction(z)
            self.add_residual(residual)


class LangfordProblem(Problem):
    def __init__(self):
         super().__init__()
         self.mu=self.define_global_parameter(mu=1.6)

    def define_problem(self):
        eqs=LangfordSystem(self.mu)
        eqs+=ODEFileOutput()
        self.add_equations(eqs@"langford")

    def get_analytical_nontrivial_floquet_multiplier(self):
        # Calculate the analytical nontrivial Floquet multiplier
        muv=self.mu.value
        z=2.5*(1-numpy.sqrt(0.8*muv-1.24))
        r=numpy.sqrt(z*(muv-z))
        exponent=(muv-2*z+numpy.emath.sqrt((muv-2*z)**2-8*r*(r-0.4*r*z)))/2
        T=4*2*numpy.pi
        multiplier=numpy.exp(exponent*T)
        if numpy.imag(multiplier)<0:
            multiplier=numpy.conjugate(multiplier) # We always consider the one with␣
↪positive imaginary part
        return multiplier
```

Note how we also provide a function to calculate the analytical nontrivial Floquet multiplier [25], i.e. a complex Floquet multiplier which is not the trivial one $\lambda = 1$.

We will then again find the Hopf bifurcation, switch to the orbit and continue in the parameter μ. But at each continuation step, we also calculate the Floquet multipliers and write the non-trivial one (along with the corresponding analytical solution) to the output:

```
with LangfordProblem() as problem:
   # Use again an analytic Hessian for the determination of the first Lyapunov␣
↪coefficient
   problem.setup_for_stability_analysis(analytic_hessian=True)
   # We also need the SLEPc eigensolver here
   problem.set_eigensolver("slepc").use_mumps()

   problem+=InitialCondition(x=0.01,z=1.1)@"langford"  # Some non-trivial initial␣
↪position

   # Find the Hopf bifurcation as usual
   problem.solve()
   problem.solve_eigenproblem(3)
   problem.activate_bifurcation_tracking("mu")
   problem.solve()

   # Output file to compare the numerical and analytical Floquet multipliers
   floquet_output=problem.create_text_file_output("floquet.txt",header=["mu","num_real
```

(continues on next page)

(continued from previous page)

```
↪","num_imag","ana_real","ana_imag"])

  # Switch again to the orbits originating from the Hopf bifurcation
  with problem.switch_to_hopf_orbit(NT=50,order=3) as orbit:
          ds=orbit.get_init_ds()
          maxds=ds*100 # Limit the maximum step size
          while problem.mu.value<2.05:
                  ds=problem.arclength_continuation("mu",ds,max_ds=maxds)
                  F=orbit.get_floquet_multipliers(n=3,shift=3) # Calculate some␣
↪Floquet multipliers
                  # However, not always three multipliers are found. We have to␣
↪consider the cases
                  if len(F)==3:
                          # Three multipliers found: The trivial one and two complex␣
↪conjugate ones
                          F=numpy.delete(F,numpy.argmin(numpy.abs(F-1)))
                          nontrivial_floquet=F[0] # Take one of the complex␣
↪conjugate multipliers
                  elif len(F)==2:
                          # Only two multipliers found: The trivial one and one real␣
↪one
                          F=numpy.delete(F,numpy.argmin(numpy.abs(F-1)))
                          nontrivial_floquet=F[0] # Take the remaining multiplier
                  else:
                          # Only one multiplier found: The trivial one
                          nontrivial_floquet=0 # The others are then very close to 0

                  if numpy.imag(nontrivial_floquet)<0:
                          # conjugate a multiplier with negative imaginary part
                          nontrivial_floquet=numpy.conjugate(nontrivial_floquet)

                  # Output the orbit
                  odir="orbit_{:.3f}".format(problem.mu.value)
                  orbit.output_orbit(odir)

                  # Write to output for comparison
                  floq_ana=problem.get_analytical_nontrivial_floquet_multiplier()
                  floquet_output.add_row(problem.mu, nontrivial_floquet.real,
↪nontrivial_floquet.imag,floq_ana.real,floq_ana.imag)
```

Floquet multipliers can be calculated via the method `get_floquet_multipliers()` of the `PeriodicOrbit` class. The internals work analogously to the way proposed in Ref. [19]. However, multipliers close to zero will be discarded. The usually do not give any information on the stability anyways. We carefully have to select the interesting Floquet multiplier and write it to the output. As depicted in Fig. 3.19, the results agrees well with the analytical Floquet multiplier.

Since the Floquet multipliers at $\mu = 2$ cross the stability condition $|\lambda| = 1$ by a complex-conjugated pair, this corresponds to a Neimark-Sacker bifurcation. The orbit becomes unstable to a torus. We can check this by performing time integration. The moment we leave the `with` statement of the `orbit`, pyoomph will initialize the degrees of freedom to the starting point of the orbit. A trivial `run()` statement will then perform a time integration along the orbit. However, if we start at $\mu > 2$ (here e.g. $\mu = 2.005$), it will be unstable and we can see the torus developing. We just have to replace the orbit loop by (i.e. the code after solving for the Hopf bifurcation) by:

```
with problem.switch_to_hopf_orbit(NT=50,order=3) as orbit:
        ds=orbit.get_init_ds()
        maxds=ds*100 # Limit the maximum step size
```

(continues on next page)

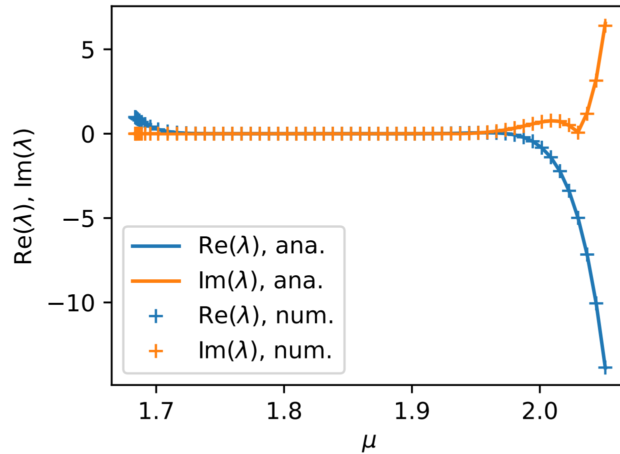

Fig. 3.19: Floquet multipliers of the Langford ODE system

(continued from previous page)

```
        problem.go_to_param(mu=2.005,startstep=ds,max_step=maxds,call_after_step=␣
↪lambda ds: orbit.output_orbit("orbit_at_mu_"+str(problem.mu.value)))
        T=orbit.get_T() # Get the period
        NT=orbit.get_num_time_steps() # Get the number of time steps
        dt=T/NT # And calculate a good time step

# Running a transient integration starting on the orbit
problem.run(40*T,outstep=dt/4)
```

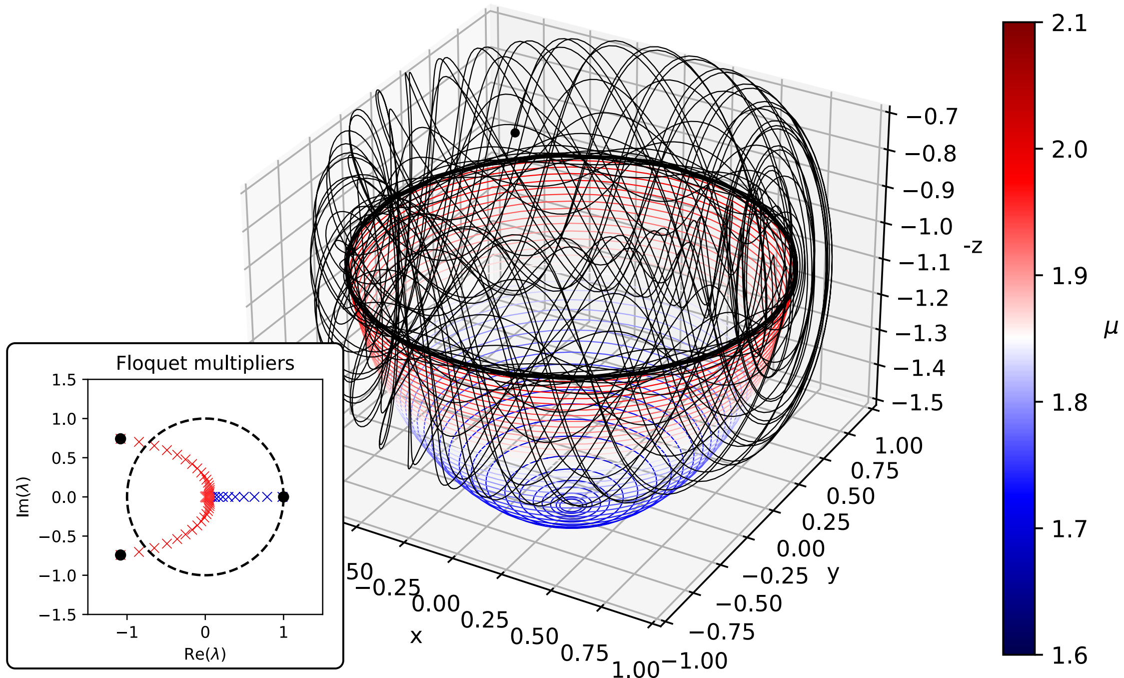

Fig. 3.20: Stable orbits (color-coded by μ) and time integration at $\mu = 2.005$ (black) showing the unstable dynamics building a torus. Also, the path of the Floquet multipliers as function of μ is shown.

CHAPTER

FOUR

SPATIAL DIFFERENTIAL EQUATIONS

While the previous chapter was dedicated on temporal ODEs, this section will focus on pure spatial differential equations, i.e. without temporal derivatives, but with spatial derivatives instead. This necessitates a discretization in space of a given particular geometry in either one, two or three spatial dimensions.

4.1 The Poisson equation

The Poisson equation is the most common example to illustrate the finite element method. It is the most trivial equation which directly allows to discuss the central ideas of the finite element method. Based on the Poisson equation, we will discuss the derivation of the weak formulation, how to incorporate boundary conditions, and how to solve the equation in different dimensions and coordinate systems. Furthermore, adaptive mesh refinement will be discussed. All these topics are illustrated by simple example codes which can be used as a starting point for more complex simulations.

4.1.1 Weak formulation

The most basic example is usually the *Poisson equation*, which is usually taken as prime example for the *finite element method*. The Poisson equation for an unknown function $u(\vec{x})$ with a source function $g(\vec{x})$, both defined on an n-dimensional domain Ω with position vector $\vec{x}$ reads

$$-\nabla^2 u = g\,. \tag{4.1}$$

Here, ∇^2 is the Laplace operator, reading $\sum_i^n \partial_{x_i}^2$ in Cartesian coordinates. Of course, this equation requires boundary conditions at the boundaries of Ω, which can be split into Dirichlet boundaries, $u = u_\mathrm{D}$ on Γ_D, and Neumann boundaries, $\nabla u \cdot \vec{n} = j_\mathrm{N}$ on Γ_N with outward normal $\vec{n}$, so that $\partial\Omega = \Gamma = \Gamma_\mathrm{D} \cup \Gamma_\mathrm{N}$.

The key point in the *finite element method* is to cast the strong formulation (4.1) to a weak formulation. This is achieved as follows: Let v be an arbitrary function defined on Ω. Upon multiplication of (4.1) by v, followed by a spatial integration over the domain Ω leads to:

$$-\int_\Omega \nabla^2 u\, v\, \mathrm{d}^n x = \int_\Omega g\, v\, \mathrm{d}^n x\,. \tag{4.2}$$

Now, we treat the term on the lhs by integration by parts to arrive at

$$\int_\Omega \nabla u \cdot \nabla v\, \mathrm{d}^n x - \int_\Gamma \nabla u \cdot \vec{n}\, v\, \mathrm{d}S = \int_\Omega g\, v\, \mathrm{d}^n x\,. \tag{4.3}$$

When we now demand that $v = 0$ on the Dirichlet boundaries Γ_D (which is a necessary restriction for Dirichlet boundary conditions), the Neumann boundary condition $\nabla u \cdot \vec{n} = j_\mathrm{N}$ can be inserted directly in the surface integral:

$$\int_\Omega \nabla u \cdot \nabla v\, \mathrm{d}^n x - \int_{\Gamma_\mathrm{N}} j_\mathrm{N}\, v\, \mathrm{d}S = \int_\Omega g\, v\, \mathrm{d}^n x\,. \tag{4.4}$$

Finally, upon introducing the shorthand integral notations

$$\begin{aligned} (a,b) &= \int_\Omega a\, b\, \mathrm{d}^n x & \left(\vec{a},\vec{b}\right) &= \int_\Omega \vec{a}\cdot\vec{b}\, \mathrm{d}^n x \\ \langle a,b\rangle &= \int_{\Gamma_\mathrm{N}} a\, b\, \mathrm{d}S & \langle \vec{a},\vec{b}\rangle &= \int_{\Gamma_\mathrm{N}} \vec{a}\cdot\vec{b}\, \mathrm{d}S \end{aligned} \tag{4.5}$$

and putting all in residual form, i.e. putting all terms on one side, we arrive at

$$(\nabla u, \nabla v) - (g, v) - \langle j_\mathrm{N}, v\rangle = 0\,. \tag{4.6}$$

It can be shown that the weak formulation (4.6) together with the Dirichlet conditions $u|_{\Gamma_\mathrm{D}} = u_\mathrm{D}$ is equivalent to the strong formulation (4.1) together with both types of boundary conditions, as long as v may be chosen arbitrary (with the requirement to have $v = 0$ on Γ_D).

However, the weak formulation has several features that make it appealing: The order of the spatial derivatives has been reduced from second to first order and the implementation of the Neumann conditions is now just an interface integral. Of course, it also comes at the price that spatial integrals have to be carried out to obtain a solution and that the arbitrary test function v is appearing in the weak formulation. However, the combination of integrals and test functions provides a neat way of solving the equations numerically on versatile geometries and in all kinds of dimensions.

4.1.2 One-dimensional Poisson equation with Dirichlet boundary conditions

Let us first consider the Poisson equation on a simple interval, i.e. on the domain $\Omega = [-1, 1]$. Furthermore, we initially restrict to impose Dirichlet boundary conditions $u = 0$ at $x = -1$ and $x = 1$. The source function is simply $g = 1$. If there are no Neumann conditions, the weak formulation (4.6) reduces to

$$(\nabla u, \nabla v) - (g, v) = 0\,.$$

This weak form is implemented in a `Equations` class as follows:

```python
from pyoomph import *
from pyoomph.expressions import *  # Import grad & weak


class PoissonEquation(Equations):
    def __init__(self, *, name="u", space="C2", source=0):
        super(PoissonEquation, self).__init__()
        self.name = name  # store the variable name
        self.space = space  # the finite element space
        self.source = source  # and the source function g

    def define_fields(self):
        self.define_scalar_field(self.name, self.space)  # define the unknown scalar␣
↪field

    def define_residuals(self):
        u, v = var_and_test(self.name)  # get the unknown field and the corresponding␣
↪test function
        # weak formulation in residual form: (grad(u),grad(v))-(g,v)
        residual = weak(grad(u), grad(v)) - weak(self.source, v)
        self.add_residual(residual)  # add it to the residual
```

We define a new class `PoissonEquation`, inherited from the `Equations` class. In the constructor, we allow to pass the source function g, a name of the unknown field u (defaults to `"u"`) and a finite element space, which defaults to `"C2"`. More on finite element spaces will be discussed later on in Section 4.2. The passed arguments are stored in the object.

In the `define_fields()` method, we use `define_scalar_field()` to create a field with the desired name `self.name` on the finite element space `self.space`. Finally, in the method `define_residuals()`, the residual form is defined. To that end, we first bind the local variables `u` and `v` to the unknown field u and the corresponding test function v. The shorthand notations $(\ldots,\ldots)$ for the spatial integrals (cf. (4.5)) are written in python via the function `weak()`. The gradients of both the unknown field u and the test function v are obtained `grad()`, i.e. by `grad(u)` and `grad(v)`, respectively. The weak form in residual formulation, stored in the local variable `residual` is eventually added by the method `add_residual()`, which completes the definition of the weak form of the `PoissonEquation`.

To actually use this equation, we again have to write a problem class, inherited from the generic `Problem` class:

```
class PoissonProblem(Problem):
    def define_problem(self):
        mesh = LineMesh(minimum=-1, size=2, N=100)  # Line mesh from [-1:1] with 100␣
↪elements
        # Add the mesh (default name is "domain" with boundaries "left" and "right")
        self.add_mesh(mesh)

        # Assemble the system
        equations = PoissonEquation(source=1)  # create a Poisson equation with␣
↪source g=1
        equations += DirichletBC(u=0) @ "left"  # Dirichlet condition u=0 on the left␣
↪boundary
        equations += DirichletBC(u=0) @ "right"  # and u=0 on the right boundary
        equations += TextFileOutput()  # Add a simple text file output
        self.add_equations(equations @ "domain")  # Add the equation system on the␣
↪domain named "domain"
```

Again, the work is done in the `define_problem()` method. First of all, we need to define the geometry where the equation should be solved. Geometries in pyoomph are always defined via mesh templates. A mesh template provides spatially discretized geometric domains with named boundaries. The simplest mesh template is the `LineMesh`, which is just an interval subdivided into N elements. To create the desired domain $\Omega = [-1, 1]$, we pass the keyword arguments `minimum=-1` and `size=2` and divide it into `N=100` elements. This mesh template is added to the problem with the `add_mesh()` method. A mesh template has named domains and boundaries. The default names for the `LineMesh` are `"domain"` for the domain, i.e. here $[-1, 1]$, `"left"` for the left boundary, i.e. $x = -1$ and `"right"` for the right boundary, here $x = 1$.

Then, the equation system is assembled. We create the previously implemented equation class `PoissonEquation`, setting the `source` function g to 1. We then add `DirichletBC` objects and use the `@` operator to restrict these Dirichlet conditions to the boundaries `"left"` and `"right`. Also a `TextFileOutput` is added to the equation system to provide output as a simple text file. Finally, the entire system stored in `Equations` is added to the problem via `add_equations()`. Note that we have to restrict the equations once more to the desired domain `"domain"` which is provided by the previously added mesh template, i.e. the added `LineMesh`.

Finally, we just have to create the problem, solve it and write the output via

```
if __name__ == "__main__":
    with PoissonProblem() as problem:
        problem.solve()  # Solve the problem
        problem.output()  # Write output
```

Opposed to the ODEs in the previous chapter, there is no need for a temporal integration, i.e. the `run()` method of the problem is not required, but instead we use `solve()`. We also have to manually call the `output()` method to write the output to file, which is plotted in Fig. 4.1.

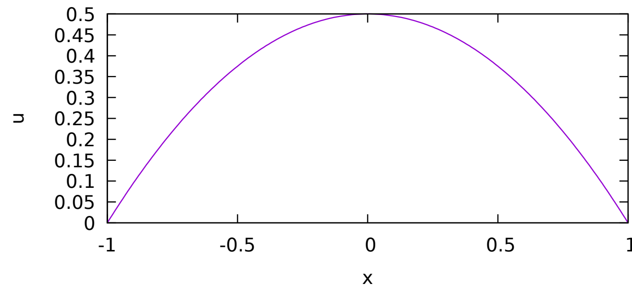

Fig. 4.1: One-dimensional Poisson equation with Dirichlet boundaries.

4.1.3 Coupled one-dimensional Poisson equations with Dirichlet boundary conditions

The power of defining the equations in classes easily let you combine multiple equations. Let us now solve the following system

$$-\nabla^2 u = w$$
$$-\nabla^2 w = -10u$$

subject to $u(-1) = u(1) = 0$ and $w(-1) = -w(1) = 1$. The code just create two instances of the `PoissonEquation` from the previous file `poisson.py` with different names, combines both and couple both equations via the source terms:

```
from pyoomph import *
from poisson import PoissonEquation  # Load the Poisson equation for the previous␣
↪class


class CoupledPoissonProblem(Problem):
    def define_problem(self):
        mesh = LineMesh(minimum=-1, size=2, N=100)
        self.add_mesh(mesh)

        u, w = var(["u", "w"])  # Bind the variables to use them mutually as sources
        # Create two instances of Poisson equations with different names and coupled␣
↪sources
        equations = PoissonEquation(name="u", source=w) + PoissonEquation(name="w",␣
↪source=-10 * u)
        equations += DirichletBC(u=0, w=1) @ "left"  # Dirichlet conditions u=0, w=1␣
↪on the left boundary
        equations += DirichletBC(u=0, w=-1) @ "right"  # and u=0, w=-1 on the right␣
↪boundary
        equations += TextFileOutput()
        self.add_equations(equations @ "domain")


if __name__ == "__main__":
    with CoupledPoissonProblem() as problem:
        problem.solve()  # Solve the problem
        problem.output()  # Write output
```

Also note how `DirichletBC` takes multiple keyword arguments to set multiple boundary values.

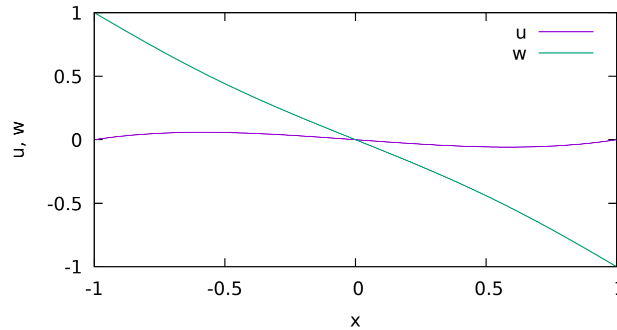

Fig. 4.2: Coupled Poisson equations with Dirchlet boundaries.

4.1.4 Poisson equation with a Neumann boundary condition

The weak formulation (4.6) has a boundary integral term stemming from the Neumann boundary conditions, which we have not accounted for yet. One cannot add this into the `PoissonEquation` class from the previous code `poisson.py` directly, since the `weak()`-terms added to the are always integrated over the entire domain, where the equation is defined on, i.e. `"domain"` ($=\Omega=[-1,1]$) in the previous two examples. Furthermore, to keep the final assembly of the equation system in the `define_problem()` method as flexible as possible, it is better to create a specific class to account for the surface integral term in (4.6):

```
from pyoomph import *
from poisson import PoissonEquation,weak  # Load the Poisson equation for the␣
↪previous class


# We create a new class, which adds the boundary integration term -<j,v>
class PoissonNeumannCondition(InterfaceEquations):
    # Makes sure that we can only use it as boundaries for a Poisson equation
    required_parent_type = PoissonEquation

    def __init__(self,name,flux):
        super(PoissonNeumannCondition, self).__init__()
        self.name=name # store the variable name and the flux
        self.flux=flux

    def define_residuals(self):
        u,v=var_and_test(self.name) # Get the test function by the name
        self.add_residual(-weak(self.flux,v)) # and add it to the residual
        # weak(j,v) is now <j,v>, i.e. a boundary integral

class NeumannPoissonProblem(Problem):
    def define_problem(self):
        mesh = LineMesh(minimum=-1, size=2, N=100)
        self.add_mesh(mesh)

        # Alternative way to assemble the system by restricting directly
        # Poisson equation and output on bulk domain
        equations = (PoissonEquation(source=1)+TextFileOutput()) @ "domain"
        equations += DirichletBC(u=0) @ "domain/left" # Dirichlet BC on domain/left
        equations += PoissonNeumannCondition("u",-1.5) @ "domain/right" # Neumann BC␣
↪on domain/right
        self.add_equations(equations)
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    with NeumannPoissonProblem() as problem:
        problem.solve()  # Solve the problem
        problem.output()  # Write output
```

Besides loading again the `PoissonEquation` from the previous code, we define a new class `PoissonNeumannCondition` inherited from the `InterfaceEquations` type. `InterfaceEquations` are a subclass of the `Equations` class, but provides some additional features which are only relevant at interfaces, i.e. at boundaries. For instance by adding `required_parent_type=PoissonEquation`, we can make sure that we are only allowed to attach this boundary condition to domains, where the `PoissonEquation` is solved. The constructor takes the name of the variable from the bulk as argument and the desired flux j to impose. We do not need to `define_fields()`, since the field has been already defined in the bulk and the `InterfaceEquations` can access these field and the corresponding test functions. Therefore, we can access the test function v of the unknown field u directly in the `define_residuals()` method. Again `weak()` is used to express the integral contribution $-\langle j_\mathrm{N}, v\rangle$. As before in the bulk contribution, it will be expanded to a spatial integral, however, since the `PoissonNeumannCondition` will be restricted to the boundary later on, the integral will be not carried out over the domain Ω, but just over the right boundary at $x = 1$. Of course, since Ω is just an interval, the boundary integral just comprises the point at $x = 1$, so that the integral will just evaluate to identity.

In the `define_problem()` method of the `Problem` class, there is another way shown how to restrict equations. Instead of restricting any boundary condition, stored in a local variable `bc` in the following, twice, i.e. `(bc @ "right") @ "domain"`, we can also restrict equivalently via a slash, i.e. by `bc @ "domain/right"` to apply it on the `"right"` boundary of the domain `"domain"`.

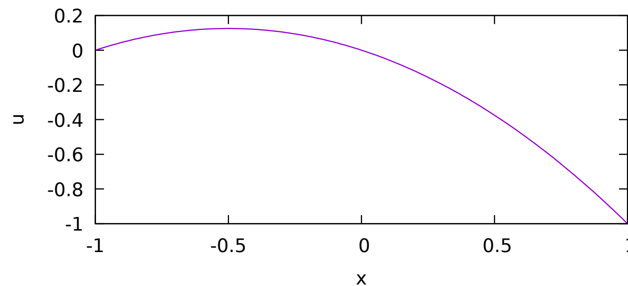

Fig. 4.3: One-dimensional Poisson equation with Dirichlet boundary at the left and Neumann boundary at the right.

It is not necessary to implement a Neumann condition for each equation by hand as done with the `PoissonNeumannCondition` class. Instead, pyoomph offers the class `NeumannBC`, which allows to add the weak contribution $\langle j_\mathrm{N}, v\rangle$ directly. However, one has to consider the sign: In the weak form of the Poisson equation (4.6), the $\langle \dots \rangle$ term has a negative sign, which is accounted for in the implementation of the `PoissonNeumannCondition`. The generic class `NeumannBC` uses a positive sign instead. One hence have to use `NeumannBC(u=1.5)` to get the same effect as `PoissonNeumannCondition("u",-1.5)`. Note that the keyword argument in `NeumannBC(u=1.5)` should be read as *impose the Neumann flux* 1.5 *for* u, **not** *set* $u = 1.5$, which would be Dirichlet-like.

Note that we have used one Dirichlet and one Neumann condition in the above example. Two Neumann conditions require special treatment, as discussed in the following example.

> **Warning:** If you do not add any boundary condition on a boundary, i.e. neither a `DirichletBC` nor any additional boundary integral term, there is no additional contribution to the residuals. This is equivalent to setting the Neumann flux to zero, i.e. a no-flux boundary condition.

4.1.5 Pure Neumann boundary conditions for the Poisson equation - Using a Lagrange multiplier to remove the nullspace

First of all, in order to have only Neumann conditions, the source term j of the Poisson equation and the imposed Neumann fluxes have to fulfill a relation, which can be seen by setting the test function $v = 1$ (possible, since test functions are allowed to be arbitrary) in (4.6), which gives

$$\int_\Omega g\,\mathrm{d}^n x = -\int_\Gamma j_\mathrm{N}\,\mathrm{d}S\,. \tag{4.7}$$

If this relation is not fulfilled, the equation in not well posed. If we would have a single Dirichlet condition, setting $v = 1$ is not allowed, since test functions have to vanish on the Dirichlet boundaries and this requirement is not necessary any more.

There is another complication to obtain the solution, namely the fact that the solution u is not unique: If u is a solution, $u + c$ for any constant c is a solution as well. Any Dirichlet condition would again specify a unique constant c for which this Dirichlet condition is fulfilled, which renders the solution unique again, but in absence of any Dirichlet condition, there is an infinite number of solutions.

Both problems can be tackled simultaneously by introducing a Lagrange multiplier λ, which fixes the spatial average of u, i.e. for some prescribed u_avg, we demand that the spatial average of u is u_avg, i.e.

$$\frac{\int_\Omega u\,\mathrm{d}^n x}{\int_\Omega 1\,\mathrm{d}^n x} = u_\mathrm{avg}$$

Given the fact that u_avg is a constant, we can write this as an implicit constraint

$$\int_\Omega \left(u - u_\mathrm{avg}\right)\,\mathrm{d}^n x = 0$$

As before in Section 3.7, this constraint can be enforced by the Lagrange multiplier λ by minimizing

$$\lambda \int_\Omega \left(u - u_\mathrm{avg}\right)\,\mathrm{d}^n x = \int_\Omega \lambda \left(u - u_\mathrm{avg}\right)\,\mathrm{d}^n x$$

with respect to u and λ, which gives by variation the additional weak contributions

$$(\lambda, v) + (u - u_\mathrm{avg}, \mu)$$

where μ is the test function corresponding to λ. Therefore, we need to augment our Poisson equation from `poisson_neumann.py` by adding these terms to the residual as well:

```
from pyoomph import *
from pyoomph.expressions import *
# Load the Poisson equation for the previous class
from poisson_neumann import PoissonEquation,PoissonNeumannCondition

# Create a new Poisson equation that fixes the average with a Lagrange multiplier
class PoissonEquationWithNullspaceRemoval(PoissonEquation):
    def __init__(self,lagrange_multiplier,average_value,*,name="u",space="C2",
↪source=0):
        # Initialize as before
        super(PoissonEquationWithNullspaceRemoval, self).__init__(name=name,
↪source=source,space=space)
        # And store the lagrange multiplier reference
        self.lagrange_multiplier=lagrange_multiplier
        self.average_value=average_value # and the desired average value
```

(continues on next page)

(continued from previous page)

```
    def define_residuals(self):
        # Add the normal Poisson residuals
        super(PoissonEquationWithNullspaceRemoval, self).define_residuals()

        # Add the contributions from the Lagrange multiplier
        l,ltest=self.lagrange_multiplier,testfunction(self.lagrange_multiplier)
        u,utest=var_and_test(self.name)
        self.add_residual(weak(u-self.average_value,ltest)+weak(l,utest))
```

Obviously, we load the old class `PoissonEquation` and inherit from it. In the `define_residuals()`, we first use a `super` call to call `define_residuals()` of the non-augmented `PoissonEquation` to add the normal residuals to the weak form. Then, the missing terms are added. However, the Lagrange multiplier λ is not defined yet and it cannot be defined in the `PoissonEquationWithNullspaceRemoval` class, since the latter will be restricted on the domain Ω. Any field definitions in the `define_fields()` method would hence add fields on Ω, but λ is just a simple real number, not a field.

Therefore, λ will be introduced by an own class. Since λ has no spatial dependence, it can be best done by using the `ODEEquations` class (see chapter Section 3), which allows the definition of real valued variables:

```
# A single "ODE", which is used as storage for the Lagrange multiplier value
class LagrangeMultiplierForPoisson(ODEEquations):
    def __init__(self,name):
        super(LagrangeMultiplierForPoisson, self).__init__()
        self.name=name

    def define_fields(self):
        self.define_ode_variable(self.name)
```

Note that we do not add any residuals for λ inside this class. The single purpose of this class is to provide the storage of a real-valued variable λ, whereas the contributions to the residuals are entirely done in the weak form of the augmented Poisson equation.

Finally, we need to couple both parts in the definition of the problem:

```
class PureNeumannPoissonProblem(Problem):
    def define_problem(self):
        mesh = LineMesh(minimum=-1, size=2, N=100)
        self.add_mesh(mesh)

        # Create the Lagrange multiplier (just a single value)
        lagrange = LagrangeMultiplierForPoisson("lambda")
        lagrange += ODEFileOutput() # Output it to file as well
        self.add_equations(lagrange@"lambda_space") # And add it to a space called
↪"lambda_space"

        l=var("lambda",domain="lambda_space") # Bind it. Important to pass the domain␣
↪name here!
        # and pass it to the augmented Poisson equation
        equations = PoissonEquationWithNullspaceRemoval(l,10,source=0)
        equations += TextFileOutput() # output
        # And the Neumann conditions
        equations += PoissonNeumannCondition("u",-1) @ "left"
        equations += PoissonNeumannCondition("u",1) @ "right"
        self.add_equations(equations@"domain")


if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```
with PureNeumannPoissonProblem() as problem:
    problem.solve()  # Solve the problem
    problem.output()  # Write output
```

In the `define_problem()` method, it is noteworthy that the `LagrangeMultiplierForPoisson` object is stored in another domain (named `"lambda_space"`) than the Poisson equation. This is necessary, since the domain `"domain"` is already bound to the interval $\Omega=[-1,1]$ of the line mesh. By the `var()` statement, we bind the just allocated variable λ to the local variable `l` and pass it together with the desired average value $u_\text{avg}=10$ to the `PoissonEquationWithNullspaceRemoval` object. Thereby, both parts, the augmented Poisson equation on Ω and the separately defined Lagrange multiplier λ are coupled. The remainder is as before, but now two `PoissonNeumannCondition` objects are created.

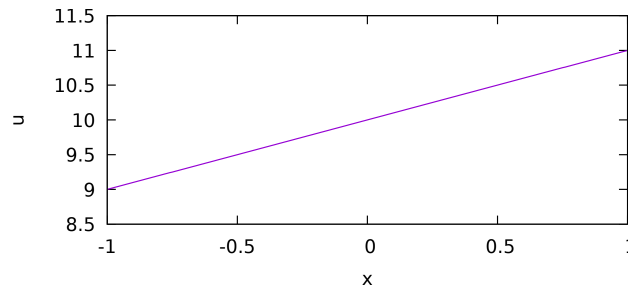

Fig. 4.4: Poisson equation with pure Neumann conditions, i.e. Neumann boundary conditions at the left and right. We have to enforce an average value (here 10) to make the solution unique.

The output (plotted in Fig. 4.4) indeed shows that the average of u is 10 and that the value of the Lagrange multiplier $\lambda=0$. If one instead violates the condition (4.7) by imposing an ill-posed combination of the Neumann fluxes and the source g, the Lagrange multiplier will attain a non-zero value, which can be seen as follows: Let us first write down the full augmented residual form:

$$(\nabla u,\nabla v)-(g,v)+(\lambda,v)+(u-u_\text{avg},\mu)-\langle j_\text{N},v\rangle=0\,.$$

Upon selection $v=1$ and $\mu=0$ as well as $v=0$ and $\mu=1$, we arrive at

$$\int_\Omega (g-\lambda)\,\mathrm{d}^n x=-\int_\Gamma j_\text{N}\,\mathrm{d}S$$
$$\int_\Omega (u-u_\text{avg})\,\mathrm{d}^n x=0$$

Obviously, the previous constraint (4.7) can now be fulfilled by solving for the correct value λ, which introduces a new source function $g^*=g-\lambda$, so that this constraint is fulfilled. The test space of λ on the other hand is used to set the average of u to u_avg.

4.1.6 Robin boundary conditions

Robin boundary conditions are sort of a weighted average of a Dirichlet and Neumann condition, i.e. one imposes

$$\alpha u+\beta\nabla u\cdot\vec{n}=g$$

for some nonzero coefficients α and β. The easiest way to implement these conditions is to rewrite this definition to

$$j=\nabla u\cdot\vec{n}=\frac{1}{\beta}\left(g-\alpha u\right)$$

and just add this term in the same manner as the Neumann condition, i.e. via $\langle j,v\rangle$:

```
from pyoomph import *
from poisson_neumann import PoissonEquation, PoissonNeumannCondition


# The Robin condition is just inherited from the Neumann condition
class PoissonRobinCondition(PoissonNeumannCondition):
    def __init__(self,name,alpha,beta,g):
        u=var(name) # Get the variable itself
        flux=1/beta*(g-alpha*u) # Calculate the Neumann flux term to impose
        super(PoissonRobinCondition, self).__init__(name,flux) # and pass it to the␣
↪Neumann class


class RobinPoissonProblem(Problem):
    def define_problem(self):
        mesh = LineMesh(minimum=-1, size=2, N=100)
        self.add_mesh(mesh)
        equations = PoissonEquation(source=1)
        equations+=TextFileOutput()
        equations += PoissonRobinCondition("u",1,0.5,1) @ "left"
        equations += PoissonRobinCondition("u",-1,2,-1) @ "right"
        self.add_equations(equations@ "domain")


if __name__ == "__main__":
    with RobinPoissonProblem() as problem:
        problem.solve()  # Solve the problem
        problem.output()  # Write output
```

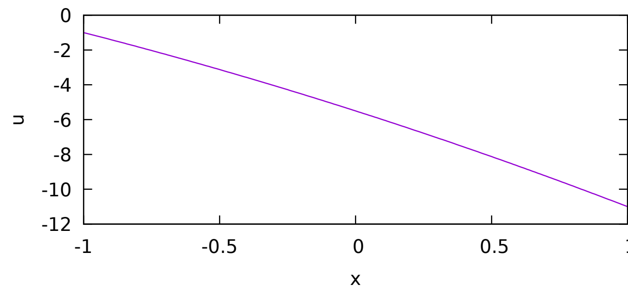

Fig. 4.5: Poisson equation with Robin boundary conditions.

Of course, you can recover the Neumann condition as special case by setting $\alpha = 0$, but you cannot recover the Dirichlet condition, since $\beta = 0$ will induce a division by zero.

To overcome this, one can enforce this particular boundary condition with arbitrary values of α and β, by introducing a Lagrange multiplier at the boundary to enforce the condition

$$\alpha u + \beta \nabla u \cdot \vec{n} - g = 0\,.$$

The same idea also works for other kinds of generalized boundary conditions, also non-linear ones. One just has to exchange the definition of the `PoissonRobinCondition` as follows:

```
# Inherit from the normal InterfaceEquations
class PoissonRobinCondition(InterfaceEquations):
    def __init__(self,name,alpha,beta,g):
        super(PoissonRobinCondition, self).__init__()
```

(continues on next page)

(continued from previous page)

```
        self.name=name # Store all passed values
        self.alpha=alpha
        self.beta=beta
        self.g=g

    def define_fields(self):
        # Define a Lagrange multiplier (field) at the interface with some unique name
        self.define_scalar_field("_lagr_robin_"+self.name,"C2")

    def define_residuals(self):
        l,ltest=var_and_test("_lagr_robin_"+self.name) # get the Lagrange multiplier
        u,utest=var_and_test(self.name) # the value of u on the interface
        # For the gradient grad(u), we need the function u inside the domain as well␣
↪to calculate the gradient there
        # This is done by changing the domain to the parent domain, i.e. the domain␣
↪where this InterfaceEquation is attached to
        ubulk,ubulk_test=var_and_test(self.name,domain=self.get_parent_domain())
        n=self.get_normal() # Normal to calculate dot(grad(u),n)
        condition=self.alpha*u+self.beta*dot(grad(ubulk),n)-self.g # The condition to␣
↪enforce
        self.add_residual(weak(condition,ltest)+weak(l,utest)) # Lagrange multiplier␣
↪pair to enforce it
```

The main idea is to create a Lagrange multiplier λ with test function μ on the interface and add the weak contributions

$$\langle \alpha u + \beta \nabla u \cdot \vec{n} - g, \mu \rangle + \langle \lambda, v \rangle \,.$$

Thereby, the value of u on the interface is adjusted until this condition holds.

Important: It is important to note that the term $\nabla u \cdot \vec{n}$ requires some extra caution in pyoomph. To calculate the bulk gradient, it is required to evaluate u also in the bulk. Therefore, it is required to obtain the bulk field u by using `var(...,domain=self.get_parent_domain())`. Without the specification of the `domain` via `get_parent_domain()`, one would obtain the value u on the interface and the calculation of the gradient will hence give the *surface gradient*, i.e. it would lead to wrong results here. Alternatively, one can use `domain=".."` instead of `domain=self.get_parent_domain()`.

The outward unit normal is obtained by `get_normal()` (or by `var("normal")`) and `dot()` represents the dot product of two vectors.

For the latter approach, there is also a generic class `EnforcedBC`, which allows to enforce arbitrary boundary conditions. To get the same result as with the custom implemented class `PoissonRobinCondition("u",alpha,beta,g)`, the generic class requires to cast it into residual form, i.e. `EnforcedBC(u=alpha*var("u")+beta*dot(grad(var("u",domain="..")), var("normal"))-g)`.

4.1.7 Cauchy boundary condition

A Cauchy boundary condition consist of enforcing a Dirichlet value u_D and a Neumann flux j_N simultaneously, i.e.

$$u|_\Gamma = u_\mathrm{D} \qquad \text{and} \qquad (\nabla u \cdot \vec{n})\,|_\Gamma = j_\mathrm{N}\,.$$

This kind of boundary condition is hard to impose in the finite element method. Enforcing the Dirichlet condition would require for the test function v to vanish on the boundary. Then, however, any Neumann-type boundary integral contributions $\langle j_\mathrm{N}, v\rangle$ cannot contribute, since v is 0. Additionally, for e.g. the 1d Poisson equation discussed here, one can impose in total two boundary conditions. If e.g. a Cauchy boundary condition is imposed on the left side, the right side cannot have any boundary conditions. However, omitting the specification of a boundary condition in finite elements automatically leads to a zero-flux Neumann boundary condition, as discussed earlier. This would be in total three boundary conditions, more than possible to impose. In principle, one can however enforce a Cauchy boundary condition by enforcing the value strongly, i.e. with a `DirichletBC`, add a custom `InterfaceEquations` class that monitors the slope and adjusts the opposite boundary condition accordingly, i.e. via a Lagrange multiplier that is defined in an `ODEEquations` helper class. Effectively, this would be a kind of a *shooting method*. However, this is not discussed here.

4.1.8 Two-dimensional Poisson equation

All previous discussions so far were exemplified on a 1d domain. Pyoomph makes it very simple to use equations on arbitrary domains. Since we have formulated the `PoissonEquation` and the Neumann boundary conditions in `poisson.py` and `poisson_robin_via_neumann.py` with `grad()`, the definition is not restricted to any particular number of the dimensions. To solve it on a 2d rectangular domain, we can hence directly reuse the equation classes and the boundary conditions defined above. To solve i.e. the system

$$\begin{aligned}
-\nabla^2 u(x,y) &= 100\exp\left(-100(x-0.5)^2 - 100(y-0.5)^2\right) \\
u &= 0 \quad \text{at} \quad x=0 \\
u &= 0 \quad \text{at} \quad x=1 \\
\partial_y u &= -2 \quad \text{at} \quad y=0 \\
u + \partial_y u &= -1 \quad \text{at} \quad y=1
\end{aligned}$$

We just have to assemble the system on a 2d geometry, which is predefinned in pyoomph in the `RectangularQuadMesh`:

```
# Import all PoissonEquation, Neumann and Robin condition as before
from poisson_robin_via_neumann import *
from pyoomph.expressions import * # Import vector


class PoissonProblem2d(Problem):
    def define_problem(self):
        # Create a 2d mesh, 10x10 elements, spanning from (0,0) to (1,1)
        mesh=RectangularQuadMesh(size=[1,1],lower_left=[0,0],N=10)
        self.add_mesh(mesh)

        # Assemble the system, bulk, then boundaries
        x=var("coordinate")-vector([0.5,0.5]) # position vector shifted by 0.5,0.5
        peak_source=100*exp(-100*dot(x,x))
        equations=PoissonEquation(source=peak_source)
        equations += DirichletBC(u=0) @ "left"
        equations += DirichletBC(u=0) @ "right"
        equations += PoissonNeumannCondition("u",2) @ "bottom"
        equations += PoissonRobinCondition("u", 1,1,-1) @ "top"
```

(continues on next page)

(continued from previous page)

```
        # Also add an output. This VTU output can be viewed with Paraview
        equations += MeshFileOutput()

        self.add_equations(equations@"domain")

if __name__=="__main__":
    with PoissonProblem2d() as problem:
        problem.solve()
        problem.output()
```

Obviously, the definition of the system in pyoomph is almost identical to the mathematical definition above. One only needs to know the default names of the `RectangularQuadMesh` class, which are `"domain"` for the inner domain and `"left"`, `"right"`, `"bottom"` and `"top"` for the boundaries. The source function now depends on the coordinate vector $\vec{x}$. This one can be accessed via `var("coordinate")`. Since it is a vector, one has to use e.g. `dot()` to calculate the square and subtract also the vectorial offset $(0.5, 0.5)$ via `vector([0.5,0.5])`. Elements of vectors can be accessed by e.g. `var("coordinate")[0]` and `var("coordinate")[1]`.

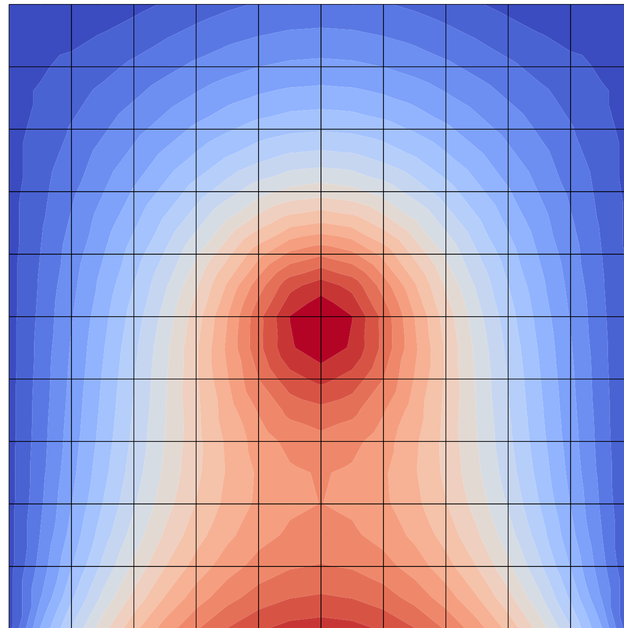

Fig. 4.6: Two-dimensional Poisson equation

4.1.9 Spatial adaptivity

The previous example has a very steep source function and due to the boundary conditions, also steep gradients in the solution u can be expected near the corners due to the different types of imposed boundary conditions. Pyoomph offers a simple way to automatically refine the mesh where necessary by adding a `SpatialErrorEstimator` object to the equations. We could either modify the previous example directly or use inheritance to re-use the previous example and just add the `SpatialErrorEstimator` object to the equation system. The latter approach reads:

```
# Load the previous code
from poisson_2d import *

# Inherit from the previous problem
```

(continues on next page)

(continued from previous page)

```
class AdaptivePoissonProblem2d(PoissonProblem2d):
    def define_problem(self):
        # define the previous problem
        super(AdaptivePoissonProblem2d, self).define_problem()

        # add a spatial error estimator for u (errors weighted by the coefficient 1.0)
        additional_equations = SpatialErrorEstimator(u=1.0)
        self.add_equations(additional_equations @ "domain")


if __name__=="__main__":
    with AdaptivePoissonProblem2d() as problem:
        # Maximum refinement level
        problem.max_refinement_level = 5
        # Refine elements with error larger than that
        problem.max_permitted_error = 0.0005
        # Unrefine elements with elements smaller than that
        problem.min_permitted_error = 0.00005
        # Solve with full refinement
        problem.solve(spatial_adapt=problem.max_refinement_level)
        problem.output()
```

By using the `super` call, the original non-adaptive problem is defined. Then, the `SpatialErrorEstimator` is applied on the field `u` with a weighting factor of `1.0`. If you have multiple fields, you can weight the estimated errors of the individual fields differently. Finally, the `solve()` call has to be augmented with a `spatial_adapt` keyword to specify the number of spatial adaption steps. However, the problem will never refine to a level finer than `max_refinement_level`. To check whether any element has to be refined, the estimated error is compared to the `max_permitted_error` value. If this threshold is exceeded, the element is refined. When any element got refined, the solution has to be recalculated and the next adaption step starts. If neighboring previously refined elements have an error lower than `min_permitted_error`, they also might be recombined to a coarser element within these adaption routine.

The result is depicted in Fig. 4.7. Obviously, the adaption is done near the corners and in the center, where the source function is prominent.

Tip: More details on how the error is estimated can be found e.g. in the oomph-lib documentation, e.g. here: https://oomph-lib.github.io/oomph-lib/doc/the_data_structure/html/classoomph_1_1Z2ErrorEstimator.html

4.1.10 Changing the coordinate system

By default, the coordinate system is just the normal Cartesian one. This holds true for 1d, 2d and 3d geometries. However, a lot of physical problems are in fact 3d, but have a rotational symmetry, i.e. can be considered to be axisymmetric so that axisymmetric cylindrical coordinates can effectively reduce the problem to a computationally favorable 2d problem. With a single line of code, i.e. by calling `set_coordinate_system()`, one can change the coordinate system from 2d Cartesian to axisymmetric cylindrical coordinates:

```
# Load the previous code
from poisson_2d_adaptive import *


if __name__=="__main__":
    with AdaptivePoissonProblem2d() as problem:
        # Change the coordinate system to axisymmetric
```

(continues on next page)

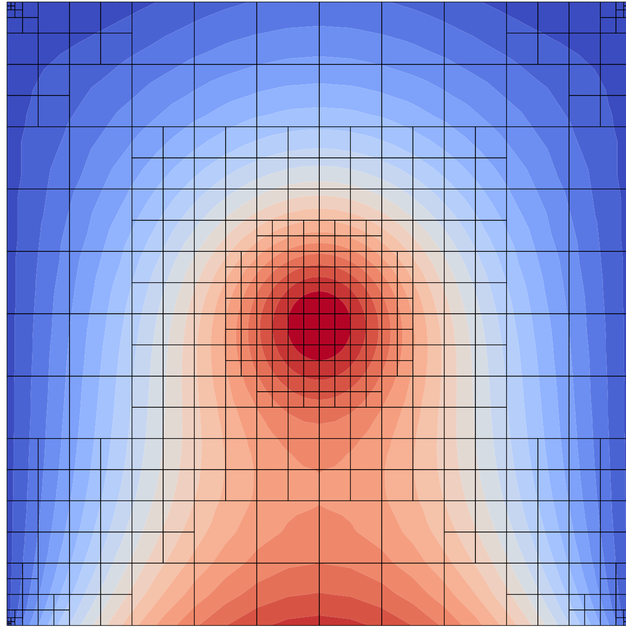

Fig. 4.7: Automatic spatial adaptivity based or error estimation in the Poisson problem.

(continued from previous page)

```
        problem.set_coordinate_system(axisymmetric)
        # The rest is the same
        problem.max_refinement_level = 5
        problem.max_permitted_error = 0.0005
        problem.min_permitted_error = 0.00005
        problem.solve(spatial_adapt=problem.max_refinement_level)
        problem.output()
```

Note that the radial coordinate r is given by the x-direction, which can be accessed in both cases by `var("coordinate_x")`, whereas the y-axis becomes the coordinate along the axis of symmetry (often called z in cylindrical coordinates). In pyoomph, this coordinate is hence accessible via `var("coordinate_y")`. The variable `var("coordinate")` will expand to the vector consisting of these coordinates.

When changing the coordinate system, pyoomph will intrinsically modify the weighting in the spatial integral terms in the weak form and will evaluate the spatial derivatives accordingly.

Besides the coordinate system `axisymmetric`, which works in 1d and 2d geometries, there is also the `radialsymmetric` coordinate system, which works in 1d geometries only. It can be used to simulate problems that have the full rotational symmetry of a sphere by just solving for the radial direction. In both cases, it is important to make sure that the geometry (i.e. the mesh) has only non-negative x-coordinates (i.e. radial coordinates). Furthermore, one cannot impose Neumann conditions at $x = 0$ (i.e. $r = 0$), since the spatial integrals will $\int \ldots \mathrm{d}S$ will expand to $2\pi \int \ldots r \mathrm{d}l$. Since $r = 0$, there cannot be any contribution from Neumann conditions here. However, the symmetry assumption usually requires to have a vanishing Neumann flux at $r = 0$ anyhow.

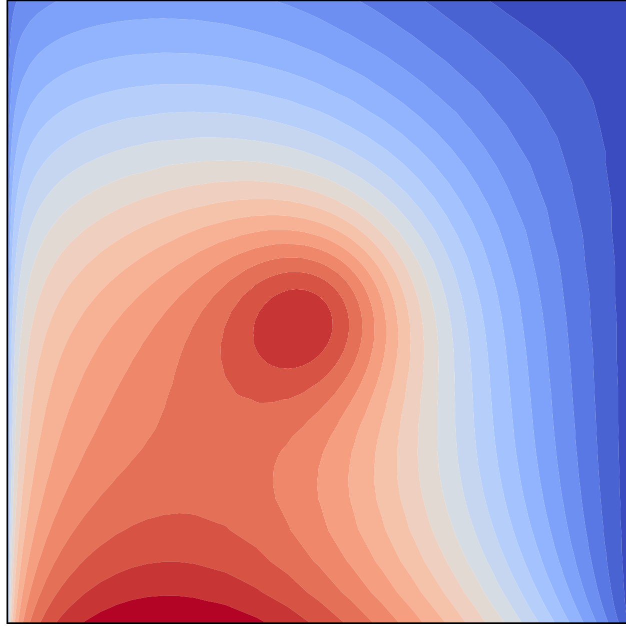

Fig. 4.8: Poisson equation with an axisymmetric coordinate system.

4.2 Mathematical details on the solution procedure

So far, we have discussed the weak form and the implementation in pyoomph. This section is devoted to provide some insight into the mathematical details of the solution procedure, i.e. what is going on under the hood in pyoomph. Pyoomph provides a high level python front-end based on oomph-lib, i.e. most of the details of assembling matrices and residuals are hidden to the user. If you want to go to the very bottom of the solution procedure, it is advised to get familiar with oomph-lib directly.

4.2.1 Continuous basis functions, spatial discretization and solution procedure

Until now, we always have allocated the unknown function u with `define_scalar_field("u","C2")`. While the first argument is just the name, the second argument needs more elaboration. It defines the *finite element space* where the function is defined upon. In pyoomph, there are two kinds on continuous spaces, namely `"C1"` and `"C2"`. Discontinuous spaces `"D0"` (constant per element) and `"DL"` (affine linear per element, cf. Section 4.4.8) are also available, but not discussed here. To understand these two continuous spaces, let us delve slightly into the basics of the *finite element method*. In fact, the unknown function u is spatially discretized by so-called *shape functions* $\psi_l(\vec{x})$.

$$u(\vec{x}) = \sum_l u_l \psi_l(\vec{x})\,. \tag{4.8}$$

This linear expansion in the shape functions obviously separates the spatial dependency of the function u into amplitudes u_l and spatially varying basis functions ψ_l. The second argument in `define_scalar_field()` selects the particular choice of these shape functions ψ_l, namely `"C1"` for linear basis functions and `"C2"` for basis functions of second order, i.e. quadratic ones.

In principle, the basis functions are quite arbitrary. In fact, one could select e.g. $\psi_l(\vec{x}) = \exp(i\vec{k}_l \cdot x)$ to obtain a Fourier decomposition. However, since the conventional finite element method is solved in the spatial domain, not in the spectral one, this choice of the basis functions is problematic. Instead, it is beneficial to choose them that the *support* only covers a few neighboring elements, i.e. they should be zero almost everywhere in the entire domain, except in the vicinity of a single position. Thereby, the degrees of freedom, i.e. u_l, are sufficiently localized in space. This idea leads to the

conclusion that the basis functions should be defined in a piece-wise manner - zero almost everywhere, but non-zero in the vicinity of a position $\vec{x}_l$ associated with the degree of freedom u_l.

The simplest idea on a one-dimensional domain, discretized by positions at $x_1 < x_2 < \ldots < x_n$ is hence to use linear slopes, i.e. the l-th basis function corresponding to the point x_l reads

$$\psi_l(x) = \begin{cases} 0 & \text{for} \quad x < x_{l-1} \\ (x - x_{l-1})/(x_l - x_{l-1}) & \text{for} \quad x_{l-1} \leq x < x_l \\ (x_{l+1} - x)/(x_{l+1} - x_l) & \text{for} \quad x_l < x \leq x_{l+1} \\ 0 & \text{for} \quad x \geq x_{l+1} \end{cases},$$

where the missing points x_0 and x_{n+1} are not required if x is confined to the range of the domain, i.e. between x_1 and x_l. The basis functions are shown for a 1d mesh in figure Fig. 4.9.

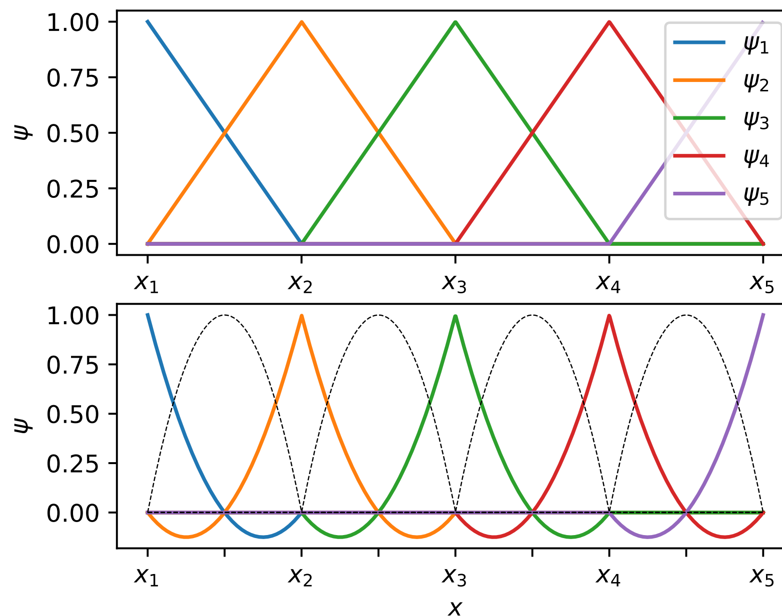

Fig. 4.9: Top: linear basis functions (`"C1"`). Bottom: quadratic basis functions (`"C2"`), where the dashed parabolas are the shape functions at the intermediate nodes in the center of each element.

These functions are in fact used if the space `"C1"` is used. Let us see how the weak form of the Poisson equation (4.6) in one spatial dimension reads with these basis functions:

$$\left(\nabla \sum_l u_l \psi_l, \nabla v\right) - (g, v) - \langle j_\mathrm{N}, v \rangle =$$
$$\left(\sum_l u_l \partial_x \psi_l, \partial_x v\right) - (g, v) - \langle j_\mathrm{N}, v \rangle = 0$$

The derivative of $u(x)$ is not required, just the derivative of the shape functions ψ_l, which can be calculated easily except on x_{l-1}, x_l and x_{l+1}, where the derivative is not defined due to the piece-wise nature of the chosen basis functions. However, these points are are *null set* with respect to the spatial integration $(.,\ .)$ so that it is not required to consider these points within the integration.

Finally, we have not yet addressed the test function v. As mentioned before, the above weak form has to hold for all (quite arbitrary) choices of v. In the discretization (4.8), we have used n unknows u_l (for $l = 1, \ldots, n$). So to get a discretized

system of equations, we should choose n linear independent test functions v_k (with $k = 1, \ldots, n$). Furthermore, we have to make sure that the *mass matrix* $\mathbf{M} = (M_{lk}) = (\phi_l, v_k)$ has a full rank. If the l-th row of this matrix is entirely zero, it means that we have selected our n test functions v_k in a manner that there is no support for the degree of freedom u_l. Thereby, we would not obtain a discretized equation for this degree of freedom.

The trivial choice of the test functions is the *Galerkin method*, where we just take the same basis functions, i.e. $v_k = \psi_k$. Thereby, both requirements on the test functions hold automatically. The Poisson equation hence reads

$$\left(\sum_l u_l \partial_x \psi_l, \partial_x \psi_k\right) - (g, \psi_k) - \langle j_\mathrm{N}, \psi_k \rangle = 0 \quad \text{for} \quad k = 1, \ldots, n\,.$$

Due to the reasonable choice of our basis functions (and hence test functions), the integrands are zero almost everywhere except for the neighborhood of the corresponding point x_k. Thus, the spatial integrals can be restricted to the support of each ψ_k:

$$\begin{aligned}\int_{x_{k-1}}^{x_{k+1}} \sum_l u_l (\partial_x \psi_l)(\partial_x \psi_k)\mathrm{d}x - \int_{x_{k-1}}^{x_{k+1}} g\psi_k \mathrm{d}x \\ + j_\mathrm{N}(x_1)\psi_1(x_1)\delta_{k,1} - j_\mathrm{N}(x_n)\psi_n(x_n)\delta_{k,n} = 0 \quad \text{for} \quad k = 1, \ldots, n\,.\end{aligned}$$

The Neumann flux terms appear only at the boundaries for $k = 1$ and $k = n$, which is indicated by the *Kronecker deltas*.

The next benefit of the choice of the basis functions is that also ψ_l are zero almost everywhere, i.e. the sum over l can be simplified depending on k. This eventually gives the system of equations

$$\begin{aligned} u_1(\partial_x\psi_1, \partial_x\psi_1) + u_2(\partial_x\psi_2, \partial_x\psi_1) &= (g, \psi_1) - j_\mathrm{N}(x_1)\psi_1(x_1) \quad \text{for} \quad k = 1 \\ u_{k-1}(\partial_x\psi_{k-1}, \partial_x\psi_k) + u_k(\partial_x\psi_k, \partial_x\psi_k) + u_{k+1}(\partial_x\psi_{k+1}, \partial_x\psi_k) &= (g, \psi_k) \quad \text{for} \quad 2 \le k \le n-1 \\ u_{n-1}(\partial_x\psi_{n-1}, \partial_x\psi_n) + u_n(\partial_x\psi_n, \partial_x\psi_n) &= (g, \psi_n) + j_\mathrm{N}(x_n)\psi_m(x_n) \quad \text{for} \quad k = n\,, \end{aligned}$$

where the integrals $(..)$ only have to be carried out over a small section of the entire domain where both arguments are non-zero. Denoting the symmetric *stiffness matrix* $\mathbf{K} = (K_{lk}) = (\partial_x\psi_l, \partial_k\psi_l)$, the vector of degrees of freedom $\vec{u} = (u_l)$ and the vector comprising the right hand side $\vec{b}$, one can rewrite this as matrix equation

$$\mathbf{K}\vec{u} = \vec{b}$$

In principle this equation can be solved by inverting $\mathbf{K}$, but due to the pure Neumann conditions and the shift-invariance $u_l \to u_l + \mathrm{const}$ of the Poisson equation, det($\mathbf{K}$) is actually 0. We have discussed how one can overcome this issue in Section 4.1.5.

Let us now see how Dirichlet boundary conditions are treated in discretized equations. When we impose a Dirichlet condition at the right side, i.e. we set $u(x_n) = a$, it means that the amplitude u_n of the expansion (4.8) is fixed by $u_n = a$. Hence, it is not an unknown anymore. Therefore, we just remove the n-th equation from the system (4.9). Removing this equation will automatically remove all connection to the Neumann flux at the right side, i.e. $j_\mathrm{N}(x_n)$. This reflects the fact that one can either impose a Dirichlet or a Neumann condition at the boundaries. Imposing both simultaneously, i.e. a Cauchy condition, is not feasible (cf. Section 4.1.7). However, when this equation is removed, $\mathbf{K}$ will become invertible, i.e. det($\mathbf{K}$) $\neq 0$. Furthermore, in the $(n-1)$-th equation, the occurrence of u_n can be replaced by a, which connects the entire system to the value of the Dirichlet condition. A unique solution is now feasible and depends on the value a.

This is exactly what happens internally in pyoomph when a one-dimensional Poisson equation on the space `"C1"` is solved. First of all, the mesh is constructed with a storage of u_l at each node located at x_l for $l = 1, \ldots, n$. Then, the Dirichlet boundary condition at x_n is applied setting the value of $u_n = a$ and removing it from the system. Finally, the spatial integrals over the shape functions are carried out numerically (using *Gauss quadrature*, cf. Section 13.1) and the resulting system is solved with a linear solver back-end. We will discuss this more generally in the next section.

At the end of this section, we still have to elaborate on the space `"C2"` and how the shape functions are defined in higher spatial dimensions. With the second order basis functions, i.e. `"C2"`, the approximated discretized solution is not piece-wise linear, but piece-wise quadratic. Since a parabola requires three points (x, u) to be uniquely defined, additional

points $x_{l+1/2}$ are introduced between each interval x_l and x_{l+1}, each of them associated with an own degree of freedom $u_{l+1/2}$. The basis functions ϕ_l and the test functions $v_k = \phi_k$ are now quadratic, but the rest of the approach is essentially the same, except that also the new degrees of freedom at $l+1/2$ have to be added to the discretized system. The quadratic basis functions of the space `"C2"` for a 1d mesh are also depicted in Fig. 4.9.

In higher dimensions, we limit ourselves to plot the basis functions, which are plotted for 2d elements in Fig. 4.10 and Fig. 4.11. In three dimensions, it is generalized analogously.

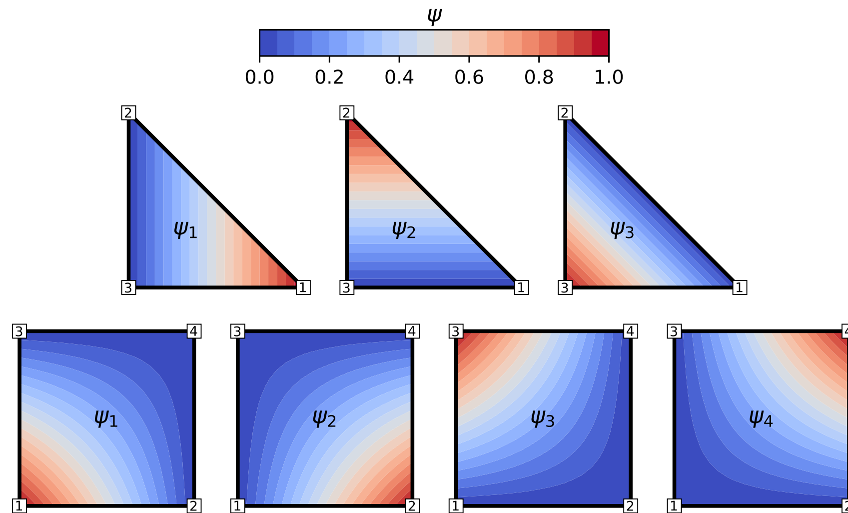

Fig. 4.10: First order shape functions (`"C1"`) on triangular and quadrilateral elements.

4.2.2 Internals: What is happening in pyoomph

When the problem is initialized - which either happens by an explicit call of `initialise()` or implicitly by the first call of e.g. `run()`, `solve()`, `output()`, etc., the `define_problem()` method is invoked. Depending on the present problem, meshes and equations are added to the problem.

The equations are assembled in an equation tree. This tree has the bulk domains as base nodes and each bulk domain can have further sub-nodes by equations defined on the boundary interfaces of this domain. If further equations are added to the boundaries of interfaces, the tree can go deeper. A text file containing the equation tree can be found relative to the output directory in `_ccode/_equation_tree.txt` . For each entry of the tree, all equations defined on this domain/interfaces are merged and a corresponding C code is generated. This C code performs the assembly of the discretized residual vector and its Jacobian on the basis of a single element of the domain mesh. It furthermore instructs the core of pyoomph to allocate all fields defined in the `define_fields()` methods of all equations defined on this particular domain. Furthermore, initial conditions and the values of the Dirichlet conditions are incorporated in the C code as well.

For the C code generation, all fields occurring in the weak expressions of the residuals added to this domain are expanded by sums over the shape functions of the selected basis functions. Likewise, the corresponding test functions are expanded to sums of the same basis functions. Spatial derivatives are carried out on the basis functions within these expansions (considering the selected coordinate system) and potential temporal derivative schemes are applied on the nodal values. During this expansion, also physical dimensions are plugged in (if used), i.e. the residual form is nondimensionalized based on the set scales.

Whenever the user is solving the system, for each element in each domain, the C code is called to assemble the residual (and the Jacobian) based on the very same steps as discussed in the previous section, but of course more general, i.e. allowing also for non-linear systems. Instead of the stiffness matrix $\mathbf{K}$, the Jacobian will enter the system (in case of the Poisson problem, both matrices are the same). The Jacobian $\mathbf{J}$ can be constructed from the residual vector (entries R_i)

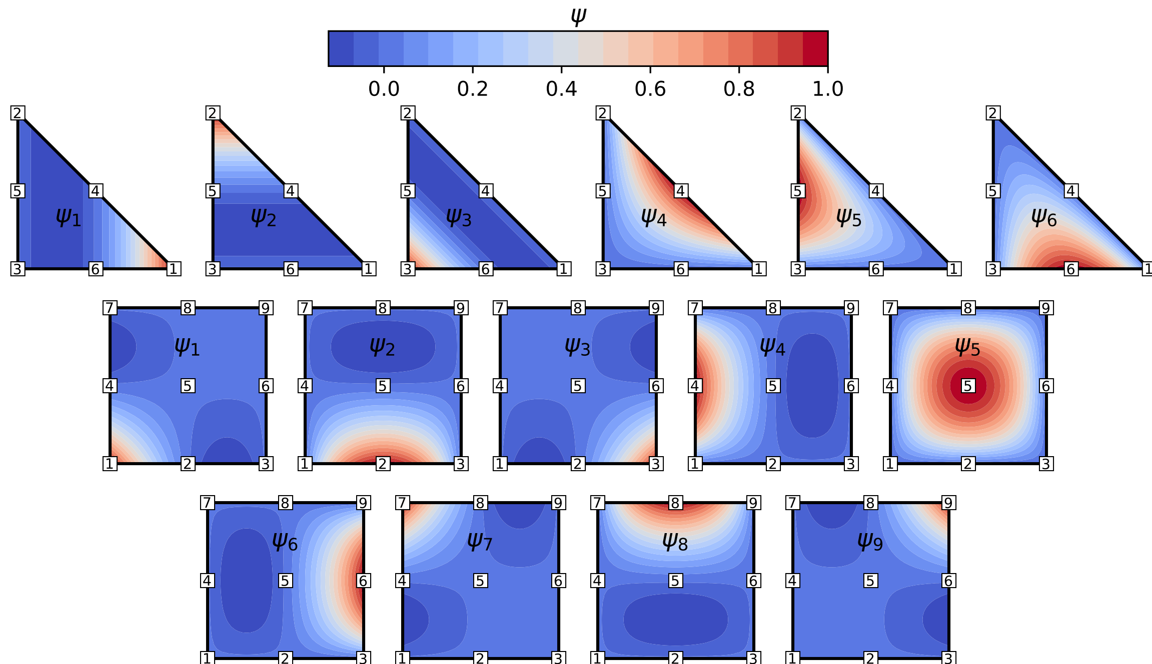

Fig. 4.11: Second order shape functions (`"C2"`) on triangular and quadrilateral elements.

by the entries $J_{ij} = \partial_{u_j} R_i$. For nonlinear problems, the assembly of the residual and the Jacobian must be repeated iteratively (Newton's method), since J_{ij} may change with the vector of unknowns $\vec{u}$. Therefore, $\vec{u}$ is updated according to

$$\mathbf{J}\,\delta\vec{u} = -\vec{R} \quad \text{and afterwards update} \quad \vec{u} \leftarrow \vec{u} + \delta\vec{u} \tag{4.9}$$

until the residual vector vanishes (i.e. its maximum entry by modulus falls below a specific threshold, controlled by the `newton_solver_tolerance` property of the `Problem` class).

To assemble the Jacobian and the residual vector, the contribution of each element is added to the global matrix/vector. To that end, each element has a map of the local degrees of freedom to the global equation numbers. The generated C code just gets passed the information of the values of each field stored at the nodes of the current element and the equation mapping from the pyoomph core and calculates the contribution to the Jacobian and residuals for a single element. The core iterates over all elements and accumulates the individual contributions of all elements, which usually overlap due to the shared nodes between different elements. This gives finally the global Jacobian and residual vector. The solution of the matrix equation (4.9) is done by a linear solver back-end, which can be selected by the `set_linear_solver()` method of the `Problem`.

4.3 Creating custom meshes

So far, we only used predefined mesh classes, namely the `LineMesh` and `RectangularQuadMesh`. In general, the problem to solve has a more intricate geometry, and the mesh has to be created by the user. This section will show how to create a custom mesh, either by specifying each element by hand or by using pyoomph's interface to the mesh generator Gmsh (gmsh.info).

4.3.1 Defining nodes and elements by hand

Until now, only trivial meshes, e.g. the 1d `LineMesh` or the 2d `RectangularQuadMesh` have been considered. However, in reality, these will rarely be sufficient for all kinds of problems. There are essentially two ways of creating meshes in pyoomph. The first way is to add all elements by hand. This can be a demanding task, but gives the full control over the desired mesh. This approach can also be used to write custom classes to load a mesh from a file in any format.

To do so, we write a mesh class that inherits from the `MeshTemplate` class. Let us create a mesh that resembles an L-shape:

```
from pyoomph import *
from pyoomph.equations.poisson import * # use the pre-defined Poisson equation


class LShapedMesh(MeshTemplate):
    def __init__(self,Nx=10,Ny=5,H=1,W=1,domain_name="domain"):
        super(LShapedMesh,self).__init__()
        self.Nx=Nx
        self.Ny=Ny
        self.H=H
        self.W=W
        self.domain_name=domain_name

    def define_geometry(self):
        domain=self.new_domain(self.domain_name)
        # row of quads in x direction
        for ix in range(self.Nx):
            x_l,x_u=self.W*ix,self.W*(ix+1) # lower and upper x coordinate
            y_l, y_u = 0, self.H # lower and upper y coordinate
            # add the corner nodes. These will be unique, i.e. no additional␣
↪node will be added if it is already present
            node_ll=self.add_node_unique(x_l,y_l)
            node_ul = self.add_node_unique(x_u, y_l)
            node_lu = self.add_node_unique(x_l, y_u)
            node_uu = self.add_node_unique(x_u, y_u)
            # add a quadrilateral element from (x_l,y_l) to (x_u,y_u)
            domain.add_quad_2d_C1(node_ll,node_ul,node_lu,node_uu)
            if ix==0: # Marking the left boundary:
                self.add_nodes_to_boundary("left",[node_ll,node_lu])

        # row of quads in y direction
        for iy in range(1,self.Ny): # we must start from 1, since the element in␣
↪the corner is already present
            x_l,x_u=self.W*(self.Nx-1),self.W*self.Nx # lower and upper x␣
↪coordinate
            y_l, y_u = self.H*iy, self.H*(iy+1) # lower and upper y␣
↪coordinate
            node_ll=self.add_node_unique(x_l,y_l)
            node_ul = self.add_node_unique(x_u, y_l)
            node_lu = self.add_node_unique(x_l, y_u)
            node_uu = self.add_node_unique(x_u, y_u)
            domain.add_quad_2d_C1(node_ll,node_ul,node_lu,node_uu)
            if iy == self.Ny-1: # Marking the top boundary:
                self.add_nodes_to_boundary("top",[node_lu, node_uu])
```

Again, we pass arguments to the constructor, e.g. the number of elements `Nx`×`Ny` in x and y direction. These are stored as properties of the class. The generation happens in the method `define_geometry()`. A `MeshTemplate` consists of *nodes* and *elements*. Nodes are part of the `MeshTemplate` itself, whereas elements are stored in domains. This will

allow to create multiple domains in the very same `MeshTemplate`, which is relevant for multi-domain problems in Section 7 later on. Therefore, before adding elements, we must create a domain to store these. This is done with the `new_domain()` method, which takes a name of this domain as argument. The name is arbitrary, but it is relevant to identify the domain. In particular, in the `Problem` class, we have to restrict the equations to the very same domain name, e.g. in the call `add_equations(eqs@"domain")`.

The, we perform a loop over the x-direction. We calculate the corner x and y coordinates of each quad. Then, we add four nodes with the `add_node_unique()` call. During the loop over `ix`, a lot of nodes will be created multiple times. If the node is already present, `add_node_unique()` will notice that and return the already previously added node instead of creating a new one. Thereby, adjacent elements will indeed share the very same mutual nodes. Finally in the `ix` loop, we add elements to the `domain`. Here, we add first order quadrilateral elements with the method `add_quad_2d_C1()`. It takes the four corner nodes as arguments, where it is necessary to add them in a zigzag order, i.e. starting with one corner (e.g. lower left), move to the next corner (e.g. lower right), then go diagonally (e.g. upper left) and finally the last corner (e.g. upper right). It is furthermore important to add the nodes in the order that the corner points in the argument order 1, 2, 4, 3 form a counter-clockwise loop.

We do the same in y-direction. However, opposed to nodes, it is not checked whether elements were already added. Therefore, it is important that the y-loop starts with `iy=1`, not `iy=0`, to prevent the dual generation of the element in the corner of the L-shape.

Lastly, we also can add boundary markers. This is done with the `add_nodes_to_boundary()` method, that takes first an interface name and then a list of nodes that should be added to this boundary. It is not required to add all nodes to one boundary in a single call of `add_nodes_to_boundary()`. You can also e.g. replace `add_nodes_to_boundary("left",[node_ll,node_lu])` by two calls `add_nodes_to_boundary("left",[node_ll])` and `add_nodes_to_boundary("left",[node_lu])`, which do exactly the same.

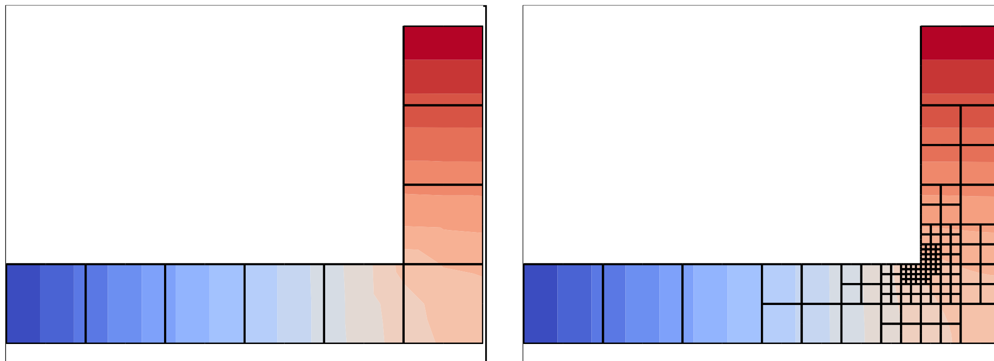

Fig. 4.12: Poisson equation on an L-shaped custom mesh without and with spatial adaptivity.

As an example how to use this mesh, we solve again a Poisson equation on this mesh, however, this time using the predefined `PoissonEquation` from the module `pyoomph.equations.poisson`:

```
class MeshTestProblem(Problem):
    def define_problem(self):
        self.add_mesh(LShapedMesh(Nx=6,Ny=4))
        eqs=MeshFileOutput()
        eqs+=PoissonEquation(name="u",source=0)
        eqs+=DirichletBC(u=0)@"left"
        eqs += DirichletBC(u=1) @ "top"
        eqs+=SpatialErrorEstimator(u=1)
        self.add_equations(eqs@"domain")

if __name__=="__main__":
    with MeshTestProblem() as problem:
```

(continues on next page)

(continued from previous page)

```
            problem.initial_adaption_steps=0
            problem.solve(spatial_adapt=4)
            problem.output_at_increased_time()
```

The predefined `PoissonEquation` works as the one developed in Section 4.1. Note how we initially suppress the mesh adaption by setting `initial_adaption_steps` to zero. Otherwise, we would get redundant adaption near the `"top"` boundary due to the non-zero `DirichletBC`. The custom mesh and the problem class in action can be seen in Fig. 4.12.

> **Warning:** The orientation of the elements can matter, in particular for refineable meshes. Therefore, it is advised to make sure that all elements are constructed by node indices in the same orientation. E.g. for a 2d mesh, the order of the nodes passed to `add_quad_2d_C1()` can either lead to an element facing in positive or negative z-direction. If elements of different orientation are connected in the very same mesh, this leads to issues upon spatial refinement. Therefore, make sure that all elements are oriented in the same direction by adjusting the order of the nodes passed to the construction of the elements. If it is wrong, pyoomph will raise an error, unless you set `check_mesh_integrity` of the `Problem` class to `False`. After doing so, you can easily check the mesh by *Paraview*. After outputting the mesh with `MeshFileOutput`, you can open it with Paraview and search for *Backface Representation* in the search box of the *Properties* box (hidden by default). Then, select *Cull Frontface* or *Cull Backface*. The entire mesh should be visible in one of this settings and entirely invisible in the other setting.

4.3.2 A helical line mesh & differential operators on manifolds

Until now the meshes have always be conforming in the number of dimensions, i.e. either one-dimensional meshes with one-dimensional elements or two-dimensional meshes with two-dimensional elements. However, you can also have a mesh with one-dimensional elements embedded in a two-dimensional or three-dimensional space. The same holds for a mesh consisting of two-dimensional elements embedded in a three-dimensional space. These meshes represent manifolds with a non-vanishing *co-dimension*. We will now create a line mesh that resembles a helical shape, i.e. has a co-dimension of 2:

```python
from pyoomph import *
from pyoomph.equations.poisson import *  # use the pre-defined Poisson equation


class HelicalLineMesh(MeshTemplate):
    def __init__(self, N=100, radius=1, length=5, windings=4, domain_name="helix"):
        super(HelicalLineMesh, self).__init__()
        self.N = N
        self.radius = radius
        self.length = length
        self.windings = windings
        self.domain_name = domain_name

    def define_geometry(self):
        domain = self.new_domain(self.domain_name, 3)  # Domain, but with 3d nodes

        # function to get the node based on a parameter l from [0:1]
        def node_at_parameter(l):
            x = self.radius * cos(2 * pi * self.windings * l)
            y = self.radius * sin(2 * pi * self.windings * l)
            z = self.length * l
            return self.add_node_unique(x, y, z)

```

(continues on next page)

(continued from previous page)

```
        # loop to generate the elements
        for i in range(self.N):
            n0 = node_at_parameter(i / self.N)  # constructing nodes
            n1 = node_at_parameter((i + 0.5) / self.N)
            n2 = node_at_parameter((i + 1) / self.N)
            domain.add_line_1d_C2(n0, n1, n2)  # add a second order line element
            if i == 0:  # Marking the start boundary:
                self.add_nodes_to_boundary("start", [n0])
            elif i == self.N - 1:  # Marking the end boundary:
                self.add_nodes_to_boundary("end", [n2])
```

Note how we name the domain by default `"helix"`, so that we must add equations to the domain `"helix"` in the `add_equations()` method of the problem class to restrict them on this helix. Furthermore, in the `new_domain()` calls, we add a second argument, `3`, which sets the nodal dimension space of this domain to 3. The rest works analogous to the previous example, however, this time we create second order line elements instead of first order quadrilateral elements by the call of `add_line_1d_C2()`. Due to the second order, we must supply a third node so that we in total have a start, a center and an end node of each line element. pyoomph automatically converts first order elements to second order elements, if equations on the space `"C2"` are defined on this domain. Vice versa, if we only have `"C1"` fields defined on a domain, pyoomph will simplify all generated second order elements to first order elements.

A potential driver code could read

```
class MeshTestProblem(Problem):
    def define_problem(self):
        self.add_mesh(HelicalLineMesh())
        eqs = MeshFileOutput()
        x, y, z = var(["coordinate_x", "coordinate_y", "coordinate_z"])
        source = x ** 2 + 5 * y * z + z
        eqs += PoissonEquation(name="u", source=source, space="C2")
        eqs += DirichletBC(u=0) @ "start"
        eqs += DirichletBC(u=0) @ "end"
        self.add_equations(eqs @ "helix")


if __name__ == "__main__":
    with MeshTestProblem() as problem:
        problem.solve(spatial_adapt=4)
        problem.output_at_increased_time()
```

At this stage, one might wonder, what the `PoissonEquation` actually does. It should solve

$$-\nabla^2 u = g$$

or likewise in weak formulation

$$(\nabla u, \nabla v) - (g, v) = 0\,.$$

However, what should $\nabla^2 u$ or ∇u mean here? If the helix is unfolded, it is just a one-dimensional function. We can parameterize u by a single parameter s, i.e. $u = u(s)$, where s could be e.g. the arc length along the helix. However, how should we apply a gradient or a Laplacian here? The conventional definition of $\nabla = (\partial_x, \partial_y, \partial_z)$ cannot work, since we cannot derive u in all these directions, but only along the helix, i.e. with respect to the parameterization s, i.e. $\partial_s u$.

Whenever pyoomph notices that we have a co-dimension, all spatial derivatives like `grad()` or `div()` will be internally expanded in a different manner. Instead of the conventional gradient or divergence, the corresponding differential operator reasonable for the actual manifold is selected. These are defined as follows: Let n be the dimension of the embedding space, i.e. the dimension of the nodal coordinates, and e be the element dimension. The co-dimension is obviously $k = n - e$. The manifold spanned by the elements can be parameterized (at least locally) by e parameters ξ^α for

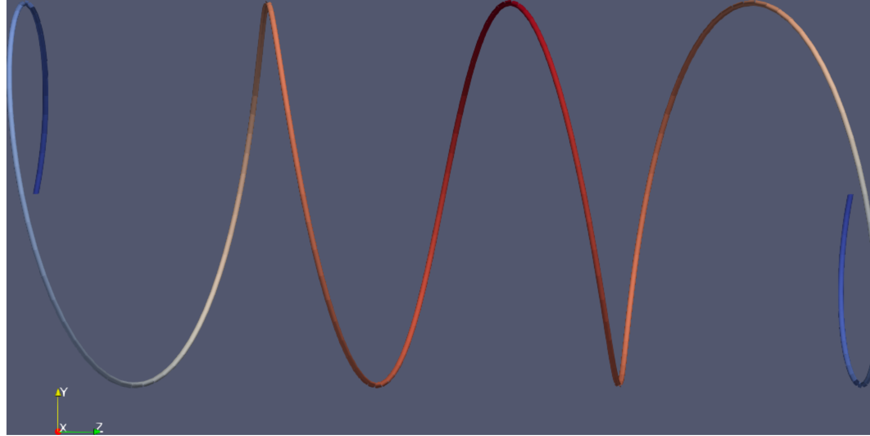

Fig. 4.13: Poisson equation solved on a helical manifold, where the differential operators have been implicitly replaced by their counterparts acting on manifolds.

$\alpha = 1, \dots, e$. The n-dimensional position vector hence reads $\vec{R}(\xi^\alpha)$. We construct the tangents, i.e. the *covariant basis vectors* by

$$\vec{g}_\alpha = \frac{\partial \vec{R}}{\partial \xi^\alpha}$$

And define the *covariant metric tensor* **g** by $g_{\alpha\beta} = \vec{g}_\alpha \cdot \vec{g}_\beta$. The inverse of **g**, the *contravariant metric tensor*, is denoted by the components $g^{\alpha\beta}$. With that, we define the scalar gradient of a field ϕ on the manifold as

$$\text{grad(phi)} = \nabla_S \phi = g^{\alpha\beta} \vec{g}_\alpha \frac{\partial \phi}{\partial \xi^\beta} \tag{4.10}$$

where we sum over all α and β. The result is an n-dimensional vector in the local tangent space of the manifold, pointing in the direction of the largest increase of ϕ along the surface. The choice of the particular parameterization of the manifold does not influence the result.

To illustrate that, let us consider the case of a 1d manifold embedded in a 3d space, as in our helix. Let ξ be our only parameter ξ^1. We get the covariant basis vector, which is just a non-normalized tangent to the helix, by

$$\vec{g} = \vec{g}_1 = \frac{\partial \vec{R}(\xi)}{\partial \xi}$$

is indeed the local normalized tangent on the helix. The metric tensor is just $\mathbf{g} = [g_{11}] = [\vec{g} \cdot \vec{g}] = [(\partial_\xi \vec{R})^2]$. Hence, the contravariant metric tensor is just given by the single component $g^{11} = 1/g_{11}$. With that, $\nabla_S \phi$ reads according to the definition (4.10)

$$\nabla_S \phi = \frac{1}{(\partial_\xi \vec{R})^2} \left(\partial_\xi \vec{R} \right) \frac{\partial \phi}{\partial \xi} \, .$$

When defining the normalized tangent to the helix as $\vec{t} = \partial_\xi \vec{R} / \|\partial_\xi \vec{R}\|$, we get

$$\nabla_S \phi = \left(\frac{1}{\|\partial_\xi \vec{R}\|} \frac{\partial \phi}{\partial \xi} \right) \cdot \vec{t} \, .$$

Upon reparameterization with the arc length $s = \xi \|\partial_\xi \vec{R}\|$, we arrive at

$$\text{grad(phi)} = \nabla_S \phi = \frac{\partial \phi}{\partial s} \cdot \vec{t} \, ,$$

which is indeed the slope of ϕ along the manifold pointing in tangential direction.

The divergence of a vector field $\vec{\psi}$ defined on a manifold reads similar to (4.10), namely

$$\text{div(psi)} = \nabla_S \cdot \vec{\psi} = g^{\alpha\beta} \vec{g}_\alpha \cdot \frac{\partial \vec{\psi}}{\partial \xi^\beta} \tag{4.11}$$

Finally, when changing the coordinate system by `set_coordinate_system()`, the corresponding scale factors are also considered in the differential operators on manifolds.

As conclusion, we can just use `grad()` and `div()` in our equations. When any equation involving these differential operators is restricted to a manifold, the only reasonable differential operator is selected automatically. This allows to use the same `PoissonEquation` either in the bulk (with co-dimension 0) or at any manifold with co-dimension > 0. In the latter case, the Poisson equation reads

$$-\nabla_S^2 u = g$$

with the *Laplace-Beltrami operator* ∇_S^2 or likewise in weak formulation

$$(\nabla_S u, \nabla_S v) - (g, v) = 0\,.$$

Warning: Spatial adaptivity does not work yet on domains with a co-dimension, but it will be implemented soon. Hence, we cannot use at `SpatialErrorEstimator` here yet.

Tip: oomph-lib also has a tutorial on the definition and calculation of surface gradients and surface divergences at https://oomph-lib.github.io/oomph-lib/doc/navier_stokes/surface_theory/html/index.html.

4.3.3 Mesh with metric dimensions a curved boundaries

Sometimes, we want to use physical dimensions, i.e. specify the size of the mesh in meters instead of `float` numbers. Furthermore, we also have frequently curved boundaries, that should remain resemble the very same smooth boundary curve also upon refinement. Both aspects will be handled in the following example mesh.

We will implement a fish mesh, which was inspired by the fish mesh example of oomph-lib. The mesh definition is analogous to the L-shaped mesh from Section 4.3.1:

```
from pyoomph import *
from pyoomph.equations.poisson import *  # use the pre-defined Poisson equation

from pyoomph.expressions.units import *

class FishMesh(MeshTemplate):
    def __init__(self,  size=1, mouth_angle=45*degree, fin_angle=50*degree,mouth_
↪depth_factor=0.5,fin_length_factor=0.45,fin_height_factor=0.8,domain_name="fish"):
        super(FishMesh, self).__init__()
        self.size = size # overall size of the fish (potentially dimesional)
        self.mouth_angle=mouth_angle # angle of the mouth-opening (with respect to␣
↪the body center)
        self.mouth_depth_factor=mouth_depth_factor # depth of the mouth
        self.fin_angle=fin_angle # angle of the fin-body-connection (with respect to␣
↪the body center)
        self.fin_length_factor=fin_length_factor
```

(continues on next page)

(continued from previous page)

```
        self.fin_height_factor = fin_height_factor
        self.domain_name = domain_name # name of the fish domain

    def define_geometry(self):
        domain = self.new_domain(self.domain_name)

        S=self.nondim_size(self.size) # Important: Nondimensionalize the potentially␣
↪dimensional size

        # Corner nodes of the fish
        n_mouth_center=self.add_node_unique(-(1-self.mouth_depth_factor)*S,0)
        n_upper_jaw = self.add_node_unique(-cos(self.mouth_angle / 2) * S, sin(self.
↪mouth_angle / 2)*S)
        n_lower_jaw=self.add_node_unique(-cos(self.mouth_angle/2)*S,-sin(self.mouth_
↪angle/2)*S)
        n_upper_body_fin=self.add_node_unique(cos(self.fin_angle/2)*S,sin(self.fin_
↪angle/2)*S)
        n_lower_body_fin = self.add_node_unique(cos(self.fin_angle / 2) * S, -
↪sin(self.fin_angle / 2) * S)
        n_center_body_fin = self.add_node_unique(cos(self.fin_angle / 2) * S, 0)
        n_upper_fin_corner=self.add_node_unique((cos(self.fin_angle / 2)+self.fin_
↪length_factor) * S, self.fin_height_factor * S)
        n_lower_fin_corner = self.add_node_unique((cos(self.fin_angle / 2) + self.fin_
↪length_factor) * S,-self.fin_height_factor * S)
        n_center_fin_end = self.add_node_unique((cos(self.fin_angle / 2) + self.fin_
↪length_factor) * S, 0)

        # Elements
        domain.add_quad_2d_C1(n_lower_jaw,n_lower_body_fin,n_mouth_center,n_center_
↪body_fin) # lower body part
        domain.add_quad_2d_C1(n_mouth_center, n_center_body_fin,n_upper_jaw, n_upper_
↪body_fin ) # upper body part
        domain.add_quad_2d_C1(n_lower_body_fin,n_lower_fin_corner,n_center_body_fin,n_
↪center_fin_end) # lower fin part
        domain.add_quad_2d_C1(n_center_body_fin,n_center_fin_end,n_upper_body_fin,n_
↪upper_fin_corner) # upper fin part

        # Curved entities
        upper_body_curve=self.create_curved_entity("circle_arc",n_upper_jaw,n_upper_
↪body_fin,center=[0,0])
        lower_body_curve = self.create_curved_entity("circle_arc", n_lower_jaw, n_
↪lower_body_fin, center=[0, 0])
        self.add_facet_to_curve_entity([n_upper_jaw,n_upper_body_fin],upper_body_
↪curve) # top body curve
        self.add_facet_to_curve_entity([n_lower_body_fin, n_lower_jaw], lower_body_
↪curve) # bottom body curve

        # Add nodes to boundaries
        self.add_nodes_to_boundary("curved",[n_upper_body_fin, n_lower_body_fin,n_
↪lower_jaw,n_upper_jaw]) # nodes on curved body parts
        self.add_nodes_to_boundary("mouth", [ n_lower_jaw,n_upper_jaw,n_mouth_
↪center])  # nodes of the mouth
        self.add_nodes_to_boundary("fin",[n_upper_body_fin,n_lower_body_fin,n_center_
↪fin_end,n_upper_fin_corner,n_lower_fin_corner]) # fin
```

The first new aspect is the call of `nondim_size()`, will calculate a corresponding non-dimensional size of an optionally dimensional argument. The dimensional argument will just be divided by `scale_factor("spatial")`, i.e. the

spatial scale set by set_scaling() in the Problem class. Every potentially metric argument passed to the mesh should be handled that way. Thereby, the mesh will be generated in the correct non-dimensional coordinates.

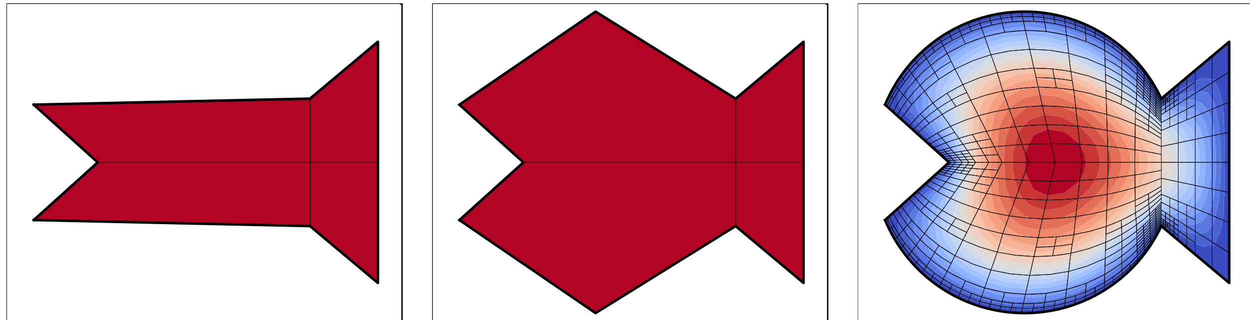

Fig. 4.14: (left) Fish mesh as initially defined. (middle) mesh after converting the elements to "C2" space: The additional nodes will be mapped on the circular boundaries. (right) Final adaptive solution of the Poisson equation on the fish mesh.

The definition of the corner node looks more complicated than it is. It are just the corners of the fish mesh, but the calculation of the coordinates from the parameters is a bit longish. The basic fish mesh without any adaption can be seen in Fig. 4.14. Also the elements are the same as before, but then we have to tell the FishMesh, that we have facets that are located on curved boundaries. To that end, we construct the curved boundaries by the calls of create_curved_entity(). The first argument "circle_arc" tells that we want to have a curved boundary in shape of a circle segment. Then we specify the start and end node and the center, which can either be a node nor, as here, a list of coordinates. We then still have to inform the FishMesh which facets shall be mapped onto this curve, since in principle there could be multiple facets sharing the same curved entity. This is done with the add_facet_to_curve_entity() call, one for each facet.

Finally, we assign the boundary names. We split it into different names to separate between the "curved" boundaries and the straight lines.

As a driver code, we use the following with a dimensional fish_size:

```
class MeshTestProblem(Problem):
    def __init__(self):
        super(MeshTestProblem, self).__init__()
        self.fish_size=1*meter # quite large fish, isn't it...?
        self.max_refinement_level = 5 # maximum level of refinements
        self.space="C2"

    def define_problem(self):
        self.add_mesh(FishMesh(size=self.fish_size))
        self.set_scaling(spatial=self.fish_size) # Nondimensionalize space by the␣
↪fish size

        eqs = MeshFileOutput()
        # We must set a meter^2 coefficient to be consistent with the units
        eqs += PoissonEquation(name="u", source=1, space=self.space,
↪coefficient=1*meter**2)

        # Boundaries all u=0
        eqs += DirichletBC(u=0)@"fin"
        eqs += DirichletBC(u=0) @ "mouth"
        eqs += DirichletBC(u=0) @ "curved"

        # refine the curved boundary to the highest order (i.e. max_refinement_level)␣
↪during adaptive solves
        eqs += RefineToLevel("max")@"curved"
```

(continues on next page)

(continued from previous page)

```
        eqs += SpatialErrorEstimator(u=1) # and adapt on all other elements based on␣
↪the error

        self.add_equations(eqs @ "fish")


if __name__ == "__main__":
    with MeshTestProblem() as problem:
        problem.solve(spatial_adapt=problem.max_refinement_level)
        problem.output_at_increased_time()
```

Since the `fish_size` is dimensional, we have to use `set_scaling()` to set a good spatial scale for non-dimensionalization of the coordinates. This also implies, that the coefficient of the Poisson equation has to be dimensional, since the `PoissonEquation` involves a ∇^2, which has to be compensated for by a `coefficient` with the unit m^2. The `coefficient` c enters the `PoissonEquation` as $-\nabla \cdot (c\nabla u) = g$.

The rest is trivial with the exception that we enforce the `"curved"` boundaries to be refined to maximum level. Thereby, the curvature is well resolved. The results are shown in Fig. 4.14.

We started with a rather simple mesh with just four elements and the final mesh is an accurate representation of the domain including all well resolved curved boundaries and refined singularities at sharp corners.

4.3.4 Generating meshes from points and lines via Gmsh

As we have seen on the basis of the fish mesh, one can create rather complicated domains already by hand. However, it can be quite cumbersome to add all elements by hand and map facets on curved boundaries, in particular if the geometry is rather complex. Once call intrinsically let the meshing tool gmsh do this job for you, which will be discussed in the following. To that end, we have to inherit our mesh from the `GmshTemplate` class.

The constructor looks - besides the chosen class name `GmshFishMesh` - the same:

```
from pyoomph import *
from pyoomph.equations.poisson import *  # use the pre-defined Poisson equation

from pyoomph.expressions.units import *

class GmshFishMesh(GmshTemplate):
    def __init__(self,  size=1, mouth_angle=45*degree, fin_angle=50*degree,mouth_
↪depth_factor=0.5,fin_length_factor=0.45,fin_height_factor=0.8,domain_name="fish"):
        super(GmshFishMesh, self).__init__()
        self.size = size # all as before
        self.mouth_angle=mouth_angle
        self.mouth_depth_factor=mouth_depth_factor
        self.fin_angle=fin_angle
        self.fin_length_factor=fin_length_factor
        self.fin_height_factor = fin_height_factor
        self.domain_name = domain_name
```

In the `define_geometry()` method, however, there are multiple changes:

```
def define_geometry(self):
    S=self.size # gmsh does not require to nondimensionalize the size, it will be␣
↪done automatically
    # Corner nodes of the fish: instead of "add_node_unique", we use "point".
    # We do not need p_center_body_fin and p_center_fin_end here
    p_mouth_center=self.point(-(1-self.mouth_depth_factor)*S,0)
```

(continues on next page)

(continued from previous page)

```
    p_upper_jaw = self.point(-cos(self.mouth_angle / 2) * S, sin(self.mouth_angle /
↪2)*S)
    p_lower_jaw=self.point(-cos(self.mouth_angle/2)*S,-sin(self.mouth_angle/2)*S)
    p_upper_body_fin=self.point(cos(self.fin_angle/2)*S,sin(self.fin_angle/2)*S)
    p_lower_body_fin = self.point(cos(self.fin_angle / 2) * S, -sin(self.fin_angle /
↪2) * S)
    p_upper_fin_corner=self.point((cos(self.fin_angle / 2)+self.fin_length_factor) *
↪S, self.fin_height_factor * S)
    p_lower_fin_corner = self.point((cos(self.fin_angle / 2) + self.fin_length_
↪factor) * S,-self.fin_height_factor * S)

    # Instead of starting with the elements, we start with the outlines
    # Create lines from lower jaw, to mouth center and to upper jaw, all named "mouth"
    self.create_lines(p_lower_jaw,"mouth",p_mouth_center,"mouth",p_upper_jaw)
    # Create the fin, also here, just a chain of straight lines, all named "fin"
    self.create_lines(p_lower_body_fin,"fin",p_lower_fin_corner,"fin",p_upper_fin_
↪corner,"fin",p_upper_body_fin)
    # Create the body curves
    self.circle_arc(p_lower_jaw,p_lower_body_fin,center=[0,0],name="curved")
    self.circle_arc(p_upper_jaw,p_upper_body_fin,center=[0,0],name="curved")

    # Now, generate the surface, i.e. the domain
    self.plane_surface("mouth","fin","curved",name=self.domain_name)
```

First of all, we do not need to call `nondim_size()` to get nondimensional coordinates. In fact, the `GmshTemplate` expects dimensional coordinates if a spatial dimension is set via `set_scaling(spatial=...)` in the problem class where the mesh is used. Next, instead of using `add_node_unique()`, we add points via `point()`. We then do not create any elements by hand at all. Also we do not create any domain via `new_domain()`. Instead, we just form the outlines, which can be done with the `create_lines()` method for a chain of straight lines. The arguments must be first a start point, then the name of the line, then the end point of the first line, which is also the start point of the next line, etc. For circular parts, we can use `circle_arc()` using start and end point as arguments and the `name` and the `center` as keyword arguments.

Finally, we can mesh the surface, i.e. the `"fish"` domain, with `plane_surface()`. Here, first the lines or the line names have to be passed as argument and the keyword `name` sets the name of the resulting domain.

The driver code is essentially the same as before:

```
class MeshTestProblem(Problem):
    def __init__(self):
        super(MeshTestProblem, self).__init__()
        self.fish_size=0.5*meter # quite large fish, isn't it...?
        self.resolution = 0.1 # Resolution of the mesh
        self.mesh_mode="quads" # Try to use quadrilateral elements
        self.space="C2"

    def define_problem(self):
        mesh=GmshFishMesh(size=self.fish_size)
        mesh.default_resolution=self.resolution
        mesh.mesh_mode=self.mesh_mode
        self.add_mesh(mesh)
        self.set_scaling(spatial=self.fish_size) # Nondimensionalize space by the
↪fish size

        eqs = MeshFileOutput()
        eqs += PoissonEquation(name="u", source=1, space=self.space,
↪coefficient=1*meter**2)
```

(continues on next page)

(continued from previous page)

```
        # Boundaries all u=0
        eqs += DirichletBC(u=0)@"fin"
        eqs += DirichletBC(u=0) @ "mouth"
        eqs += DirichletBC(u=0) @ "curved"

        self.add_equations(eqs @ "fish")


if __name__ == "__main__":
    with MeshTestProblem() as problem:
        problem.solve()
        problem.output_at_increased_time()
```

The differences are that we do not allow for spatial adaptivity and introduce new parameters `resolution` and `mesh_mode`, which will be passed to the mesh properties `default_resolution` and `mesh_mode`, respectively. Thereby, we can control the resolution of the mesh (the smaller, the finer) and also whether gmsh should try to create quadrilateral elements (`mesh_mode="quads"`) or should just create triangular elements (`mesh_mode="tris"`). However, there no guarantee that `mesh_mode="quads"` generate only quadrilateral elements. In particular at the sharp corners of the fin, gmsh will likely produce a triangle instead, leading to a mixed mesh. Some representative generated meshes are depicted in Fig. 4.15.

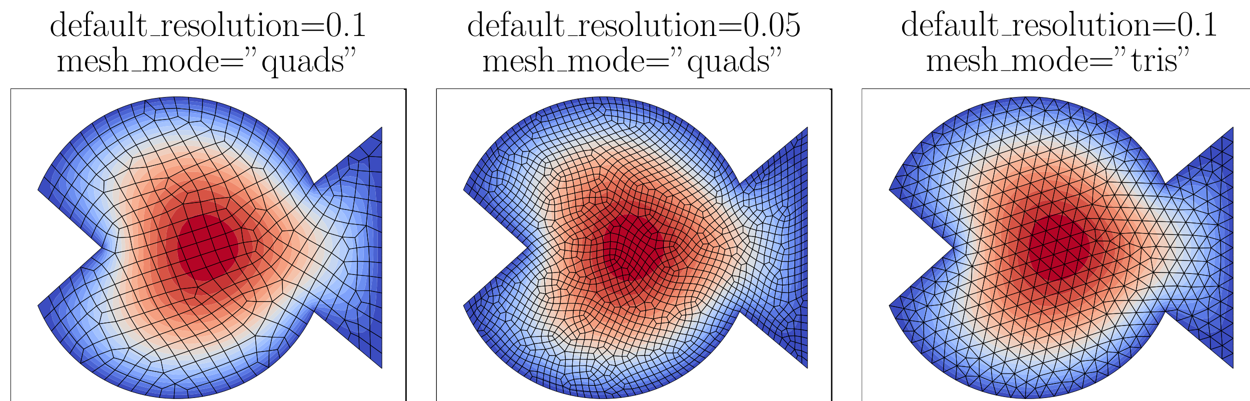

Fig. 4.15: Influence of `default_resolution` and `mesh_mode` on the meshes generated `GmshTemplate`.

> **Warning:** At the moment, spatial adaptivity does not work for triangular elements. The moment one triangular element is present in the mesh, spatial adaptivity is entirely deactivated. This will change in future to also also for adaptivity of triangular and mixed meshes. As a workaround, you can set the `mesh_mode` to `"only_quads"`. It will force gmsh to create only quadrilateral elements, but it will also lead to a less optimal mesh.

> **Warning:** Again, the orientation of the elements can matter, in particular for refineable meshes. *Gmsh* will select the element facing based on the order of the boundaries passed to `plane_surface()`. When using refineable meshes, make sure that all elements in a mesh are oriented in the same direction by adjusting the order of the boundaries passed here. If it is wrong, pyoomph will raise an error, unless you set `check_mesh_integrity` of the `Problem` class to `False`. After doing so, you can easily check the mesh by *Paraview*. After outputting the mesh with `MeshFileOutput`, you can open it with Paraview and search for *Backface Representation* in the search box of the *Properties* box (hidden by default). Then, select *Cull Frontface* or *Cull Backface*. The entire mesh should be visible in one of this settings and entirely invisible in the other setting. If not, permute the order of the boundaries

passed.

4.3.5 Splines, adding holes and locally controlling the mesh resolution

Actually, the previous mesh is not a fish - it does not have an eye! One could excuse this by arguing that it is a flatfish, viewed from the bottom, so that the migrated eye is not visible. However, then the mouth is wrong…

So, we must add an eye and we will introduce spline boundaries and how to control the local mesh size. To that end, only slight modifications are necessary. In the `define_geometry()`, we now add the eye points and boundary lines:

```
eye_size=0.125*S
eye_center_x,eye_center_y=-S*0.25,S*0.3
eye_resolution=self.default_resolution*0.2 # Fine mesh near the eye
p_eye_center=self.point(eye_center_x,eye_center_y,size=eye_resolution)
p_eye_north=self.point(eye_center_x,eye_center_y+eye_size,size=eye_resolution)
p_eye_west=self.point(eye_center_x-eye_size,eye_center_y,size=eye_resolution)
p_eye_east=self.point(eye_center_x+eye_size,eye_center_y,size=eye_resolution)
p_eye_south=self.point(eye_center_x,eye_center_y-0.25*eye_size,size=eye_resolution)
self.circle_arc(p_eye_west,p_eye_north,center=p_eye_center,name="eye")
self.circle_arc(p_eye_north,p_eye_east,center=p_eye_center,name="eye")
self.spline([p_eye_west,p_eye_south,p_eye_east],name="eye")
```

With the keyword argument `size` in the `point()` calls, the local mesh size (resolution) can be set. If it is not passed, it will default to `default_resolution` of the `GmshTemplate`. The upper part of the eye are just circular segments. We have split it into two segments, since the total angle of the circular arc is 180 °. With a single py:meth:~*pyoomph.meshes.gmsh.GmshTemplate.circle_arc* from `p_eye_west` to `p_eye_east`, it would be ambiguous, whether the arc should cover the north or the south direction. For that reason, any py:meth:~*pyoomph.meshes.gmsh.GmshTemplate.circle_arc* with an opening angle ≥ 180 ° must be slit into multiple segments. The bottom part of the eye is a `spline()`, which takes a list of points it has to pass.

We have to tell gmsh that the eye should be removed from the fish. It is actually a hole in the mesh, so we have to pass the keyword `holes` to the `plane_surface()`:

```
self.plane_surface("mouth","fin","curved",name=self.domain_name,holes=[["eye"]])
```

`holes` must take a list holes, which again are each lists of boundary lines or boundary line names.

In the problem class, a `NeumannBC` is added to the `"eye"` boundary

```
eqs += NeumannBC(u=1*meter) @ "eye"
```

so that the result looks as depicted in Fig. 4.16. Since our `PoissonEquation` has a `coefficient` in `meter**2`, the `NeumannBC` boundary condition has to be in units of `meter`.

4.4 The Stokes equations

The Stokes equations generalizes the simple example of the Poisson equation on various levels. First of all, the Stokes equations are a system of two partial differential equations, one for the velocity field and one for the pressure field. Secondly, the Stokes equations involves the velocity as vectorial unknown. Additionally, particular care has to be taken on the choice of the used finite element spaces for the velocity-pressure discretization.

Therefore, the Stokes equations constitute a perfect example to progress from the simple Poisson equation to more complex problems.

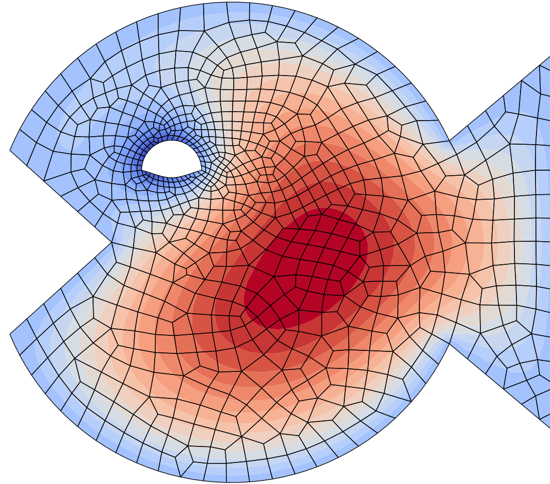

Fig. 4.16: Using a spline and local mesh size control, we add a hole to the fish mesh to create its eye.

4.4.1 Strong and weak formulation

The Stokes-equations for the velocity field $\vec{u}$ and the pressure p read

$$
\begin{aligned}
-\nabla p + \nabla \cdot \left[\mu \left(\nabla \vec{u} + (\nabla \vec{u})^t\right)\right] &= 0 \\
\nabla \cdot u &= 0
\end{aligned}
$$

Here, μ is the dynamic viscosity of the liquid. Obviously, we have to solve two field, one vector field for the velocity $\vec{u}$ and a scalar pressure field p. The second equation, the continuity equation, is a scalar equation and reads as a constraint to the velocity field $\vec{u}$, namely that its divergence vanishes. For an n-dimensional Stokes flow problem, there are hence n momentum equations and 1 constraining equation. To solve this, we have a n-dimensional vector field $\vec{u}$ and a single scalar p, which gives rise to the generic weak form of the Stokes equations by solving the vectorial momentum equation on the test spaces of the velocity and the continuity equation on the test space of the pressure. Let $\vec{v}$ and q be the corresponding test functions, the weak form thus reads

$$
\left(-p\mathbf{1} + \mu \left(\nabla \vec{u} + (\nabla \vec{u})^t\right), \nabla \vec{v}\right) + (\nabla \cdot \vec{u}, q) - \left\langle \vec{n} \cdot \left[-p\mathbf{1} + \mu \left(\nabla \vec{u} + (\nabla \vec{u})^t\right)\right], \vec{v}\right\rangle = 0 \tag{4.12}
$$

Note how the partial integration also works for tensor-valued equations by using the tensor contraction in the first term of the lhs. Furthermore, the momentum equation has been negated, since it is beneficial for the Navier-Stokes equation later on in Section 5.3.

> **Warning:** Due to the continuity equation, one can simplify the strain tensor term, i.e. do not consider the term $(\nabla \vec{u})^t$. However, this gives rise to a different, usually non-physical, natural Neumann boundary condition, i.e. the $\langle ., . \rangle$ term. If one wants to impose tractions, i.e. Neumann conditions, it is hence important to keep the $(\nabla \vec{u})^t$ in the weak formulation of the Stokes equations.

4.4.2 Implementation of the Stokes equations and the inf-sup condition

Let us now implement the aforementioned weak form of the Stokes equation. As usual, this requires only a few lines in pyoomph. However, there is some important requirement on the selection on the basis functions for the velocity and the pressure fields. To try out which choices of the discretizations work, we will allow the user to pass the spaces, e.g. `"C1"` or `"C2"`, for the velocity and the pressure.

```
from pyoomph import *
from pyoomph.expressions import *

class StokesEquations(Equations):
    # Passing the viscosity and the basis function space ("C1" or "C2") for velocity␣
↪and pressure
    def __init__(self,mu,uspace,pspace):
            super(StokesEquations, self).__init__()
            self.mu=mu # Store viscosity and the selected spaces
            self.uspace=uspace
            self.pspace=pspace

    def define_fields(self):
            self.define_vector_field("velocity",self.uspace) # define a vector field␣
↪called "velocity" on space uspace
            self.define_scalar_field("pressure",self.pspace) # and a scalar field
↪"pressure"

    def define_residuals(self):
            u,v=var_and_test("velocity") # get the fields and the corresponding test␣
↪functions
            p,q=var_and_test("pressure")
            # stress tensor, sym(A) applied on a matrix gives 1/2*(A+A^t), so this␣
↪is -p+mu*(grad(u)+grad(u)^t)
            stress=-p*identity_matrix()+2*self.mu*sym(grad(u))
            self.add_residual(weak(stress,grad(v)) + weak(div(u),q)) # weak form of␣
↪Stokes flow
```

We pass the viscosity as `mu` and the two spaces `uspace` and `pspace` to the constructor of the `StokesEquations` and store them internally. A vector field can be defined with `define_vector_field()` within the specific implementation of `define_fields()`. This will automatically create a one-dimensional vector field on a mesh with one spatial dimension and correspondingly two-dimensional and three-dimensional vector fields on two-dimensional and three-dimensional meshes, respectively.

As a test problem, we implement the typical parabolic *Poiseuille flow*. This is achieved by considering a rectangular domain and imposing zero velocity at the top and at the bottom, a parabolic inflow in x-direction at $x = 0$ and an open outflow with $u_y = 0$:

```
eqs=StokesEquations(self.mu,self.uspace,self.pspace) # Stokes equation using the␣
↪viscosity and the spaces
eqs+=MeshFileOutput() # Add output to write PVD/VTU files to be viewed in paraview

# Inflow: Parabolic u_x=y*(1-y), u_y=0
y=var("coordinate_y")
u_x_inflow=y*(1-y)
# Components of vector quantities can be accessed with the suffix "_x" or "_y" (or "_z
↪" in 3d)
eqs+=DirichletBC(velocity_x=u_x_inflow,velocity_y=0)@"left"

# Outflow, u_y=0, no Dirichlet on u_x, i.e. stress free outlet
```

(continues on next page)

(continued from previous page)

```
eqs+=DirichletBC(velocity_y=0)@"right"

# No slip conditions at top and bottom
eqs+=DirichletBC(velocity_x=0,velocity_y=0)@"bottom"
eqs+=DirichletBC(velocity_x=0,velocity_y=0)@"top"

# Adding this to the default domain name "domain" of the RectangularQuadMesh above
self.add_equations(eqs@"domain")
```

Note how the components of the vectorial field u can be accessed by `u_x` and `u_y` in two dimensions for setting e.g. Dirichlet boundary conditions. Furthermore, we take the viscosity and the discretization spaces for the velocity and the pressure as arguments for the problem constructor and pass them to the `StokesEquations`.

By that, we can now easily try out different combinations for the finite element spaces by constructing a corresponding problem and solve it:

```
if __name__ == "__main__":
    # Create a Stokes problem with viscosity 1, quadratic velocity basis functions␣
↪and linear pressure basis functions
    with StokesSpaceTestProblem(1.0,"C2","C1") as problem:
            problem.solve() # solve and output
            problem.output()
```

It is trivial to try out other pairs of spaces, but it turns out that for the choice of the continuous spaces `"C1"` and `"C2"`, the only combination that leads to convergence and reasonable solutions is `"C2"` for the velocity and `"C1"` for the pressure. These combination is called *Taylor-Hood element*. Mathematically, the convergence or divergence/oscillations can be proven and the corresponding theorem is the so-called *inf-sup criterion* or *Ladyzhenskaya-Babuška-Brezzi condition*. In simple words, we have to be aware of the fact that the pressure acts as a field of Lagrange multipliers that enforces the incompressibility, i.e. $\nabla \cdot \mathbf{u} = 0$. The pressure field adjusts hence that way and couples back to the momentum equation that this condition is fulfilled. However, if we choose equal spaces for both velocity and pressure, we are actually over-constraining the requirement of $\nabla \cdot \mathbf{u} = 0$, i.e. we enforce it at too many localizations so that a solution cannot be found or suffers from checkerboard-like oscillations.

Warning: Since the pressure acts as Lagrange multiplier field for the incompressibility, one should not be tempted to impose a Dirichlet boundary condition for the pressure, except for a single degree of freedom in a special case discussed later on in Section 4.4.5.

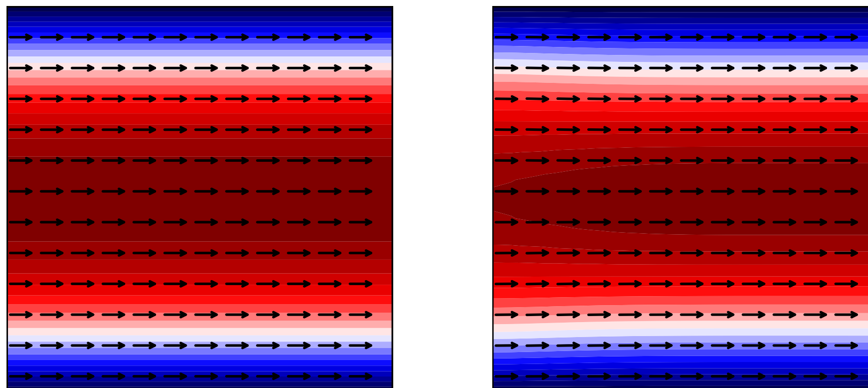

Fig. 4.17: Stokes flow example with a Newtonian fluid (left) and a shear-thickening fluid (right, see next page).

4.4.3 Non-Newtoninan fluids

We now want to consider a nonlinear Stokes equation by letting the viscosity depend on the shear rate. Thereby, the normally linear Stokes equation becomes nonlinear and highly dependent on the initial guess. However, shear-thickening or -thinning fluids can be found in reality. We can easily reuse our previous implementation of the Stokes problem by passing a shear dependent viscosity to the `Problem` class which forwards it to the `Equations` class, where it is ultimately evaluated in the weak form implementation. For simplicity, we set the viscosity to $\mu = 1 + \dot{\gamma}$, where $\dot{\gamma} = \sqrt{2\mathbf{S} : \mathbf{S}}$ with the shear tensor $\mathbf{S} = \frac{1}{2}\left(\nabla\vec{u} + \nabla\vec{u}^{\mathrm{t}}\right)$:

```
from pyoomph import *
from stokes import * # Import the previous Stokes problem

if __name__ == "__main__":

     # bind the velocity as variable
     u=var("velocity")
     # symmetrized shear
     S=sym(grad(u))
     # get the scalar shear rate by sqrt(2*S:S)
     shear_rate=square_root(2*contract(S,S))
     # Shear thickening fluid -> viscosity increases with increasing shear
     # we wrap it in a subexpression since the viscosity appears in the equation for␣
↪u_x and u_y
     # Thereby, the viscosity will be only evaluated once in the compiled code, which␣
↪speeds up the calculation
     mu=subexpression(1+shear_rate)

     # pass it to the Stokes problem, which will pass it to the equations
     with StokesSpaceTestProblem(mu,"C2","C1") as problem:

             # Since the problem is nonlinear, it is essential to provide a good␣
↪guess for the initial condition
             # Here, we use the one of the Poiseuille flow for Newtonian liquids
             y=var("coordinate_y")
             ux_init=y*(1-y)
             # We can add arbitrary equations/conditions to the problem by adding␣
↪them to additional_equations
             # Here, we set the initial condition to help the nonlinear problem to␣
↪converge
             problem.additional_equations+=InitialCondition(velocity_x=ux_init)@
↪"domain"

             problem.solve() # solve and output
             problem.output()
```

This example again shows the power of pyoomph: At any stage, we can define quite arbitrary expressions and pass them via the `Problem` class to the `Equations` class, where they are evaluated in the weak form. To get the strain rate, we first bind the variable field by `u=var("velocity")`, although it is not known here at all. The symmetrized strain tensor is again calculated via `sym()` and `grad()`, `S=sym(grad(u))`, and we get $\dot{\gamma}$ via applying `square_root()` on the contraction $\mathbf{S} : \mathbf{S}$ (using `contract()`). To speed up the code, the dynamic viscosity is again wrapped in a `subexpression()`.

Since the problem is now strongly nonlinear, it is essential to provide a reasonable initial guess. We could write our own `Problem` class, but we also can add further equations to a given `Problem` class by adding those to the property `additional_equations`. Of course, we have to restrict them to the `"domain"`, i.e. on the domain where the equation should be solved.

4.4.4 A case with pure Dirichlet boundary conditions

In Section 4.1.5 we have learned that the Poisson equation shows some caveats when considering pure Neumann conditions. We learned that the solution is in that case not unique due to a shift-invariance with respect to the addition of an arbitrary constant. A similar issue also occurs in the Stokes equation the velocities in normal direction are prescribed by Dirichlet boundary conditions. In that case, all potential Neumann contributions vanish and we are left with the weak form

$$\left(-p\mathbf{1}+\mu\left(\nabla\vec{u}+(\nabla\vec{u})^t\right),\nabla\vec{v}\right)+(\nabla\cdot\vec{u},q)=0$$

To illustrate the issue, let use revert the partial integration on the pressure contribution in the first term, where the arising Neumann flux is again not present due to the pure Dirichlet boundary conditions for the velocity:

$$(\nabla p,\vec{v})+\left(\mu\left(\nabla\vec{u}+(\nabla\vec{u})^t\right),\nabla\vec{v}\right)+(\nabla\cdot\vec{u},q)=0$$

Since only gradients of the pressure entering the equation, it is invariant with respect to $p\to p+\text{const}$, i.e. a unique solution for the pressure cannot determined. Any Neumann term would depend on the absolute value of the pressure and hence remove this ambiguity.

If one only has Dirichlet conditions, one hence should either impose a global Lagrange multiplier enforcing the average pressure, directly analogous to the procedure described in Section 4.1.5 for the Poisson equation with pure Neumann conditions. Alternatively, often easier, one can prescribe one single pressure degree. On a `RectangularQuadMesh`, one can e.g. fix the pressure in the lower left corner by adding a `DirichletBC(pressure=0)@"bottom/left"` to the domain. One has to make sure that only a single degree of freedom is specified, i.e. the domain where the `DirichletBC` is applied must be a single point.

4.4.5 Using physical dimensions and imposing a traction

If one wants to replicate an actual experimental result, one usually has to deal with dimensional quantities, i.e. with velocities given in $\mathrm{m/s}$, dynamic viscosities given in $\mathrm{Pa/s}$, pressures in Pa and so on. While it is often beneficial to nondimensionalize the problem by hand and identify the relevant nondimensional numbers (Reynolds, Rayleigh, Marangoni number, etc.), sometimes the problem is too complex to consider all these details. In particular for multi-component flow dynamics, each property (mass density, viscosity, diffusivity, surface tension) will change with the local composition - and usually in a nonlinear manner. Then, all these characteristic numbers are only limited in their meaning since it will vary a lot with the local composition.

For these cases, pyoomph allows for automatic nondimensionalization. In numerics, eventually everything will be nondimensional since we are dealing with floating point numbers internally. pyoomph allows to treat the nondimensionalization in the following way: Let us again start with the weak form of the Stokes equation (cf. (4.12)), but now with every quantity being dimensional, i.e.

$$\left(-p\mathbf{1}+\mu\left(\nabla\vec{u}+(\nabla\vec{u})^t\right),\nabla\vec{v}\right)+(\nabla\cdot\vec{u},q)-\left\langle-p+\mu\left(\nabla\vec{u}+(\nabla\vec{u})^t\right),\vec{v}\right\rangle=0$$

We introduce nondimensional variables $\tilde{\vec{u}}$, $\tilde{p}$ and the nondimensional spatial coordinate $\tilde{\vec{x}}$, which are connected by typical scale factors U, P and X:

$$\vec{u}=U\tilde{\vec{u}}\,,\quad p=P\tilde{p}\,,\quad \vec{x}=X\tilde{\vec{x}}$$

These leading to the very same equation, with nondimensional variables, but still a dimensional result

$$\left(-P\tilde{p}\mathbf{1}+\frac{\mu U}{X}\left(\tilde{\nabla}\tilde{\vec{u}}+(\tilde{\nabla}\tilde{\vec{u}})^t\right),\frac{1}{X}\tilde{\nabla}\vec{v}\right)+\left(\frac{U}{X}\tilde{\nabla}\cdot\tilde{\vec{u}},q\right)-\left\langle-P\tilde{p}+\frac{\mu U}{X}\left(\tilde{\nabla}\tilde{\vec{u}}+(\tilde{\nabla}\tilde{\vec{u}})^t\right),\vec{v}\right\rangle=0$$

However, in numerics, eventually no dimensions shall be present anymore, i.e. the entire lhs shall just be a numeric scalar. To do so, we also introduce scales for the test functions

$$\vec{v} = V\tilde{\vec{v}}, \quad q = Q\tilde{q}$$

which gives after collecting the scales in the first arguments of the weak contributions

$$\begin{aligned}\left(-\frac{PV}{X}\tilde{p}\mathbf{1} + \frac{\mu UV}{X^2}\left(\tilde{\nabla}\tilde{\vec{u}} + (\tilde{\nabla}\tilde{\vec{u}})^t\right), \tilde{\nabla}\tilde{\vec{v}}\right) + \left(\frac{UQ}{X}\tilde{\nabla}\cdot\tilde{\vec{u}}, \tilde{q}\right) - \\ \left\langle -\frac{PV}{X}\tilde{p} + \frac{\mu UV}{X^2}\left(\tilde{\nabla}\tilde{\vec{u}} + (\tilde{\nabla}\tilde{\vec{u}})^t\right), \tilde{\vec{v}}\right\rangle = 0\end{aligned} \tag{4.13}$$

Note that in the Neumann boundary term $\langle .,. \rangle$ an additional factor $1/X$ arises. This one stems from the fact that the boundary integral covers one spatial dimension less than the bulk integrals $(.,.)$. In pyoomph, these integrals are in fact nondimensional by default, i.e. $\int \ldots \mathrm{d}^n\tilde{x}$ instead of $\int \ldots \mathrm{d}^n x$. This comes with the benefit that the particular nondimensionalization is independent on the number of dimensions n.

A good choice for the pressure test scale Q in (4.13) is obviously $Q = X/U$, since it will then lead to a factor of unity in the incompressibilty constraint. For V we can choose differently, either $V = X/P$ or $V = X^2/(U\mu)$, leading to either unity as factor for the pressure term or for the shear term in the momentum equation. The better choice depends usually on the boundary conditions desired to be imposed. If the velocity is imposed, we know a typical scale for the velocity, i.e. U. Then, we can easily choose $V = X^2/(U\mu)$ and set the (typically unknown) pressure scale $P = \mu U/X$. Thereby, both factors will become unity. Similarly, if one imposes pressures (or better tractions), it might be beneficial to set $V = X/P$ with the typical imposed traction magnitude P and setting the a priori unknown resulting velocity scale to $U = PX/\mu$. Eventually, both can be equivalent, provided both scales U and P are selected accordingly.

Let us now see how to implement this in pyoomph. We have to modify our Stokes equations a bit to consider the process of nondimensionalization:

```
from pyoomph import *
from pyoomph.expressions import *
from pyoomph.expressions.units import * # Import units as e.g. meter, second, etc.


class StokesEquations(Equations):
    # Passing the viscosity
    def __init__(self,mu):
        super(StokesEquations, self).__init__()
        self.mu=mu # Store viscosity

    def define_fields(self):

        X=scale_factor("spatial") # spatial scale (will be supplied by the␣
↪Problem class)
        U=scale_factor("velocity")
        P=scale_factor("pressure")
        mu=self.mu

        #Taylor-Hood pair, with testscale we can set the definition of the test␣
↪scales V and Q
        self.define_vector_field("velocity","C2",testscale=X**2/(mu*U))
        self.define_scalar_field("pressure","C1",testscale=X/U)

    def define_residuals(self):
        # Fields and test functions are dimensional here!
        u,v=var_and_test("velocity")
        p,q=var_and_test("pressure")
        stress=-p*identity_matrix()+2*self.mu*sym(grad(u))
        self.add_residual(weak(stress,grad(v)) + weak(div(u),q))
```

Note that we have direcly used the Taylor-Hood combination ("C2","C1"). The only other difference is in the define_fields() method. Here, we pass testscale arguments, which depend on the scale_factor() of the space X, the velocity U and the pressure P. These are not known at this moment and will be supplied by the Problem class. If they are not supplied by the problem, they will default to unity. Thereby, one still can use this implementation of the Stokes equation for nondimensional calculations, provided that the passed viscosity mu is nondimensional as well.

The problem class will now use dimensional units:

```
class DimStokesProblem(Problem):
    def __init__(self):
        super(DimStokesProblem, self).__init__()
        # we are now using units for the viscosity
        self.mu=1*milli*pascal*second
        self.boxsize=1*milli*meter # the size of the box
        self.imposed_traction=1*pascal # and the imposed traction on the left


    def define_problem(self):
        # setting the spatial scale X by the boxsize and the pressure scale P by␣
↪the imposed traction
        self.set_scaling(spatial=self.boxsize,pressure=self.imposed_traction)
        # the velocity scale is now calculated based on these scales. scale_
↪factor will expand to P and X, respectively
        self.set_scaling(velocity=scale_factor("pressure")*scale_factor("spatial
↪")/self.mu)
        # alternatively, you can just set directly
        # self.set_scaling(velocity=self.imposed_traction*self.boxsize/self.mu)

        self.add_mesh(RectangularQuadMesh(size=self.boxsize)) # we have to tell␣
↪the mesh that it has a dimensional size now
        eqs=StokesEquations(self.mu) # passing the dimensional viscosity to the␣
↪Stokes equations
        eqs+=MeshFileOutput()

        # A traction is just the Neumann term
        eqs+=NeumannBC(velocity_x=-self.imposed_traction)@"left"
        # zero y velocity at left and right
        eqs+=DirichletBC(velocity_y=0)@"left"
        eqs+=DirichletBC(velocity_y=0)@"right"
        # No slip conditions at top and bottom
        eqs+=DirichletBC(velocity_x=0,velocity_y=0)@"bottom"
        eqs+=DirichletBC(velocity_x=0,velocity_y=0)@"top"


        # Adding this to the default domain name "domain" of the␣
↪RectangularQuadMesh above
        self.add_equations(eqs@"domain")


if __name__ == "__main__":
    # Create a Stokes problem with viscosity 1, quadratic velocity basis functions␣
↪and linear pressure basis functions
    with DimStokesProblem() as problem:
        problem.solve() # solve and output
        problem.output()
```

We use set_scaling() to set the corresponding scales for X (spatial), U and P (both identified by the name "velocity" and "pressure" we named the fields in the Stokes equation class). Furthermore, the RectangularQuadMesh now gets a dimensional size passed, which will be canceled out internally by the spatial scale. The

imposed traction is exactly the Neumann term in the Stokes equation for the momentum equation.

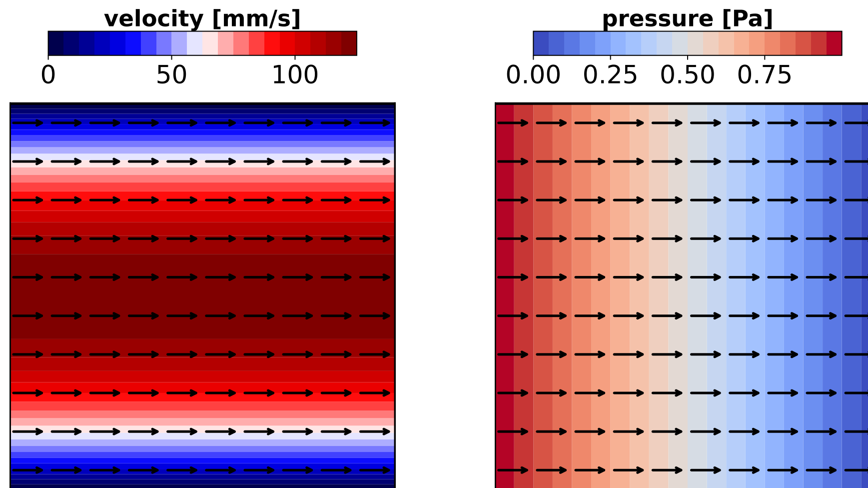

Fig. 4.18: Velocity and pressure field of the dimensional Stokes flow example.

Internally, pyoomph will now create the Stokes equations. All quantities bound by `var()` or `var_and_test()` will be treated as dimensional quantities and successively expanded into the scale and the nondimensional quantity, e.g. $p = P\tilde{p}$. In pyoomph, this means that `var("pressure")` will be replaced by `scale_factor("pressure")*nondim("pressure")`. The same applies for the test functions and also the spatial differential operators `grad()` and `div()` will be nondimensionalized by yielding $1/X$ (`1/scale_factor("spatial")`). When assembling the weak form, all units will cancel out and just numerical factors will survive, provided that all units and scales are selected correctly. If any unit survives this process, an error will be thrown. Thereby, one can easily identify whether the used units are in agreement. Since this happens before the C code generation, there is no additional overhead in the assembly of the system and hence in the calculation time when dimensional quantities are used.

The `MeshFileOutput` will write the result in dimensional units again, i.e. the velocity in $\mathrm{m/s}$, pressure in Pa and the spatial dimensions of the mesh in m.

4.4.6 Enforcing zero normal flow

Until now, the mesh was always aligned with the axes so that it was easy to just impose e.g. `DirichletBC(velocity_x=0)` to prevent any flow in the x-direction. However, on curved interfaces of a mesh, one sometimes want to enforce that there is no in- or outflow. This occurs e.g. when one tries to enforce a kinematic boundary condition $\vec{u}\cdot\vec{n}=0$ on a curved quasi-stationary free surface, e.g. of a droplet. Then, one cannot fix neither `velocity_x` nor `velocity_y`, but only the projection `dot(var("velocity"),var("normal"))` must be zero, which is one constraint for two separate degrees of freedom. In these cases, the typical approach is to use a field of Lagrange multipliers λ (with test function η) to enforce this constraint, one adds

$$\langle \vec{u}\cdot\vec{n},\, \eta\rangle + \langle \lambda,\, \vec{n}\cdot\vec{v}\rangle \tag{4.14}$$

to the interface residuals where the normal flow should vanish. This weak form can only be fulfilled if $\vec{u}\cdot\vec{n}=0$. λ is then the normal traction ($\approx$ pressure), required to push or pull the fluid so that the constraint is fulfilled.

We want to combine it with the dimensional equations from the previous section to illustrate how this can be done:

```
from stokes_dimensional import * # Import the dimensional Stokes equation from the previous section
from pyoomph.meshes.simplemeshes import CircularMesh # Import a curved mesh
```

(continues on next page)

(continued from previous page)

```
class StokesFlowZeroNormalFlux(InterfaceEquations):
    required_parent_type = StokesEquations # Must be attached to an interface of a␣
↪Stokes equation

    def define_fields(self):
            # Velocity space is C2, so we must create the Lagrange multipliers on␣
↪the same space
            # Note how we set the scale and the testscale here: In both cases, we␣
↪absorb the test scale or the scale of the velocity
            self.define_scalar_field("noflux_lambda","C2",scale=1/test_scale_factor(
↪"velocity"),testscale=1/scale_factor("velocity"))

    def define_residuals(self):
            # Binding variables
            l,ltest=var_and_test("noflux_lambda")
            u,utest=var_and_test("velocity")
            n=var("normal")
            # Add the residual: The scales will cancel out: u~U, ltest~1/U and l~1/V,
↪ utest~V
            self.add_residual(weak(dot(u,n),ltest)+weak(l,dot(utest,n)))

    # This will be called before the equations are numbered. This is the last chance␣
↪to apply any pinning (i.e. Dirichlet conditions)
    def before_assigning_equations_postorder(self, mesh):
            # If the velocity is entirely pinned at any node (e.g. no slip), we also␣
↪have to set the Lagrange multiplier to zero
            # This can be done with the helper function: we set noflux_lambda=0␣
↪whenever "velocity" (i.e. "velocity_x" & "velocity_y) are pinned
            self.pin_redundant_lagrange_multipliers(mesh, "noflux_lambda", "velocity
↪")
```

We introduce new interface equations `StokesFlowZeroNormalFlux` which must be attached to a domain having the `StokesEquations` in the bulk. These equations will introduce a new Lagrange multiplier field at the interface on space `"C2"`, i.e. the same space as the velocity is defined on. Previously, we have set the scaling on a problem level, using the method `set_scaling()` in the `define_problem()` method of the `Problem` class. However, it is complicated for the user to set also the scales (i.e. the dimensions) for the Lagrange multipliers by hand. Instead, we see immediately from (4.14) that the test function η should scale inverse to the velocity scale, i.e. $\eta = \tilde{\eta}/U$ to cancel out the dimensions in the first term. The Lagrange multiplier field λ in the second term of (4.14) must scale as $1/V$ i.e. the inverse of the velocity test function scale to kill all dimensional contributions. Thereby, the nondimensionalization is automatically done and the used just have to set the velocity scale on a problem level.

Furthermore, there is one caveat here: When the interface meets with another interface where e.g. a no-slip boundary condition $\vec{u} = 0$ is set, we run into troubles. In fact, the first term of (4.14) will be automatically zero, since $\vec{u} \cdot \vec{n} = 0$ holds due to the no-slip condition. Hence, we do not add any contribution to the test space of the Lagrange multiplier. Also, the second term will be problematic, since the Dirichlet condition for the velocity requires that the velocity test function $\vec{v}$ vanishes at this intersection of the two interfaces. Therefore, the entire residual will be zero irrespectively of the local value of λ here. This eventually leads to a degenerate matrix (i.e. having a zero row/column) and finding a unique solution becomes impossible.

One either can leave this caveat to the user, who has to make sure at the problem level that also $\lambda = 0$ is imposed at these particular interface intersections. A better way, which leads to less complications, is to give this responsibility to the `InterfaceEquations` class itself. The method `before_assigning_equations_postorder()` will be called whenever the degrees of freedom are about to be numbered internally. This is the last chance to pin individual degrees of freedom, i.e. setting $\lambda = 0$ here. Therefore, we call the helper function

pin_redundant_lagrange_multipliers(), which will check if indeed both degrees of freedom for the velocity are pinned. If so, we set the local value of $\lambda = 0$ and remove it from the list of unknowns. Note that it might not always work entirely automatically, namely in the case that we e.g. have only a Dirichlet condition for the velocity in x-direction, but not in y-direction. Since the y-velocity is still an unknown, the Lagrange multiplier λ will not be pinned to zero by pin_redundant_lagrange_multipliers(). However, if the normal $\vec{n}$ happens to have a vanishing y-component, the entire issue persists and is not resolved. Due to $n_y = 0$, $u_x = 0$ and $v_x = 0$, all terms in (4.14) are again zero and the system cannot be solved for a unique value of λ for this particular interface intersection. However, this rarely happens and in this case, the responsibility to treat for it is by the user.

Next, we also want to add a bulk force density $\vec{f}$ to the Stokes flow, so we write another bulk equation:

```
class StokesBulkForce(Equations):
    def __init__(self,force_density):
        super(StokesBulkForce, self).__init__()
        self.force_density=force_density

    def define_residuals(self):
        utest=testfunction("velocity")
        self.add_residual(-weak(self.force_density,utest))
```

We can just use this equation to add e.g. gravity or other bulk forces to the momentum equation. Both new equations are now used in the problem class:

```
class NoFluxStokesProblem(Problem):
    def __init__(self):
        super(NoFluxStokesProblem, self).__init__()
        self.mu=1*milli*pascal*second # dynamic viscosity
        self.radius=1*milli*meter # the radius of the circular mesh

    def define_problem(self):
        # Setting reasonable scales
        self.set_scaling(spatial=self.radius,velocity=1*milli*meter/second,
↪pressure=1*pascal)

        # Changing to an axisymmetric coordinate system
        self.set_coordinate_system("axisymmetric")

        # Taking the north east segment of a circle as mesh, set the radius and␣
↪rename the interfaces
        mesh=CircularMesh(radius=self.radius,segments=["NE"],straight_interface_
↪name={"center_to_north":"axis","center_to_east":"bottom"},outer_interface="interface
↪")
        self.add_mesh(mesh)

        eqs=StokesEquations(self.mu) # passing the dimensional viscosity to the␣
↪Stokes equations
        eqs+=RefineToLevel(3) # Refine the mesh, which is otherwise too coarse
        eqs+=MeshFileOutput() # and output

        #Imposing gravity as bulk force
        rho=1000*kilogram/meter**3 # mass density
        g=9.81*meter/second**2 # gravity
        gdir=vector(0,-1)       # direction of the gravity
        eqs+=StokesBulkForce(rho*g*gdir)

        #adding some artificial bulk force as well
        f=1000*rho/second**2 * vector(-var("coordinate_y"),var("coordinate_x"))
        eqs+=StokesBulkForce(f)
```

(continues on next page)

(continued from previous page)

```
        # No slip at substrate
        eqs+=DirichletBC(velocity_x=0,velocity_y=0)@"bottom"
        # No flow through the axis of symmtery
        eqs+=DirichletBC(velocity_x=0)@"axis"
        # Use our zero flux interface
        eqs+=StokesFlowZeroNormalFlux()@"interface"

        # Adding these equations to the default domain name "domain" of the␣
↪CircularMesh above
        self.add_equations(eqs@"domain")
```

We use a quarter circle mesh with the `CircularMesh` class. This one has a curved interface and is hence ideal to test the no-flux condition. It is also important to set the spatial scale and typical scales for the velocity and the pressure here. Since both are not imposed strongly, it is hard to determine a good scale a priori. We just take any reasonable values (which can later on be checked by comparing with the typical orders of magnitude in the output). We switch to an axisymmetric coordinate system, so our quarter circle mesh is in fact an axisymmetric hemisphere with the y-axis as axis of symmetry. The `CircularMesh` has options to rename the interfaces, which we use here to name the interface aligned with the x-axis as `"bottom"`, the one aligned with the y-axis as `"axis"` and the curved interface is named as `"interface"`. Furthermore, the `CircularMesh` is very coarse by default, but we can refine it three times by adding a `RefineToLevel` to the equation class. We add some bulk forces, i.e. the gravity $-\rho g\vec{e}_y$ and some artificial force to bring the fluid into motion. Since we apply a no-slip condition at the `"bottom"` interface, the contact line between `"bottom"` and `"interface"` is actually a case where the discussed problem with the Lagrange multiplier of the `StokesFlowZeroNormalFlux` class arises. However, the implemented method `before_assigning_equations_postorder()` will take care of this and pin the local Lagrange multiplier at this point automatically.

The run code is trivial:

```
if __name__ == "__main__":
    with NoFluxStokesProblem() as problem:
        problem.solve()
        problem.output()
```

The results are shown in Fig. 4.19. It is apparent how the spatially varying body force drives the flow. At the curved interface, the action of the no-flux condition prevents any in-/outflow, but allows for free tangential flow.

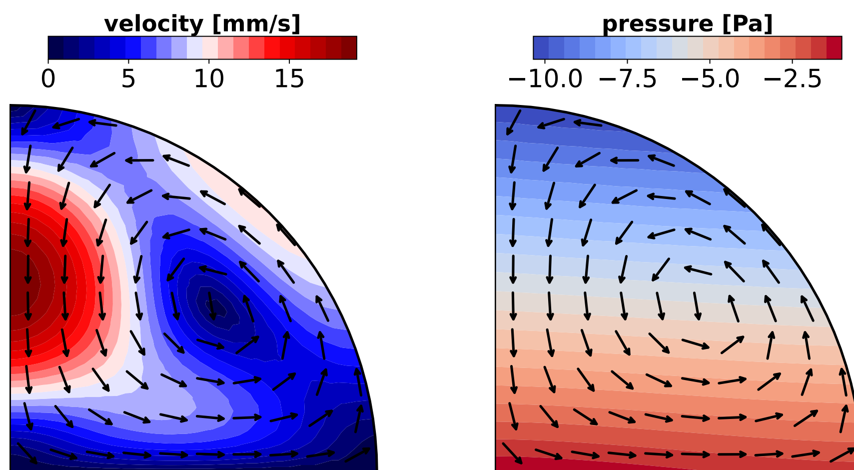

Fig. 4.19: Velocity and pressure field of the Stokes flow with enforced zero outflow at the curved interface and bulk force driving the flow.

4.4.7 Stokes' law - Obtaining forces by traction integrals and using global Lagrange multipliers

Stokes' law describes the flow field around a spherical rigid object. Here, we consider a solid sphere sinking down due to buoyancy. Obviously, treating this in lab frame would require to solve for the motion of the object directly, which can be done with a moving mesh method as described later on in Section 6. However, we can also transform into the coordinate system co-moving with the object. In this case, a fixed mesh can be used.

Stokes' law states that the terminal velocity will be given by

$$U = \frac{2}{9}\frac{\rho_\mathrm{o}-\rho_\mathrm{f}}{\mu}gR^2\,, \tag{4.15}$$

where ρ_o and ρ_f are the mass densities of the spherical object and the fluid, respectively, g is the gravitational acceleration, μ is the fluid's dynamic viscosity and R is the radius of the object. This equation can be derived by balancing the net force due to gravity

$$F_g = \Delta\rho g V = (\rho_\mathrm{o}-\rho_\mathrm{f})\,g\frac{4}{3}\pi R^3$$

acting on the object with the drag force F_d into the direction of $\vec{e}_z$, i.e. in direction of the gravity. The drag force can be obtained by the integration over the z-projected traction of the object

$$F_\mathrm{d} = \int_\mathrm{obj} \vec{n}\cdot\left[-p\mathbf{1}+\mu\left(\nabla\vec{u}+(\nabla\vec{u})^\mathrm{t}\right)\right]\cdot\vec{e}_z\,\mathrm{d}S \tag{4.16}$$

When solving this, the analytical radial and axial velocity (in frame of the object) reads

$$\begin{aligned} u_r &= \frac{3R^3}{4}\frac{rzU}{d^5}-\frac{3R}{4}\frac{rzU}{d^3}\\ u_z &= \frac{R^3}{4}\left(\frac{3Uz^2}{d^5}-\frac{U}{d^3}\right)+U-\frac{3R}{4}\left(\frac{U}{d}+\frac{Uz^2}{d^3}\right)\\ \text{with } d &= \sqrt{r^2+z^2} \end{aligned}$$

In our problem, we will not use the analytical velocity (4.15), but indeed solve for it. In fact, we use the terminal velocity U as global Lagrange multiplier (with test function V) to enforce the force balance. Hence, we minimize with respect to the constraint

$$U\cdot(F_\mathrm{d}-F_g)=0$$

to determine U. This value of U is then used as far field condition by virtue of (4.17). U is hence determined by the weak form

$$V\cdot(F_\mathrm{d}-F_g)=0 \tag{4.17}$$

and the feedback of U via the traction is given by the far field, which depends on U and changes the flow to modify the value of F_d until this constraint is met.

As a first step, we must build a mesh with a spherical object in the center. The far field boundary may be more or less arbitrary, but we chose a larger spherical shell. According to the mesh creation tutorial in Section 4.3, this can be done e.g. by

```
from stokes_dimensional import * # Import dimensional Stokes from before

# Mesh: Two concentrical hemi-circles => axisymmetric => concentric spheres
class StokesLawMesh(GmshTemplate):
    def define_geometry(self):
```

(continues on next page)

(continued from previous page)

```
        self.default_resolution=0.05 # make it a bit finer
        self.mesh_mode="tris"
        p=self.get_problem() # get the problem to obtain parameters
        Rs=p.sphere_radius # bind sphere radius
        Ro=p.outer_radius # and outer radius
        self.far_size = self.default_resolution*float(Ro/Rs) # Make the far␣
↪field coarser
        p00=self.point(0,0) # center
        pSnorth=self.point(0,Rs) # points along the sphere
        pSeast=self.point(Rs,0)
        pSsouth=self.point(0,-Rs)
        pOnorth=self.point(0,Ro,size=self.far_size) # points of the far field
        pOeast=self.point(Ro,0,size=self.far_size)
        pOsouth=self.point(0,-Ro,size=self.far_size)
        self.line(pOsouth,pSsouth,name="axisymm_lower") # axisymmetric lines, we␣
↪have two since we
        self.line(pOnorth,pSnorth,name="axisymm_upper") # want to fix p=0 at a␣
↪single p-DoF at axisymm_upper
        self.circle_arc(pOsouth,pOeast,center=p00,name="far_field") # far field␣
↪hemi-circle
        self.circle_arc(pOnorth,pOeast,center=p00,name="far_field")
        self.circle_arc(pSsouth,pSeast,center=p00,name="liquid_sphere") # sphere␣
↪hemi-circle
        self.circle_arc(pSnorth,pSeast,center=p00,name="liquid_sphere")
        self.plane_surface("axisymm_lower","axisymm_upper","far_field","liquid_
↪sphere",name="liquid") # liquid domain
```

We split the axis of symmetry into two parts, namely the lower and upper one. Thereby, we can later on pin a single degree of the pressure at e.g. `"liquid_object/axisymm_lower"` to remove the pressure nullspace.

Next, we require a possibility to calculate the drag force F_d and add this contribution to the test space of U, i.e. add it to the residuals with test function V. To that end, we will later pass V to the constructor of our new class

```
class DragContribution(InterfaceEquations):
    required_parent_type = StokesEquations
    def __init__(self,lagr_mult,direction=vector(0,-1)):
        super(DragContribution, self).__init__()
        self.lagr_mult=lagr_mult # Store the destination Lagrange multiplier U
        self.direction=direction # and the e_z direction

    def define_residuals(self):
        u=var("velocity",domain=self.get_parent_domain()) # Important: we want␣
↪to calculate grad with respect to the bulk
        strain=2*self.get_parent_equations().mu*sym(grad(u)) # get mu from the␣
↪parent equations
        p=var("pressure")
        stress = -p * identity_matrix() + strain  # T=-p*1 + mu*(grad(u)+grad(u)^
↪t))
        n = var("normal")  # interface normal pointing outwards
        traction = matproduct(stress, n)  # traction vector by projection
        ltest=testfunction(self.lagr_mult) # test function V of the Lagrange␣
↪multiplier U
        self.add_residual(weak(dot(traction,self.direction),ltest,dimensional_
↪dx=True)) # Integrate dimensionally over the traction
```

One important trick is here that we pass `domain=self.get_parent_domain()` when we bind the field `"velocity"` to `"u"`. Thereby, we do not get the interfacial velocity, but the full velocity of the bulk. While the values of the bulk

and interfacial velocity coincide on the interface, spatial derivatives do not! If we would bind `u=var("velocity")` without the `domain` argument, $\nabla\vec{u}$ would take the surface gradient $\nabla_S\vec{u}$, not the bulk gradient $\nabla\vec{u}$. Alternatively, we could have used `u=var("velocity",domain="..")` as shortcut to bind the bulk velocity.

Then we add the integral (4.16) to the test space of U, i.e on `testfunction(U)`, which is V. However, since `weak()` by default calculates integrals to the non-dimensional differential, i.e. to $\mathrm{d}\tilde{S}$ instead of $\mathrm{d}S$, we would not get the unit of a force. Therefore, we have to tell `weak()` by passing `dimensional_dx=True` that we want to integrate dimensionally.

The `Problem` class uses physical dimensions and we set the default values in the constructor. Furthermore, we add a method that allows to calculate the analytical terminal velocity according to (4.15):

```
class StokesLawProblem(Problem):
    def __init__(self):
        super(StokesLawProblem, self).__init__()
        self.sphere_radius=1*milli*meter # radius of the spherical object
        self.outer_radius=10*milli*meter # radius of the far boundary
        self.gravity=9.81*meter/second**2 # gravitational acceleration
        self.sphere_density=1200*kilogram/meter**3 # density of the sphere
        self.fluid_density=1000*kilogram/meter**3 # density of the liquid
        self.fluid_viscosity=1*milli*pascal*second # viscosity

    def get_theoretical_velocity(self): # get the analytical terminal velocity
        return 2 / 9 * (self.sphere_density - self.fluid_density) / self.fluid_
↪viscosity * self.gravity * self.sphere_radius ** 2
```

The problem definition will now use our mesh, set an axisymmetric coordinate system and introduces scalings, namely the object radius as spatial scale and the theoretical velocity as velocity scale. The pressure scale is set by the viscous pressure scale and we furthermore introduce a scale for any `"force"`, which is initialized by the buoyancy force. This one will be used in a minute.

```
def define_problem(self):
        self.set_coordinate_system("axisymmetric") # axisymmetric
        self.set_scaling(spatial=self.sphere_radius) # use the radius as spatial scale

        # Use the theoretical value as scaling for the velocity
        UStokes_ana=self.get_theoretical_velocity()
        self.set_scaling(velocity=UStokes_ana)
        self.set_scaling(pressure=scale_factor("velocity")*self.fluid_viscosity/scale_
↪factor("spatial"))
        # Buoyancy force
        F_buo=(self.sphere_density-self.fluid_density)*self.gravity*4/3*pi*self.
↪sphere_radius**3
        self.set_scaling(force=F_buo) # define the scale "force" by the value of the␣
↪gravity force

        self.add_mesh(StokesLawMesh())          # Mesh
```

The first part of the equations is trivial, just `StokesEquations` with output and a few boundary conditions:

```
eqs=StokesEquations(self.fluid_viscosity) # Stokes equation and output
eqs+=MeshFileOutput()

eqs+=DirichletBC(velocity_x=0)@"axisymm_lower" # No flow through the axis of symmetry
eqs += DirichletBC(velocity_x=0) @ "axisymm_upper"  # No flow through the axis of␣
↪symmetry
eqs+=DirichletBC(velocity_x=0,velocity_y=0)@"liquid_sphere" # and no-slip on the␣
↪object
```

Then, the Lagrange multiplier, i.e. the terminal velocity U, is introduced. We use `GlobalLagrangeMultiplier` for that, which will introduce a single global degree of freedom `UStokes`. Furthermore, the constant offset of F_g (`F_buo`) is subtracted, i.e. accounting for this term in (4.17). Both, the definition of `UStokes` and the offset term are simultaneously done by passing `UStokes=-F_buo` to the `GlobalLagrangeMultiplier`. The Lagrange multiplier equation is then augmented by a `Scaling` and a `TestScaling`, which sets the scale of `UStokes` to the `"velocity"` scale and the scale of its test function, i.e. V, to an inverse of the `"force"` scale. With the latter, (4.17) will become nondimensional, i.e. the units of force will cancel out upon the internal replacement of the variables and test functions by its non-dimensional counterparts:

```
# Define the Lagrange multiplier U
U_eqs = GlobalLagrangeMultiplier(UStokes=-F_buo) # name if "UStokes" and add an offset of -F_buo to test space of U
U_eqs += Scaling(UStokes=scale_factor("velocity")) # "UStokes" scales as a velocity
U_eqs += TestScaling(UStokes=1/scale_factor("force")) # and V scales as 1/[F]
self.add_equations(U_eqs @ "globals") # add it to an ODE domain named "globals"
```

Note: The `Problem` class has a method `add_global_dof()`, which simplifies the addition of a `GlobalLagrangeMultiplier` with a `Scaling` and a `TestScaling` and a potential global contribution to its residual.

Since the Lagrange multiplier is global, we cannot add it to any mesh. Instead, it has to be added to an own domain, which we call `"globals"` here.

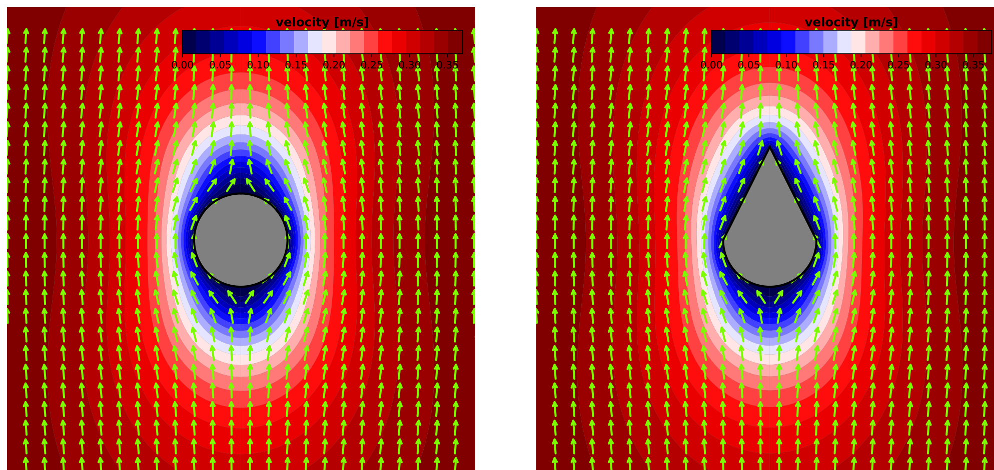

Fig. 4.20: (left) Velocity around a spherical object according to Stokes law. (right) With adjustments of the mesh, one easily can replace the shape of the object.

We then bind this variable, where again the `domain` argument is crucial and pass it to our developed class `DragContribution`. The `DragContribution` has to be attached to the `"liquid/liquid_sphere"` interface, since we must integrate over this interface to obtain the drag:

```
U=var("UStokes",domain="globals") # bind U from the domain "globals"
# Add the traction integral, i.e. the drag force to U
eqs += DragContribution(U)@"liquid_sphere" # The constraint is now fully assembled
```

Finally, the value of U must be used as far field condition. To that end, we implement the analytical solution (4.17) into pyoomph and enforce it at the far field boundary. We cannot use a `DirichletBC` here, since the analytical solution depends on U, which is part of the unknowns, but `DirichletBC` terms should only depend on independent variables as e.g. `"time"`:

```
# Far field condition
R=self.sphere_radius
r,z=var(["coordinate_x","coordinate_y"])
d=subexpression(square_root(r**2+z**2)) # precalcuate d in the generated C code for␣
↪faster computation
ur_far=3*R**3/4*r*z*U/d**5-3*R/4*r*z*U/d**3 # u_r as function of U
uz_far=R**3/4*(3*U*z**2/d**5-U/d**3)+U-3*R/4*(U/d+U*z**2/d**3) # u_z as function of U

# Since U is an unknown, DirichletBC should not be used here. Instead, we enforce the␣
↪velocity components to the far field by Lagrange multipliers
eqs+=EnforcedBC(velocity_x=var("velocity_x")-ur_far,velocity_y=var("velocity_y")-uz_
↪far)@"far_field"
eqs += DirichletBC(pressure=0) @ "liquid_sphere/axisymm_upper"  # fix one pressure␣
↪degree

self.add_equations(eqs@"liquid")
```

The run code is again short, but we compare the analytical and numerical value, leading to an error of $\sim 0.024\,\%$ for this mesh resolution:

```
if __name__ == "__main__":
    with StokesLawProblem() as problem:
        problem.solve() # solve and output
        problem.output()
        # Compare numerical and analytical velocity
        U_num=problem.get_ode("globals").get_value("UStokes")
        U_ana=problem.get_theoretical_velocity()
        print("NUMERICAL: ",U_num,"ANALYTICAL:",U_ana,"ERROR [%]:",abs(float((U_
↪num-U_ana)/U_ana*100)))
```

The result is plotted in Fig. 4.20. We can easily change the mesh to calculate the terminal velocity around differently shaped objects. The far field solution won't be exact, but for a sufficiently large exterior mesh, the made error becomes small due to the convergence of (4.17) to $(u_r, u_z) = (0, U)$.

4.4.8 Using a discontinuous pressure - Crouzeix-Raviart elements

So far, we have only addressed the continuous spaces "C1" and "C2". Pyoomph has additional finite element spaces which can be used. In particular for solving flow problems, there is - besides the Taylor-Hood element - the Crouzeix-Raviart element [8]. In fact, a Taylor-Hood approximation (i.e. using "C2" for the velocity and "C1" for the pressure) does guarantee to fulfill the continuity equation in each element, but only globally on the entire domain, i.e. in- and outfluxes through domain boundaries are balanced, but not necessarily in each element.

The reason for that can be found in the continuous pressure space. The pressure acts as a Lagrange multiplier field enforcing the incompressibility. However, since the degrees of freedom of the pressure are shared by neighboring elements, the incompressibility is not enforced on a elemental basis. To achieve this, one must make sure that all pressure degrees of freedom are associated with a single element, which means that the pressure field may become discontinuous across the elements. While there are several choices for Crouzeix-Raviart elements, we will focus on the same as oomph-lib provides, where the pressure is represented by an affine linear function in each element.

In pyoomph, the discontinuous space "DL" introduces exactly these degrees of freedom, i.e. in each element, the pressure field will follow an affine linear relation

$$p^{\mathrm{el}}(\vec{x}) = \vec{a}^{\mathrm{el}} \cdot (\vec{x} - \vec{x}_{\mathrm{c}}^{\mathrm{el}}) + b^{\mathrm{el}} \tag{4.18}$$

where $\vec{a}^{\mathrm{el}}$ accounts for the slopes of the pressure within the element, $\vec{x}_{\mathrm{c}}^{\mathrm{el}}$ is the center of the element and b^{el} is the offset to account for the degree of freedom in the absolute pressure. The superscript $^{\mathrm{el}}$ is used to indicate that these quantities

change when moving to the next element, while they are constants within each element. As a *nota bene*, pyoomph also has the discontinuous space `"D0"`, which is just constant in each element, i.e. (4.18) with $\vec{a}^{\mathrm{el}} = 0$. For an illustration, of the continuous spaces `"C1"` and `"C2"` and the discontinuous spaces `"DL"` and `"D0"`, please refer to Fig. 4.21.

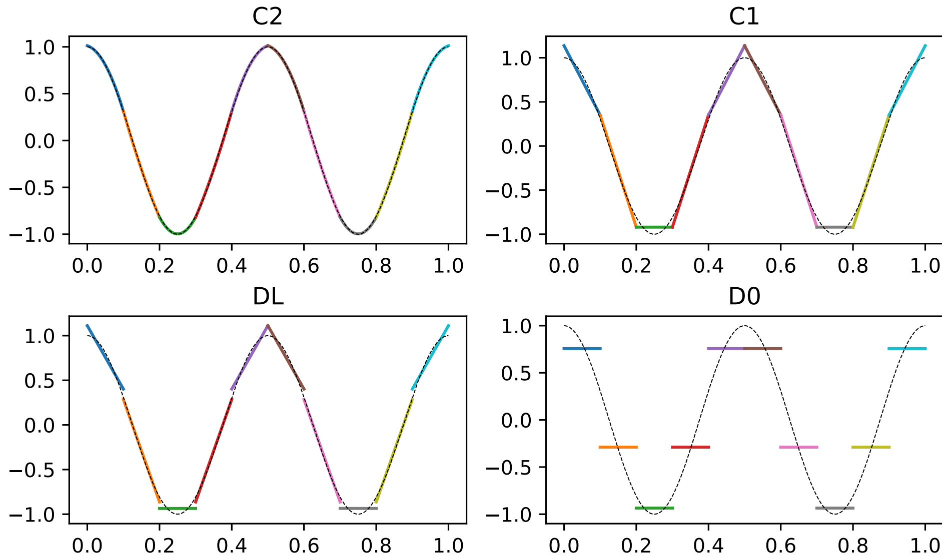

Fig. 4.21: Approximations of $\cos(4\pi x)$ (dashed line) by different spaces. The different colors correspond to the elements.

The Crouzeix-Raviart elements of oomph-lib combine the `"C2"` space for the velocity with the `"DL"` space for the pressure. This is at least true for quadrilateral elements, whereas for triangular elements, the velocity would be over-constrained by the incompressibility enforced on the `"DL"` space. For these elements, a cubic *bubble* velocity degree must be added to the `"C2"` space. This additional bubble degree is located in the center of the triangular element (see Fig. 4.22). Likewise, a three-dimensional tetrahedral element must be enriched by bubbles.

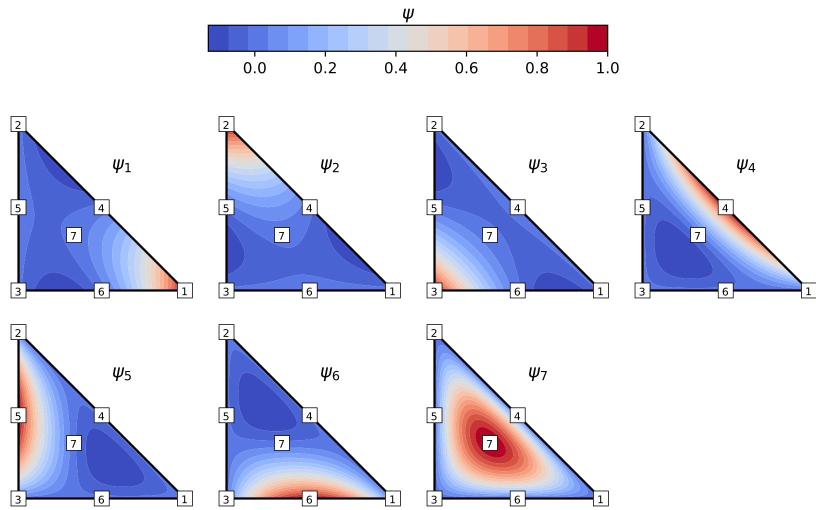

Fig. 4.22: Bubble-enriched triangular element, i.e. showing the space `"C2TB"`.

Since pyoomph separates the spaces and the elements, an additional space `"C2TB"` is introduced, which is `"C2"` on each quadrilateral element, but will enrich each triangular or tetrahedral element by bubble degrees. Hence, we can use the Crouzeix-Raviart elements on both quadrilateral and triangular elements, by just passing the combination `"C2TB"`, `"DL"` to the Stokes problem with space selection from Section 4.4.2:

```
from stokes import *

if __name__ == "__main__":
    # Create a Stokes problem with viscosity 1, on a Crouzeix-Raviart element
    with StokesSpaceTestProblem(1.0, "C2TB", "DL") as problem:
        problem.solve()  # solve and output
        problem.output()
```

The benefit of Crouzeix-Raviart elements is the elementwise valid continuity equation, but it comes at the price of an increased number of degrees of freedom.

Warning: When using `MeshFileOutput` for the discontinuous space `"DL"`, it will not show the gradients, but only the value at the centroid of the element. This is due to the fact that VTU files do not support cell data that varies in space.

Warning: Continuous spaces are directly transferred to the interfaces. Thereby, we can access the velocity on the interface by `var_and_test("velocity")` on an `InterfaceEquations` class, as e.g. in the example in Section 4.4.6. Discontinuous fields, like the pressure here, are internally stored in the bulk elements and hence cannot be accessed directly by the attached interfaces, but you can access it by e.g. `var_and_test("pressure", domain=self.get_parent_domain())` on interfaces.

For the same reason, you cannot set `DirichletBC` terms for discontinuous fields at interfaces. While usually you do not set `DirichletBC`s for the pressure, this can be problematic for the case discussed in Section 4.4.4, where one degree of the pressure had to be eliminated to remove the null space. To that end, the predefined `StokesEquations` in `pyoomph.equations.navier_stokes` have a function to fix one pressure degree for both cases, Taylor-Hood and Crouzeix-Raviart elements. See `stokes_pressure_fix.py` for an example how to use it.

Tip: oomph-lib covers the Taylor-Hood and the Crouzeix-Raviart elements in the example https://oomph-lib.github.io/oomph-lib/doc/navier_stokes/driven_cavity/html/index.html.

4.5 The Helmholtz equation with PML

Here, we consider the Helmholtz equations which arises when e.g. applying separation of variables to the wave equation. The solution of the Helmholtz equation then gives the amplitude of e.g. a periodic sound field. However, as we will see in this tutorial, boundaries will automatically reflect the wave, either by posing a node for Dirichlet conditions or an antinode for zero Neumann conditions. We will also introduce the concept of *perfectly matched layers* (PML), which can be used to mimick an infinite domain, i.e. where an outgoing wave just leaves the boundary without any reflection.

But let's first start with the basis, namely the Helmholtz equation

$$\nabla^2 u + k^2 u = 0$$

The weak formulation is obviously analogous to the Poisson equation, but the source term is now dependent on the unknown field u as well, entering linearly with the squared wave number k as factor. Using the test function v, the weak formulation reads

$$(\nabla u, \nabla v) - \left(k^2 u, v\right) - \langle \vec{n} \cdot \nabla u, v \rangle = 0$$

While this is trivial to implement with the knowledge obtained so far throughout this tutorial, it is not trivial to pose appropriate boundary conditions that allow for the wave to leave the considered domain without reflection. While this can in principle be done by solving a Dirichlet-to-Neumann problem along the boundaries, pyoomph does not allow for this. Pyoomph is a local finite element method, whereas Dirichlet-to-Neumann problems are always nonlocal and usually solved by boundary integral methods instead. In particular, the resulting Jacobian won't be sparse (at least the boundary contributions) and the linear solvers pyoomph is using are optimized for sparse matrices only. Moreover, pyoomph just does not allow to pose nonlocal integral equations like convolution integrals.

The alternative are the perfectly matched layers (PML), which can be written as local contribution. However, this comes at some expenses: Firstly, the domain has to be extended by a PML regions and secondly, the real Helmholtz equation must be generalized to complex values. Once this is done, a complex coordinate transformation according to

$$\frac{\partial}{\partial x_i} \to \frac{1}{\gamma_i}\frac{\partial}{\partial x_i}$$

with complex spatially dependent functions $\gamma_i(\vec{x})$ can be utilized to alter the oscillatory behavior of the wave to an increasingly damped wave which vanishes once it reaches the end of the PML region. Obviously, when $\gamma_i = 1$, the equation behaves like the conventional Helmholtz equation, so the complex coordinate transformation with $\gamma_i(\vec{x}) \neq 1$ is only applied in the PML regions, not in the region of interest. If the wave leaves the domain of interest through a boundary in x-direction, we therefore set

$$\gamma_x = 1 + \frac{i}{k}\sigma_x(x)$$

where $\sigma_x(x)$ is an inverse distance to the exterior boundary of the PML region, e.g. $\sigma_x = |X_{\text{PML}} - x|^{-1}$, where X_{PML} is the x-position of the PML exterior boundary. For this particular case, we keep $\gamma_y = 1$, where it is vice versa for wave leaving the domain of interest in y-direction. If the wave leave the domain through a corner, both tranformations are active. We introduce a complex unknown field $U(\vec{x})$ by the splitting $U = u + iu_{\text{Im}}$ with corresponding complex test function $V = v + iv_{\text{Im}}$ and generalize the Helmholtz equation (for the 2d Cartesian case) to

$$(\mathbf{T}\nabla U, \nabla V) - \left(\gamma_x\gamma_y k^2 U, V\right) - \langle \vec{n}\cdot\mathbf{T}\nabla U, V\rangle = 0$$

here, $\mathbf{T} = \operatorname{diag}(\gamma_y/\gamma_x, \gamma_x/\gamma_y)$ applies the necessary transformation of the differential operators. Thereby, in the PML regions, the waves get exponentially damped but are still coupled with the surroundings, i.e. thereby still accounting for the coupling which leads to a nonlocal Dirichlet-to-Neumann problem.

To make all these aspects more clear, let's just implement a circular hole in a rectangular domain and solve the Helmholtz equation with a prescribed Dirichlet value on the circle boundary. We will add an option to activate or deactivate the PML to see the effect. We skip the mesh class here for brevity. It is quite some work if you want to add the individual PML regions. However, the mesh class is of course part of the example code file.

Our Helmholtz equation now takes, besides the wavenumber k, the potential scalings γ_x and γ_y:

```
class HelmholtzEquation(Equations):
    def __init__(self, k,gamma_x=1, gamma_y=1):
        super().__init__()
        self.k = k # wavenumber
        self.gamma_x = gamma_x # PML complex coordinate transformation coefficient in␣
↪x-direction
        self.gamma_y = gamma_y # PML complex coordinate transformation coefficient in␣
↪y-direction

    def has_PML(self):
        # Check if PML is used, i.e., if the gamma_x or gamma_y are not equal to 1
        return self.gamma_x != 1 or self.gamma_y != 1

    def define_fields(self):
        # Define the scalar fields for the Helmholtz equation
```

(continues on next page)

(continued from previous page)

```
        self.define_scalar_field('u', 'C2')
        if self.has_PML():
            # If PML is used, define an additional scalar field for the imaginary part
            self.define_scalar_field('u_Im', 'C2')

    def define_residuals(self):
        u,v=var_and_test("u")
        if not self.has_PML():
            # Standard Helmholtz equation without PML
            self.add_weak(grad(u),grad(v)).add_weak(-self.k**2 * u, v)
        else:
            # Helmholtz equation with PML, only works for 2D Cartesian coordinates␣
↪like here
            if self.get_nodal_dimension()!=2 or self.get_coordinate_system().get_id_
↪name() != "Cartesian":
                raise ValueError("PML Helmholtz equations only implemented for␣
↪Cartesian 2D problems")

            uIm,vIm=var_and_test("u_Im")
            # complex field and test function
            U,V=u+imaginary_i()*uIm,v+imaginary_i()*vIm
            # scaled gradient
            mygrad=lambda f: vector(self.gamma_y/self.gamma_x*grad(f)[0], self.gamma_
↪x/self.gamma_y*grad(f)[1])
            # complex weak form
            R=weak(mygrad(U),grad(V))-weak(self.k**2 * U, V*self.gamma_x*self.gamma_y)
            # add real and imaginary parts separately
            self.add_residual(real_part(R)+imag_part(R))
```

If no scaling is set, the function `has_PML` will return `False` and we only define a real-valued field `u` with the conventional weak form of the Helmholz equation. Otherwise, we also add a field for the imaginary part of U and define the weak form of the PML-modified Helmholtz equation instead. Note how we can access the imaginary unit by `imaginary_i()`. This allows us to assemble the complex-valued field U by adding the real and imaginary part. Since pyoomph only handles real-valued residuals, we have to cast it back to a real-valued residual by applying `real_part()` and `imag_part()` on it. Again, since the test functions of the real and imaginary part can be chosen arbitarily, the superposition of both residuals is sufficient in the `add_residual()` call.

As usual, the problem class just defines some reasonable parameters in the constructor. We also add a flag here whether we want to consider the PML part or not. If we have PML activated, we first must set up the expressions for γ_x and γ_y, which depend on the distances to the exterior PML boundaries. In order to activate the coordinate transformation only in the PML regions, we introduce two helper fields, `"PML_indicator_x"` and `"PML_indicator_y"`. These are `D0` fields, i.e. having a constant value within each element. By combining it with a `DirichletBC` without any boundary restriction, the values of these indicator fields will be set without having to solve for additional unknowns. By the indicator fields, we can blend in the PML coordinate transformation where necessary. Also note that we set $U = 0$ at the exterior boundary of the PML region and suppress the imaginary part at the circle, where we prescribe the Dirichlet condition for u.

```
class HelmholtzProblem(Problem):
    def __init__(self):
        super().__init__()
        self.k = square_root(50) # wavenumber
        self.a=0.2 # radius of the circle
        self.L=2 # half the side length of the square domain
        self.use_PML=True # use PML or not
        self.N_PML=5 # number of nodes in the PML region
        self.d_PML=0.2 # thickness of the PML region
```

(continues on next page)

(continued from previous page)

```
        self.mesh_coeff_PML=1 # node placement coefficient for the PML region

    def define_problem(self):
        self+=RectangularMeshWithHoleAndPMLBoundary()

        if self.use_PML:
            x,y=var(["coordinate_x","coordinate_y"])
            # Inverse PML distance coefficients. Diverge at the far boundary
            sigma_x=subexpression(maximum(1/((self.L+self.d_PML)-x),1/((self.L+self.d_
↪PML)+x)))
            sigma_y=subexpression(maximum(1/((self.L+self.d_PML)-y),1/((self.L+self.d_
↪PML)+y)))
            # PML complex coordinate transformations, only active in the PML region␣
↪by the indicator functions
            gamma_x=1+var("PML_indicator_x")*imaginary_i()/self.k*sigma_x
            gamma_y=1+var("PML_indicator_y")*imaginary_i()/self.k*sigma_y
            # Indicator functions for PML, elementally constant, set to 1 in PML␣
↪region, 0 in physical domain
            pml_eqs=ScalarField("PML_indicator_x", "D0")+DirichletBC(PML_indicator_
↪x=(heaviside(absolute(x)-self.L)))
            pml_eqs+=ScalarField("PML_indicator_y", "D0")+DirichletBC(PML_indicator_
↪y=(heaviside(absolute(y)-self.L)))
            pml_eqs+=DirichletBC(u_Im=0)@"circle"
            pml_eqs+=DirichletBC(u=0,u_Im=0)@"PML_outer"
        else:
            pml_eqs=0 # No PML equations if not used
            gamma_x, gamma_y=1, 1 # No PML scaling factors if not used

        eqs=HelmholtzEquation(self.k,gamma_x,gamma_y)
        eqs+=MeshFileOutput()
        eqs+=DirichletBC(u=0.1)@"circle"
        eqs+=pml_eqs # Add PML equations if used

        self+=eqs@"domain"
```

The run code is again trivial:

```
with HelmholtzProblem() as problem:
    problem.solve()
    problem.output()
```

The results are depicted in Fig. 4.23 and speak for themselves. Indeed the radial symmetry of the wave without any reflection on the boundaries can be achieved by PML.

Note: This example is a direct adaption of the case discussed in oomph-lib here: https://oomph-lib.github.io/oomph-lib/doc/pml_helmholtz/scattering/scattering.pdf

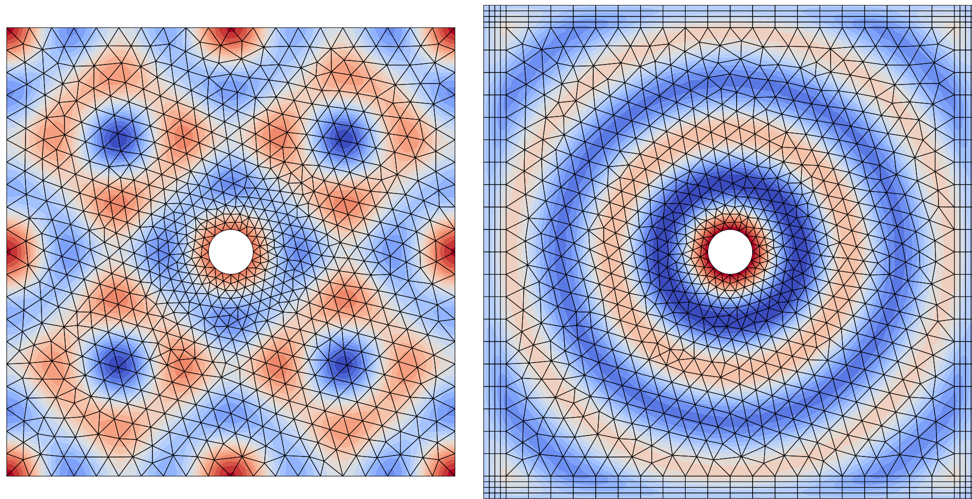

Fig. 4.23: (left) Solution with zero Neumann conditions and (right) with PML.

CHAPTER

FIVE

SPATIO-TEMPORAL DIFFERENTIAL EQUATIONS

Spatio-temporal differential equations are partial differential equations involving both derivatives with respect to space and time. We started our journey with temporal ODEs in Section 4. Now, it is time to combine both parts. Essentially, when discretizing spatio-temporal differential equations in space, one obtains a large system of temporal ODEs, which are then solved by temporal integration.

5.1 The wave equation

The wave equation can be considered as coupling between the harmonic oscillators discussed in Section 3 and the Poisson equation discussed in Section 4.1. As such, we will combine both temporal and spatial derivatives, which are both extensively discussed in the aforementioned sections.

5.1.1 Simple wave equation in one dimension

Let us start with a simple wave equation

$$\partial_t^2 u - c^2\nabla^2 u = 0\,. \tag{5.1}$$

Upon multiplication with the test function w and partial integration, one obtains the weak form

$$\left(\partial_t^2 u, w\right) + \left(c^2\nabla u, \nabla w\right) - \left\langle c^2\nabla u\cdot\vec{n}, w\right\rangle = 0\,. \tag{5.2}$$

In pyoomph, we just can use `partial_t()` for the time derivative and do the spatial part as before in Section 4. For the bulk contributions, i.e. the $(\cdot,\cdot)$ terms, we write again a `Equations` class:

```
from pyoomph import *
from pyoomph.expressions import *


class WaveEquation(Equations):
    def __init__(self,c=1):
        super(WaveEquation, self).__init__()
        self.c=c # speed

    def define_fields(self):
        self.define_scalar_field("u","C2")

    def define_residuals(self):
        u,w=var_and_test("u")
        self.add_residual(weak(partial_t(u,2),w)+weak(self.c**2*grad(u),grad(w)))
```

It is essentially the same approach as in Section 3. The problem class could read as follows

```
class WaveProblem(Problem):
    def __init__(self):
        super(WaveProblem, self).__init__()
        self.c=1 # speed
        self.L=10 # domain length
        self.N=100 # number of elements

    def define_problem(self):
        # interval mesh from [-L/2 : L/2 ] with N elements
        self.add_mesh(LineMesh(N=self.N,size=self.L,minimum=-self.L/2))

        eqs=WaveEquation() # equation
        eqs+=TextFileOutput() # output
        eqs+=DirichletBC(u=0)@"left" # fixed knots at the end points
        eqs+=DirichletBC(u=0)@"right"

        # Initial condition
        x,t=var(["coordinate_x","time"])
        #u_init=exp(-x**2) # This has no initial motion, i.e. partial_t(u)=0
        u_init=exp(-(x-self.c*t)**2) # This one moves initially to the right
        eqs+=InitialCondition(u=u_init) # setting the initial condition

        self.add_equations(eqs@"domain") # adding the equation

if __name__=="__main__":
    with WaveProblem() as problem:
        problem.run(20,outstep=True,startstep=0.1)
```

Note how the initial condition `u_init` depends on time `t`, which is bound by `var("time")` again. Thereby, we ensure that we have a traveling wave solution. Besides the initial condition $u(x,t{=}0)$, the additionally required first derivative $\partial_t u(x,t{=}0)$ is automatically evaluated. Indeed the result is, as expected, a traveling wave which is reflected at the boundaries, cf. Fig. 5.1. Without specifying the time dependency of the initial condition, $\partial_t u(x,t{=}0)=0$ would hold, yielding a different solution.

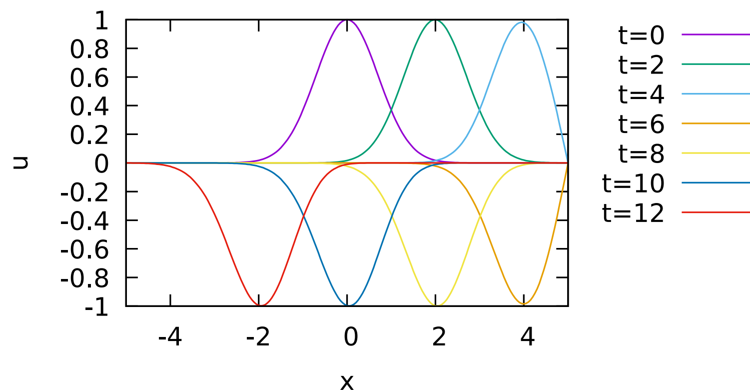

Fig. 5.1: Traveling wave solution, which is reflected at the boundaries.

Without the `DirichletBC(u=0)` terms, the $\langle\cdot,\cdot\rangle$ terms in (5.2) would become relevant. Since we do not add any contributions at the boundaries by some `InterfaceEquations` or `NeumannBC`, the term $\langle c^2\nabla u\cdot\vec{n},w\rangle$ is zero. This can only hold for arbitrary w, if $\partial_x u=0$. Thereby, the wave will have free ends on both sides. The wave gets reflected, but without changing sign.

5.1.2 Playing drums - Wave equation on a circular domain

Since pyoomph supports the definition of the `Equations` class independently of the number of spatial dimensions and coordinate systems, the same wave equation can be reused to solve it on other domains. Let us just solve the equation on a circular domain with radius R now. The analytical result is well known and is expressed as a *Fourier-Bessel series*:

$$u(r,\theta)=\sum_m\left(\alpha_m\cos(m\theta)+\beta_m\sin(m\theta)\right)\sum_n A_{mn}J_m(\lambda_{mn}r/R)$$

Here, (r,θ) are the polar coordinates and α_m and β_m are amplitudes of the polar modes of index m. A_mn is the amplitude of the radial mode (m,n), where J_m is the Bessel functions of the first kind. λ_{mn} is the nth positive root of J_m.

pyoomph does not have the Bessel functions implemented, but the Python package scipy does. However, since pyoomph compiles the equations to C code, we must wrap the scipy implementation of the Bessel functions in a `CustomMathExpression` callback function:

```
from wave_eq import * # Import the wave equation from the previous example
from pyoomph.meshes.simplemeshes import CircularMesh # Import the circle mesh

# Required for Bessel functions
import scipy.special


# Expose the Bessel function from scipy to pyoomph
class BesselJ(CustomMathExpression):
    def __init__(self,m):
        super(BesselJ,self).__init__()
        self.m=m # index of the Bessel function

    def eval(self,arg_arry):
        return scipy.special.jv(self.m,arg_arry[0]) # Return the scipy result
```

The problem class can then use the `BesselJ` class to setup a single angular mode m with some excited radial modes (m,n) as `InitialCondition`:

```
class WaveProblemCircularMesh(Problem):
    def __init__(self):
        super(WaveProblemCircularMesh, self).__init__()
        self.c=1 # speed
        self.R=10 # domain length
        self.m=3 # angular mode
        self.alpha=1 # coefficient of cos
        self.beta=0 # coefficient of sin
        self.radial_amplitudes=[1,-0.4,0.8] # radial amplitudes of R_mn

    def define_problem(self):
        self.add_mesh(CircularMesh(radius=self.R)) # Circular mesh

        eqs=WaveEquation(self.c) # equation
        eqs+=MeshFileOutput() # output
        eqs+=DirichletBC(u=0)@"circumference" # fixed knots at the rim
        eqs+=RefineToLevel(4) # the CircularMesh is by default coarse, refine it␣
↪4 times

        # Initial condition
        x, y = var(["coordinate_x","coordinate_y"]) # Cartesian coordinates
        r, theta = square_root(x**2+y**2), atan2(y,x)   # polar coordinates
        J_m=BesselJ(self.m) # bind the Bessel function with integer index m
```

(continues on next page)

(continued from previous page)

```
            bessel_roots=scipy.special.jn_zeros(self.m, len(self.radial_amplitudes))
↪# get the Bessel roots lambda_mn

            Theta=self.alpha*cos(self.m*theta)+self.beta*sin(self.m*theta) # angular␣
↪solution
            R=sum([A*J_m(r*lambd/self.R) for A,lambd in zip(self.radial_amplitudes,
↪bessel_roots)]) # radial solution
            eqs+=InitialCondition(u=Theta*R) # setting the initial condition

            self.add_equations(eqs@"domain") # adding the equation


if __name__=="__main__":
    with WaveProblemCircularMesh() as problem:
            problem.run(20,outstep=True,startstep=0.1)
```

First of all, we use the `CircularMesh`, which is very coarse by default. However, since we add a `RefineToLevel` object to the equations of the circular domain, the mesh will be refined (4 times here). The initial condition is then assembled to match a single angular mode m with a few excited radial modes (m, n). The amplitudes can be controlled by the `alpha`, `beta` and `radial_amplitudes` properties of the `Problem` class. We use scipy's functionality to find the first roots of J_m and eventually pass the assembled initial exitation as `InitialCondition`.

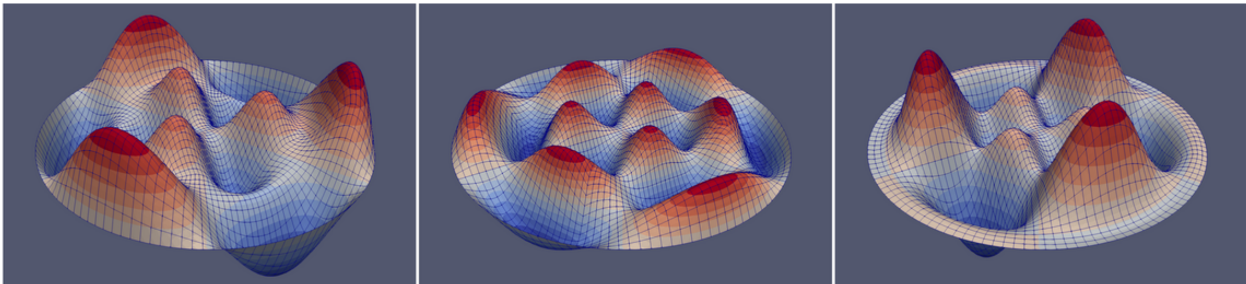

Fig. 5.2: Numerically obtained solution of the wave equation on a circular domain at three different time instants.

5.1.3 Double-slit

We will now solve the wave equation on a setting that resembles the famous *double-slit experiment*. To get a double-slit domain, we obviously have to create a custom mesh by interfacing Gmsh via the `GmshTemplate` class:

```
from wave_eq import *  # Import the wave equation from the previous example


class DoubleSlitMesh(GmshTemplate):
    def __init__(self,problem):
            super(DoubleSlitMesh,self).__init__()
            self.problem=problem # store the problem to access the properties
            self.default_resolution=self.problem.resolution
            self.mesh_mode="tris" # use triangles here

    def define_geometry(self):
            p=self.problem

            # corner points of the inlet
```

(continues on next page)

(continued from previous page)

```
            p_in_0=self.point(-p.inlet_length,0)
            p_in_H = self.point(-p.inlet_length, p.domain_height)
            p_wl_0=self.point(0,0)
            p_wl_H = self.point(0, p.domain_height)

            # slit corner points, upper slit
            p_wl_ub = self.point(0, (p.domain_height-p.slit_width+p.slit_distance)*0.
↪5)
            p_wl_uu = self.point(0, (p.domain_height+p.slit_width+p.slit_distance)*0.
↪5)
            p_wr_ub = self.point(p.wall_thickness, (p.domain_height - p.slit_width +␣
↪p.slit_distance) * 0.5)
            p_wr_uu = self.point(p.wall_thickness, (p.domain_height + p.slit_width +␣
↪p.slit_distance) * 0.5)

            # slit corner points, bottom slit
            p_wl_bb = self.point(0, (p.domain_height - p.slit_width - p.slit_
↪distance) * 0.5)
            p_wl_bu = self.point(0, (p.domain_height + p.slit_width - p.slit_
↪distance) * 0.5)
            p_wr_bb = self.point(p.wall_thickness, (p.domain_height - p.slit_width -␣
↪p.slit_distance) * 0.5)
            p_wr_bu = self.point(p.wall_thickness, (p.domain_height + p.slit_width -␣
↪p.slit_distance) * 0.5)

            # corner points of the domain behind the slit
            p_wr_0 = self.point(p.wall_thickness, 0)
            p_wr_H = self.point(p.wall_thickness, p.domain_height)
            p_scr_0 = self.point(p.screen_distance, 0)
            p_scr_H = self.point(p.screen_distance, p.domain_height)

            # Create a line loop of the exterior boundary
            lines=self.create_lines(p_in_0,"inlet",p_in_H,"top",p_wl_H,"wall_left",p_
↪wl_uu,"wall",p_wr_uu,"wall",p_wr_H,"out_top",p_scr_H,"screen",p_scr_0,"out_bottom",
↪p_wr_0,"wall",p_wr_bb,"wall",p_wl_bb,"wall_left",p_wl_0,"bottom",p_in_0)
            # The center part of the wall is a hole in the mesh
            holes=self.create_lines(p_wl_bu,"wall_left",p_wl_ub,"wall",p_wr_ub,"wall
↪",p_wr_bu,"wall",p_wl_bu)
            self.plane_surface(*lines,name="domain",holes=[holes]) # create the␣
↪domain
```

A lot of parameters are directly accessed from the `Problem` class, which will be defined soon. Therefore, the constructor of the `DoubleSlitMesh` gets the `Problem` passed, which is then accessed in the `define_mesh` method to get the chosen settings as e.g. `slit_width` or `slit_distance`. The corner points are created and a line loop is assembled with the `create_lines()` method. It takes alternating arguments of points and interface names for the interfaces between. The interface `"inlet"` will be used to impose a planar incoming wave, `"top"` and `"bottom"` are at the top and bottom of the domain before the slit. `"wall_left"` is the left side of the wall containing the slits, i.e. the side facing towards the incoming wave. At this side, we have to make sure to complete remove any reflections of the incoming wave to prevent an undesired interference with the incoming wave. All other interfaces of the slits and the right side of the wall are marked as `"wall"`. There, reflection is admitted. The `"out_top"` and `"out_bottom"` are the interfaces at the top and bottom after the two slits. Also here, reflection will be prevented. Finally, at the far right side of the mesh, the `"screen"` interface is set where the intensity will be measured.

To measure the intensity, a new `InterfaceEquations` class is defined, which will be added to the interface `"screen"`. On that interface, a new field I will be added, which is calculated by the accumulation of the wave in-

tensity over time, i.e.

$$I = \int u^2 \mathrm{d}t$$

or, in weak formulation, $(\partial_t I - u^2, J)$ with test function J.

```
# Measure the intensity of the waves at the screen
class WaveEquationScreen(InterfaceEquations):
    required_parent_type =  WaveEquation

    def define_fields(self):
        self.define_scalar_field("I","C2") # intensity as interface field

    def define_residuals(self):
        I,J=var_and_test("I")
        self.add_residual(weak(partial_t(I)-var("u")**2,J)) # I=integral of u^2␣
↪dt
```

Before the `Problem` class will be described, let us consider how any reflection can be prevented as e.g. on the screen and the wall of the double-slit facing towards the incoming wave. In the example in Section 5.1.1, we have seen that imposing a `DirichletBC(u=0)` reflects the wave with a change in sign. Without imposing any boundary condition, which is equivalent to impose a zero Neumann flux, the wave gets reflected as well, but without any sign flip. So what is the correct boundary condition if we just want the wave to be absorbed without any reflection? In that case, we have to make sure that any incoming wave just passed through the interface as if the domain would just continue after the interface. To see a good solution, we factorize the differential operator in (5.1) and consider the normal direction:

$$(\partial_t - c\vec{n}\cdot\nabla)(\partial_t + c\vec{n}\cdot\nabla)\,u = 0\,.$$

The equation is obviously fulfilled if $\partial_t u \pm c\nabla u\cdot\vec{n} = 0$, reflecting the fact that the wave equation allows for traveling solutions. As a Neumann flux, however, we can impose $-c^2\nabla u\cdot\vec{n}$ at interfaces. Hence, when imposing $c\partial_t u$ as Neumann flux, we will not influence the wave equation due to the presence of the boundary, however, only if the wave approaches in normal direction. More sophisticated solutions are e.g. *perfectly matched layers*, as discussed in Section 4.5.

This finally brings us the specification of the `Problem` class:

```
class DoubleSlitProblem(Problem):
    def __init__(self):
        super(DoubleSlitProblem, self).__init__()
        self.c = 1  # speed
        self.omega=10 # wave frequency
        self.inlet_length=0.4 # length of the inlet
        self.wall_thickness=0.1 # thickness of the wall
        self.screen_distance=2 # distance of the screen
        self.domain_height=4 # height of the entire domain
        self.slit_width=0.2 # width of a slit
        self.slit_distance=0.5 # distance of the slits
        self.resolution=0.04 # mesh resolution, the smaller the finer



    def define_problem(self):
        mesh=DoubleSlitMesh(self) # mesh

        #mesh=LineMesh(size=6,N=200)
        self.add_mesh(mesh)

        eqs = WaveEquation(c=self.c)  # wave equation
```

(continues on next page)

(continued from previous page)

```
            eqs += MeshFileOutput()  # mesh output

            # initial condition: incoming wave wave, exponentially damped
            u0=cos(self.omega*(var("coordinate_x")+self.inlet_length-self.c*var("time
↪")))*exp(-10*(var("coordinate_x")+self.inlet_length))
            eqs +=InitialCondition(u=u0)
            eqs += DirichletBC(u=u0) @ "inlet"  # incoming wave

            # Let the waves just flow out/absorb without any reflection at some␣
↪interfaces
            u=var("u")
            eqs += NeumannBC(u=self.c*partial_t(u)) @ "wall_left"
            eqs += NeumannBC(u=self.c*partial_t(u)) @ "screen"
            eqs += NeumannBC(u=self.c*partial_t(u)) @ "out_top"
            eqs += NeumannBC(u=self.c*partial_t(u)) @ "out_bottom"

            # Measure the intesity at the screen
            eqs += WaveEquationScreen()@"screen"
            eqs += TextFileOutput() @ "screen"

            self.add_equations(eqs @ "domain")


if __name__ == "__main__":
    with DoubleSlitProblem() as problem:
            problem.c=3 # increase the wave speed
            problem.omega=20 # and the wave frequency
            problem.run(1, outstep=True, startstep=0.01)
```

In the constructor, we have several parameters to allow for a custom wave and slit geometry. Since the problem itself is passed to the `DoubleSlitMesh`, the latter parameters are used to construct a mesh based on these. We make use of the absorption (no reflection) boundary conditions and add the `WaveEquationScreen` as well as a `TextFileOutput` to the interface `"screen"`. Thereby, we get the results of the intensity on the screen written to a file.

In the results (cf. Fig. 5.3) we indeed see that the incoming wave is not reflected at the wall, i.e. there is no self-interference. The same is true for the top and bottom boundary of the domain beyond the double-slit and the screen itself. The screen intensity I shows the expected pattern, i.e. a maximum in the center of the slits with additional smaller maxima and minima off-center.

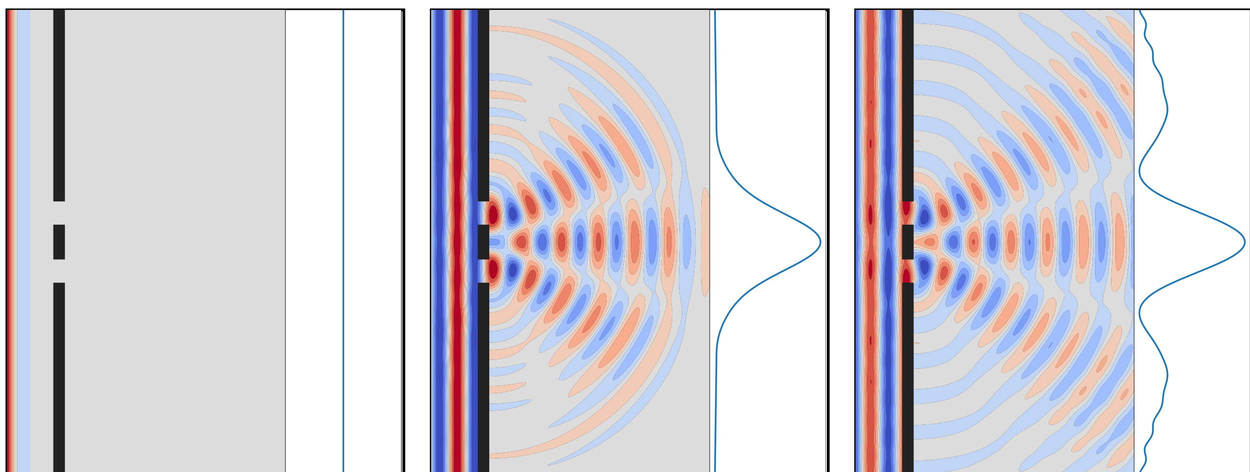

Fig. 5.3: Double-slit result at three different times along with the monitored intensity at the screen.

5.2 Convection-diffusion equation

A very important equation for transport phenomena is the convection diffusion equation

$$\partial_t c + \nabla \cdot (c\vec{u}) = \nabla \cdot (D\nabla c) \,.$$

Here c can be understood as e.g. some concentration field or a temperature field, $\vec{u}$ is the advecting velocity field and D is a diffusion coefficient. After multiplication with a test function ϕ, we can have two different weak formulations, depending on whether the advection term is included in the partial integration or not, namely:

$$(\partial_t c, \phi) + (\nabla \cdot (c\vec{u}), \phi) + (D\nabla c, \nabla\phi) + \langle -D\nabla c \cdot \vec{n}, \phi\rangle = 0 \tag{5.3}$$

and

$$(\partial_t c, \phi) - (c\vec{u}, \nabla\phi) + (D\nabla c, \nabla\phi) + \langle (\vec{u}c - D\nabla c) \cdot \vec{n}, \phi\rangle = 0\,. \tag{5.4}$$

Note that, if the velocity $\vec{u}$ is divergence free (incompressible), the second term in (5.3) reads $(\vec{u} \cdot \nabla c, \phi)$. The first observation between both variants is that the Neumann term $\langle \cdot, \cdot \rangle$ is different. In (5.3), we impose pure diffusive fluxes as Neumann conditions, whereas in (5.4) total fluxes, i.e. the sum of advection and diffusion, are imposed by Neumann conditions.

The second observation is more severe: The advection terms are not symmetric with respect to the spatial derivative order. While the time derivative has zeroth order spatial derivatives on both c and ϕ and the diffusion term has both first order spatial derivatives, i.e. ∇c and $\nabla\phi$, the advection term is mixed. Either the field c or the test function ϕ is derived. This asymmetry leads to considerable complications if the equation is advection-dominated.

5.2.1 Naive implementation

Let us first ignore this complication and implement the equation naively. We will add a flag `advection_in_partial_integration` to choose between both different weak forms of the advection term:

```
from pyoomph import *
from pyoomph.expressions import *


class ConvectionDiffusionEquation(Equations):
    def __init__(self,u,D,advection_in_partial_integration,space="C2"):
        super(ConvectionDiffusionEquation, self).__init__()
        self.u=u # advection velocity
        self.D=D # diffusivity
        self.space=space # space of the field c
        self.advection_in_partial_integration=advection_in_partial_integration #␣
↪Which weak form to use

    def define_fields(self):
        self.define_scalar_field("c",self.space) # The scalar field to advect

    def define_residuals(self):
        c,phi=var_and_test("c")
        # Advection either intergrated by parts or not
        advection=-weak(self.u*c,grad(phi)) if self.advection_in_partial_
↪integration else weak(div(self.u*c),phi)
        # TPZ or MPT time stepping can be of advantage compared to BDF2
        self.add_residual(time_scheme("TPZ",weak(partial_t(c),
↪phi)+advection+weak(self.D*grad(c),grad(phi))))
```

Depending on the value of `advection_in_partial_integration`, we either use (5.3) or (5.4) for the weak form. Furthermore, we changed the time stepping from the default `"BDF2"` to `"TPZ"`, which can be of advantage (cf. Section 3.6.3 for time stepping methods).

As a problem class, we use bump which is swirled around by one period. When the diffusivity is low, we expect the bump to be only slightly smaller in amplitude and only slightly coarser due to diffusion. The problem class hence reads:

```
class ConvectionDiffusionProblem(Problem):
    def __init__(self):
        super(ConvectionDiffusionProblem, self).__init__()
        self.u=2*pi*vector([-var("coordinate_y"),var("coordinate_x")]) #␣
↪Circular flow, one rotation at t=1
        self.D=0.001 # diffusivity
        self.L=1 # size of the mesh
        self.N=4 # number of elements of the coarsest mesh in each direction
        self.max_refinement_level=5 # max refinement level
        self.advection_in_partial_integration=False # which weak form to choose

    def define_problem(self):
        self.add_mesh(RectangularQuadMesh(lower_left=-self.L/2,size=self.L,
↪N=self.N))

        eqs=ConvectionDiffusionEquation(self.u,self.D,self.advection_in_partial_
↪integration)
        eqs+=MeshFileOutput() # output

        # use a bump as initial condition
        bump_pos=vector([-self.L/5,0]) # center pos of the bump
        bump_width=0.005*self.L # width of the bump
        bump_amplitude=1 # amplitude of the bump
        xdiff=var("coordinate")-bump_pos # difference between coordinate and␣
↪bump center
        bump=bump_amplitude*exp(-dot(xdiff,xdiff)/bump_width) # Gaussian bump
        eqs+=InitialCondition(c=bump)

        # Set the boundaries to 0
        for b in ["top","left","right","bottom"]:
            eqs+=DirichletBC(c=0)@b

        # Errors: We evaluate the jumps in the gradients of c at the element␣
↪boundaries, i.e. when crossing to the next element
        # this is not only done at the current time step, but also on the␣
↪previous one
        error_fluxes=[grad(var("c")),evaluate_in_past(grad(var("c")))]
        eqs+=SpatialErrorEstimator(*error_fluxes)

        self.add_equations(eqs@"domain") # adding the equation
```

The most interesting thing here is that we can also define a `SpatialErrorEstimator` based on history values. Instead of passing keyword arguments, we can also pass positional arguments to the `SpatialErrorEstimator`. The error estimator requires gradients, but these can also be evaluated at previous time steps. This ensures that the wake remains finer resolved.

The run code is again simple:

```
if __name__=="__main__":
    with ConvectionDiffusionProblem() as problem:
        problem.advection_in_partial_integration=True # Can also set it to false
```

(continues on next page)

(continued from previous page)

```
            problem.D=0.0001 # diffusivity
            problem.run(1,outstep=0.01,maxstep=0.0025,spatial_adapt=1,temporal_
↪error=1)
```

Using `outstep=0.01` in the `run()`, we will get 100 outputs, but due to `maxstep=0.0025`, we solve at least 4 times per output. `spatial_adapt=1` will perform, as usual, one spatial adaption per solve, whereas `temporal_error=1` just ensures that the time step gets reduced when it does not converge.

Results at different times are depicted in Fig. 5.4.

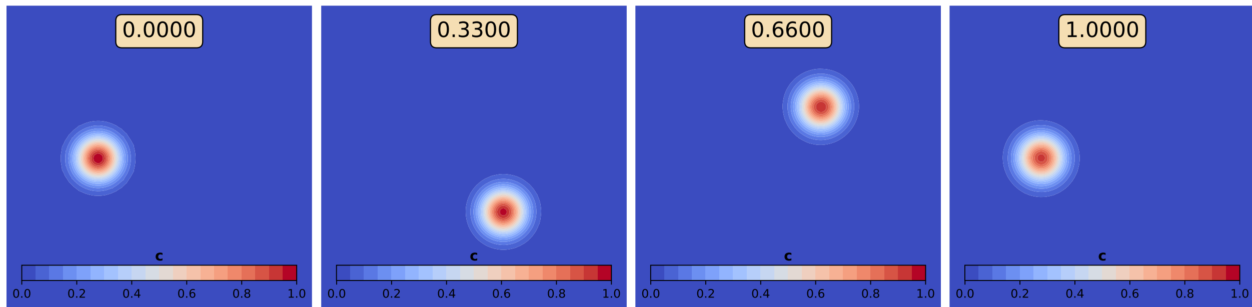

Fig. 5.4: Advecting a bump with low diffusivity.

5.2.2 SUPG implementation

An analogous approach to the *upwind scheme* in finite differences is the *SUPG* stabilization in the finite element method. In the *upwind scheme*, the first order advective derivative is evaluated in upwind direction, i.e. taking the slope in the upwind direction, which stabilizes the scheme for high *Péclet numbers*. The *SUPG* method (streamline upwind Petrov-Galerkin) does essentially the same in the finite element method. However, while it is trivial to find the degrees of freedom in upwind direction on a regular line or 2d/3d grid in the finite difference method, for arbitrary meshes, as commonly used in the finite element method, it is not that trivial.

The one-dimensional problem is best to illustrate the idea, so we will stick to it here. The general idea is to weight the upwind residuals more, i.e. to modify the localization of the residual by the projection on the test function to be enhanced in upwind direction. This can be achieved by replacing the test function ϕ in (5.3) to $\phi + \tau\vec{u}\cdot\nabla\phi$. The choice of the parameter τ is crucial and will be discussed in a minute. The SUPG variant of (5.3) (with $\nabla\cdot\vec{u}=0$ and for pure Dirichlet boundary conditions) hence reads (cf. e.g. [3]):

$$(\partial_t c + \vec{u}\cdot\nabla c, \phi) + (D\nabla c, \nabla\phi) + (\partial_t c + \vec{u}\cdot\nabla c, \tau\vec{u}\cdot\nabla\phi) = 0 \qquad (5.5)$$

We have assumed a first order space (`"C1"`) here (or vanishing diffusivity D), which removes the term $D\nabla^2 c$ in the last weak contribution. Pyoomph cannot handle second order derivatives yet, so this approximation is also necessary at the moment for `"C2"` spaces, where $\nabla^2 c$ does not vanish in each element. The selection of τ is important. There are several approaches, but usually it is defined as a constant per element. It must have a finite value for $\vec{u}\to 0$, must compensate the $\vec{u}$ term in the test function argument of the last term of (5.5) to prevent a dominance of the stabilization term for high velocities. Furthermore, it must vanish for infinitely refined meshes (so it must depend on the element size). The idea is to introduce the mesh Peclet number

$$\mathrm{Pe}_h = \frac{h\|\vec{u}\|}{2D}$$

where h is the size of the current element (e.g. the length of a one-dimensional element or the circumference for 2d elements). When $\mathrm{Pe}_h \to 0$, we do not require any stabilization, which happens for low velocities, high diffusivities or

small elements. If Pe_h becomes large (typically >3), stabilization is necessary to prevent the spurious oscillations for advection-dominated problems. Hence, a good selection of τ is

$$\tau_h = \frac{h}{2\|\vec{u}\|}\left(\coth\left(\mathrm{Pe}_h\right) - \frac{1}{\mathrm{Pe}_h}\right)$$

The term in the brackets indeed is 0 for $\mathrm{Pe}_h = 0$ and goes to unity for large Pe_h. The factor $\frac{h}{\|\vec{u}\|}$ compensates for the ∇ and the velocity appearing in the stabilization projection on $\tau\vec{u}\cdot\nabla\phi$.

To augment the advection-diffusion equations with the stabilization term, we can use the class `ElementSizeForSUPG` from `pyoomph.equations.SUPG`. It will calculate the Cartesian measure (i.e. length/area/volume) of each element and store it in a `"D0"` space. Since in moving mesh methods (cf. Section 6) the elements can change in size, the element size becomes part of the degrees of freedom. One can access the typical element length scale by the method `get_element_h()` of the `ElementSizeForSUPG` object.

The implementation of the augmented form (5.5) reads:

```
from pyoomph import *
from pyoomph.equations.SUPG import * # To calculate the element size


class ConvectionDiffusionEquationWithSUPG(Equations):
    def __init__(self, u, D,with_SUPG=True):
        super(ConvectionDiffusionEquationWithSUPG, self).__init__()
        self.u = u  # advection velocity
        self.D = D  # diffusivity
        self.scheme="TPZ" # Time scheme, trapezoidal rule
        self.with_SUPG=with_SUPG # do we activate SUPG?

    def define_fields(self):
        self.define_scalar_field("c", "C1") # Take the coarse space C1

    def get_supg_tau(self):
        # We must find an equation of the type ElementSizeForSUPG, which calculates␣
↪the element size
        elsize_eqs = self.get_combined_equations().get_equation_of_
↪type(ElementSizeForSUPG, always_as_list=True)
        if len(elsize_eqs)!=1: # User must combine it with a single␣
↪ElementSizeForSUPG instance
            raise RuntimeError("SUPG only works if combined with a single␣
↪ElementSizeForSUPG equation")
        elsize_eqs=elsize_eqs[0] # get the ElementSizeForSUPG object, which is␣
↪combined with this equation
        h = elsize_eqs.get_element_h() + 1e-15 # element size, add a tiny offset to␣
↪prevent errors
        u_mag=square_root(dot(self.u,self.u))+1e-15 # velocity magnitude , add a tiny␣
↪offset to prevent errors
        Pe_h=u_mag*h/(2*self.D) # Mesh Peclet number
        beta=1/tanh(Pe_h)-1/Pe_h # coefficient activating SUPG if Pe becomes large
        tau = subexpression(beta*h/(2*u_mag)) # returning the tau coefficient
        return tau

    def define_residuals(self):
        c, ctest = var_and_test("c")
        # This term occurs multiple times, so wrap it into a subexpression for␣
↪performance gain
        radv = subexpression(time_scheme(self.scheme,partial_t(c) + dot(self.u,␣
↪grad(c))))
```

(continues on next page)

(continued from previous page)

```
        self.add_residual(weak(radv, ctest))  # time derivative and advection
        self.add_residual(time_scheme(self.scheme,weak(self.D * grad(c),␣
↪grad(ctest))))  # diffusion
        if self.with_SUPG: # SUPG stabilization
            self.add_residual(time_scheme(self.scheme,weak(radv,self.get_supg_tau() *␣
↪dot(self.u, grad(ctest)))))
```

In the method `get_supg_tau` we check if the equation is combined with a single `ElementSizeForSUPG` object and bind the size h. We calculate Pe_h and thereby τ_h according to the relations discussed above. Finally, this is used for the stabilization term, but only if `with_SUPG` is `True`.

As a test class, we advect again a bump, but this time in one dimension:

```
class OneDimAdvectionDiffusionProblem(Problem):
    def __init__(self):
        super(OneDimAdvectionDiffusionProblem, self).__init__()
        self.u=vector(1,0)
        self.D=0.0001
        self.with_SUPG=True

    def define_problem(self):
        self.add_mesh(LineMesh(N=100,size=100,minimum=-20)) # coarse mesh from [-
↪20:80]

        eqs=TextFileOutput()
        eqs+=ConvectionDiffusionEquationWithSUPG(u=self.u,D=self.D,with_SUPG=self.
↪with_SUPG)
        if self.with_SUPG:
            eqs+=ElementSizeForSUPG() # We must add the element size

        x=var("coordinate_x")
        cinit=exp(-x**2*0.25)
        eqs+=InitialCondition(c=cinit)

        eqs+=DirichletBC(c=0)@"left"
        eqs += DirichletBC(c=0) @ "right"

        self.add_equations(eqs@"domain")
```

It is necessary to add a `ElementSizeForSUPG` object to calculate the element size if SUPG is active. The rest is trivial, but note that we again use `DirichletBC` on both sides. Neumann conditions would have to be augmented by SUPG corrections terms stemming from the consistent partial integration that leads to (5.5).

With a simple run code, we can compare the results with and without SUPG:

```
if __name__=="__main__":
    with OneDimAdvectionDiffusionProblem() as problem:
        problem.with_SUPG=True
        problem.run(50,outstep=1,maxstep=0.1)
```

Results are depicted in Fig. 5.5.

Note: An alternative way of getting the typical element size is just using `var("cartesian_element_size_Eulerian")` or `var("element_size_Eulerian")` instead of `ElementSizeForSUPG`. See `var()` for more information on such keyword variables.

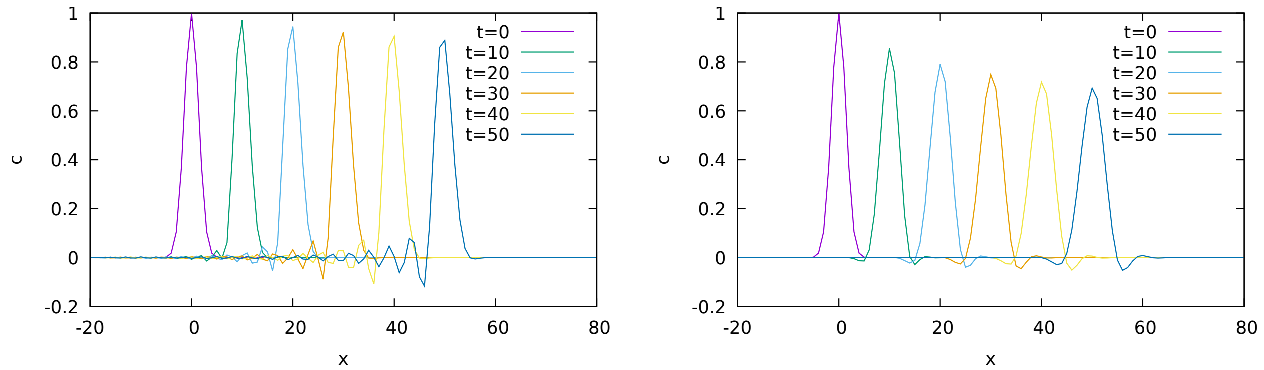

Fig. 5.5: Without (left) and with SUPG (right). Note how spurious oscillations are suppressed by SUPG, but the bump diffuses too fast. When the number of elements is increased, both problems vanish, even without SUPG.

5.3 Navier-Stokes equation

The Navier-Stokes equation is the same as the Stokes equation in Section 4.4, but with the addition of the inertia term $\partial_t\vec{u} + \vec{u} \cdot \nabla\vec{u}$. Again, we have to make sure to meet the inf-sup criterion, i.e. we select a Taylor-Hood formulation:

```
from pyoomph import *
from pyoomph.expressions import *


class NavierStokesEquations(Equations):
    def __init__(self,rho,mu):
        super(NavierStokesEquations, self).__init__()
        self.rho, self.mu= rho,mu # Store viscosity and density

    def define_fields(self):
        self.define_vector_field("velocity","C2") # Taylor-Hood pair
        self.define_scalar_field("pressure","C1")

    def define_residuals(self):
        u,v=var_and_test("velocity") # get the fields and the corresponding test␣
↪functions
        p,q=var_and_test("pressure")
        stress=-p*identity_matrix()+2*self.mu*sym(grad(u)) # see Stokes equation
        inertia=self.rho*material_derivative(u,u) # lhs of the Navier-Stokes eq.␣
↪rho*Du/dt
        self.add_residual(weak(inertia,v)+ weak(stress,grad(v)) + weak(div(u),q))
```

The inertia term is obtained by using the function `material_derivative()`. Calling this function with any scalar, vectorial or tensorial expressions F and the velocity **u** will return $\partial_t F + \mathrm{grad}(F) \cdot \mathbf{u}$.

> **Warning:** The trivial implementation of the Navier-Stokes equation does not allow for high Reynolds numbers. The reason is similar to the problem discussed with the advection-diffusion equation in the previous example. There are improved schemes using SUPG and pressure stabilization to accurately account for the term $\vec{u} \cdot \nabla\vec{u}$.

We discuss a few examples of the Navier-Stokes equation in the following sections.

5.3.1 Womersley flow

As an example for the action of inertia in this equation, we calculate a Womersley pipe flow, i.e. a flow through a pipe driven by an oscillating pressure:

```
class WomersleyFlowProblem(Problem):
    def __init__(self):
        super(WomersleyFlowProblem, self).__init__()
        self.rho,self.mu=10,1 # density and viscosity
        self.omega,self.delta_p=10,10 # frequency and pressure amplitude
        self.L,self.R=1,1 # size of the pipe
        # Corresponding to a Womersley number of 10
        self.max_refinement_level=3 # refine due to the velocity profile

    def define_problem(self):
        self.set_coordinate_system("axisymmetric") # Pipe: Axisymmetric
        Nr=4 # number of radial mesh elements
        self.add_mesh(RectangularQuadMesh(N=[Nr,int(self.L/self.R*Nr)],
↪size=[self.R,self.L]))

        eqs=NavierStokesEquations(self.rho,self.mu)
        eqs+=MeshFileOutput()
        eqs+=DirichletBC(velocity_x=0)@"left" # no r-velocity at the axis
        eqs+=DirichletBC(velocity_x=0,velocity_y=0)@"right" # no-slip at the wall
        eqs+=DirichletBC(velocity_x=0)@"top" # no r-velocity at in and outflow
        eqs+=DirichletBC(velocity_x=0)@"bottom"
        # impose oscillating pressure
        eqs+=NeumannBC(velocity_y=-self.delta_p*cos(self.omega*var("time")))@
↪"bottom"
        eqs+=SpatialErrorEstimator(velocity=1) # Refine where necessary
        # eqs+=DirichletBC(velocity_x=0) # We can also deactivate the entire x-
↪velocity in this problem
        self.add_equations(eqs@"domain") # adding the equation

if __name__=="__main__":
    with WomersleyFlowProblem() as problem:
        problem.run(1,outstep=True,startstep=0.01,spatial_adapt=1)
```

Due to the inertia, the flow reversal does not happen instantaneously, but shows a Womersley flow profile, see Fig. 5.6.

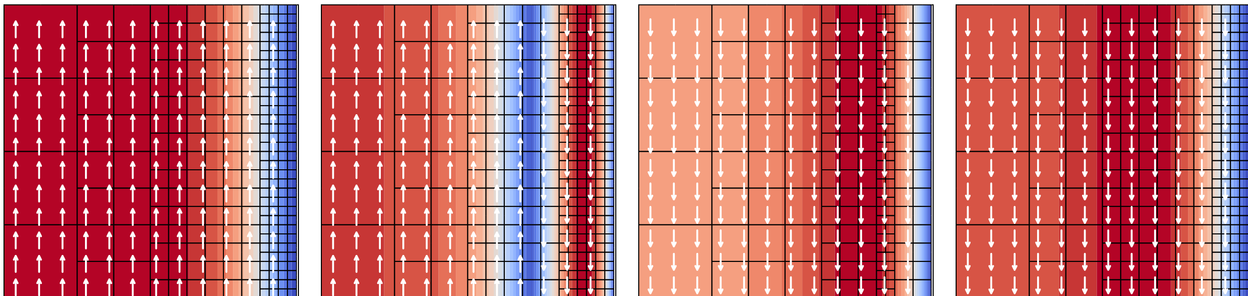

Fig. 5.6: Womersley flow in a pipe

5.3.2 Transient and nonlinear generalization of Stokes' law

Please refer to Section 4.4.7 for understanding the problem setup. We will only address the required changes here.

First of all, we do not only have to balance the drag force F_d with the buoyancy force F_g, but also the inertia $M\dot{U}$ matters, where M is the mass of the spherical object. To that end, instead of using the `GlobalLagrangeMultiplier` we write a simple `ODEEquations` class to account for Newton's law of motion:

```
class NewtonEquation(ODEEquations):
    def __init__(self,M,F_buo):
        super(NewtonEquation, self).__init__()
        self.M, self.F_buo = M, F_buo

    def define_fields(self):
        self.define_ode_variable("UStokes",scale="velocity",testscale=1/scale_factor(
↪"force"))

    def define_residuals(self):
        U,V=var_and_test("UStokes")
        self.add_residual(weak(self.M*partial_t(U)-self.F_buo,V))
```

As in the previous example, to this equation of motion, the drag force will be added externally by the `DragContribution` added to the interface of fluid and object.

In the problem class, to minimize errors stemming from the imposition of the pure Stokes solution at the far field, we shrink the spherical object and increase the radius of the far field:

```
self.sphere_radius = 0.25 * milli * meter  # radius of the spherical object
self.outer_radius = 50 * milli * meter  # radius of the far boundary
```

Since we have a temporal problem, also a time scale has to be set

```
self.set_scaling(temporal=self.sphere_radius/UStokes_ana)
```

We then have to exchange the `StokesEquations` by the `NavierStokesEquations` to account for the inertia. However, when transforming into the co-moving frame of reference, we have to correct for the accelation of the co-moving system. This can be added via the `bulkforce` argument, which adds a contribution proportional to $\rho_\mathrm{l}\dot{U}\vec{e}_z$ to the `NavierStokesEquations`. This effectively changes the time derivative in the inertia to $\rho_\mathrm{l}(\partial_t\vec{u}-\dot{U}\vec{e}_z)$ so that the acceleration of the coordinate system $\dot{U}$ cancels out:

```
U = var("UStokes", domain="globals")  # bind U from the domain "globals"
inertia_correction=self.fluid_density*vector([0,1])*partial_t(U)
eqs = NavierStokesEquations(dynamic_viscosity=self.fluid_viscosity,mass_density=self.
↪fluid_density,bulkforce=inertia_correction)  # Stokes equation and output
```

Instead of the `GlobalLagrangeMultiplier` we add the developed `NewtonEquation`:

```
# Define the Lagrange multiplier U
U_eqs=NewtonEquation(4/3*pi*self.sphere_radius**3*self.sphere_density,F_buo)
U_eqs+=ODEFileOutput()
self.add_equations(U_eqs @ "globals")  # add it to an ODE domain named "globals"
```

Note that we combine it with an `ODEFileOutput` to write the time evolution of $U(t)$ to a file.

Finally, the run code must be transient now:

```
if __name__ == "__main__":
    with TransientNonlinearStokesLawProblem() as problem:
        problem.run(0.5*second,startstep=0.05*second,outstep=True)  # solve and output
```

As seen in Fig. 5.7, the final velocity field is not symmetric anymore and we see a transient dynamics of $U(t)$ is plotted.

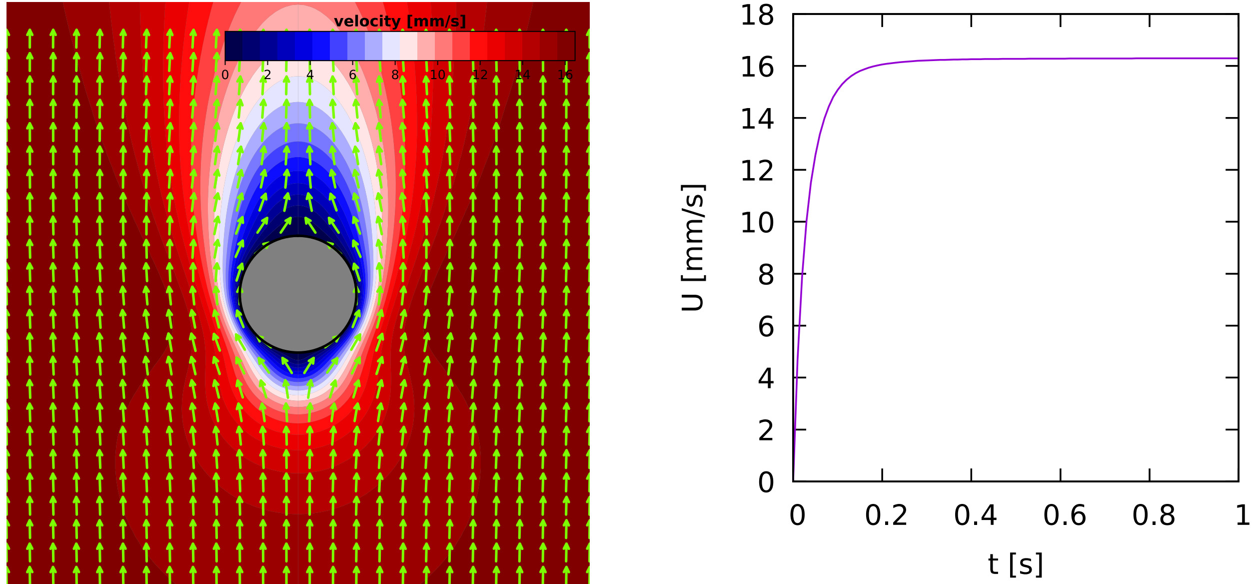

Fig. 5.7: (left) Velocity around a spherical object with consideration of inertia. (right) Evolution of the velocity $U(t)$.

5.3.3 Rayleigh-Taylor instability

Now, we can make use of the power of pyoomph to easily couple the Navier-Stokes equations with an advection-diffusion equation. Here, we will solve a denser fluid residing atop of a less dense fluid. We apply the *Boussinesq approximation*, i.e. the mass density ρ is assumed to be constant in the continuity equation and also in the inertia term, but the bulk force will depend on the varying density. We set the bulk force to $\vec{f} = -100c\vec{e}_y$ and solve for the composition field c that ranges from 0 (bottom light fluid) to 1 (heavier top fluid). We choose a small diffusivity so that advection can dominate. For the equations for the advection-diffusion of c and the velocity-pressure pair, we use the predefined equations from pyoomph. The corresponding classes are `NavierStokesEquations` and `AdvectionDiffusionEquations`. However, we could also use the equations we have developed in the previous sections instead.

```
from pyoomph import *
# Use the pre-defined equations for Navier-Stokes and advection-diffusion
from pyoomph.equations.navier_stokes import *
from pyoomph.equations.advection_diffusion import *


class RayleighTaylorProblem(Problem):
    def __init__(self):
        super(RayleighTaylorProblem,self).__init__()
        self.W,self.H=0.25, 1 # Size of the box
        self.rho,self.mu=0.01, 1 # density and viscosity
        self.Nx=4 # elmenents in x-direction
        self.max_refinement_level=4 # max. 4 times refining

    def define_problem(self):
```

(continues on next page)

(continued from previous page)

```
                # add the mesh
                self.add_mesh(RectangularQuadMesh(size=[self.W,self.H],N=[self.Nx,
↪int(self.Nx*self.H/self.W)]))
                eqs=MeshFileOutput() # output
                bulkforce=100*var("c")*vector(0,-1) # bulkforce: depends on the␣
↪composition c
                # Advection diffusion equation: Advected by the velocity
                eqs+=AdvectionDiffusionEquations(fieldnames="c",wind=var("velocity"),
↪diffusivity=0.0001,space="C1")
                # Navier-Stokes with the c-dependent bulk formce
                eqs+=NavierStokesEquations(bulkforce=bulkforce,dynamic_viscosity=self.mu,
↪mass_density=self.rho)
                # Initial condition
                xrel,yrel=var("coordinate_x")/self.W,var("coordinate_y")/self.H-0.5
                eqs+=InitialCondition(c=tanh(100*(yrel-0.0125*cos(2*pi*xrel))))
                # Refinements based on c and velocity
                eqs+=SpatialErrorEstimator(c=1,velocity=1)
                # Adding no-slip conditions
                for wall in ["left","right","top","bottom"]:
                        eqs+=DirichletBC(velocity_x=0,velocity_y=0) @ wall
                # Fix one pressure degree
                eqs+=DirichletBC(pressure=0)@"bottom/left"
                self.add_equations(eqs@"domain")


if __name__=="__main__":
        with RayleighTaylorProblem() as problem:
                problem.run(10,numouts=50,spatial_adapt=1)
```

If you have read the tutorial up to here, you should understand all steps. The idea is to create a `NavierStokesEquations` object with a body force depending on the variable `var("c")`. This variable is defined and solved in the `AdvectionDiffusionEquations`, which in turn get the field `var("velocity")` for the advection. Thereby, both parts are coupled. Since we allow no normal outflow, we have to fix a single pressure degree, which we do in the lower left corner. The results are depicted in Fig. 5.8.

5.3.4 Marangoni instability

Another interesting instability that can arise in a system that combines the Navier-Stokes equation (velocity $\vec{u}$, pressure p) with an advection-diffusion equation (scalar field c) is the *Marangoni instability*. This instability is driven by a combination of the non-linear advection term $\vec{u}\cdot\nabla c$ and a surface tension $\sigma=\sigma(c)$ that depends on c. Obviously, we must impose a surface tension force, in particular a tangential traction due to potential surface tension gradient along the interface. As in the previous example, we use the predefined classes `NavierStokesEquations` and `AdvectionDiffusionEquations`, where the velocity of the former is used to advect the field c in the latter. The field c just couples back via a tangential traction at the top interface. We will additionally use physical dimensions for this example:

```
from pyoomph import *
# Use the pre-defined equations for Navier-Stokes and advection-diffusion
from pyoomph.equations.navier_stokes import *
from pyoomph.equations.advection_diffusion import *

# Dimensional problem
from pyoomph.expressions.units import *

class MarangoniProblem(Problem):
```

(continues on next page)

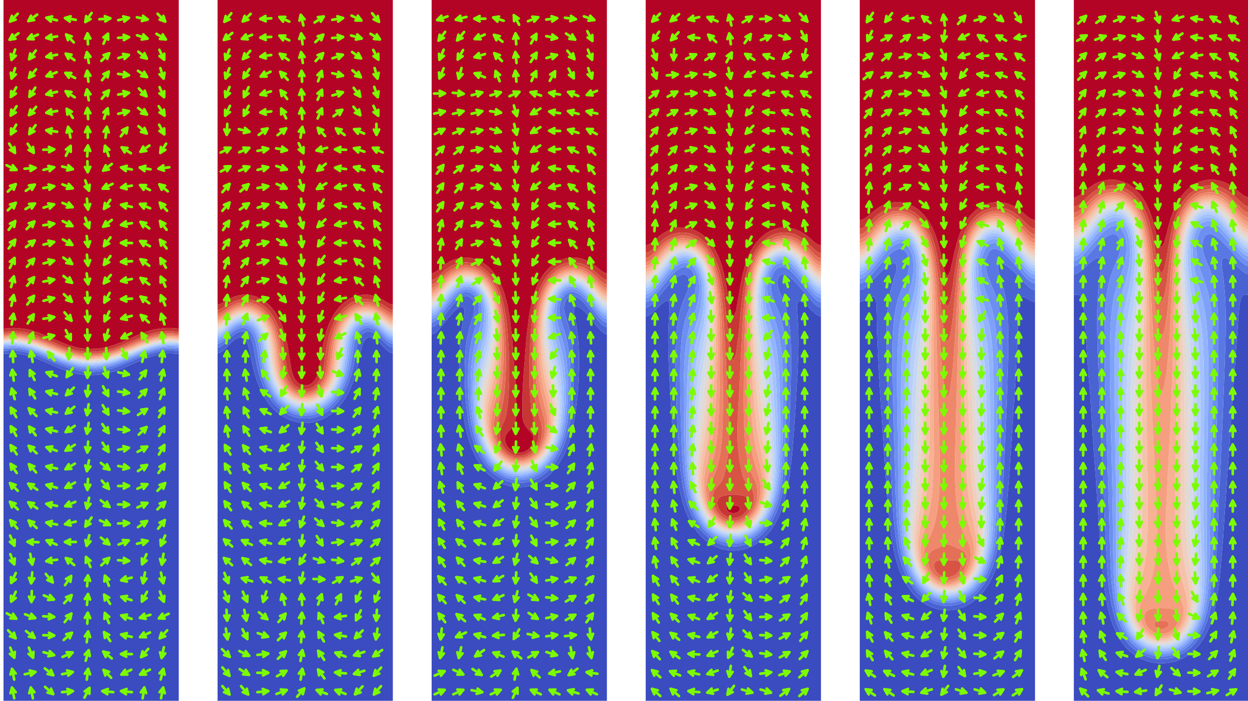

Fig. 5.8: Miscible Rayleigh-Taylor instability by coupling the Navier-Stokes equations with an advection-diffusion equation.

(continued from previous page)

```
    def __init__(self):
            super(MarangoniProblem,self).__init__()
            self.W,self.H=1*milli*meter, 0.25*milli*meter # Size of the box
            self.rho,self.mu=1000*kilogram/meter**3, 1*milli*pascal*second  #␣
↪density and viscosity
            self.D=1e-9*meter**2/second # diffusivity
            self.Nx=10 # elmenents in x-direction
            self.max_refinement_level=3 # max. 4 times refining
            self.dsigma_dc=-0.1*milli*newton/meter

    def define_problem(self):
            self.set_scaling(spatial=self.W,temporal=0.1*second)
            self.set_scaling(velocity=scale_factor("spatial")/scale_factor("temporal
↪"))
            self.set_scaling(pressure=1*pascal)
            # add the mesh
            self.add_mesh(RectangularQuadMesh(size=[self.W,self.H],N=[self.Nx,
↪int(self.Nx*self.H/self.W)]))
            eqs=MeshFileOutput()  # output

            eqs+=AdvectionDiffusionEquations(fieldnames="c",wind=var("velocity"),
↪diffusivity=self.D,space="C1")
            eqs+=NavierStokesEquations(dynamic_viscosity=self.mu,mass_density=self.
↪rho)
            # Initial condition
            xrel,yrel=var("coordinate_x")/self.W,var("coordinate_y")/self.H
            eqs+=InitialCondition(c=yrel*(1+0.01*cos(2*pi*xrel)+0.
↪001*sin(4*pi*xrel)))
```

(continues on next page)

(continued from previous page)

```
            # Refinements based on c and velocity
            eqs+=SpatialErrorEstimator(c=1,velocity=1)
            # Adding no-slip conditions
            for wall in ["left","right","bottom"]:
                    eqs+=DirichletBC(velocity_x=0,velocity_y=0)@wall

            # "Free" surface: fixed y-velocity, Marangoni force
            sigma=self.dsigma_dc*var("c")            # Surface tension
            eqs+=(DirichletBC(velocity_y=0)+NeumannBC(velocity=-grad(sigma)))@"top"
            # Fix one pressure degree
            eqs+=DirichletBC(pressure=0)@"bottom/left"
            self.add_equations(eqs@"domain")


if __name__=="__main__":
     with MarangoniProblem() as problem:
            # problem.dsigma_dc=0.1*milli*newton/meter
            problem.run(1*second,outstep=True,startstep=0.01*second,maxstep=0.
↪1*second,spatial_adapt=1,temporal_error=1)
```

Since we have a dimensional problem now, all quantities as e.g. `rho` and `mu` and also the size of the box `W×H` are dimensional. To nondimensionalize, we have to set the `spatial` and `temporal` scales, as well as the `pressure` and `velocity` scale. The scale of the field `c` is not set, since we use a non-dimensional field for c. Thereby, the problem gets nondimensionalized internally. As initial condition for c, we have a linear gradient in y direction (lower c in the bulk as compared to the `"top"` interface) with a tiny tangential pertubation. The `"left"`, `"right"` and `"bottom"` sides are just no-slip boundary conditions.

At the `"top"` interface, we just set the y-velocity to zero, i.e. the liquid is not allowed to flow out of the domain. Furthermore, we define the surface tension $\sigma = c\,\partial_c\sigma$, i.e. linearly dependent on the field c. The factor `dsigma_dc` controls the direction, i.e. whether the surface tension increases or decreases with c. The absolute value of the surface tension does not matter in this setting, since we only calculate tangential gradients thereof. To apply the Marangoni force, we just add a `NeumannBC` that applies the traction. The minus sign stems from the fact, that the Neumann term $\langle\cdot,\cdot\rangle$ in (4.12) is negative. Since the gradient of the surface tension will only have contributions in x-direction, it is indeed just a tangential contribution, i.e. applied on the x-direction of the velocity. In total, this means that we apply the tangential Marangoni traction $t_x = \partial_x\sigma$.

As apparent from Fig. 5.9, we see a drastic effect of whether $\partial_c\sigma$ is positive or negative. When $\partial_c\sigma$ is negative, any perturbation at the interface will be damped out. Any Marangoni flow will pull up liquids with higher surface tension to the positions with previously lower surface tension. Thereby, the Marangoni flow gets hampered over time. If, however, $\partial_c\sigma > 0$, it is vice versa: Any perturbation will pull up liquid with lower surface tension to spots where already a lower surface tension was before. Thereby, the Marangoni instability is triggered, leading to a self-enhancing Marangoni effect that eventually breaks up into chaotic flow. Of course, if the gradient of c in bulk direction would be inverted, i.e. higher c in the bulk as compared to the interface, also the dynamics would be unstable for $\partial_c\sigma < 0$ and stable for $\partial_c\sigma > 0$.

It is remarkable, that the tiny perturbation and the tiny dependence $\partial_c\sigma$ of the surface tension on the field c is sufficient to trigger this chaotic dynamics. However, the *Marangoni number*, i.e. the nondimensional number to estimate the Marangoni effect, has the viscosity and in particular the diffusivity in the denominator. These both quantities are so small, so that the Marangoni number is in fact large.

The Marangoni instability is the explanation why e.g. evaporating droplets consisting of ethanol and water are chaotic, whereas evaporating droplets consisting of glycerol and water show regular flow.

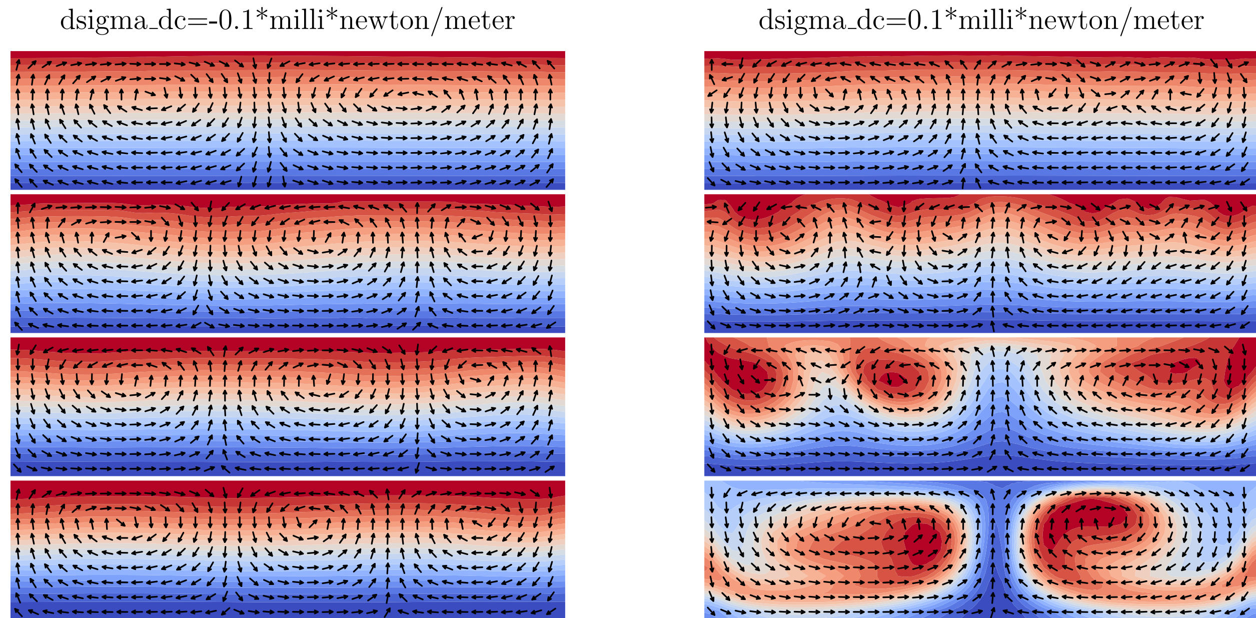

Fig. 5.9: Marangoni (in)stability. (left) When the surface tension decreases with c, i.e. $\partial_c\sigma < 0$, the surface is stable. (right) when σ increases with c, we have initially a higher surface tension than the liquid in the bulk would yield. Any tiny perturbation can pull up liquid from the bulk, locally reducing the surface tension even more and enhancing the perturbation. This can easily result in chaotic dynamics.

5.4 Lubrication equation

The *lubrication approximation* is commonly used for low-Reynolds number flow with a shallow aspect ratio. Instead of solving the (Navier-)Stokes equations with a free interface (which will be done later in Section 6.5), one instead solves the equation for the fluid height $z = h(\vec{x}, t)$ as function of the lateral coordinate $\vec{x}$ and time t. The height of the fluid changes due to pressure gradients, while the viscosity μ and the no-slip boundary condition at $z = 0$ hampers the flow velocity in relaxing these pressure gradients. The corresponding equations, where the Laplace pressure p due to the surface tension σ and potentially the Marangoni effect are driving the flow, read

$$\begin{aligned}\partial_t h + \nabla\cdot\left(-\frac{h^3}{3\mu}\nabla p + \frac{h^2}{2\mu}\nabla\sigma\right) &= 0\\ p + \nabla\cdot(\sigma\nabla h) - \Pi &= 0\end{aligned} \tag{5.6}$$

The pressure is obviously proportional to the approximated curvature in *Monge form*, i.e. approximating the Laplace pressure, and an optional contribution of a *Derjaguin pressure/disjoining pressure* Π. The total volume, i.e. the spatial integral over the height h, is obviously conserved over time.

A weak form with test functions η and q for h and p, respectively, reads

$$\begin{aligned}&(\partial_t h, \eta) + \left(\frac{h^3}{3\mu}\nabla p - \frac{h^2}{2\mu}\nabla\sigma, \nabla\eta\right) + (p - \Pi, q) - (\sigma\nabla h, \nabla q)\\ &-\left\langle \vec{n}\cdot\left(\frac{h^3}{3\mu}\nabla p - \frac{h^2}{2\mu}\nabla\sigma\right), \eta\right\rangle + \langle\vec{n}\cdot(\sigma\nabla h), q\rangle = 0\end{aligned}$$

and the corresponding implementation could look like this:

```
from pyoomph import *
from pyoomph.expressions import *
```

(continues on next page)

(continued from previous page)

```
class LubricationEquations(Equations):
    def __init__(self,sigma=1,mu=1,disjoining_pressure=0):
        super(LubricationEquations, self).__init__()
        self.sigma=sigma
        self.mu=mu
        self.disjoining_pressure=disjoining_pressure

    def define_fields(self):
        self.define_scalar_field("h","C2")
        self.define_scalar_field("p","C2")

    def define_residuals(self):
        h,eta=var_and_test("h")
        p,q=var_and_test("p")
        self.add_residual(weak(partial_t(h),eta)+weak(1/self.mu*(h**3/3*grad(p)-
↪h**2/2*grad(self.sigma)),grad(eta)))
        self.add_residual(weak(p-self.disjoining_pressure,q)-weak(self.
↪sigma*grad(h),grad(q)))
```

Note: It can be beneficial to solve the height evolution equation on the test function of the pressure field and the pressure definition on the test functions of the height field, i.e. swap `eta` and `q` in the `add_residual()` calls. This approach respects the fact that we then choose the test function according to the field with the highest spatial derivative. It is also beneficial if e.g. a Dirichlet condition for the height is imposed somewhere, since it does not require any appropriate Neumann term for the pressure.

With this equation class, we will discuss a few examples in the following:

5.4.1 Relaxation of a perturbation

As a first problem class, let us calculate the relaxation of a thin film with a modulation:

```
class LubricationProblem(Problem):
    def define_problem(self):
        self.add_mesh(LineMesh(N=100)) # simple line mesh
        eqs=LubricationEquations() # equations
        eqs+=TextFileOutput() # output
        eqs+=InitialCondition(h=0.05*(1+0.25*cos(2*pi*var("coordinate_x"))))   #␣
↪small height with a modulation
        self.add_equations(eqs@"domain") # adding the equation


if __name__=="__main__":
    with LubricationProblem() as problem:
        problem.run(50,outstep=True,startstep=0.25)
```

The result is depicted in Fig. 5.10.

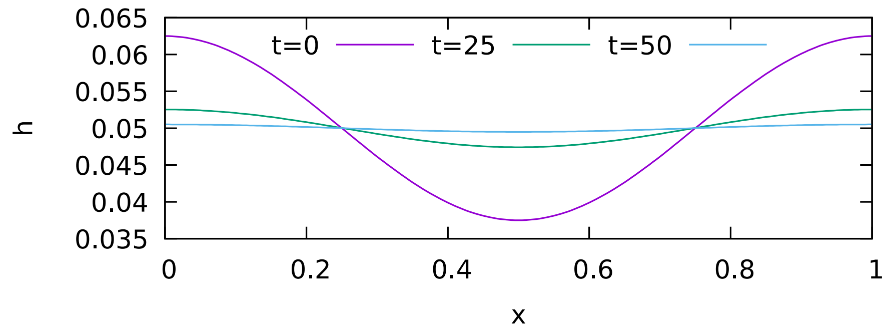

Fig. 5.10: Relaxation of a perturbed surface h in the lubrication limit.

5.4.2 Spreading of a droplet

We can use the same equation class to calculate the spreading of a droplet. For that, we have to switch to an axisymmetric coordinate system and also make sure that the droplet can spread at all. For the latter, we must make sure that the droplet does not end at its radius R with $h(R,t)=0$, since this would exclude any change in the height according to (5.6). A conventional way to resolve this is the addition of a precursor film with a thin thickness compared to the droplet. The thickness of the this film will control the spreading velocity. Additionally, one may add a disjoining pressure. Thereby, one can e.g. enforce the spreading to stop at a finite contact angle:

```
from lubrication import *

class DropletSpreading(Problem):
    def __init__(self):
        super(DropletSpreading,self).__init__()
        self.hp=0.0075 # precursor height
        self.sigma=1 # surface tension
        self.R,self.h_center=1,0.5 # initial radius and height of the droplet
        self.theta_eq=pi/8  # equilibrium contact angle


    def define_problem(self):
        self.set_coordinate_system("axisymmetric")
        self.add_mesh(LineMesh(N=500,size=5)) # simple line mesh

        h=var("h") # Building disjoining pressure
        disjoining_pressure=5*self.sigma*self.hp**2*self.theta_eq**2*(h**3 -
↪self.hp**3)/(3*h**6)

        eqs=LubricationEquations(sigma=self.sigma,disjoining_pressure=disjoining_
↪pressure) # equations
        eqs+=TextFileOutput() # output
        h_init=maximum(self.h_center*(1-(var("coordinate_x")/self.R)**2),self.
↪hp) # Initial height
        eqs+=InitialCondition(h=h_init)

        self.add_equations(eqs@"domain") # adding the equation


if __name__=="__main__":
    with DropletSpreading() as problem:
        problem.run(1000,outstep=True,startstep=0.01,maxstep=10,temporal_error=1)
```

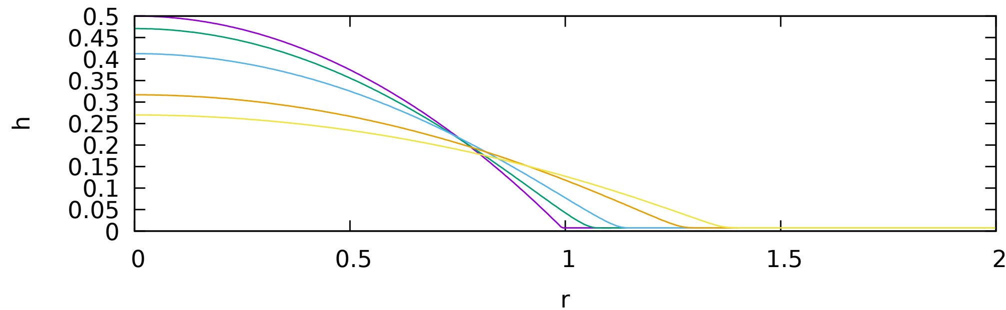

Fig. 5.11: Spreading of a droplet until the equilibrium contact angle is reached, which is enforced by the disjoining pressure.

5.4.3 Coalescence of droplets

For a reasonable coalescence, we have to solve the lubrication equation on a two-dimensional lateral plane. Due to symmetry, only one half of this plane is solved. Of course, it is beneficial to use the spatial adaptivity to resolve the domain accurately and optimizing the required computational effort:

```
from lubrication_spreading import * # Import the previous example problem

class DropletCoalescence(DropletSpreading):
    def __init__(self):
        super(DropletCoalescence,self).__init__()
        self.distance=2.5 # droplet distance
        self.Lx=7.5
        self.max_refinement_level=6


    def define_problem(self):
        self.add_mesh(RectangularQuadMesh(N=[10,5],size=[self.Lx,self.Lx/2],
↪lower_left=[-self.Lx*0.5,0]))

        h=var("h") # Building disjoining pressure
        disjoining_pressure=5*self.sigma*self.hp**2*self.theta_eq**2*(h**3 -␣
↪self.hp**3)/(3*h**6)

        eqs=LubricationEquations(sigma=self.sigma,disjoining_pressure=disjoining_
↪pressure) # equations
        eqs+=MeshFileOutput() # output
        x=var("coordinate")
        dist1=x-vector(-self.distance/2,0) # distance to the centers of the␣
↪droplets
        dist2=x-vector(self.distance/2,0)
        h1=self.h_center*(1-dot(dist1,dist1)/self.R**2) # height functions of␣
↪the droplets
        h2=self.h_center*(1-dot(dist2,dist2)/self.R**2)
        h_init=maximum(maximum(h1,h2),self.hp) # Initial height: maximum of h1,␣
↪h2 and precursor
        eqs+=InitialCondition(h=h_init)

        eqs+=SpatialErrorEstimator(h=1) # refine based on the height field

        self.add_equations(eqs@"domain") # adding the equation
```

(continues on next page)

(continued from previous page)

```
if __name__=="__main__":
    with DropletCoalescence() as problem:
        problem.run(1000,outstep=True,startstep=0.01,maxstep=10,temporal_error=1,
↪spatial_adapt=1)
```

We just reuse the previous problem by inheritance to get access to the parameters as e.g. `R`, `sigma`, etc. Of course, the parameter `distance` and the size of the mesh `Lx` is additionally required. With the `max_refinement_level` of the `Problem` base class, the maximum refinement is controlled. The rest is analogous to the previous example, however, in Cartesian coordinates with a 2d mesh and with two droplets.

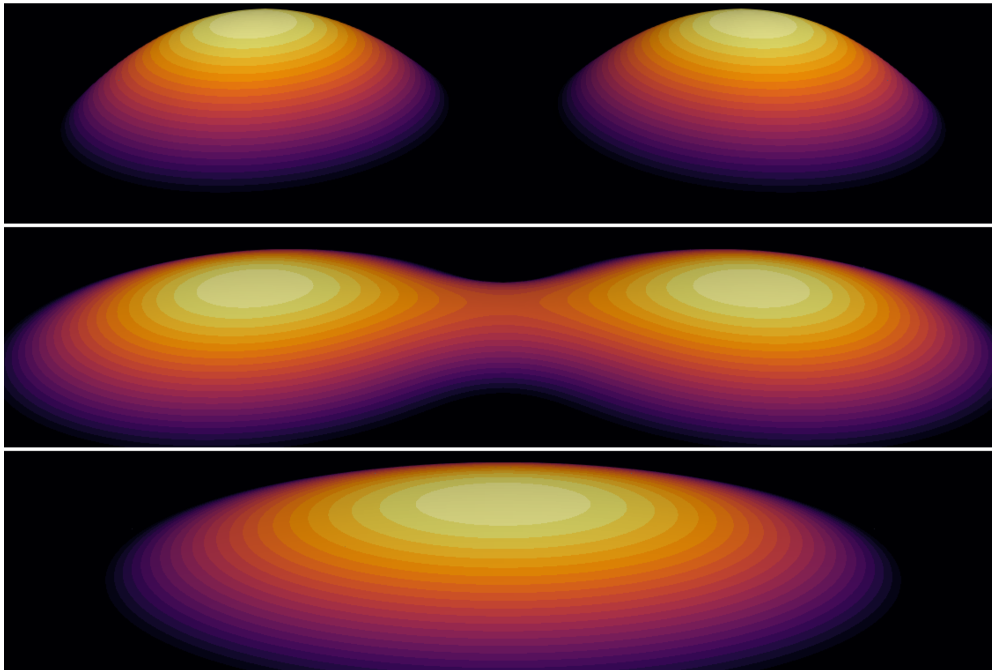

Fig. 5.12: Coalescence of two droplets.

One can rather easily add e.g. (in)soluble surfactants or a mixture composition field by adding a corresponding advection-diffusion field on the domain. When redefining the surface tension `sigma` to be dependent on this additional field, it is easy to reproduce *delayed coalescence* due to Marangoni dynamics. Similarly, it is also straight-forward to use dimensions here and use the non-dimensionalization in pyoomph to solve the dynamics of real droplets.

5.5 Continuing a stopped simulation

Spatio-temporal simulations can be time consuming and you might have to reboot for whatever reasons, causing your simulation to stop. In that case, you can continue the simulation at the last written output by adding the command line argument `--runmode c` to the call:

```
python MY_SIMULATION.py --runmode c
```

where `MY_SIMULATION.py` is of course your simulation script. You can only continue if the property `write_states` is `True`, which is the default. The state files (found in the sub-directory `_states` of your output directory) contain all information of the current state of the simulation which allows to continue.

Warning: Continuing a stopped simulation does not really work if you do stationary solves, arc length continuations, etc. It only works correcly if you just use the `run()` call to perform a temporal integration of the system. In future, it might change to be more flexible.

5.6 Pattern formation, stability analysis and bifurcation tracking

Nonlinear PDEs are often analyzed in terms of bifurcations, stability, hysteresis, etc. To that end, one usually finds stationary solutions and investigate the stability of these by linear stability analysis. However, whenever the stationary solution is complicated or not even analytically available, this process has to be done numerically. This can easily be done in pyoomph.

5.6.1 Damped Kuramoto-Sivashisky equation with periodic boundaries

Before delving into the stability analysis and bifurcation tracking, let us first define an interesting equation and integrate it temporally with pyoomph to see potential behavior of the equation. The example equation here will be the linearly and quadratically damped *Kuramoto-Sivashinsky equation*

$$\partial_t h = -\gamma h - \delta h^2 - \nabla^2 h - \nabla^4 h + (\nabla h)^2 \tag{5.7}$$

This equation is already in its rescaled form with two nondimensional parameters γ and δ controlling the damping. The field h can be interpreted as a height function. This equation has been proposed to reproduce pattern formation found in low-energy ion beam erosion of semiconductor surfaces, e.g. in Refs. [2, 15, 18].

When casting the equation to a weak form for pyoomph, we have to bear in mind that pyoomph does not allow for spatial derivatives beyond first order. Hence, we define $g = \nabla^2 h$ so that we obtain

$$\begin{aligned} \partial_t h &= -\gamma h - \delta h^2 - \nabla^2 h - \nabla^2 g + (\nabla h)^2 \\ g &= \nabla^2 h \end{aligned}$$

and cast it into the weak formulation with test functions v and w for h and g, respectively:

$$\begin{aligned} \left(\partial_t h + \gamma h + \delta h^2 + (\nabla h)^2, v\right) + (g, w) - (\nabla h + \nabla g, \nabla v) + (\nabla h, \nabla w) \\ + \langle \nabla h + \nabla g, \vec{n} v \rangle - \langle \nabla h, \vec{n} w \rangle = 0 \end{aligned}$$

Alternatively, of course, one could also add g to the first weak contribution term instead of the ∇h term in the third contribution to account for the $-\nabla^2 h$ term in (5.7), which would yield different Neumann terms.

The implementation is straight-forward:

```
from pyoomph import *
from pyoomph.expressions import *
from pyoomph.expressions.utils import DeterministicRandomField  # for the random␣
↪initial condition


class DampedKuramotoSivashinskyEquation(Equations):
    def __init__(self, gamma=0.0, delta=0.0, space="C2"):
        super(DampedKuramotoSivashinskyEquation, self).__init__()
        self.gamma, self.delta, self.space = gamma, delta, space

    def define_fields(self):
```

(continues on next page)

(continued from previous page)

```
        self.define_scalar_field("h", "C2")  # h
        self.define_scalar_field("lapl_h", "C2")  # projection of div(grad(h))

    def define_residuals(self):
        h, v = var_and_test("h")
        lapl_h, w = var_and_test("lapl_h")
        self.add_residual(weak(partial_t(h) + self.gamma * h + self.delta * h ** 2 -␣
↪dot(grad(h), grad(h)), v))
        self.add_residual(-weak(grad(h) + grad(lapl_h), grad(v)))
        self.add_residual(weak(lapl_h, w) + weak(grad(h), grad(w)))
```

For the problem, we want to use two new features, namely periodic boundaries and a random initial condition. We use a `RectangularQuadMesh` and connect the `"left"` with the `"right"` interface and the `"top"` with the `"bottom"` interface, so that the domain is virtually infinite in all directions due to periodicity. Thereby, there is no single Neumann term relevant. This can be done with the `PeriodicBC`, which must be added to an interface and gets the opposite interface as first argument. Furthermore, we must tell pyoomph, how to find the corresponding node on the other boundary to connect these. We can just pass an `offset`, so that each pair of nodes on both connected periodic boundary pair is found by applying this offset to the position of the source node to the destination node:

```
class KSEProblem(Problem):
    def __init__(self):
        super(KSEProblem, self).__init__()
        self.L = 50  # domain length
        self.N = 40  # number of elements
        self.gamma,self.delta=0.24,0.05 # parameters
        self.random_amplitude=0.01 # Initial random initial condition amplitude

    def define_problem(self):
        self.add_mesh(RectangularQuadMesh(N=self.N, size=self.L))

        eqs = DampedKuramotoSivashinskyEquation(gamma=self.gamma, delta=self.delta)
        eqs += MeshFileOutput()
        # Adding periodic boundaries: nodes at "bottom" will be merged by the nodes␣
↪at top (found by applying offset to the position)
        eqs += PeriodicBC("top", offset=[0, self.L]) @ "bottom"
        # Same for the left<->right connection
        eqs += PeriodicBC("right", offset=[self.L, 0]) @ "left"

        # Create a deterministic random field. We must pass the corners of the domain
        # All random values will be pre-allocated so that successive evaluations of
        # the functions at the same point yield the same value
        h_init = DeterministicRandomField(min_x=[0, 0], max_x=[self.L, self.L],␣
↪amplitude=self.random_amplitude)
        x, y = var(["coordinate_x", "coordinate_y"])
        eqs += InitialCondition(h=h_init(x, y))

        self.add_equations(eqs @ "domain")  # adding the equation
```

> **Warning:** The `PeriodicBC` object enforces periodicity to all fields defined on this domain. It is hence not possible to have e.g. one field periodic and another one discontinuous across the interface with the `PeriodicBC` object.

Additionally, note that we use a `DeterministicRandomField` to create our initial condition. Since pyoomph requires that successive function calls with the same arguments yield the same values (i.e. deterministic functions), it is necessary to precalculate the random numbers in advance. This is done internally in the `DeterministicRandom-`

`Field`. To that end, we must specify the minimum and maximum coordinates, so that internally an n-dimensional array of random numbers with the prescribed `amplitude` is created. Whenever the function is evaluated, it is interpolated between the initially generated random numbers to ensure the deterministic requirement.

The problem code is simple and representative results of the pattern formation are shown in Fig. 5.13:

```
if __name__ == "__main__":
    with KSEProblem() as problem:
        problem.run(2000, outstep=True, startstep=0.1, temporal_error=1, maxstep=50)
```

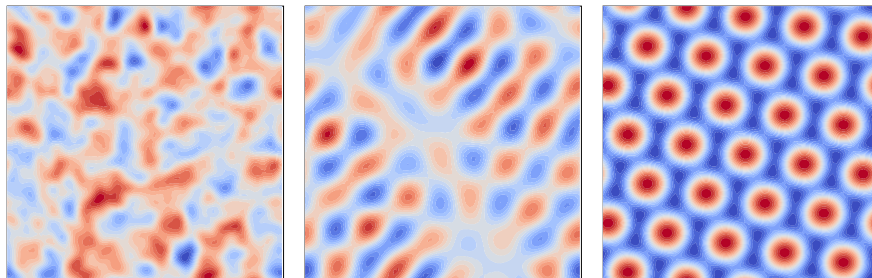

Fig. 5.13: Emergence of a hexagonal dot pattern by the damped Kuramoto-Sivashinsky equation starting from a random initial condition.

5.6.2 Stability via eigenvalues

When simulating the damped Kuramoto-Sivashinsky for different values of $\gamma \geq 0$ and $\delta \geq 0$, one will notice that not only hexagonal dot patterns can emerge, but also hexagonal hole patters, stripe patters, but also even chaotic solutions can occur. Furthermore, when starting with a small `amplitude` of the random initial condition, just the simple solution $h = 0$ will be present if $\gamma > 1/4$. It is cumbersome to temporally integrate the equation for each combination of γ and δ to create a phase diagram of the emerging solutions. Instead, one can use pyoomph's features of eigenvalue calculation and bifurcation tracking to investigate the parameter space quickly.

To that end, we start with stationary solutions and test them for stability. However, we must have a good initial guess for the stationary solutions, since the stationary `solve()` command would not converge if we are too far off. In particular, it is helpful to know the typical length scale (or wave number k) of the emerging patterns. This can be done by linearizing (5.7) around $h = 0$ and investigate the spatially Fourier transformed equation:

$$\partial_t \tilde{h}(\vec{k}, t) = \left(-\gamma + k^2 - k^4\right) \tilde{h}(\vec{k}, t)$$

Obviously, when $\gamma > 1/4$, the dispersion relation $-\gamma + k^2 - k^4$ is negative for all $k = \|\vec{k}\|$. Therefore, all initial modes will decay over time. If γ is slightly below $1/4$, the dispersion relation will have a positive maximum at the critical wave number $k_\mathrm{c} = k = 1/\sqrt{2}$. Hence, we expect that the typical size of the emerging patterns, at least close to $\gamma = 1/4$, will be controlled by this wave number. Thereby, we can formulate several initial conditions, which will be used to find stationary solutions:

$$\begin{aligned}
h_\text{flat} &= 0 \\
h_\text{stripes} &= \frac{A}{2} \cos\left(k_\mathrm{c} x\right) \\
h_\text{hexdots} &= \frac{2A}{9} \left(\cos\left(k_\mathrm{c} x\right) + 2\cos\left(\frac{k_\mathrm{c}}{2} x\right) \cos\left(\frac{\sqrt{3}}{2} k_\mathrm{c} y\right)\right) \\
h_\text{hexholes} &= -\frac{2A}{9} \left(\cos\left(k_\mathrm{c} x\right) + 2\cos\left(\frac{k_\mathrm{c}}{2} x\right) \cos\left(\frac{\sqrt{3}}{2} k_\mathrm{c} y\right)\right)
\end{aligned}$$

where the amplitude A can be used to control the amplitude of the initial conditions.

Of course, the domain size must be chosen that the initial conditions fit in perfectly. We therefore construct a new `Problem` class:

```
from kuramoto_sivanshinsky import * # Import the previous script

class KSEBifurcationProblem(Problem):
    def __init__(self):
        super(KSEBifurcationProblem, self).__init__()
        self.periods,self.period_y_factor=2,1 # consider two full periods in both␣
↪directions
        self.N_per_period=20 # elements per period and direction
        self.kc=1/square_root(2) # wave number of the pattern
        # Introduce parameters
        # for both linear and quadratic damping with some initial settings
        self.param_gamma=self.define_global_parameter(gamma=0.24)
        self.param_delta=self.define_global_parameter(delta=0)


    def define_problem(self):
        kc, N = self.kc, self.N_per_period*self.periods
        Lx = self.periods * 4 * pi / kc # Calulate fitting mesh size
        Ly = self.period_y_factor*self.periods* 4 * square_root(1/3) * pi / kc
        mesh = RectangularQuadMesh(size=[Lx, Ly], N=[N, int(N* Ly / Lx)])
        self.add_mesh(mesh)

        eqs=MeshFileOutput()
        eqs+=DampedKuramotoSivashinskyEquation(gamma=self.param_gamma,delta=self.
↪param_delta)

        # Register different initial conditions
        A=3
        x,y=var(["coordinate_x","coordinate_y"])
        eqs += InitialCondition(h=0,IC_name="flat")
        eqs += InitialCondition(h=2*A/9*(cos(kc*x)+2*cos(kc/2*x)*cos(kc*square_
↪root(3)/2*y)),IC_name="hexdots")
        eqs += InitialCondition(h=-2 * A / 9 * (cos(kc * x) + 2 * cos(kc / 2 * x) *␣
↪cos(kc * square_root(3) / 2 * y)),IC_name="hexholes")
        eqs += InitialCondition(h= A / 2 * cos(kc * x),IC_name="stripes")

        # And integral observables, in particular h_rms
        eqs += IntegralObservables(_area=1,_h_integral=var("h"),_h_sqr_integral=var("h
↪")**2)
        eqs += IntegralObservables(h_avg=lambda _area,_h_integral : _h_integral/_area)
        eqs += IntegralObservables(h_rms=lambda _area, h_avg,_h_sqr_integral: square_
↪root(_h_sqr_integral/_area - h_avg**2))
```

The parameters γ and δ are now bound as parameters, i.e. we can change the values dynamically during the simulation. Moreover, we pass `IC_name` arguments to the `InitialCondition` objects. Thereby, we can later on set the different starting conditions by `set_initial_condition()`. The four initial conditions are plotted in Fig. 5.14.

One might wonder why we do not add `PeriodicBC` boundaries here. The reason is that we later on want to calculate stationary solutions and eigenvalues. Since (5.7) is invariant with respect to a shift of the coordinate system, any stationary solution h_0 would automatically imply an infinite set of stationary solutions $h_{0,\vec{s}}(\vec{x}) = h_0(\vec{x} - \vec{s})$. And each of these solutions would have eigenvalues of zero corresponding to this shift, i.e. with eigenfunctions $\nabla h_0 \cdot s$. This would hamper the stability analysis tremendously. Instead, we fix the arbitrary shift (and the rotational freedom due to the isotropy of (5.7)) by omitting the `PeriodicBC`. Thereby, zero Neumann fluxes will be imposed a the boundaries, i.e. $\partial_x h = \partial_x^3 h = 0$ at the `"left"` and `"right"` boundaries and $\partial_y h = \partial_y^3 h = 0$ at the `"top"` and `"bottom"` boundaries will be present.

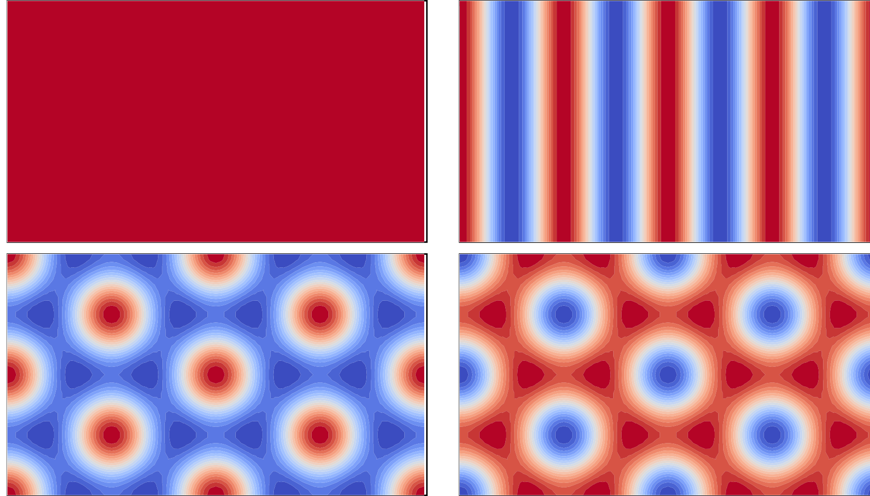

Fig. 5.14: Initial conditions used as initial guesses for the stationary solutions.

Furthermore, we add `IntegralObservables` here. These are observables of the type

$$I = \int f(\vec{x})\, \mathrm{d}^{(n)}x\,, \tag{5.8}$$

i.e. we integrate the argument over the `"domain"` here. Thereby, we can calculate the `"_area"` of the `"domain"` by integrating over `1`. Furthermore, we integrate over h and store it in `"_h_integral"`. The underscore just indicates that its output should be suppressed, since both the `"_area"` and the `"_h_integral"` are just helper observables we require to determine the root mean square of the field. The root mean square (rms) is given by

$$\mathrm{rms}(h) = \sqrt{\frac{\int (h-h_\mathrm{avg})^2\,\mathrm{d}^2x}{\int 1\,\mathrm{d}^2x}} \quad \text{with the avg. height} \quad h_\mathrm{avg} = \frac{\int h\,\mathrm{d}^2x}{\int 1\,\mathrm{d}^2x}$$

Obviously, the terms cannot be written in a form like (5.8), but we can use `lambda` calls to evaluate mathematical expressions based on integral observables. h_avg is obviously just the quotient of `"_h_integral"` and `"_area"`, so we bind it via the `lambda` as such. For the rms, we first write it as equivalent

$$\mathrm{rms}(h) = \sqrt{\frac{\int h^2\,\mathrm{d}^2x}{\int 1\,\mathrm{d}^2x} - h_\mathrm{avg}^2}$$

and bind this observable via a `lambda`.

First, let us investigate only the case $\delta = 0$ for varying γ:

```
# slepc eigensolver is more reliable here
import pyoomph.solvers.petsc # Requires petsc4py, slepc4py. Might not work in Windows

if __name__ == "__main__":
    with KSEBifurcationProblem() as problem:
        problem.initialise()
        problem.param_gamma.value=0.24
        problem.param_delta.value = 0.0
        problem.set_initial_condition(ic_name="hexdots") # set the hexdot initial␣
↪condition
        problem.solve(timestep=10) # One transient step to converge towards the␣
↪stationary solution
        problem.solve() # stationary solve
```

(continues on next page)

(continued from previous page)

```
        problem.set_eigensolver("slepc") # Set the slepc eigensolver
        # Write eigenvalues to file
        eigenfile=open(os.path.join(problem.get_output_directory(),"hexdots.txt"),"w")
        def output_with_eigen():
            eigvals,eigvects=problem.solve_eigenproblem(6,shift=0) # solve for 6␣
↪eigenvalues with zero shift
            h_rms=problem.get_mesh("domain").evaluate_observable("h_rms") # get the␣
↪root mean square
            line=[problem.param_gamma.value,h_rms,eigvals[0].real,eigvals[0].imag] #␣
↪line to write
            eigenfile.write("\t".join(map(str,line))+"\n") # write to file
            eigenfile.flush()
            problem.output_at_increased_time() # and write the output

        # Arclength continuation
        output_with_eigen()
                while problem.param_gamma.value>0.23:
                    ds=problem.arclength_continuation(problem.param_gamma,ds,max_
↪ds=0.005)
                    output_with_eigen()
```

We use another eigensolver, provided by the PETSc/SLEPc package. These can be installed as explained in Section 2.4. These packages might not be available in Windows. Just give it a try. If these packages cannot be installed, you can omit the import and the `set_eigensolver()` call to use the default `scipy` eigensolver.

We then jump on the stationary solution by a stationary `solve()` command. However, before we step a bit in the direction with a transient solve command, since we might otherwise converge into the flat solution $h = 0$. We perform an `arclength_continuation()` along γ and output the eigenvalue with the largest real part and the calculated rms to a text file. Based on the real part of the eigenvalue, we can determine whether the stationary solution is stable or not. The results are depicted in Fig. 5.15, where we also include the flat solution, which stability has been investigate analytically before.

The rms is used as y-axis to show the amplitude of the patterns. Obviously, for $\delta = 0$, hexagonal dot structures cease to exist for $\gamma \geq 0.2826$ in a fold bifurcation. Between $\gamma = 0.25$ and this value, both the flat solution and hexagonal dot solutions co-exists with a hysteretic behavior in this range.

5.6.3 Stability via bifurcation tracking

We could now perform similar scans for different δ, but there is a simpler route, namely bifurcation tracking. We can instruct pyoomph to find the fold bifurcation and its corresponding value of $\gamma \approx 0.2826$ directly:

```
from kuramoto_sivanshinsky_arclength_eigen import * # Import the previous problem

if __name__ == "__main__":
    with KSEBifurcationProblem() as problem:
        # Output the zeroth eigenvector. Will only output if the eigenvalue/vector is␣
↪calculated either by
        # solve_eigenproblem or by bifurcation tracking
        problem.additional_equations+=MeshFileOutput(eigenvector=0,eigenmode="real",
↪filetrunk="eigen0_real")@"domain"

        problem.initialise()
        problem.param_gamma.value=0.24
        problem.param_delta.value = 0.0
```

(continues on next page)

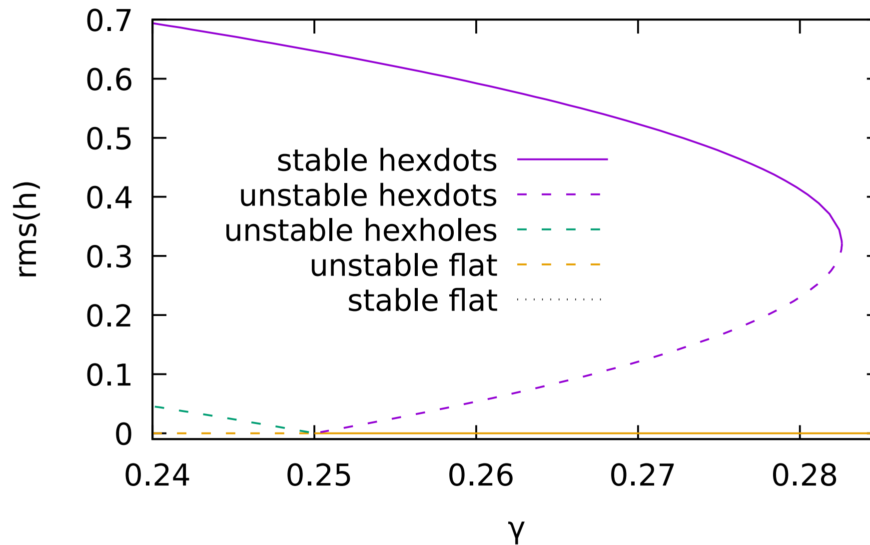

Fig. 5.15: Fold bifurcation at $\gamma \approx 0.2826$, where hexagonal dots stops to exist. For higher γ, only the flat solution exists. Below $\gamma < 0.25$, hexagonal dot structures are the generic solution, at least in the shown interval. The hexagonal hole branch is unstable and meets the flat solution in a transcritical bifurcation.

(continued from previous page)

```
        problem.set_initial_condition(ic_name="hexdots")
        problem.solve(timestep=10) # One transient step to converge towards the␣
↪stationary solution
        problem.solve() # stationary solve

        # from the previous example we know that the fold bifurcation happens close␣
↪to 0.28
        problem.param_gamma.value=0.28
        problem.solve() # solve at gamma=0.28

        # Activate bifurcation tracking
        problem.activate_bifurcation_tracking(problem.param_gamma,"fold")
        problem.solve()
        print("FOLD BIFURCATION HAPPENS AT",problem.param_gamma.value)
```

To that end, we first move close to the bifurcation, i.e. to $\gamma = 0.28$ and `solve()` to find a good guess. Then, we `activate_bifurcation_tracking()` for a `"fold"` bifurcation in γ. Within the next `solve()` command, the value of γ will be adjusted (i.e. γ is in fact a degree of freedom) so that the system is directly at the fold bifurcation. We also output the eigenvector directly at the fold bifurcation. To that end, another `MeshFileOutput` is added, but with the arguments `eigenvector=0` (meaning the zeroth eigenvector) and `eigenmode="real"` (i.e. considering the real part, although this particular eigenvector is real anyhow). We furthermore must supply a `filetrunk` to prevent overwriting of the output files of the solution itself.

Once we are on the bifurcation, we can sweep over δ and follow the position of the fold bifurcation. As long as the bifurcation tracking is active γ will be adjusted to stay on the fold bifurcation, i.e. we get a curve $\gamma_{\text{fold}}(\delta)$, which is written to file:

```
hexfold_file = open(os.path.join(problem.get_output_directory(), "hexfold.txt"), "w")
def output_with_params():
    h_rms = problem.get_mesh("domain").evaluate_observable("h_rms")  # get the root␣
↪mean square
    line = [problem.param_gamma.value, problem.param_delta.value,h_rms]  # line to␣
```

(continues on next page)

(continued from previous page)

```
↪write
    hexfold_file.write("\t".join(map(str, line)) + "\n")   # write to file
    hexfold_file.flush()
    problem.output_at_increased_time()   # and write the output

output_with_params()
ds = 0.025
while problem.param_delta.value < 0.5:
    ds = problem.arclength_continuation(problem.param_delta, ds, max_ds=0.025)
    output_with_params()
```

The result, i.e. the location of the fold bifurcation, is depicted in Fig. 5.16.

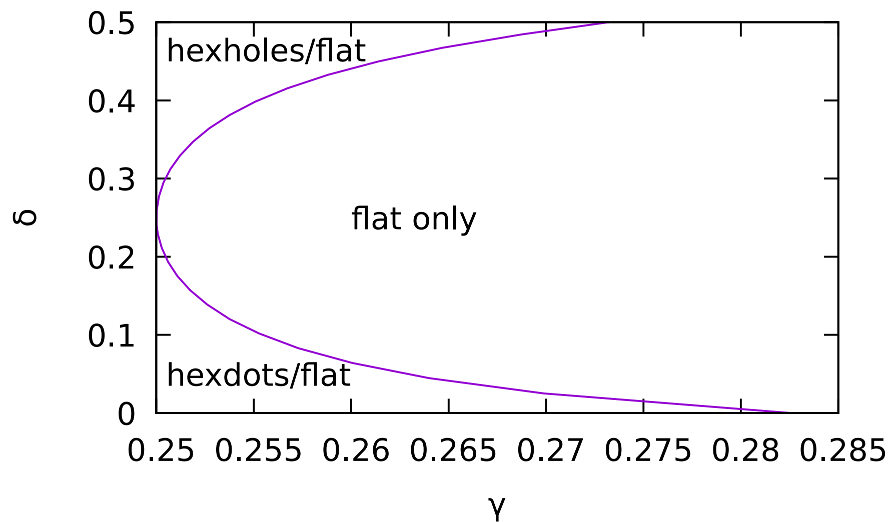

Fig. 5.16: Emergence of a hexagonal dot pattern by the damped Kuramoto-Sivashinsky equation starting from a random initial condition.

Similarly, we can set the other `InitialCondition` to start with hexagonal holes or stripe patterns and find the bifurcations.

CHAPTER SIX

MOVING MESH (ALE) METHODS

Until now, we have always considered a static mesh. However, when e.g. a droplet evaporates, it will shrink over time. Of course, one can always take a static domain and solve e.g. a phase field along with the flow, i.e. with a Navier-Stokes-Cahn-Hilliard model (cf. e.g. Demont *et al.* [11]). While this has the benefit that the position of the interface is not required to be known at all and topological changes (coalescence, pinch-off) are possible, it is more complicated to impose e.g. Marangoni forces or to add surfactants directly on the interface. As an alternative, pyoomph allows to formulate equations for the mesh coordinates, which are solve along with all other equations. Thereby, one can e.g. define an equation system that solves the governing equations on a moving mesh and couple the mesh motion with the underlying dynamics, i.e. via a kinematic boundary condition. With this approach, it is possible to account for an evaporating droplet with a sharp and well resolved interface, where Marangoni stresses and surfactants can be added easily. Since the mesh is neither fixed in space (Eulerian), nor necessarily co-moving with the fluid velocity (Lagrangian), the approach discussed here is also called *Arbitrary Lagrangian-Eulerian*, abbreviated *ALE*.

We will develop this method in the following, address the necessary correction of the temporal derivative and mesh reconstruction at larger deformations. Afterwards, we provide a few examples:

6.1 Lagrangian coordinates

When equations for the mesh position are formulated, the positions of the mesh nodes become part of the equations, i.e. they become dependent variables. The corresponding interpolated field can be obtained by the variable `var("mesh")` (vectorial) or `var("mesh_x")`, `var("mesh_y")` and `var("mesh_z")` for the individual components. However, since these are now dependent variables, it is beneficial to also have a fixed coordinate system, i.e. independent variables. These are the so-called *Lagrangian coordinates*, accessible with `var("lagrangian")` and `var("lagrangian_x")`, `var("lagrangian_y")`, `var("lagrangian_z")`, respectively (see `var()` for more details). These coordinates are initialized by default with the initial mesh positions, but they do not change when the mesh moves. A mesh node at a coordinate $\vec{x}_0$ may move to another position, e.g. $\vec{x}_1$, but it will still have the same Lagrangian coordinate afterwards. In that sense, the Lagrangian coordinates move along with the mesh motion - they are attached to the mesh. Let us denote the Lagrangian coordinates with $\vec{\xi}$, then the Eulerian mesh coordinates $\vec{x}$ can be described as a function of $\vec{\xi}$ and time t, i.e.

$$\vec{x} = \vec{x}\left(\vec{\xi}, t\right)$$

We can hence formulate equations for the mesh coordinates by describing them by Lagrangian coordinates. To that end, we can also calculate spatial derivatives, e.g. `grad()` and `div()`, with respect to the Lagrangian coordinates instead the Eulerian ones. This is done by adding the keyword argument `lagrangian=True` to the calls of `grad()` and `div()`. Likewise, when assembling weak forms, we can also integrate over the Lagrangian domain instead of the Eulerian one by adding the keyword argument `lagrangian=True` to the calls of `weak()`.

6.2 Laplace smoothed mesh

Since the Lagrangian coordinates are initialized with the undeformed initial Eulerian coordinates, we can define the displacement from the initial configuration as $\vec{d} = \vec{x} - \vec{\xi}$. One can smooth this displacement by solving a Laplace equation for $\vec{d}$, i.e. $\nabla_\xi^2 \vec{d} = 0$, where ∇_ξ denotes the derivatives with respect to the Lagrangian coordinates. Thereby, any deformation that is imposed e.g. on the boundaries, will be smooth out along the mesh.

The weak formulation with test function $\vec{\chi}$ reads

$$\left(\nabla_\xi \left(\vec{x}-\vec{\xi}\right), \nabla_\xi \vec{\chi}\right)_\xi - \left\langle \vec{n}_\xi \cdot \nabla_\xi \left(\vec{x}-\vec{\xi}\right), \vec{\chi}\right\rangle_\xi = 0 \tag{6.1}$$

Let use hence define this equation class:

```
from pyoomph import *
from pyoomph.expressions import *

class LaplaceSmoothedMesh(Equations):

    def define_fields(self):
        # let the mesh coordinates become a variables, approximated with second␣
↪order Lagrange basis functions
        self.activate_coordinates_as_dofs(coordinate_space="C2")

    def define_residuals(self):
        x,xtest=var_and_test("mesh") # Eulerian mesh coordinates
        xi=var("lagrangian") # fixed Lagrangian coordinates
        d=x-xi # displacement
        # Weak formulation: gradients and integrals are carried out with respect␣
↪to the Lagrangian coordinates
        self.add_residual(weak(grad(d,lagrangian=True), grad(xtest,␣
↪lagrangian=True),lagrangian=True) )
```

We do not define fields, but we activate the mesh coordinates as dependent variables with the call `activate_coordinates_as_dofs()`. You can pass an argument `coordinate_space` to select the space. If further fields are added, the coordinate space must at least comprise the highest space of all defined fields, i.e. we cannot have a `coordinate_space` of `"C1"` and defining other fields on the space `"C2"`. If the argument is omitted, the coordinate space will be automatically determined by the highest space of all added fields. The rest is trivial, except the usage of the variables `"mesh"` and `"lagrangian"` and the keyword arguments `lagrangian=True` to the `grad()` and `weak()` calls.

As an example problem, let us deform a rectangular mesh by prescribing Dirichlet boundary conditions at the interfaces and let the internal mesh relax based on the Laplace smoothing:

```
class LaplaceSmoothProblem(Problem):
    def define_problem(self):
        self.initial_adaption_steps=0
        self.add_mesh(RectangularQuadMesh(N=6))
        eqs=LaplaceSmoothedMesh()
        eqs+=MeshFileOutput()
        eqs+=DirichletBC(mesh_x=0,mesh_y=True)@"left" # fix the mesh at x=0 and␣
↪keep y in place
        eqs+=DirichletBC(mesh_x=True,mesh_y=0)@"bottom" # fix the mesh at y=0␣
↪and keep x in place
        xi=var("lagrangian") # Lagrangian coordinate
        eqs+=DirichletBC(mesh_x=1+0.5*xi[1])@"right" # linear slope at the left
        eqs+=DirichletBC(mesh_y=1+0.25*xi[0]*(1-xi[0]))@"top" # quadratic␣
↪deformation at the top
```

(continues on next page)

(continued from previous page)

```
            eqs+=SpatialErrorEstimator(mesh=1) # Adapt where large deformations are␣
↪present

            self.add_equations(eqs@"domain")

if __name__=="__main__":
    with LaplaceSmoothProblem() as problem:
            problem.output()
            problem.solve(spatial_adapt=4)
            problem.output_at_increased_time()
```

A few new things occur here. First, we set the property `initial_adaption_steps` of the problem class to `0`. This controls the initial adaption, i.e. the adaption steps taken after the first solve. We deactivate this to get the middle mesh in Fig. 6.1. If this is not set, but a `SpatialErrorEstimator` is present, pyoomph will already adapt with respect to the initial condition. Then, the `DirichletBC` terms have values that are set to `True` instead to some value. This will fix the value of the variable at the interface, but it will not influence its value. Thereby, we can e.g. fix the y-coordinates of the `"left"` interface. Finally, note that we use the Lagrangian coordinate to prescribe the deformation in the `DirichletBC` term. We cannot use the Eulerian coordinate (i.e. `var("mesh")` or `var("coordinate")`) here, since these are now unknowns. Dirichlet boundary conditions may only depend on independent variables.

Finally, the `SpatialErrorEstimator` will refine the mesh where the deformation is rather discontinuous (cf. right panel in Fig. 6.1).

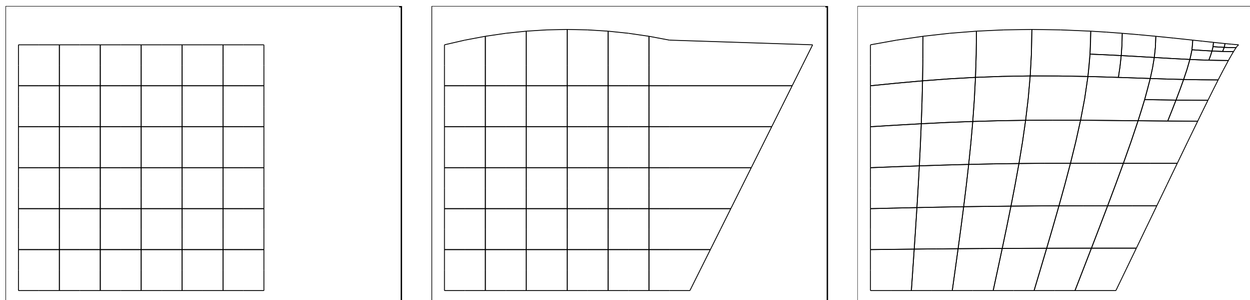

Fig. 6.1: Laplace smoothing: (left) undeformed mesh. (center) mesh after applying the Dirichlet boundary conditions that deform the mesh at the interfaces. (right) Relaxed mesh.

6.3 Time derivatives of fields on moving meshes

We will now move a mesh around and solve a diffusion equation on the moving mesh to see what happens. First of all, the mesh shall now move to the left and right by a sinus oscillation, which can be realized as follows

```
from pyoomph import *
from pyoomph.expressions import *

class MovingMesh(Equations):
    def define_fields(self):
            self.activate_coordinates_as_dofs()

    def define_residuals(self):
            x,xtest=var_and_test("mesh_x")
            xi,t=var(["lagrangian_x","time"])
```

(continues on next page)

(continued from previous page)

```
            desired_pos=xi+0.125*sin(2*pi*t)
            self.add_residual(weak(x-desired_pos,xtest,lagrangian=True) )
```

We have omitted the `coordinate_space` argument in `activate_coordinates_as_dofs()`, so that the coordinate space is automatically adjusted by the space of the diffusion equation. The diffusion equation with diffusivity 0.01 could be written as follows:

```
class DiffusionEquation(Equations):
    def define_fields(self):
            self.define_scalar_field("c","C2")

    def define_residuals(self):
            c,ctest=var_and_test("c")
            # Note the ALE=False statement here. We come to it in the text
            self.add_residual(weak(partial_t(c,ALE=False),ctest)+weak(0.01*grad(c),
↪grad(ctest)))
```

And the `Problem` class combining the equations reads

```
class ALEProblem(Problem):
    def define_problem(self):
            self.add_mesh(RectangularQuadMesh(N=32,lower_left=[-0.5,-0.5]))
            eqs=MovingMesh()
            eqs+=MeshFileOutput()
            eqs+=DiffusionEquation()
            eqs+=DirichletBC(mesh_y=True)
            x=var("coordinate")
            eqs+=InitialCondition(c=exp(-dot(x,x)*100))
            self.add_equations(eqs@"domain")

if __name__=="__main__":
    with ALEProblem() as problem:
            problem.run(1,numouts=20)
```

Note how we pin the y-coordinate by a `DirichletBC` applied on the entire `"domain"`. There is no equation for the y-coordinate, so we have to fix it. We furthermore set a Gaussian initial condition for the diffusion field.

What happens is naively counter-intuitive, namely the diffusion field c is moving along with the oscillating mesh, see Fig. 6.2. One might expect, that the maximum of the field c will be always at $\vec{x}=0$, but it is not the case. Instead, it will be always in the center of the mesh, i.e. at $\vec{\xi}=0$. This can be understood, when considering that fields are always approximated as functions of the nodal values $c_l(t)$. Here, we have

$$c(\vec{x},t)=\sum_l c_l(t)\psi(\vec{x},t)$$

where $\psi(\vec{x},t)$ are the shape/basis functions, which are due to the moving mesh now also a function of time t and l is a summation over all nodes. When adding `ALE=False` to `partial_t(c)`, we will just calculate the temporal derivatives of the coefficients $c_l(t)$, i.e.

$$\text{partial_t}^{\text{ALE=False}}(\text{c})=\sum_l \dot{c}_l(t)\psi(\vec{x},t) \tag{6.2}$$

Thereby, when the mesh moves, the entire field (including the time derivative) will co-move with the mesh. If we want to compensate for the mesh motion, we have to compensate for the term originating from the chain rule due to the time-dependence of the mesh coordinates. To that end `partial_t()` has an optional argument `ALE`, which defaults to `ALE="auto"`. If `ALE` is `False`, we calculate the time derivative according to (6.2). However, if `ALE=True` is

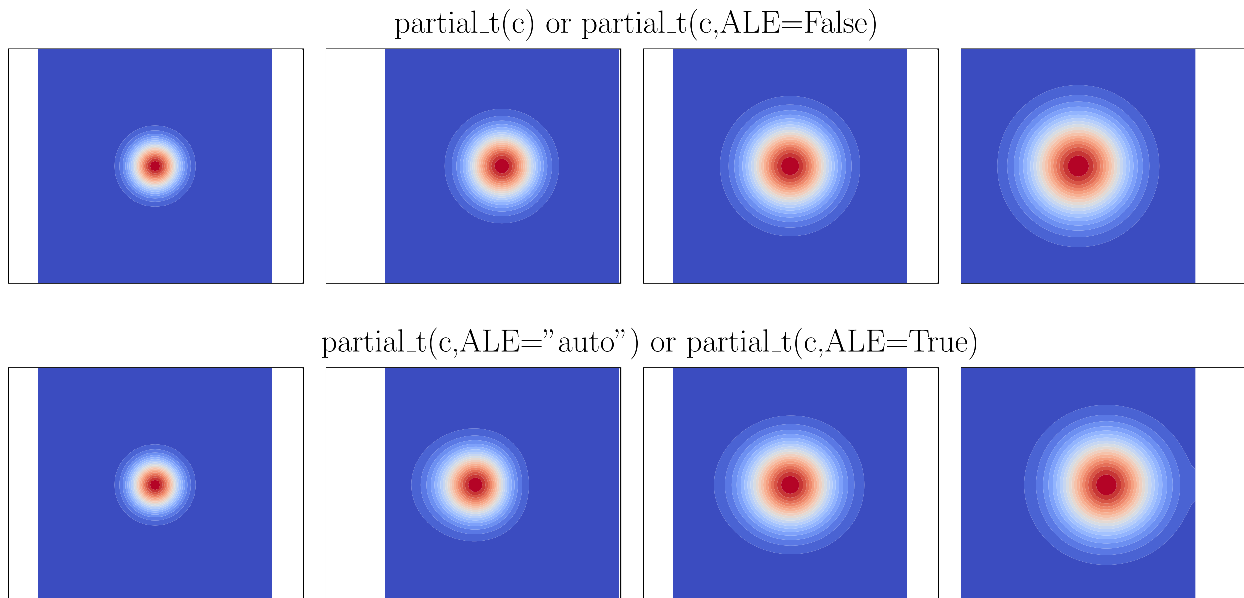

Fig. 6.2: When the mesh is moving oscillatory to the left and right, all fields move along with it. The center of the Gaussian spot is always in the center of the mesh, not of the Eulerian coordinate system. When the keyword argument `ALE` of the function `partial_t()` is set to `True` or `"auto"`, it is compensated for the mesh motion. Thereby, the maximum of the field stays in the center of the coordinates, not of the mesh. Since it gets into contact with the boundaries, the field is slightly deformed.

passed, we get

$$\text{partial_t}^{\text{ALE=True}}(\text{c}) = \text{partial_t}^{\text{ALE=False}}(\text{c}) - \dot{\vec{x}} \cdot \nabla c \tag{6.3}$$

Thereby, the field c is moved against the mesh motion and hence stays in place when the mesh moves. Since `ALE=True` requires the evaluation of the mesh velocity $\dot{\vec{x}}$, it is not required on static meshes. On a static mesh, $\dot{\vec{x}} = 0$ holds, and hence the calculation is redundantly time-consuming during the assembly of the system. Pyoomph also allows to pass `ALE="auto"` (the default value) to set it to `False` if the mesh is static, i.e. no `Equations` are added that call `activate_coordinates_as_dofs()`. Thereby, the redundant computation of $\dot{\vec{x}}$ is not carried out. If the mesh is moving, i.e. an equation for the mesh coordinates is present, `ALE="auto"` will become `ALE=True`, i.e. expanding according to (6.3).

> **Warning:** The predefined variables `var("coordinate")` and `var("mesh")` are in principle the same (and so there components `var("coordinate_x")`, `var("mesh_x")`, etc). However, there are two fundamental differences: `var("mesh")` can have test functions, when `activate_coordinates_as_dofs()` has been called. Furthermore, `partial_t(var("mesh"))` may be non-zero on a moving mesh, i.e. yielding the mesh velocity $\dot{\vec{x}}$, whereas `partial_t(var("coordinate"))` is always zero.

In conclusion, if one has a moving mesh, but want to keep spatio-temporal fields independent of the mesh motion, one should augment all `partial_t()` calls with an `ALE="auto"` (or leave it out, since it is the default value). This has been done in the bottom row of Fig. 6.2, where the weak formulation of the diffusion equation has been changed to

```
self.add_residual(weak(partial_t(c,ALE="auto"),ctest)+weak(0.01*grad(c),grad(ctest)))
```

The field c is now not following the oscillatory motion of the mesh, but stays in place.

6.4 Mesh reconstruction upon large deformations

It happens frequently that a moving mesh deforms strongly so that the elements are not well suited for solving equations on it. In that case, it might be necessary to regenerate the mesh by fitting the interfaces with splines, recreate a new mesh and interpolate all defined fields, including their history values for time stepping, of the previous mesh to the new mesh. pyoomph can do this automatically with the `RemeshWhen` class, which must be added to bulk domains. At the moment, this only works for two-dimensional meshes. Furthermore, it is necessary to set the property `remesher` of the `MeshTemplate` to a `Remesher2d` instance from the module `pyoomph.meshes.remesher`. Thus, for an example problem, we first have to make sure to import the module and set the `remesher` attribute to a `Remesher2d`, which requires the mesh template as first argument for the constructor:

```
from laplace_smoothed_mesh import *
from pyoomph.meshes.remesher import *

class RemeshingProblem(Problem):
    def __init__(self):
        super(RemeshingProblem, self).__init__()
        self.remeshing=True # shall we remesh or not
        self.remesh_options=RemeshingOptions(max_expansion=2,min_expansion=0.3,
↪min_quality_decrease=0.2) # when to remesh

    def define_problem(self):
        # Create a mesh and add a remesher
        mesh=RectangularQuadMesh(N=6)
        mesh.remesher=Remesher2d(mesh)
```

The `RemeshingOptions` control when a mesh reconstruction shall be invoked. The parameters `max_expansion` and `min_expansion` give a threshold for which remeshing is invoked whenever an element has grown or shrunken above `max_expansion` or below `min_expansion` with respect to its initial size. This is determined based on the Cartesian area of each element. Furthermore, a `min_quality_decrease` can be set to the `RemeshingOptions`, since the area of an element can still remain quite close to its initial area, but the element becomes strongly anisotropic, e.g. by expanding in one direction while collapsing in the perpendicular direction.

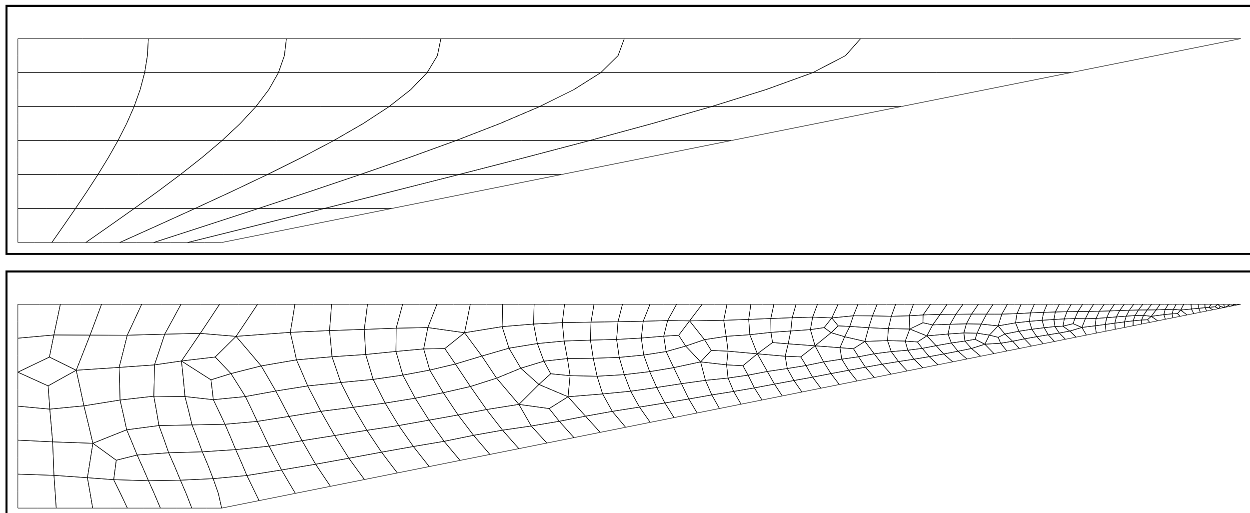

Fig. 6.3: Without remeshing (top), the mesh can strongly deform. With remeshing (bottom), one can prevent this (at the cost of some computational time and small interpolation errors). One can furthermore control the local mesh size, e.g. to ensure a fine mesh at the sharp corner.

After that, the equations are assembled as usual. Here, we fix all boundaries except the `"right"` one, which will move

with time:

```
# Add the mesh and use the Lagrange smoothed mesh
self.add_mesh(mesh)
eqs=LaplaceSmoothedMesh()
eqs+=MeshFileOutput()
# Fix some interfaces
eqs+=DirichletBC(mesh_x=0,mesh_y=True)@"left"
eqs+=DirichletBC(mesh_x=True,mesh_y=0)@"bottom"
eqs+=DirichletBC(mesh_y=1)@"top"

# Moving boundary
xi=var("lagrangian")
eqs+=DirichletBC(mesh_x=1+0.5*xi[1]*var("time"))@"right" # move the right interface␣
↪with time
```

To automatically invoke remeshing, we just must add a `RemeshWhen` object to the domain, which takes `RemeshingOptions` as argument. When there are no further instructions added, pyoomph tries to estimate the local mesh resolution based on the previous resolution. This does not always work well and sometimes it is beneficial to instruct pyoomph to use specific mesh sizes instead. To that end, one can add `RemeshMeshSize` objects to interfaces and corners, by what the local mesh size (nondimensional, in terms of the spatial scale) can be set:

```
# Remeshing based on the options
eqs+=RemeshWhen(self.remesh_options)
# optional: setting particular sizes at interfaces or corners
eqs+=RemeshMeshSize(size=0.2)@"left" # size of 0.2 at the left interface
eqs += RemeshMeshSize(size=0.02) @ "right/top" # size of 0.02 at the top right corner

self.add_equations(eqs@"domain")
```

To run the simulation, there is nothing specifically to be done:

```
if __name__=="__main__":
    with RemeshingProblem() as problem:
        # problem.remesh_options.active=False we can deactivate remeshing by the␣
↪remesh options as well
        problem.run(10,outstep=True,startstep=0.5,maxstep=0.5)
```

As usual, the `RemeshingOptions` can be modified before the `run()` statement. In particular, one can deactivate it by setting `active=False`. A comparison without and with remeshing is shown in Fig. 6.3.

One additional option is to add an instance of the class `SetLagrangianToEulerianAfterSolve` to the equations of the moving mesh domain. This will set the Lagrangian coordinates to the Eulerian coordinates after each successful time step. Thereby, the mesh displacement in (6.1) will be zero at the beginning of the next time step.

6.5 Free surface Navier-Stokes equation

We will now combine the Laplace smoothed moving mesh with a Navier-Stokes equation. Having a moving mesh allows us to track a free surface and impose a surface tension on it. We will have two boundary conditions that must be enforced, namely the kinematic and the dynamic boundary condition.

The kinematic boundary condition reads

$$\vec{n}\cdot\left(\vec{u}-\dot{\vec{x}}\right)=0\,, \tag{6.4}$$

i.e. the mesh has to move with the normal fluid velocity [5]. This is obviously a constraint which narrows the potential solutions of the mesh coordinates. We therefore add a field of Lagrange multipliers λ with test function η on each free

surface that enforces this constraint. Since we want the normal mesh motion to follow the fluid, and not the velocity following the mesh motion, we add the action of the Lagrange multiplier to the test space of the mesh, not the velocity. If we would add it to the test space of the velocity, the particular mesh equations (e.g. the Laplace smoothed mesh) would influence the flow, which is not reasonable. Hence, the weak form at the free surfaces for the kinematic boundary condition reads

$$\left\langle \vec{n}\cdot\left(\vec{u}-\dot{\vec{x}}\right),\eta\right\rangle - \langle\lambda,\vec{n}\cdot\vec{\chi}\rangle\,. \tag{6.5}$$

The implementation of the kinematic boundary condition could read as follows

```
from pyoomph import *

# use the pre-defined Navier-Stokes ( it will use partial_t(...,ALE="auto") )
from pyoomph.equations.navier_stokes import *
# and the pre-defined LaplaceSmoothedMesh
from pyoomph.equations.ALE import *

class KinematicBC(InterfaceEquations):
    def define_fields(self):
        self.define_scalar_field("_kin_bc","C2") # second order field lambda

    def define_residuals(self):
        n,u=var(["normal","velocity"])
        l,eta=var_and_test("_kin_bc") # Lagrange multiplier
        x,chi=var_and_test("mesh") # unknown mesh coordinates
        # Let the normal mesh velocity follow the normal fluid velocity
        self.add_residual(weak(dot(n,u-mesh_velocity()),eta)-weak(l,dot(n,chi)))

    def before_assigning_equations_postorder(self, mesh):
        # pin the Lagrange multiplier, when the mesh is locally entirely pinned
        self.pin_redundant_lagrange_multipliers(mesh, "_kin_bc", "mesh")
```

Note that we use `mesh_velocity()` instead of `partial_t()` to calculate the mesh velocity. In fact, `mesh_velocity()` is just `partial_t(var("mesh"),ALE=False)`. This is reasonable, since we want the mesh velocity co-moving with the interface, i.e. directly at the nodes.

Again, we use `pin_redundant_lagrange_multipliers()` in the `before_assigning_equations_postorder()` method to automatically pin the local Lagrange multiplier, if the mesh position is entirely pinned at any node. If the mesh cannot move, the Lagrange multiplier would otherwise over-constrain the problem.

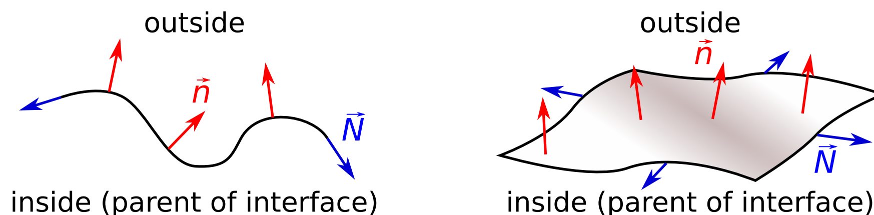

Fig. 6.4: Interface normals $\vec{n}$ (normal to the interface, pointing outside from the parent domain) and the interface boundary normals $\vec{N}$ (tangentially to the interface, pointing outwards) for a 1d interface of a 2d bulk domain and a 2d interface of a 3d bulk domain.

The second condition is the dynamic boundary condition. This one states that the traction is given by a combination of the interface curvature κ, the surface tension σ and potential tangential gradients of the latter, i.e.

$$\vec{n}\cdot\left[-p\mathbf{1}+\mu\left(\nabla\vec{u}+(\nabla\vec{u})^{\mathrm{t}}\right)\right]=\sigma\kappa\vec{n}+\nabla_S\sigma \tag{6.6}$$

∇_S is the surface gradient operator, sometimes written as $\nabla_S = (\mathbf{1} - \mathbf{nn})\,\nabla$, and will only have tangential contributions. Obviously, the lhs of this equation is the negative Neumann contribution we can add to the (Navier-)Stokes equation, cf. (4.12). It could be hence implemented by adding

$$-\left\langle \sigma\kappa\vec{n} + \nabla_S\sigma, \vec{v}\right\rangle$$

as interface contribution to the velocity test function $\vec{v}$. However, it is not trivial to calculate the curvature $\kappa = -\nabla_S \cdot \vec{n}$. In fact, pyoomph does not allow to calculate the surface divergence of the normal yet. Instead, we make use of the *surface divergence theorem*. For an arbitrary vector field $\vec{w}$ defined on the interface Γ, we have the relation

$$\int_\Gamma \nabla_S \cdot \vec{w}\,\mathrm{d}A = \int_\Gamma (\nabla_S \cdot \vec{n})\,(\vec{n}\cdot\vec{w})\,\mathrm{d}A + \int_{\partial\Gamma} \vec{w}\cdot\vec{N}\,\mathrm{d}l$$

where the last integral is comprising the boundary of the surface with outward normal $\vec{N}$ (cf. Fig. 6.4 for an illustration of both kinds of normals). When selecting $\vec{w} = \sigma\vec{v}$, this can be arranged to

$$\int_\Gamma (-\nabla_S \cdot \vec{n})\,(\sigma\vec{n}\cdot\vec{v})\,\mathrm{d}A = -\int_\Gamma \nabla_S \cdot (\sigma\vec{v})\;\mathrm{d}A + \int_{\partial\Gamma} \sigma\vec{v}\cdot\vec{N}\,\mathrm{d}l\,,$$

or, alternatively, using the product rule

$$\int_\Gamma \left[\sigma\,(-\nabla_S\cdot\vec{n})\,\vec{n}\right]\cdot\vec{v}\,\mathrm{d}A = -\int_\Gamma \left[(\nabla_S\sigma)\cdot\vec{v} + \sigma\,(\nabla_S\cdot\vec{v})\right]\,\mathrm{d}A + \int_{\partial\Gamma}\sigma\vec{v}\cdot\vec{N}\,\mathrm{d}l$$

and by moving the surface tension gradient $\nabla_S\sigma$ from the right to the left, we get

$$\int_\Gamma \left[\sigma\,(-\nabla_S\cdot\vec{n})\,\vec{n} + \nabla_S\sigma)\right]\cdot\vec{v}\,\mathrm{d}A = -\int_\Gamma \sigma\,(\nabla_S\cdot\vec{v})\;\mathrm{d}A + \int_{\partial\Gamma}\sigma\vec{v}\cdot\vec{N}\,\mathrm{d}l\,.$$

Upon negation, we can identify the weak forms

$$-\left\langle\sigma\kappa\vec{n} + \nabla_S\sigma, \vec{v}\right\rangle = \left\langle\sigma, \nabla_S\cdot\vec{v}\right\rangle - \left[\sigma\vec{N}, \vec{v}\right]\,, \tag{6.7}$$

So instead calculating the curvature, it is sufficient to add $\langle\sigma, \nabla_S\cdot\vec{v}\rangle$ to get both normal traction due to the Laplace pressure and tangential Marangoni stresses simultaneously. Additionally, there is another term $[\cdot,\cdot]$ arising, which allows weak Neumann contributions at the ends of the free surface, which will help us to impose contact angles soon.

Tip: oomph-lib also covers the boundary conditions of a free surface in the tutorial at https://oomph-lib.github.io/oomph-lib/doc/navier_stokes/surface_theory/html/index.html.

Also, an analogous implementation of the following free surface can be found in oomph-lib at https://oomph-lib.github.io/oomph-lib/doc/navier_stokes/single_layer_free_surface/html/index.html

---

But first, let us now implement the dynamic boundary condition which can be added to the free surface itself, i.e. the $\langle\cdot,\cdot\rangle$ contribution:

```python
class DynamicBC(InterfaceEquations):
    def __init__(self,sigma):
        super(DynamicBC,self).__init__()
        self.sigma=sigma

    def define_residuals(self):
        v=testfunction("velocity")
        self.add_residual(weak(self.sigma,div(v)))
```

One might wonder whether `div()` is indeed the surface divergence operator ∇_S. But when this equation is added to an interface, it will indeed expand to this. There is no other reasonable way to calculate the divergence of a field defined on a manifold embedded in a higher order space. The same applies for `grad()`: In the bulk, i.e. on domains with zero *co-dimension*, it is indeed the convectional gradient, but on manifolds (surfaces) with nonzero co-dimension, it will be the corresponding surface gradient.

Before defining the problem, we can combine both boundary conditions in a short-hand notation:

```
# Shortcut to create both conditions simultaneously
def FreeSurface(sigma):
    return KinematicBC()+DynamicBC(sigma)
```

Now, as example problem, let us do the same as before on the basis of lubrication theory in Section 5.4.1, but this time solving the exact flow and the exact free surface dynamics:

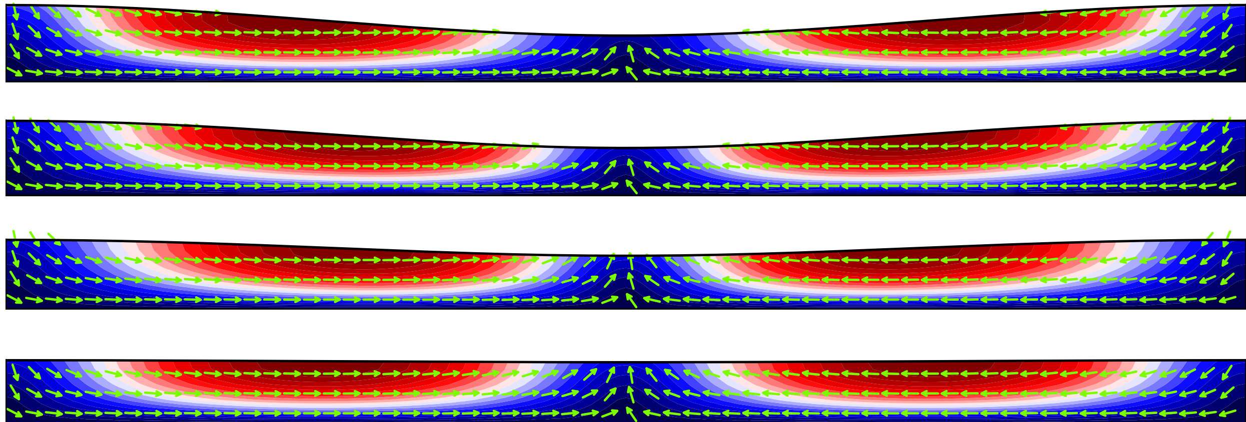

Fig. 6.5: Free surface combined with Navier-Stokes and a Laplace-smoothed mesh.

```
class SurfaceRelaxationProblem(Problem):
    def define_problem(self):
        # Shallow 2d mesh
        self.add_mesh(RectangularQuadMesh(N=[80,4],size=[1,0.05]))
        eqs=NavierStokesEquations(mass_density=0.01,dynamic_viscosity=1) #␣
↪equations
        eqs+=LaplaceSmoothedMesh() # Laplace smoothed mesh
        eqs+=DirichletBC(mesh_x=True) # We can fix all x-coordinates, since the␣
↪problem is rather shallow
        eqs+=MeshFileOutput() # output
        eqs+=DirichletBC(velocity_x=0,velocity_y=0,mesh_y=0)@"bottom" # no slip␣
↪at bottom and fix the mesh there
        eqs+=DirichletBC(velocity_x=0)@"left" # no in/outflow at the sides
        eqs+=DirichletBC(velocity_x=0)@"right"
        eqs+=FreeSurface(sigma=1)@"top" # free surface at the top
        # Deform the initial mesh
        X,Y=var(["lagrangian_x","lagrangian_y"])
        eqs+=InitialCondition(mesh_y=Y*(1+0.25*cos(2*pi*X)))  # small height␣
↪with a modulation
        self.add_equations(eqs@"domain") # adding it to the system


if __name__=="__main__":
    with SurfaceRelaxationProblem() as problem:
        problem.run(50,outstep=True,startstep=0.25)
```

Opposed to the lubrication example in Section 5.4.1, we use a `RectangularQuadMesh` to resolve the entire bulk flow. We add the predefined `NavierStokesEquations`, which also - opposed to lubrication theory - allows for inertia due to the finite mass density. In order to use the free surface equations we have just defined, we must allow the mesh nodes to move, since the `KinematicBC` requires to add weak contributions to the test function of the mesh coordinates. We therefore use the predefined `LaplaceSmoothedMesh`. However, since this particular problem is shallow, it is sufficient to only consider motion in y-direction. This can be achieved by just pinning all x-positions to their initial values by the `DirichletBC(mesh_x=True)`. We impose no slip and a zero y-coordinate at the `"bottom"` interface and prevent any in- or outflow at the `"left"` and `"right"` interfaces. Finally, we deform the initial mesh by adding an `InitialCondition`, which sets the y-position based on the `"lagrangian"` coordinate, which corresponds to the undeformed mesh by default.

6.6 Droplet spreading with equilibrium contact angle

In the following, we consider a sessile droplet on a substrate, which is allowed to spread over the substrate to form an equilibrium contact angle. We slowly generalize the example problem to more and more complex scenarios:

6.6.1 With free slip at the substrate

Similarly as done with lubrication theory in Section 5.4.2, we now want to let a droplet spread until it reaches its equilibrium contact angle. This time, however, we want to solve the full bulk flow including inertia and considering the full interface curvature. Hence, we use again the free surface equations from the previous example.

Key part to impose an equilibrium contact angle is the $[\cdot,\cdot]$ term in (6.7). It has not been considered to far, but it will become relevant now. This term can be added to boundaries of the free surface, i.e. to contact lines. The surface tension is fully balanced if $\vec{N}$ is the outward pointing normal of the contact line, which is the outward pointing tangential continuation of the free interface at the boundaries. Let θ be the equilibrium contact angle, then $\vec{N}$ will read $(\cos(\theta), -\sin(\theta))$ if the droplet is at equilibrium contact angle. Let us define the corresponding equation class that has to be added to the contact line to enforce this contact angle:

```
from free_surface import * # Load our free surface implementation
from pyoomph.meshes.simplemeshes import CircularMesh # Import a curved mesh


class EquilibriumContactAngle(InterfaceEquations):

    required_parent_type=DynamicBC # Must be attached to an interface with a␣
↪DynamicBC

    def __init__(self,N):
        super(EquilibriumContactAngle,self).__init__()
        self.N=N # equilibrium vector

    def define_residuals(self):
        # get sigma from the DynamicBC object of the interface
        sigma=self.get_parent_equations().sigma
        # Contact line contribution
        v=testfunction("velocity")
        self.add_residual(-weak(sigma*self.N,v))
```

It is rather trivial, how $-[\sigma\vec{N},\vec{v}]$ is implemented here. Note how we can access the free interface by accessing the `DynamicBC` equation of the parent domain, i.e. of the free surface, by `get_parent_equations()`. Since we have set `required_parent_type=DynamicBC`, pyoomph knows that `get_parent_equations()` should give the `DynamicBC` and not the `KinematicBC` contribution. Only the former has the surface tension property `sigma` defined.

The problem class itself is very similar to the previous example:

```
class DropletSpreadingProblem(Problem):
    def __init__(self):
        super(DropletSpreadingProblem,self).__init__()
        self.contact_angle=45*degree # equilibrium contact angle

    def define_problem(self):
        # hemi-circle mesh, i.e. initial contact angle of 90 degree, free␣
↪interface "interface", symmetry axis "axis" and bottom interface "substrate"
        mesh=CircularMesh(radius=1,segments=["NE"],straight_interface_name={
↪"center_to_north":"axis","center_to_east":"substrate"},outer_interface="interface")
        self.add_mesh(mesh)

        self.set_coordinate_system("axisymmetric") # axisymmetry

        eqs=NavierStokesEquations(mass_density=0.01,dynamic_viscosity=1) # flow
        eqs+=LaplaceSmoothedMesh() # Laplace smoothed mesh
        eqs+=RefineToLevel(4) # refine, since the CircularMesh is coarse by␣
↪default
        eqs+=DirichletBC(mesh_x=0,velocity_x=0)@"axis" # fix mesh x-position, no␣
↪flow through the axis
        eqs+=DirichletBC(mesh_y=0,velocity_y=0)@"substrate" # fix substrate at␣
↪y=0, no flow through the substrate
        # free surface at the interface, equilibrium contact angle at the␣
↪contact with the substrate
        N=vector(cos(self.contact_angle),-sin(self.contact_angle))
        eqs+=(FreeSurface(sigma=1)+EquilibriumContactAngle(N)@"substrate")@
↪"interface"
        eqs+=MeshFileOutput() # output

        self.add_equations(eqs@"domain") # adding it to the system


if __name__=="__main__":
    with DropletSpreadingProblem() as problem:
        problem.run(50,outstep=True,startstep=0.25)
```

Important differences are the mesh, which is now the north-east (`"NE"`) quarter of a `CircularMesh` with renamed boundaries and a suitable `RefineToLevel` by thereof, the selection of an `"axisymmetric"` coordinate system and the fact that now both x and y mesh coordinates are allowed to move. We set the x-coordinate (i.e. the r-coordinate) of the mesh and the x-velocity (r-velocity) to zero at the axis of symmetry and likewise the y-coordinate and y-velocity to zero at the substrate. Note that the x-velocity at the substrate is completely free, i.e. it corresponds to an (unrealistic) free slip boundary at the moment. This can be seen on the left side of Fig. 6.6. The droplet spreads quickly and the fluid can flow unhindered tangentially along the substrate.

6.6.2 With Navier-slip at the substrate

As mentioned, the free slip, i.e. the unhindered x-velocity, at the substrate is not physical. Often, one imposes no slip at solid substrates, but adding `DirichletBC(velocity_x=0)@"substrate"` would lead to a pinned contact line. Since the x-velocity will be zero, the only way to fulfill the `KinematicBC` will be that also the mesh velocity in x-direction will be zero. Hence, the contact line won't move. Furthermore, any contribution to the contact line boundary condition $[\sigma\vec{N},\vec{v}]$ would vanish in that case, since the velocity test function $\vec{v}$ must be zero on strongly enforced boundaries.

A solution to this dilemma is the consideration of a *slip length*, which essentially does what the precursor film has done in the corresponding lubrication theory example in Section 5.4.2: It allows for the motion of the contact line by introducing a small length scale.

without slip length

slip length=0.001

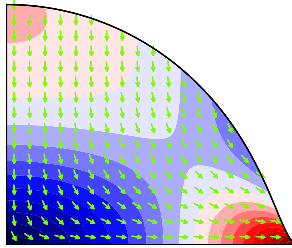
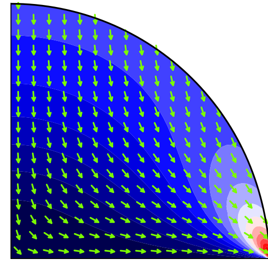
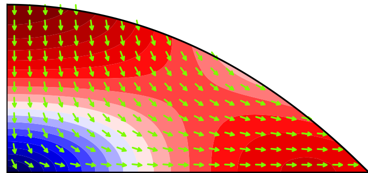
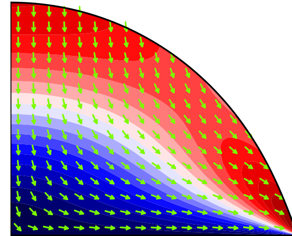
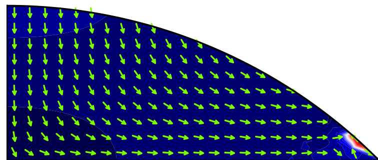
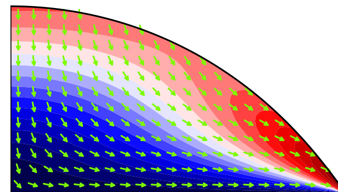
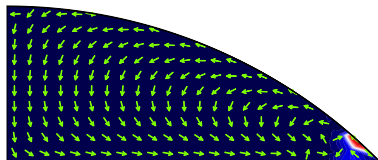
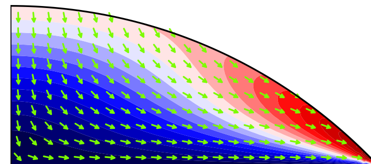

Fig. 6.6: (left) Spreading with free slip at the substrate. (right) spreading with a tiny slip length.

The *Navier-slip* boundary condition does not set the velocity to zero a the substrate but allows a damped tangential motion. This can be done by imposing a term proportional to the tangential velocity as tangential Neumann condition, i.e. as traction, we add

$$\left\langle \frac{\mu}{L_\mathrm{s}}\mathbf{P}_t\vec{u},\mathbf{P}_t\vec{v}\right\rangle$$

to the Neumann contribution, where $\mathbf{P}_t = \mathbf{1} - \mathbf{nn}$ is the tangential projector, L_s is the slip length and μ is the dynamic viscosity in the bulk. If the fluid wants to move tangentially along the substrate, a counter-acting traction proportional to $\frac{\mu}{L_\mathrm{s}}\mathbf{P}_t\vec{u}$ will hamper this motion. An implementation could read

```
from droplet_spread_free_slip import * # Load the problem without slip

class SlipLength(InterfaceEquations):
    # must be attached to a domain with NavierStokesEquations
    required_parent_type = NavierStokesEquations

    def __init__(self, slip_length):
        super(SlipLength, self).__init__()
        self.slip_length = slip_length # store the slip length

    def define_residuals(self):
        n = var("normal")
        u, utest = var_and_test("velocity")
        utang = u - dot(u, n) * n # tangential velocity
        utest_tang = utest - dot(utest, n) * n # tangential test function
        mu=self.get_parent_equations().dynamic_viscosity # get mu from the␣
↪parent equations
        factor = mu / (self.slip_length) # add the weak contribution
        self.add_residual(weak(factor * utang, utest_tang))
```

Again, we use `required_parent_type` in combination with `get_parent_equations()` to access the `dynamic_viscosity` of the bulk, i.e. of the `NavierStokesEquations` in the parent domain of this interface. Note that this equation does not add contributions normal to the substrate. Here, we rely that a `DirichletBC(velocity_y=0)` takes care of preventing any flow through the substrate in y-direction.

To add the slip length to the problem, we use inheritance, i.e. our new problem will be the same as the old problem, except that a slip length will be additionally added to the system:

```
# Inherit from the problem without slip
class DropletSpreadingWithSliplength(DropletSpreadingProblem):
    def __init__(self):
        super(DropletSpreadingWithSliplength,self).__init__()
        self.slip_length=0.001 # tiny slip length


    def define_problem(self):
        super(DropletSpreadingWithSliplength,self).define_problem() # define the␣
↪old problem
        self.add_equations(SlipLength(self.slip_length)@"domain/substrate") #␣
↪add a slip length to the substrate

        # Refinement
        self.max_refinement_level=6 # level 4 is already the base refinement,␣
↪allow additional refinment
        self.add_equations(SpatialErrorEstimator(velocity=1)@"domain") # allow␣
↪for refinement to resolve the strong stresses near the contact line
```

(continues on next page)

(continued from previous page)

```
if __name__=="__main__":
    with DropletSpreadingWithSliplength() as problem:
        problem.run(50,outstep=True,startstep=0.25,spatial_adapt=1)
```

Note that we anticipate high stresses near the contact line. The droplet wants to spread due to the stress stemming from the equilibrium contact angle contribution, but it will be hampered due to the slip length near the substrate. Hence, we add mesh refinement to resolve this more accurately.

A comparison of results without and with slip length can be seen in on the right side of Fig. 6.6 in the previous section. The case with slip length is definitely more realistic. The smaller the slip length, the slower the spreading will take place. The slip length can hence be used to match the spreading velocity with experiments.

6.6.3 Full three dimensional implementation with wetting gradients on the substrate

Finally, we also want to use the droplet spreading case in three dimensions. Since equations are defined in a coordinate-system independent way within pyoomph, it is not much additional work required. The problem class reads

```
from droplet_spread_sliplength import * # Import the previous example
from pyoomph.meshes.simplemeshes import SphericalOctantMesh # import a 3d mesh,␣
↪octant of a sphere

class DropletSpreading3d(Problem):
    def __init__(self):
        super(DropletSpreading3d, self).__init__()
        # The equilibirum contact angle will vary with the position along the␣
↪substrate
        x,y=var(["coordinate_x","coordinate_y"])
        # some equilibrium contact angle expression
        self.contact_angle = (45+80*(minimum(x,2)-1)**2-30*(minimum(y,2)-1)**2) *␣
↪degree

    def define_problem(self):
        # Eighth part of a sphere, rename the outer interface to "interface" and the␣
↪z=0 plane to "substrate"
        mesh = SphericalOctantMesh(radius=1, interface_names={"shell":"interface",
↪"plane_z0":"substrate"})
        self.add_mesh(mesh)

        eqs = NavierStokesEquations(mass_density=0.01, dynamic_viscosity=1)  # flow
        # PseudoElasticMesh is a bit more expensive to calculate, but is more stable␣
↪in terms of larger deformations than LaplaceSmoothedMesh
        eqs += PseudoElasticMesh()
        eqs += RefineToLevel(2)  # Since the SphericalOctanctMesh is really coarse

        # No flow through the boundaries and
        eqs += DirichletBC(mesh_x=0, velocity_x=0) @ "plane_x0"
        eqs += DirichletBC(mesh_y=0, velocity_y=0) @ "plane_y0"
        eqs += DirichletBC(mesh_z=0, velocity_z=0) @ "substrate"

        # free surface at the interface, equilibrium contact angle at the contact␣
↪with the substrate
        n_free=var("normal",domain="domain/interface") # normal of the free surface
        n_substrate=vector(0,0,1) # normal of the substrate
```

(continues on next page)

(continued from previous page)

```
        t_substrate=n_free-dot(n_free,n_substrate)*n_substrate # projection of the␣
↪free surface normal on the substrate
        t_substrate=t_substrate/square_root(dot(t_substrate,t_substrate)) #␣
↪normalized => tangent along the substrate locally outward
        N = cos(self.contact_angle)*t_substrate - sin(self.contact_angle)*n_substrate
↪# Assemble N vector
        eqs += (FreeSurface(sigma=1) + EquilibriumContactAngle(N) @ "substrate") @
↪"interface"

        eqs += MeshFileOutput()  # output

        self.add_equations(eqs @ "domain")  # adding it to the system


if __name__ == "__main__":
    with DropletSpreading3d() as problem:
        problem.run(50, outstep=True, startstep=0.05,temporal_error=1,maxstep=2)
```

The equilibrium contact angle may be an arbitrary function of the coordinates. Here, we chose some expression that describe some wetting gradients along the substrate. As mesh, we use the `SphericalOctantMesh`, which is one eighth of a sphere. It has to be refined with a `RefineToLevel`, since it is otherwise too coarse. However, three-dimensional ALE problems become easily very expensive in terms of computational costs. We get a high number of degrees of freedom for the velocity and also for the mesh position. Also, the motion of the mesh will couple with all equations and the Jacobian of the coupled system has quite a bunch of non-zero entries. Hence, one should not exaggerate the refinement level here. Instead of the `LaplaceSmoothedMesh`, now the predefined class `PseudoElasticMesh` is used. While both leads to a relaxtion of a mesh that is subject to deformations due to the `KinematicBC`, the `PseudoElasticMesh` behaves as a deformable solid. This is often more stable, i.e. preventing strong deformations in a better way. It is, however, slightly more expensive to calculate than the `LaplaceSmoothedMesh`.

At the planar surfaces at $x = 0$, $y = 0$ and $z = 0$ of the mesh, we deactivate the mesh motion and the velocity in this direction to prevent any outflow through these domains. The tangential continuation of the free interface at the contact line $\vec{N}$ is now more complicated than in two dimensions. We calculate it by first projecting the free surface normal $\vec{n}$ on the normal $\vec{n}_\text{s} = (0, 0, 1)$ of the substrate. This leads to a vector with zero z-component, i.e. oriented along the substrate plane and pointing outward from the contact line. When normalizing this vector $\vec{t}_\text{s}$, we can assemble our vector by

$$\vec{N} = \cos(\theta)\vec{t}_\text{s} - \sin(\theta)\vec{n}_\text{s}$$

The equilibrium contact angle θ is a function of the local coordinates. $\vec{t}_\text{s}$ depends on the local free surface. So in total, the contact line dynamics is really complicated and highly non-linear. Luckily, pyoomph does all the required internals, in particular the assembly of the analytical Jacobian, automatically.

In terms of implementation, one has to pay attention: It is important to use `n_free=var("normal", domain="domain/interface")`, whereas `n_free=var("normal")` would *not* work. `n_free` will be further evaluated at the contact line, not at the free surface. Hence, without `domain` specification, it would expand to the normal $\vec{N}$ of the contact line, which is the tangential continuation of the free surface.

Since we are mostly interested in the final state of the droplet, we have not considered a slip length here, so that the droplet attains the equilibrium shape (cf. Fig. 6.7) as quickly as possible. Also the dynamic time stepping is beneficial for that. However, it is required to delimit the maximum time step with `maxstep` since too large steps lead to unacceptable errors in the volume conservation. The reason is due to the kinematic boundary condition, which is discrete in time.

> **Warning:** Three-dimensional meshes and problems are currently still under development. Therefore, one should handle these with caution.

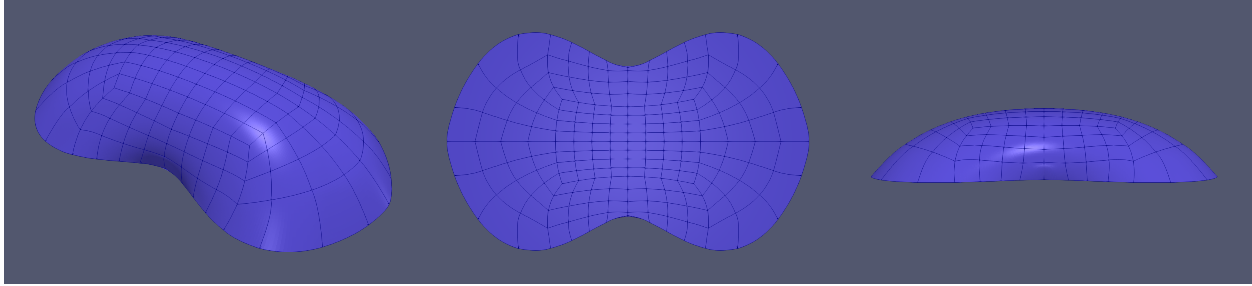

Fig. 6.7: Equilibrium shape of a three-dimensional droplet with varying equilibrium contact angle along the substrate.

6.6.4 Stationary solutions of a dimensional droplet with Marangoni flow and gravity

Lastly, we want to check how a droplet with an equilibrium contact angle behaves under the influence of gravity and Marangoni flow. Opposed to the previous cases, this time we consider physical dimensions and calculate stationary solutions, which will reveal some pitfalls to carefully consider. We will make use of the predefined equations, which essentially do the same as e.g. the `KinematicBC`, `DynamicBC` and `SlipLength` considered here, but allow for physical dimensions. We could implement it by hand into our versions of these equations, but this has been laid out in other examples in detail (cf. e.g. sections Section 3.8, Section 4.4.5, Section 4.4.7 and Section 5.3.4). For simplicity, we also revert to the axisymmetric variant.

The first problem when looking for stationary solutions is the fact that the problem is non-linear, mainly due to the mesh motion. Hence, one must have a good initial guess in order for the problem to converge with a stationary solve. Since we do not know the shape *a priori*, we start with a droplet a zero gravity, zero Marangoni flow and a contact angle of 90 °. For these parameters, we know that we just will have a hemisphere with vanishing velocity. We will then crank up first the gravity, then the contact angle to 120 ° and finally activate the Marangoni flow. To that end, we introduce three parameters to the system:

```
from pyoomph import * # main pyoomph
from pyoomph.expressions.units import * # units like meter, etc.

from pyoomph.equations.navier_stokes import * # Navier-Stokes equations, including␣
↪free surface, contact angles
from pyoomph.equations.ALE import * # Moving mesh

from pyoomph.meshes.simplemeshes import CircularMesh # Mesh


class StationaryDropletProblem(Problem):
    def __init__(self):
        super(StationaryDropletProblem, self).__init__()
        self.volume=0.25*milli*liter
        self.rho=1000*kilogram/meter**3
        self.mu=1*milli*pascal*second
        self.sigma0=72*milli*newton/meter
        self.slip_length=1*micro*meter

        # Parameter to modify the gravity (in terms of g), initially 0
        self.param_gravity_factor = self.define_global_parameter(gravity_
↪factor=0)
        # Parameter to dynamically change the contact angle at run time
        # Initially 90 deg coinciding with the initial mesh
        self.param_contact_angle=self.define_global_parameter(contact_angle=pi/2)
        # Parameter to control the surface tension, sigma=sigma0*(1+sigma_
```

(continues on next page)

(continued from previous page)

```
↪gradient*(var("coordinate_x")/R0)**2), initially 0
            self.param_sigma_gradient = self.define_global_parameter(sigma_
↪gradient=0)
            self.gravity=9.81*meter/second**2 # overall gravity when param_gravity_
↪factor.value=1
```

We use dimensional values for the volume and the fluid parameters. The gravity will be blended in by the parameter `param_gravity_factor`. When this parameter reaches 1, we will get a downwards acting gravity of 9.81 $\mathrm{m/s^2}$. The contact angle will be controlled by the global parameter `param_contact_angle`, which `value` is initialized to 90 °. Finally, `param_sigma_gradient` will activate Marangoni flow by letting the surface tension be

$$\sigma = \sigma_0 \cdot \left(1 + \sigma_1 \left(\frac{x}{R_0}\right)^2\right),$$

where σ_1 is the parameter `param_sigma_gradient` and R_0 is the radius of the initial hemispherical droplet.

The `define_problem()` method starts as follows:

```
def define_problem(self):
        R0=square_root(3*self.volume/(4*pi)*2,3) # Radius of the initial hemi-sphere
        mesh=CircularMesh(radius=R0,segments=["NE"],outer_interface="interface",
↪straight_interface_name={"center_to_north":"axis","center_to_east":"substrate"})
        self.add_mesh(mesh)

        # Find good scales to nondimensionalize the space, the time, velocity and␣
↪pressure
        self.set_scaling(spatial=R0,temporal=1*second,velocity=scale_factor("spatial
↪")/scale_factor("temporal"))
        self.set_scaling(pressure=self.sigma0/R0)

        self.set_coordinate_system("axisymmetric")

        eqs=MeshFileOutput()

        # Navier-Stokes with gravity
        g=self.gravity*self.param_gravity_factor*vector(0,-1) # We can change the␣
↪influence of gravity by the parameter
        eqs+=NavierStokesEquations(mass_density=self.rho,dynamic_viscosity=self.mu,
↪gravity=g)
        # Mesh motion and refinement of the coarse CircularMesh
        eqs+=PseudoElasticMesh()
        eqs+=RefineToLevel(self.initial_adaption_steps) # Refine slightly in the␣
↪beginning to find the solution quickly


        # Dirichlet boundary conditions and slip length
        eqs+=DirichletBC(velocity_x=0,mesh_x=0)@"axis"
        eqs += DirichletBC(velocity_y=0, mesh_y=0) @ "substrate"
        eqs +=NavierStokesSlipLength(self.slip_length)@"substrate"

        # Free surface and contact angle. Both surface tension and contact angle␣
↪depend on parameters
        sigma=self.sigma0*(1+self.param_sigma_gradient*(var("coordinate_x")/R0)**2)
        eqs+=NavierStokesFreeSurface(surface_tension=sigma)@"interface"
        eqs+=NavierStokesContactAngle(self.param_contact_angle,wall_normal=vector(0,
↪1),wall_tangent=vector(-1,0))@"interface/substrate"
```

Until here, it is quite trivial if you have read the previous examples. Note how we use e.g. `param_gravity_factor` directly to get the symbolic parameter, i.e. not its `value` but its symbolic value, that will be replaced by its instantaneous value during each calculation. The refinement `RefineToLevel` of the initial mesh will be controlled by `initial_adaption_steps`. We will later set this to a small refinement level to approximate a good guess for the initial condition of the stationary solution we are looking for on a coarse mesh. This will drastically improve the computation time. The `NavierStokesFreeSurface` and `NavierStokesContactAngle` are the predefined equivalents to the combination of `KinematicBC` and `DynamicBC` of the free surface and the `EquilibriumContactAngle` of the contact line developed in this chapter.

Now, we come to the most intricate problem: When solving for stationary solutions, the volume of the droplet will not be conserved at all. The reason lies in the implementation of the `KinematicBC`. In this, we let the mesh follow the normal fluid motion. However, it involves the time derivative of the mesh coordinates, but in stationary solves, all temporal derivatives are zero. Hence, there is no guarantee at all that the volume would be conserved or even that a solution is actually found. We therefore must enforce the volume by a single Lagrange multiplier. Therefore, we will add a new ODE domain containing just the single Lagrange multiplier λ, which will enforce the volume constraint by virtue of adding the Lagrange multiplier contribution

$$\lambda\left(\int_{\text{drop}} 1\,\mathrm{d}V - V_0\right)\,. \tag{6.8}$$

This has to be split into two contributions, namely one integral over the domain and the constant offset of the negative initial volume $-V_0$. The latter part can be done as follows:

```
# The volume is not conserved during stationary solving
# So we must add a single global Lagrange multiplier that ensures the volume

# Create the Lagrange multiplier "volume_lagrange", add an offset of -self.volume to␣
↪the residual
vol_constr_eqs=GlobalLagrangeMultiplier(volume_lagrange=-self.volume)
vol_constr_eqs+=Scaling(volume_lagrange=scale_factor("pressure")) # Introduce the␣
↪scale to nondimensionalize
vol_constr_eqs += TestScaling(volume_lagrange=1/self.volume) # And also a test scale,␣
↪which is just the inverse
self.add_equations(vol_constr_eqs@"volume_constraint") # add it to an "ODE" domain␣
↪called "volume_constraint"
```

The `GlobalLagrangeMultiplier` will just define a Lagrange multiplier with the name `volume_lagrange` and considering the offset $-V_0$ to the weak form. Since we have a dimensional problem, we also have to set the scaling of `volume_lagrange` and for the corresponding test function. This can be changed by adding `Scaling` and `TestScaling` objects to the ODE equation. One might wonder why we choose a scale of the pressure for the Lagrange multiplier, but this will become clear in a minute. Finally, the Lagrange multiplier is added to the ODE domain `"volume_constraint"`.

There is still the integral portion missing. Since it involves an integral over the droplet `"domain"`, we have to add it to the equations on `"domain"`. To that end, we use `WeakContribution`:

```
# bind the volume enforcing Lagrange multiplier
vol_constr_lagr=var("volume_lagrange",domain="volume_constraint")
# And add the dimensional integral 1*dx over the droplet to the residual of the␣
↪Lagrange multiplier
# In total, we then solve [integral_droplet 1*dx - self. volume] on the test space of␣
↪the Lagrange multiplier
# This is indeed the volume constraint
eqs+=WeakContribution(1,testfunction(vol_constr_lagr),dimensional_dx=True)
```

The `WeakContribution` will just add the integral $\int 1\,\mathrm{d}V$ to the test space of the global Lagrange multiplier λ, i.e. accounting for the first term in (6.8). Note the `dimensional_dx=True` argument, which carries out the integral with

physical dimensions, i.e. not in non-dimensional coordinates. This is required to get a dimensional volume contribution in m^3.

So far, so good. The volume constraint is assembled correctly. But there is no feedback yet on the droplet. We must impose the value of the Lagrange multiplier somewhere back to the system, otherwise it is just redundant. The idea is to add an additional pressure contribution to the droplet, which is proportional to the Lagrange multiplier. This can be understood easily if the initially hemispherical droplet without any gravity or flow is considered. As argued before, in absence of the additional Lagrange multiplier, the volume is not conserved and hence, there is an infinite number of solutions possible, namely for each volume V we get a unique Laplace pressure due to the different curvature. When we add the contribution of the Lagrange multiplier to the pressure, the system is only in agreement if the volume V is indeed the volume V_0 and the corresponding Laplace pressure in the droplet is Laplace pressure corresponding to this volume. If the volume V deviates from V_0, the additional contribution to the pressure from the non-zero Lagrange multiplier will alter the pressure throughout the droplet and forces the droplet to attain the volume V_0 with the corresponding Laplace pressure during the solution procedure.

We hence add the Lagrange pressure as additional normal traction, i.e. analogous to the Laplace pressure stemming from the surface tension and curvature.

```
# Finally, the Lagrange multiplier must also yield feedback to the problem
# It can be done easily by adding additional pressure whenever the droplet's volume␣
↪does not match
# We hence add a normal traction (pressure) proportional to the volume Lagrange␣
↪multiplier
eqs+=NeumannBC(velocity=vol_constr_lagr*var("normal"))@"interface"

eqs+=SpatialErrorEstimator(velocity=1)

self.add_equations(eqs@"domain")
```

The problem definition ends with a `SpatialErrorEstimator` for the velocity, which will be relevant when Marangoni flow is activated at the end.

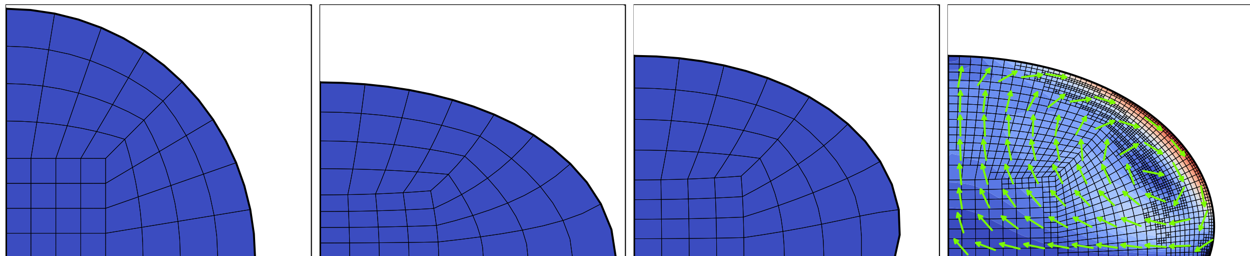

Fig. 6.8: Starting with a coarse hemispherical mesh, we first crank up the gravity, then modify the contact angle and finally solve for the Marangoni flow with refinement.

Now, we can use the solution procedure, i.e. starting with a coarse hemispherical droplet, cranking up gravity, subsequently adjusting the contact angle and finally blending in the Marangoni flow. Only at the final step, we do mesh refinement, since the other steps where just made to approach a good initial guess for the final solution:

```
if __name__=="__main__":
    with StationaryDropletProblem() as problem:

            problem.initial_adaption_steps=2 # adapt the initial mesh only a bit for␣
↪fast calculation
            problem.max_refinement_level=7 # final adaption, very fine

            # Start without gravity, no Marangoni flow and at 90 deg contact angle
```

(continues on next page)

(continued from previous page)

```
            problem.param_gravity_factor.value=0
            problem.param_sigma_gradient.value = 0
            problem.param_contact_angle.value = pi/2

            problem.solve() # This stationary solve is trivial
            problem.output() # Output first step

            # Ramp up gravity by arclength continuation
            problem.go_to_param(gravity_factor=1,startstep=0.2,final_adaptive_
↪solve=False)
            problem.output_at_increased_time()

            # Ramp up the contact angle by arclength continuation
            problem.go_to_param(contact_angle=110*degree, startstep=5*degree,final_
↪adaptive_solve=False)
            problem.output_at_increased_time()

            # Ramp up Marangoni flow by arclength continuation
            problem.go_to_param(sigma_gradient=0.001, startstep=0.001,final_adaptive_
↪solve=True)
            problem.output_at_increased_time()
```

The method `go_to_param()` will gradually increase/decrease the specified parameter (identified by the name passed to `get_global_parameter()` in the `Problem` constructor) to the desired value via arc length continuation. With the `startstep` argument, we can optionally set the first try of the parameter increment. If not set, it will try to directly reach the desired parameter value, which is not always successful and results in a lot of unsuccessful tries followed by a retry with a reduced step. The `final_adaptive_solve` is set to `False` in the first two parameter increases. This means we do not adapt after the parameter value has been reached. At the very end, we set it to `True` to ultimately adapt the final result to maximum accuracy, selected by `max_refinement_level`. The results are depicted in Fig. 6.8. The problem would never converge if one directly tries to solve for the last step starting with the hemispherical initial mesh.

Note: We have enforced the volume in (6.8) by integrating 1 over the droplet domain. While this is definitely acceptable, the volume contribution of each element depends on all nodal positions of the element. This leads to a lot of computational costs for the assembly of the Jacobian and also for the solution of the latter.

Alternatively, we can use a trick so that only the nodal positions on the boundaries contribute. To that end, first note that $\nabla\cdot\vec{x}=d$, where d is the number of dimensions (in axisymmetric coordinates, it will be 3 as well). We can hence write the contribution $(1,\eta)_{\text{drop}}$ (with η being the testfunction of λ) as $1/d\,(\nabla\cdot\vec{x},\eta)_{\text{drop}}$. Now, we can apply the divergence theorem to obtain $1/d\,(\nabla\cdot\vec{x},\eta)_{\text{drop}}=1/d\,\langle\vec{x}\cdot\vec{n},\eta\rangle$, where the interface integral has to be applied to all interfaces. However, at the substrate and at the axis of symmetry, $\vec{x}\cdot\vec{n}$ is zero, so that only the free surface contributes.

Instead of `WeakContribution(1,testfunction(vol_constr_lagr),dimensional_dx=True)`, we could hence define `x_dot_n=dot(var("coordinate"),var("normal"))` and add `WeakContribution(1/3*x_dot_n,testfunction(vol_constr_lagr),dimensional_dx=True)@"interface"`.

This will speed up the assembly and the solving.

6.6.5 Hyperelastic meshes and tangentially consistent mesh motion

If we want to increase the contact angle in the previous problem case to $150°$ instead of $110°$ by adjusting the line

```
problem.go_to_param(contact_angle=150*degree, startstep=5*degree,final_adaptive_
↪solve=False)
```

the solver will fail to converge. The reason is large deformation of the mesh, in particular the element at the contact line. Of course, we could use remeshing as described in Section 6.4, but alternatively, we can also improve the moving mesh dynamics. Pyoomph comes with several moving mesh equations, so if we use e.g. the `HyperelasticSmoothedMesh` instead of the `PseudoElasticMesh`, the compilation and numerical assembly will be a bit slower, but the mesh quality is usually better under deformation. However, the main issue that prevents convergence is actually the tangential mesh node positioning at the moving boundaries. The kinematic boundary condition only prescribes the normal motion and the tangential motion is given by the bulk equations of the particular moving mesh class. However, we only add a local point force at the contact line, which leads to strong deformation of the bulk element attached to the contact line. Thereby, the element at the contact line can easily collapse to a singular element.

To prevent this, we can make sure that also the boundary nodes close to the contact line are nicely shifted tangentially, so that the bulk element at the contact line is not collapsing. To do so, we can ensure that the nodes on the boundaries keep their relative arclength position. To that end, we add

```
# Make sure that the interface node positions keep their relative tangential positions
eqs+=EnforcedInterfacialLaplaceSmoothing().with_corners("substrate","axis")@"interface
↪" # Smooth the interface
eqs+=EnforcedInterfacialLaplaceSmoothing().with_corners("interface","axis")@"substrate
↪" # Smooth the interface
```

to the equations. The first line ensures that the relative tangential position of all nodes on the `"interface"` boundary are kept, i.e. the nodes are shifted tangentially so that each node stays at its initial position when parameterized by the normalized arclength of the boundary. We must call the `with_corners()` method of the `EnforcedInterfacialLaplaceSmoothing` class to tell pyoomph the considered end points of the interface. Thereby, a unique normalized arclength parameterization becomes possible. Internally, pyoomph just calculates the initial arclength parameterization and solves a Laplace equation along the surface to obtain the current arclength parameterization. At the corners, Dirichlet boundary conditions are automatically set by the `with_corners()` call, so that the arclength can be indeed recovered by solving the Laplace equation along the interface. This is of course only true if the Laplace equation is solved in a Cartesian coordinate system (despite of the axisymmetric coordinate system considered e.g. for the flow), but this is the default setting of `EnforcedInterfacialLaplaceSmoothing`.

Pyoomph enforces that the difference between the initial arclength parameterization and the current parameterization found by solving the Laplace equation vanishes. The corresponding field of Lagrange multipliers enforcing this condition is acting as tangential force of the mesh along the boundary, which is arbitrary, since it does not influence the physics in this case, since e.g. the kinematic boundary condition only prescribes the motion in normal direction.

Lastly, we also make use of the class `EnforceVolumeByPressure()` to do exactly what was discussed in the previous example, but as a one-liner.

The issue without shifting the tangent node positions is shown in the left part of Fig. 6.9. Clearly, the element at the contact line becomes problematic. This aspect is solved when shifting the nodes tangentially along the entire substrate (and free surface) by keeping the relative arclength positions of the nodes fixed (middle of Fig. 6.9). The final solution is shown on the right of Fig. 6.9.

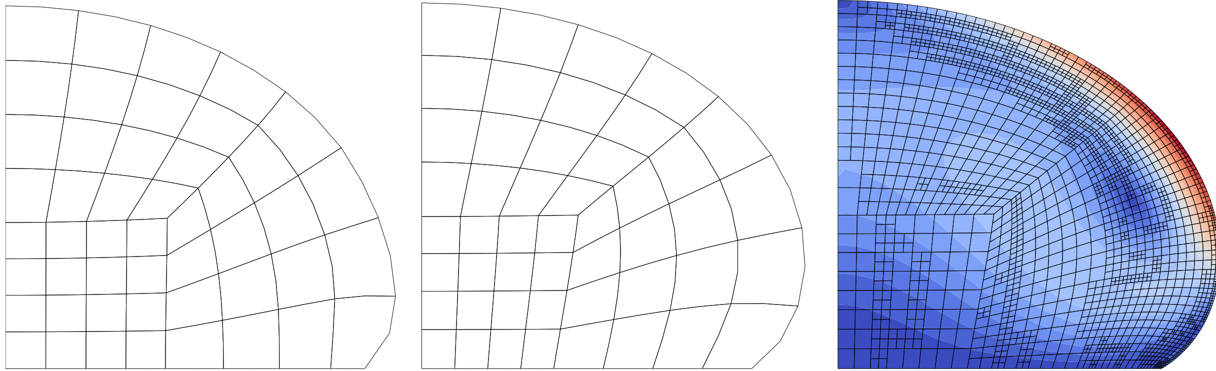

Fig. 6.9: (left) Without tangentially shifting the nodes along the substrate (and along the free surface), the element at the contact line collapses to a singular element when going to higher contact angles. (middle) Switching to hyperelastic mesh dynamics and in particular ensuring relative tangential arclength positioning allows for higher contact angles. (right) Final solution with the tricks discussed here.

6.7 Nonlinear solid mechanics

The possibility of having a moving mesh, i.e. having the mesh coordinates as degrees of freedom, is obviously a good foundation to account for deformable solids. In this section, we cover the possibilities of pyoomph for nonlinear elasticity of solid bodies. The essential idea in pyoomph's implementation is that the Lagrangian coordinates represent the undeformed body, whereas the Eulerian coordinates reflect the current deformed configuration.

Note: The implementation of nonlinear elasticity is essentially a one-to-one copy of the implementation in oomph-lib. Oomph-lib's documentation covers the individual aspects in considerably more detail, and with the exact co- and contravariant definitions, than this documentation here.

As governing equations, we have the principle of virtual displacements, which reads in pyoomph's notation

$$\left(\rho\partial_t^2\vec{x}-\vec{f},\Gamma\vec{\chi}\right)_\xi+\left(\left(\nabla_\xi\vec{x}\right)\boldsymbol{\sigma},\Gamma\nabla_\xi\vec{\chi}\right)_\xi-\langle\vec{T},\vec{\chi}\rangle=0$$

Again, $\vec{x}(\vec{\xi},t)$ represents the deformed configuration (with test function $\vec{\chi}$) and $\vec{\xi}$ is the undeformed configuration. $\vec{f}$ is a bulk force density (in the undeformed configuration), ρ is the mass density in undeformed configuration and Γ is an isotropic growth factor, i.e. the factor the undeformed solid wants to grow in terms of the undeformed length/area/volume (depending on the dimension). The bulk part of the equation is available in the class `DeformableSolidEquations`, whereas a traction $\vec{T}$ can be applied on the boundary of the solid by the class `SolidTraction`. For pressures, tractions in negative normal direction, the class `SolidNormalTraction` can be used.

The constitutive law is entirely wrapped in the second Piola-Kirchhoff stress tensor $\boldsymbol{\sigma}$. Following oomph-lib's GeneralisedHookean class, a reasonable definition is given by the following

$$\begin{aligned}
\sigma_{ij}&=E_{ijkl}\gamma_{kl}\\
E_{ijkl}&=\frac{E}{1+\nu}\left(\frac{\nu}{1-2\nu}G_{ij}^{-1}G_{kl}^{-1}+\frac{1}{2}\left(G_{ik}^{-1}G_{jl}^{-1}+G_{il}^{-1}G_{jk}^{-1}\right)\right)\\
\boldsymbol{G}&=\left(\left(\nabla_\xi\vec{x}\right)^{\mathrm{t}}\left(\nabla_\xi\vec{x}\right)\right)\\
\boldsymbol{\gamma}&=\frac{1}{2}\left(\boldsymbol{G}-\boldsymbol{g}\right)\\
\boldsymbol{g}&=\Gamma^{2/D}\left(\left(\nabla_\xi\vec{\xi}\right)^{\mathrm{t}}\left(\nabla_\xi\vec{\xi}\right)\right)
\end{aligned}$$

Please kindly excuse the sloppy notation not respecting the co- and contravariances here correctly. This is done considerably better in oomph-lib's documentation. However, the way it is written here corresponds to the implementation in pyoomph, where we heavily use matrices and the vector gradients, since they automatically adjust correctly based on the considered coordinate system (e.g. $\boldsymbol{g}$ is the (scaled) identity matrix in Cartesian coordinates, but in axisymmetric coordinates, it is not). Entering parameters are Young's modulus E and the Poisson ratio ν. For small deformations, it perfectly recovers Hook's law. D is the dimension of the solid, which is e.g. 3 for a 2d mesh with axisymmetric coordinate system. This constitutive law is implemented in the class `GeneralizedHookeanSolidConstitutiveLaw`.

If $\nu = 1/2$, the solid is incompressible and the definition becomes singular. In that case, we have to use

$$\sigma^{ij} = -pG^{-1}_{ij} + \frac{E}{3}\left(G^{-1}_{ik}G^{-1}_{jl} + G^{-1}_{il}G^{-1}_{jk}\right)\gamma_{kl} \tag{6.9}$$

instead. Here, p is a pressure field, which enforces that

$$\det \boldsymbol{G} - \det \boldsymbol{g} = 0 \tag{6.10}$$

i.e. that the deformation locally conserves the volume. For this law, we have to use the class `Incompressible-HookeanSolidConstitutiveLaw`.

Note: For compressible constitutive laws, the mass density ρ is to be understood as mass density in the undeformed configuration. The mass density in the deformed configuration can be accessed by `var("deformed_mass_density")`, which is calculated by the conservation of mass. In the incompressible case, there is no difference and `var("deformed_mass_density")` will expand to ρ.

Warning: The nonlinear elasticiy equations cannot be used in combination with azimuthal or Cartesian normal mode stability analysis as outline in section Section 10.3 and Section 10.4. For this approach, we would have to expand the mesh coordinates also by an azimuthal or additional Cartesian mode, which is not implemented (yet).

6.7.1 Bending of a cantilever beam

Note: The following case is a direct adaption of the corresponding example in oomph-lib.

We consider a 2d solid cantilever of length L and height $2H$ which is attached to a fixed wall on its left side. At the top of the cantilever, a pressure load P is applied in normal direction. The cantilever hence bends downwards. The problem class is again quite short

```
from pyoomph import *
from pyoomph.expressions import *
from pyoomph.equations.solid import * # nonlinear solid equations


class AiryCantileverProblem(Problem):
    def __init__(self):
        super().__init__()
        self.H=0.5 # half height of the cantilever
        self.L=10 # length of the cantilever
        self.E=1 # Young's modulus
        self.nu=0.3 # Poisson's ratio
        self.G=0 # Optional gravity vector (not considered here)
        self.P=self.define_global_parameter(P=0) # Pressure on the top of the␣
```

(continues on next page)

(continued from previous page)

```
↪cantilever

    def define_problem(self):
        self+=RectangularQuadMesh(size=[self.L,2*self.H],N=[20,2],lower_left=[0,-self.
↪H])

        eqs=MeshFileOutput()
        if True:
            # Take the constitutive law for a generalized Hookean solid (compressible)
            claw=GeneralizedHookeanSolidConstitutiveLaw(E=self.E,nu=self.nu)
        else:
            # Alternatively, take the constitutive law for an incompressible solid
            # This will introduce a pressure variable to ensure incompressibility,␣
↪but we must remove the null space of the pressure variable
            claw=IncompressibleHookeanSolidConstitutiveLaw(E=self.E)
            eqs+=AverageConstraint(pressure=0) # remove the null space of the␣
↪pressure variable

        eqs+=DeformableSolidEquations(constitutive_law=claw,mass_density=0,
↪bulkforce=vector(0,self.G),coordinate_space="C2",pressure_space="DL",with_error_
↪estimator=True)
        eqs+=DirichletBC(mesh_x=0,mesh_y=True)@"left" # Fix the left side of the␣
↪cantilever
        eqs+=SolidNormalTraction(self.P)@"top" # Apply pressure on the top of the␣
↪cantilever
```

Relevant parts are the import of the solid equations of `pyoomph.equations.solid`. After defining the default parameters in the problem constructor (where the pressure load P is introduced as global parameter to allow to vary it later on), the problem only consists of a simple `RectangularQuadMesh` and `DeformableSolidEquations` associated with the desired boundary conditions. `SolidNormalTraction` will introduce the varying pressure load to the top of the cantilever in normal direction. The material properties are introduced by instances of particular constitutive law classes. Here, in particular, we use `GeneralizedHookeanSolidConstitutiveLaw`, which is a compressible nonlinear generalization of Hook's law. It requires Young's modulus E and the Poisson ratio ν as parameters. Note that the latter may not be $\nu = 1/2$, since this leads to a singularity. In this case, the material is incompressible and one has to use particular incompressible constitutive laws like e.g. the `IncompressibleHookeanSolidConstitutiveLaw`. Opposed to an compressible law, an incompressible constitutive law requires the introduction of a pressure field, which acts as Langrange multiplier enforcing the incompressibility everywhere. Since the pressure offset (reference pressure) can be chosen arbitrarily, we must remove this null space, which can be done by either fixing a single pressure degree of freedom or by demanding that e.g. the average pressure should vanish. The latter approach is easily realized by an `AverageConstraint`.

Besides the constitutive law, the `DeformableSolidEquations` allows to set a mass density (which is only relevant in transient problems for the inertia term) and a potential bulk force density (note that this is meant with respect to the *undeformed* solid). With `coordinate_space="C2"`, we explicitly ask for second order elements. By default, the order of the elements will be determined by the highest order space of all other fields defined on this mesh, but since we do not define any other fields, we have to explicitly set it here. In case of a pressure field required by an incompressible constitutive law, we also can select the space of the pressure field. Here, with `pressure_space="DL"`, we select a discontinuous pressure field, but alternatively, we could have used e.g. `pressure_space="C1"` for a continuous pressure space. Finally, `with_error_estimator=True`, will introduce a functionality like the `SpatialErrorEstimator`, so that the mesh can be adapted based on the strains in the solid.

As in the corresponding example in oomph-lib, we also want to compare the numerically obtained stresses with an approximate analytical solution. This can be done by adding `LocalExpressions`, which will write the numerical and analytical stresses to the output. We therefore continue the definition of the problem by

```
# To compare the numerical solution with the analytical solution, we write the␣
↪numerical and analytical stress tensors
eqs+=LocalExpressions(sigma_num=claw.get_sigma(2,pressure_var=var("pressure"))) #␣
↪Numerical stress tensor
# Constants for exact (St. Venant) solution
a=-1.0/4.0*self.P
b=-3.0/8.0*self.P/self.H
c=1.0/8.0*self.P/self.H**3
d=1.0/20.0*self.P/self.H
xi=var("lagrangian")
xx=self.L-xi[0]
yy=xi[1]
# Approximate analytical (St. Venant) solution of the stress tensor
sigma=matrix([[c*(6.0*xx*xx*yy-4.0*yy*yy*yy)+6.0*d*yy, 2.0*(b*xx+3.0*c*xx*yy*yy)],[2.
↪0*(b*xx+3.0*c*xx*yy*yy),2.0*(a+b*yy+c*yy*yy*yy)]])
eqs+=LocalExpressions(sigma_ana=sigma) # approximate analytical stress tensor

self+=eqs@"domain"
```

More details on the analytical solution can be found in the documentation of oomph-lib.

Finally, we can just run the problem by gradually increasing the pressure load, solving the problem at each load and writing the output:

```
with AiryCantileverProblem() as problem:
     max_adapt=3
     nstep=5
     p_increment=1.0e-5

     problem.initialise()
     problem.refine_uniformly()

     for i in range(nstep):
        problem.P.value+=p_increment
        problem.solve(spatial_adapt=max_adapt)
        problem.output_at_increased_time()
```

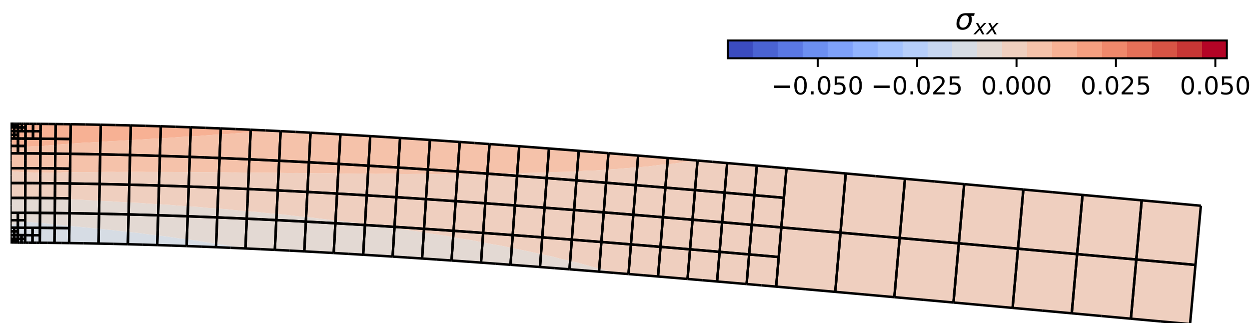

Fig. 6.10: Bending of a cantilever beam under increasing pressure load at the top

6.7.2 Compression of 2D circular disk

Note: The following case is a direct adaption of the corresponding example in oomph-lib.

We consider a 2d disc that is compressed by a uniform pressure. The disc initially has a radius of unity, however, we introduce a isotropic growth factor of $\Gamma = 1.1$, so that the disc wants to grow to a radius of $\sqrt{\Gamma}$ is absence of any external pressure. Opposed to the corresponding example in oomph-lib, we will come up with two implementations of the very same case. We either can solve the problem on a quarter circle mesh with symmetry boundary conditions in a two-dimensional Cartesian coordinate system. However, due to the coordinate-system agnostic formulation of equations, the same can be realized by a simple radial line mesh in a polar coordinate system. To that end, we introduce a flag `polar_implementation` in the problem class:

```
from pyoomph import *
from pyoomph.expressions import *
from pyoomph.equations.solid import *
from pyoomph.meshes.simplemeshes import CircularMesh

class CompressedDiscProblem(Problem):
    def __init__(self):
        super().__init__()
        self.Gamma=1.1 # isotropic growth factor
        self.claw=GeneralizedHookeanSolidConstitutiveLaw(E=1,nu=0.3) # Generalized␣
↪Hookean solid constitutive law
        self.P=self.define_global_parameter(P=0) # Pressure on the circumference of␣
↪the disc
        self.polar_implementation=False # Use radial polar coordinates only

    def define_problem(self):
        # Base equations, irrespective of the coordinate system
        eqs=MeshFileOutput()
        eqs+=DeformableSolidEquations(constitutive_law=self.claw,coordinate_space="C2
↪",isotropic_growth_factor=self.Gamma)
        eqs+=SolidNormalTraction(self.P)@"circumference"

        # Mesh, coordinate system and boundary conditions depending on whether we␣
↪solve a 2d Cartesian or polar 1d problem
        if self.polar_implementation:
            self+=LineMesh(size=1,N=20,left_name="center",right_name="circumference")
↪# Create a line mesh for the radial direction
            self.set_coordinate_system("axisymmetric") # Polar coordinate system
            eqs+=DirichletBC(mesh_x=0)@"center" # Fixed in the center of the disc
        else:
            # Case of Cartesian coordinates, we create a quarter circular mesh
            self+=CircularMesh(radius=1,segments=["NE"])
            eqs+=DirichletBC(mesh_x=0)@"center_to_north" # and fix the positions at␣
↪the symmetry axes
            eqs+=DirichletBC(mesh_y=0)@"center_to_east"
```

The basis setup is analogous to the previous example, i.e. we require a constitutive law which is then used in the `DeformableSolidEquations`. Here, however, we impose the `isotropic_growth_factor`, which lets the disc wants to grow everywhere from its undeformed configuration by Γ in terms of the area. Again, a `SolidNormalTraction` is imposed at the boundary `circumference`. If `polar_implementation==True`, we switch to an *axisymmetric*` coordinate system (which is a radial polar coordinate system for 1d meshes) and use a simple 1d mesh. In that case, the right boundary will be called `circumference`, whereas the left boundary of the interval is called `center`. At the latter, we make sure that the mesh position is fixed to the origin. If we do not solve the polar case, we

do solve it on a quarter circle mesh on a 2d Cartesian coordinate system. We have to fix the mesh coordinates on the axes of symmetry in that case.

We also want to measure that current radius r of the disc. Irrespectively of the coordinate system, we can do so by integrating over the boundary `circumference`. We calculate two integrals, namely the line length $L = \int 1\,\mathrm{d}l$ and the integral over the radius $R = \int \|\vec{x}\|\,\mathrm{d}l$. In case of the 2d Cartesian implementation, both integrals will be only a quarter of the full disc, but this does not matter, since the (averaged) radius of the disc can be obtained by the ratio $r = R/L$:

```
# To monitor the radius of the disc, we can use IntegralObservables. We integrate␣
↪over the circumference of the disc to the the line length
# and we also integrate over r*dl
eqs+=IntegralObservables(_linelength=1,_radius_integral=square_root(dot(var(
↪"coordinate"),var("coordinate"))))@"circumference"
# The radius is then given by the ratio of the integral of r and the line length
eqs+=IntegralObservables(radius=lambda _radius_integral,_linelength:_radius_integral/_
↪linelength)@"circumference"

self+=eqs@"domain"
```

In the driver code, we just iterate over the imposed pressure (starting with a negative pressure to pull the disc outwards first). To compare the actual radius r with an analytical linearized expression (see the oomph-lib example for details), we can evaluate the introduced observable and write both the numerical value and the analytical approximation to a text file in the output directory:

```
with CompressedDiscProblem() as problem:
    delta_p=0.0125
    nstep=21

    problem.P.value=-delta_p*(nstep-1)*0.5 # Start with a negative pressure (pulling␣
↪the disc outwards)
    problem.initialise()
    problem.refine_uniformly()

    # Write a comparison output file with the radius computed from the linearized␣
↪analytical solution and the numerical solution
    outf=problem.create_text_file_output("disc_output.txt",header=["P","r_numeric","r_
↪linear"])

    for i in range(nstep):
        problem.solve()
        problem.output_at_increased_time()
        rlinear=square_root(problem.Gamma)*(1-problem.P*(1+problem.claw.nu)*(1-
↪2*problem.claw.nu))
        rnumeric=problem.get_mesh("domain/circumference").evaluate_observable("radius
↪")
        outf.add_row(problem.P,rnumeric,rlinear)
        problem.P.value+=delta_p
```

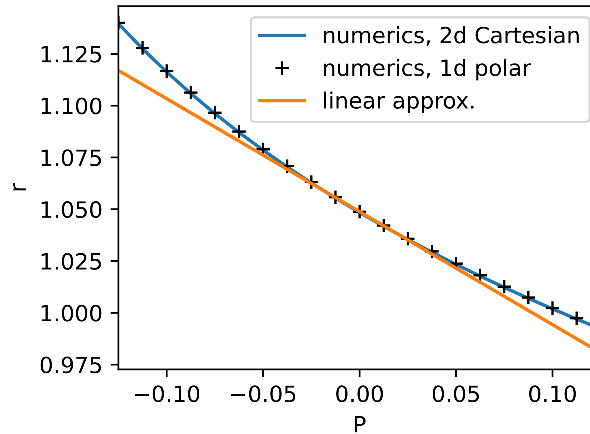

Fig. 6.11: Compressing a disc with an isotropic growth factor

6.7.3 Oscillations of a released torsion

So far, only stationary solutions of deformed solid bodies were considered. Now, we go for transient dynamics of a long three-dimensional beam. We start by a deformed configuration, specifically, by applying a torsion of the beam. On one side, the beam is fixed to a solid wall.

With the previous examples in mind, it is trivial to setup the problem case. However, here we consider physical units, i.e. we have to set typical scalings to let pyoomph nondimensionalize the equations and redimensionalize the output automatically. In particular, we need a spatial scale, a temporal scale and a scale for the mass density. With these, the equations can be nondimensionalized. A typical time scale for the solid dynamics can be obtained by Young's modulus, the density and the length:

```
from pyoomph import *
from pyoomph.expressions import *
from pyoomph.equations.solid import *
from pyoomph.meshes.simplemeshes import CuboidBrickMesh
from pyoomph.expressions.units import *

class OscillatingSolidProblem(Problem):
    def __init__(self):
        super().__init__()
        self.rho=1000*kilogram/meter**3 # Density of the solid
        self.E=2.5*giga*pascal # Young's modulus of the solid
        self.nu=0.38 # Poisson's ratio of the solid
        self.L=1*meter # Length of the beam
        self.H=5*centi*meter # thickness of the beam in the y and z direction
        self.Nh=2 # Number of elements in the y and z direction
        self.Nl=20 # Number of elements in the x direction
        self.torsion=90*degree/meter # Torsion of the beam, in torsion angle per meter

    def get_characteristic_time_scale(self):
        # Some typical time scale for the oscillation of the solid
        return self.L*square_root(self.rho/self.E)

    def define_problem(self):
        # Scales to nondimensionalize the equations
        self.set_scaling(spatial=self.L,mass_density=self.rho,temporal=self.get_
↪characteristic_time_scale())
```

(continues on next page)

(continued from previous page)

```
        self+=CuboidBrickMesh(size=[self.L,self.H,self.H],N=[self.Nl,self.Nh,self.Nh])
        eqs=MeshFileOutput()
        claw=GeneralizedHookeanSolidConstitutiveLaw(E=self.E,nu=self.nu)
        eqs+=DeformableSolidEquations(constitutive_law=claw,coordinate_space="C2",
↪mass_density=self.rho)
        # Apply the torsion to the solid by expression the mesh coordinates in terms␣
↪of the torsion angle and the Lagrangian coordinates (undeformed mesh coordinates)
        X=var("lagrangian")
        theta=self.torsion*X[0]
        eqs+=InitialCondition(mesh_y=X[1]*cos(theta)+X[2]*sin(theta),mesh_z=-
↪X[1]*sin(theta)+X[2]*cos(theta))
        eqs+=DirichletBC(mesh_x=0,mesh_y=True,mesh_z=True)@"left" # Fix the left side␣
↪of the beam to the solid wall
        self+=eqs@"domain"


with OscillatingSolidProblem() as problem:
    T=problem.get_characteristic_time_scale()
    problem.run(10*T,outstep=0.1*T,temporal_error=1)
```

For the torsion, we use the original undeformed beam by accessing the Lagrangian coordinates and apply the deformation by setting an initial condition for the Eulerian mesh coordinates. Note that the code generation and compilation takes a while, since the three-dimensional dynamics involves a lot of higher order tensors, inverse matrices (the contravariant metric tensor) and nonlinearities. In particular, the entries of the analytical Jacobian with respect to the moving mesh coordinates hence constitute long expressions, which bloat up the generated C code to over 3 megabytes.

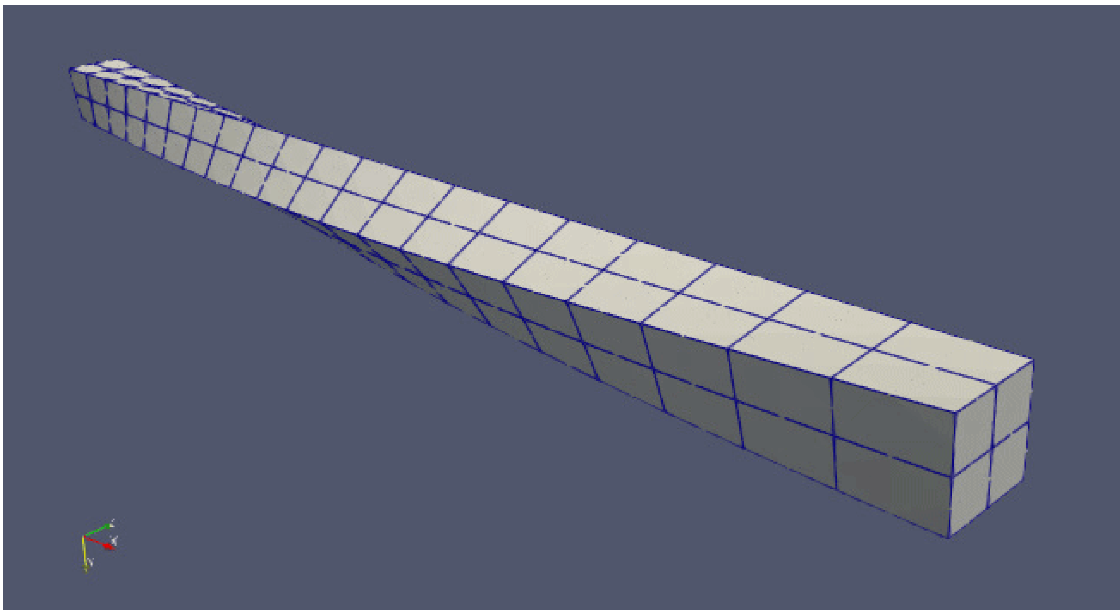

Fig. 6.12: Oscillations of a beam when releasing a torsion

CHAPTER

SEVEN

MULTI-DOMAIN METHODS

So far, we only considered a single bulk domain, sometimes with additional equations at the boundaries or with an external Lagrange multiplier. However, a lot of physical problems must be modeled by multiple domains that are coupled at the mutual interfaces. This section is devoted to these kind of problems.

7.1 Temperature conduction through two bodies of different conductivity

For a simple start, a static (i.e. non-moving) one-dimensional mesh consisting of two domains will be considered. The mesh should contain two interval domains `"domainL"` and `"domainR"`, ranging from 0 to x_I and from x_I to L respectively, which are connected at a mutual interface `"interface"` at x_I. We use the `MeshTemplate` class to provide such a mesh (cf. Section 4.3 for details):

```
from pyoomph import *
from pyoomph.equations.poisson import *

class TwoDomainMesh1d(MeshTemplate):
    def __init__(self,Ntot=100,xI=1,L=2,left_domain_name="domainA",right_domain_name=
↪"domainB"):
            super(TwoDomainMesh1d,self).__init__()
            self.Ntot, self.xI, self.L = Ntot, xI, L
            self.left_domain_name,self.right_domain_name=left_domain_name,right_
↪domain_name

    def define_geometry(self):
            xI=self.nondim_size(self.xI)
            L=self.nondim_size(self.L)
            NA=round(self.Ntot*xI/L) # number of elements on domainA calculated from␣
↪total number

            domainA=self.new_domain(self.left_domain_name)
            domainB=self.new_domain(self.right_domain_name)

            # Generate nodes
            nodesA=[self.add_node_unique(x) for x in numpy.linspace(0,xI,NA)]
            for x0,x1 in zip(nodesA, nodesA[1:]):
                    domainA.add_line_1d_C1(x0,x1)    # and elements by pairs of nodes

            # same for domainB
            nodesB=[self.add_node_unique(x) for x in numpy.linspace(xI,L,self.Ntot-
↪NA)]
            for x0,x1 in zip(nodesB, nodesB[1:]):
```

(continues on next page)

(continued from previous page)

```
                        domainB.add_line_1d_C1(x0,x1)

                # marking boundaries
                self.add_nodes_to_boundary("left",[nodesA[0]])
                self.add_nodes_to_boundary("interface",[nodesB[0]]) # coordsB[0] is␣
↪actually = coordsA[-1]
                self.add_nodes_to_boundary("right",[nodesB[-1]])
```

Note how we create two domains with the `new_domain()` calls and add line elements to both of these domains. During the latter, we use `zip` with a shifted node list to get the nodes in pairs, i.e. (node0,node1), (node1,node2), etc., to build the elements. It is important to note, since `add_node_unique()` will not create new nodes if a node is already existing at a point, that `domainA[-1]` is the very same node as `domainB[0]`. Thereby, this node, which is marked to be the interface `"interface"`, is part of both domains.

> **Warning:** If you want to couple the equations on different domains at mutual interfaces, all involved domains have to be generated within the very same `MeshTemplate` (or `GmshTemplate`). It is not possible to couple e.g. two `LineMesh` instances at an interface, even not if the position and the name of the interface is matching.

On the domains A and B, we want to solve the (nondimensional) temperature conduction equations, i.e. the Poisson equations

$$\nabla\cdot(k_\mathrm{A}\nabla T_\mathrm{A})=0 \quad \text{on domainA}$$
$$\nabla\cdot(k_\mathrm{B}\nabla T_\mathrm{B})=0 \quad \text{on domainB}$$

with the, in general different, thermal conductivities k_A and k_B. The solution shall be subject to the boundary conditions $T_\mathrm{A}(0)=0$ and $T_\mathrm{B}(L)=1$. At the mutual interface `"interface"` at x_I, we want to have a continuous temperature, i.e. $T_\mathrm{A}(x_\mathrm{I})=T_\mathrm{B}(x_\mathrm{I})$. While the former boundary conditions can be realized by trivial `DirichletBC`, the latter requires some additional consideration, since it involves the temperature field on two different domains. We can write the boundary condition as constraint with an associated Lagrange multiplier λ defined on the interface `"interface"`. As usual, the constraint can be thought as minimization of the Lagrange multiplier contribution

$$\lambda\ (T_\mathrm{A}-T_\mathrm{B})$$

with respect to λ, T_A and T_B. Let the corresponding test functions be η, Θ_A and Θ_B, then the corresponding weak terms read

$$\langle T_\mathrm{A}-T_\mathrm{B},\eta\rangle+\langle\lambda,\Theta_\mathrm{A}\rangle+\langle-\lambda,\Theta_\mathrm{B}\rangle \tag{7.1}$$

In pyoomph, we can write again an `InterfaceEquations` class for this:

```
class ConnectTAtInterface(InterfaceEquations):
    def define_fields(self):
        self.define_scalar_field("lambda","C2") # Lagrange multiplier

    def define_residuals(self):
        my_field,my_test=var_and_test("T") # T on the domain where this␣
↪InterfaceEquations object is attached to
        opp_field,opp_test=var_and_test("T",domain=self.get_opposite_side_of_
↪interface()) # T on the interface, but evaluated in the opposite domain
        lagr,lagr_test=var_and_test("lambda") # Lagrange multiplier
        self.add_residual(weak(my_field-opp_field,lagr_test)) # constraint T_my-
↪T_opp=0
        self.add_residual(weak(lagr,my_test)) # Lagrange Neumann contribution to␣
```

(continues on next page)

(continued from previous page)

```
↪the inside domain
                self.add_residual(weak(-lagr,opp_test)) # Lagrange Neumann contribution␣
↪to the outside domain
```

We introduce again the Lagrange multiplier λ at the interface and add the weak contributions to the residuals. Later on, the `ConnectTAtInterface` object will be added to the `"interface"` of either `"domainA"` or `"domainB"`, i.e. to `@"domainA/interface"` or `@"domainB/interface"`. In both domains, we will have the temperature field `var("T")` defined. To distinguish between the fields on the inside (i.e. the domain where the `ConnectAtInterface` is attached to) and the outside (i.e. the opposite domain), we must use `get_opposite_side_of_interface()` for the `domain` to clearly state that we want to get the temperature of the opposite side of the interface, i.e. the temperature `var("T")` at the `"interface"`, but evaluated at the opposite domain. Alternatively, we could have used `var("T",domain="|.")` as shortcut. It will also return the temperature field of the opposite side of the interface. To access the opposite bulk domain instead, use `get_opposite_parent_domain()` as `domain` or the shortcut `var("T",domain="|..")`.

The driver code is quite trivial

```
class TwoDomainTemperatureConduction(Problem):
    def __init__(self):
        super(TwoDomainTemperatureConduction,self).__init__()
        self.conductivityA=0.5  # thermal conductivity of domain A
        self.conductivityB=2    # thermal conductivity of domain B

    def define_problem(self):
        self.add_mesh(TwoDomainMesh1d())

        # Assemble equations domainA
        eqsA=TextFileOutput()
        eqsA+=PoissonEquation(name="T",space="C2",coefficient=self.conductivityA,
↪source=0)
        eqsA+=DirichletBC(T=0)@"left"

        # and equations of domainB
        eqsB=TextFileOutput()
        eqsB+=PoissonEquation(name="T",space="C2",coefficient=self.conductivityB,
↪source=0)
        eqsB+=DirichletBC(T=1)@"right"

        # Interface connection. Must be added to one side of the interface, i.e.␣
↪alternatively to eqsB
        eqsA+=ConnectTAtInterface()@"interface"

        self.add_equations(eqsA@"domainA")
        self.add_equations(eqsB@"domainB")


if __name__=="__main__":
    with TwoDomainTemperatureConduction() as problem:
        problem.solve()
        problem.output()
```

We add only one mesh, but assemble two Poisson equations, each with a different `coefficient` and with different `DirichletBC` terms. At the very end, the equations are restricted to `"domainA"` and `"domainB"`, respectively. The `ConnectTAtInterface` can be either added to `"domainA/interface"` or `"domainB/interface"`, but not on both simultaneously, since this would overconstrain the problem. The result is plotted in Fig. 7.1.

In terms of physics within this problem, we wonder of course, whether the heat flux $\vec{q}=-k\nabla T$ is indeed the same across

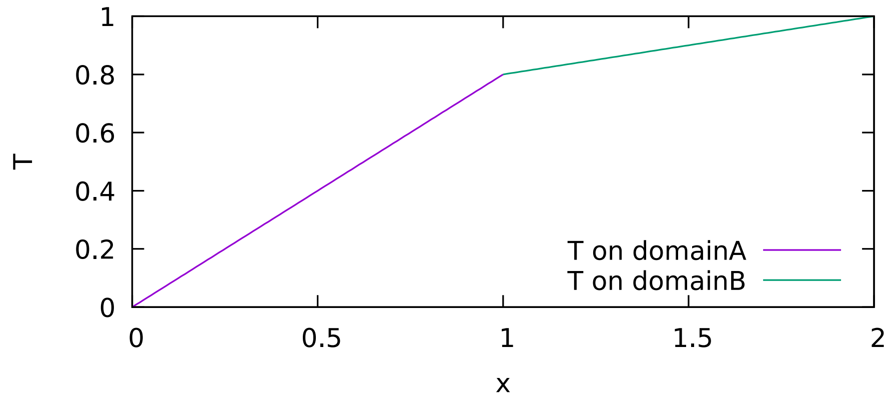

Fig. 7.1: Temperature conduction in two different domains with different conductivity, coupled at the mutual interface.

the interface. Since the normals in `"domainA"` and `"domainB"` at the `"interface"` obey the relation $\vec{n}_\mathrm{A} = -\vec{n}_\mathrm{B}$, a continuous heat flux would mean that

$$(\vec{q}_\mathrm{A} - \vec{q}_\mathrm{B}) \cdot \vec{n}_\mathrm{A} = -k_\mathrm{A}\partial_x T_\mathrm{A} + k_\mathrm{B}\partial_x T_\mathrm{B} = 0\,. \tag{7.2}$$

From the results in Fig. 7.1, we see that $\partial_x T_\mathrm{A} = 0.8$ and $\partial_x T_\mathrm{B} = 0.2$, and due to $k_\mathrm{A} = 0.5$ and $k_\mathrm{B} = 2$, it is indeed fulfilled. This is not just coincidence! From the weak form (7.1) of the enforced continuity of T, we see that we impose λ as Neumann flux to `"domainA"` and $-\lambda$ to `"domainB"`. The Neumann flux is exactly the heat flux and so continuity of this relation is actually a result of the enforcing. This also works, if the temperatures have a prescribed offset ΔT in the enforcing, which would read

$$\langle T_\mathrm{A} - T_\mathrm{B} - \Delta T, \eta\rangle + \langle \lambda, \Theta_\mathrm{A}\rangle + \langle -\lambda, \Theta_\mathrm{B}\rangle$$

Hence, when enforcing continuity of fields across interfaces this way, one automatically gets the correct physics, here the continuity of the transported heat across the interface. However, (7.2) would be violated if the weak forms of the Poisson equations would be e.g. multiplied by different factor in both domains, since this factor would also affect the Neumann term. Therefore, one has to pay attention.

Warning: Due to the above argument, one should not be tempted to set different scalings via `set_scaling()` of the `Problem` class or by the `Scaling` in dimensional problems. This can easily invalidate the continuity of the Neumann flux, which can lead to unphysical behavior. If one uses the same scale for non-dimensionalization of e.g. the temperature, e.g. by setting `set_scaling(T=1*kelvin)` at `Problem` level, this issue can be circumvented.

Note: It is cumbersome to write a coupling interface like the `ConnectTAtInterface` here for every field you want to connect at inter-domain interfaces. pyoomph has already the predefined class `ConnectFieldsAtInterface`, which allows enforcing continuity of scalar fields. In the current problem, we could just use `ConnectFieldsAtInterface("T")` instead of our custom class `ConnectTAtInterface`.

7.2 Propagation of an ice front

The next problem is very related to the previous one. We will again solve two temperature conduction equations, but this time, condition (7.2) will be slightly different. Furthermore, we will consider also the temporal behavior and use physical dimensions.

In fact, we want to solve the propagation of an ice front, i.e. how ice is solidifying or melting in presence of a temperature gradient. Since this phase transition obviously leads to a growth of the ice domain and a corresponding shrinkage of the liquid domain or vice versa, an moving mesh/ALE method will be used.

Mathematically, we have the transient heat conduction equations

$$\rho^\phi c_p^\phi \partial_t T^\phi = \nabla \cdot \left(k^\phi \nabla T\right) \quad (7.3)$$

where the phase superscript ϕ can be either ice or liquid, depending of whether we apply this equation on either the `"ice"` or the `"liquid"` domain. The equation can be easily cast into its weak form and implemented:

```
from temperature_conduction import * # To get the mesh from the previous example
from pyoomph.expressions.units import * # units for dimensions
from pyoomph.equations.ALE import * # moving mesh equations


class ThermalConductionEquation(Equations):
    def __init__(self,k,rho,c_p):
        super(ThermalConductionEquation,self).__init__()
        # store conductivity, mass density and spec. heat capacity
        self.k,self.rho,self.c_p=k,rho,c_p

    def define_fields(self):
        # Note the testscale here: We want to nondimensionalize the entire equation␣
↪by the scale "thermal_equation"
        # which will be set at the problem level
        self.define_scalar_field("T","C2",testscale=1/scale_factor("thermal_equation
↪"))

    def define_residuals(self):
        T,T_test=var_and_test("T")
        self.add_residual(weak(self.rho*self.c_p*partial_t(T),T_test)+weak(self.
↪k*grad(T),grad(T_test)))
```

Note that we bind the test scale of the temperature field `"T"` to `1/scale_factor("thermal_equation")`. This means essentially, that (7.3) is multiplier by a factor $1/S$ during non-dimensionalization, i.e. that we actually solve

$$S^{-1}\rho^\phi c_p^\phi \partial_t T^\phi = S^{-1}\nabla \cdot \left(k^\phi \nabla T\right) \quad (7.4)$$

One could now choose e.g. $S = \rho^{\text{ice}} c_p^{\text{ice}}[T]/[t]$, where $[T]$ is the scaling of the temperature field, e.g. $1\,\text{K}$ and $[t]$ is the characteristic time scale, e.g. $1\,\text{s}$. Thereby, the nondimensional lhs would have a unity factor. Alternatively, we can also set the factor of the nondimensional conduction term on the rhs to unity by selecting $S = k^{\text{ice}}[T]/[X]^2$ with the spatial scale X. Of course, one can also use the properties of the `"liquid"` domain instead of the `"ice"` domain. Eventually, S will be set at `Problem` level with the `set_scaling(thermal_equation=...)` method. Thereby, on both domains, the equations will have the same test scale, i.e. are nondimensionalized with respect to the same scale. That way, the problem regarding the consistency of the heat flux at the interface, as discussed in the previous example, will be circumvented. Therefore, this approach is a good practice.

Next, we must couple the interface motion, i.e. the propagation of the ice front, with the heat fluxes. The interface x_{I} will move, according to

$$\partial_t x_{\text{I}} = \frac{k^{\text{ice}} \partial_x T^{\text{ice}} - k^{\text{liq}} \partial_x T^{\text{liq}}}{\rho^{\text{ice}} \Lambda},$$

where Λ is the latent heat of solidification. We have used ρ^{ice} in the denominator, since the liquid will actually be subject to a tiny normal velocity at the interface due to the density difference. But this small contribution is disregarded here, since only conduction equations are solved.

As usual in pyoomph, we should write this equation independent of the chosen coordinate system to make this equation applicable to any problem. This is obviously given by

$$\vec{n}\cdot\partial_t\vec{x}_{\text{I}}=\frac{k^{\text{ice}}\nabla T^{\text{ice}}-k^{\text{liq}}\nabla T^{\text{liq}}}{\rho^{\text{ice}}\Lambda}\cdot\vec{n}\,,$$

In this formulation with interface normal $\vec{n}$, we also notice that it is a constraint for the normal motion of the mesh, whereas the tangential motion is not affected. Since it is a constraint, the typical Lagrange multiplier approach is again the way to take. As usual, with $\vec{\chi}$ and η being the test functions of the mesh position and the Lagrange multiplier λ, respectively, we get the weak formulation for the constraint:

$$\left\langle \vec{n}^{\text{ice}}\cdot\partial_t\vec{x}-\frac{k^{\text{ice}}\nabla T^{\text{ice}}\cdot\vec{n}-k^{\text{liq}}\nabla T^{\text{liq}}\cdot\vec{n}^{\text{ice}}}{\rho^{\text{ice}}\Lambda},\eta\right\rangle+\left\langle\lambda,\vec{n}^{\text{ice}}\cdot\vec{\chi}\right\rangle \tag{7.5}$$

The implementation is rather straight-forward:

```python
class IceFrontSpeed(InterfaceEquations):
    required_parent_type=ThermalConductionEquation    # Must have ThermalConductionEquation on the inside bulk
    required_opposite_parent_type = ThermalConductionEquation # and ThermalConductionEquation on the outside bulk

    def __init__(self,latent_heat):
        super(IceFrontSpeed, self).__init__()
        self.latent_heat=latent_heat

    def define_fields(self):
        self.define_scalar_field("_lagr_interf_speed","C2",scale=1/test_scale_factor("mesh"),testscale=scale_factor("temporal")/scale_factor("spatial"))

    def define_residuals(self):
        n=var("normal")
        x,xtest=var_and_test("mesh")
        l,ltest=var_and_test("_lagr_interf_speed")
        k_in=self.get_parent_equations().k           # conductivity of the inside domain
        rho_in=self.get_parent_equations().rho       # density of the inside domain
        k_out=self.get_opposite_parent_equations().k # conductivity of the outside domain
        T_bulk_in=var("T",domain=self.get_parent_domain())    # temperature in the inside bulk
        T_bulk_out = var("T", domain=self.get_opposite_parent_domain()) # temperature in the outside bulk
        speed=dot(k_in*grad(T_bulk_in)-k_out*grad(T_bulk_out),n)/(rho_in*self.latent_heat)
        self.add_residual(weak(dot(mesh_velocity(),n)-speed,ltest))
        self.add_residual(weak(l,dot(xtest,n)))
```

with the `required_parent_type` and `required_opposite_parent_type`, we inform pyoomph that it is only allowed to attach this constraint to an interface that has as `TemperatureConductionEquation` on both the inside bulk and the outside bulk of this interface. Otherwise, an error will be thrown. Due to these statements, we also get automatically the inside and outside `TemperatureConductionEquation` of the bulk phases when calling `get_parent_equations()` and `get_opposite_parent_equations()`. This is used to obtain the required properties k^{ϕ} and ρ in the `define_residuals()` method here. The interface property `latent_heat`, however, has to be passed to the constructor and is stored internally.

The scaling has to fit, i.e. upon non-dimensionalization of (7.5), all weak forms must yield non-dimensional results. Indeed, if we scale λ with the inverse of the scaling of χ and nondimensionalize the test function η as $\eta = [T]/[X]\tilde{\eta}$, all units will cancel out in (7.5).

There is another very relevant aspect to consider, namely:

> **Warning:** One fundamental aspect is that we want to take bulk gradient for the ∇T terms in (7.5). Since we are on an interface, i.e. on a manifold with co-dimension 1, the simple statement `grad(var("T"))` would expand to the surface gradient $\nabla_S T$ of temperature field of the inside domain (cf. Section 4.3.2), which will be always tangential to $\vec{n}$. The bulk gradients are only obtained if the temperature fields of the bulk phases are passed to `grad()`. These can be obtained by adding `get_parent_domain()` and `get_opposite_parent_domain()` (for the inside and outside bulk, respectively) as `domain=` keyword argument in the bindings via `var()`. Alternatively, you can also use `domain=".."` instead of `domain=self.get_parent_domain()` and `domain="|.."` instead of `domain=self.get_opposite_parent_domain()`.

In the constructor of the `Problem` class, nothing spectacular happens. We just initialize a few default parameters:

```
class IceFrontProblem(Problem):
    def __init__(self):
        super(IceFrontProblem,self).__init__()

        # properties of the ice
        self.rho_ice=915*kilogram/(meter**3) # mass density
        self.k_ice=2.22*watt/(meter*kelvin) # thermal conductivity
        self.cp_ice=2.050*kilo*joule/(kilogram*kelvin)       # spec. heat capacity

        # properties of the liquid
        self.rho_liq=999.87*kilogram/(meter**3)
        self.k_liq=0.5610*watt/(meter*kelvin)
        self.cp_liq=4.22*kilo*joule/(kilogram*kelvin)

        self.T_eq=0*celsius # Melting point
        self.latent_heat= 334 *joule/gram # Latent heat of melting/solidification

        self.L=1*milli*meter # domain length
        self.front_start_fraction=0.3 # initial relative position of the front
        self.T_left=-1*celsius # left and right temperatures
        self.T_right=1*celsius
```

In the `define_problem()` method, we have to set the scales for nondimensionalization and we make use of a `for` loop to construct similar equations on both domains:

```
def define_problem(self):
    # Mesh: a dimensional size and xI is set, also the domains are renamed
    self.add_mesh(TwoDomainMesh1d(L=self.L,xI=self.L*self.front_start_fraction,left_
↪domain_name="ice",right_domain_name="liquid"))
    self.set_scaling(spatial=self.L,temporal=100*second) # Nondimensionalize space␣
↪and time by these quantities
    self.set_scaling(T=kelvin) # Temperature scale
    # Now, we define the scale "thermal_equation", by what both thermal equations␣
↪will be divided
    # We take the conduction term of the ice as reference here
    self.set_scaling(thermal_equation=scale_factor("T")*self.k_ice/scale_factor(
↪"spatial")**2)

    # Create similar equations on both domains
```

(continues on next page)

(continued from previous page)

```
    # wrap the domain name and the corresponding properties
    domain_props=[["ice",self.k_ice,self.rho_ice,self.cp_ice,self.T_left],
                  ["liquid",self.k_liq,self.rho_liq,self.cp_liq,self.T_right]]
    for (domain_name,k,rho,cp,T_init) in domain_props: # iterate over the entries
        eqs=TextFileOutput() # Output
        eqs+=ThermalConductionEquation(k,rho,cp) # thermal transport eq
        eqs+=LaplaceSmoothedMesh()  # mesh motion
        eqs+=InitialCondition(T=T_init) # initial condition
        eqs+=SpatialErrorEstimator(T=1) # spatial adaptivity
        eqs+=DirichletBC(T=self.T_eq)@"interface" # melting point at the interface
        self.add_equations(eqs@domain_name) # add the equations

    # Dirichlet conditions
    self.add_equations(DirichletBC(T=self.T_left,mesh_x=0)@"ice/left")
    self.add_equations(DirichletBC(T=self.T_right,mesh_x=self.L)@"liquid/right")

    # Interface equations
    interf_eqs=IceFrontSpeed(self.latent_heat) # Front speed equation
    interf_eqs+=ConnectMeshAtInterface() # Connect the mesh at xI

    # We could also add it on "liquid/interface", but then we must use -self.latent_
↪heat in the IceFrontSpeed
    self.add_equations(interf_eqs@"ice/interface")
```

The interface equations consist of an instance of our just developed class `IceFrontSpeed` and the predefined class `ConnectMeshAtInterface`. The latter will introduce Lagrange multipliers so that the nodes of the `"liquid"` and `"ice"` domain at the mutual `"interface"` will be enforced to coincide. Without this, only the `"ice"` mesh would move, whereas the `"liquid"` mesh would remain static. Alternatively to adding `interf_eqs@"ice/interface"` to the problem, we could also add `interf_eqs` to the `"liquid"` side of the `"interface"`. In that case, however, we would have to negate the `latent_heat`.

The code to run this problem is simple, but we use temporal and spatial adaptivity to well resolve the initial temperature discontinuity at the `"interface"`:

```
if __name__=="__main__":
    with IceFrontProblem() as problem:
        problem.run(1000*second,startstep=0.00001*second,outstep=True,temporal_
↪error=1,spatial_adapt=1,maxstep=2*second)
```

The results are shown in Fig. 7.2.

7.3 Melting of an ice cylinder with natural convection

While is has been a lot of work to develop the rather simple example of an propagating ice front, we can now harvest the fruits of our labor by re-using these equations in higher dimensions and different geometries and add even more equations to the system.

As an example system, we consider the melting of an ice cylinder in a cylindrical bath of water. This has been investigated in Ref. [46], resulting in intriguing scalloped ice shapes due to a Kelvin-Helmholtz instability. We hence transfer the previous system into an axisymmetric variant, with an ice cylinder in the center and a liquid domain outside. In the liquid, also buoyancy driven flow will be relevant, where the density anormaly of water is important.

First of all, a mesh is required. We are lazy guys here: Although the geometry would allow to build the elements by hand, we just use the `GmshTemplate` to construct a mesh via gmsh (cf. Section 4.3.4). :

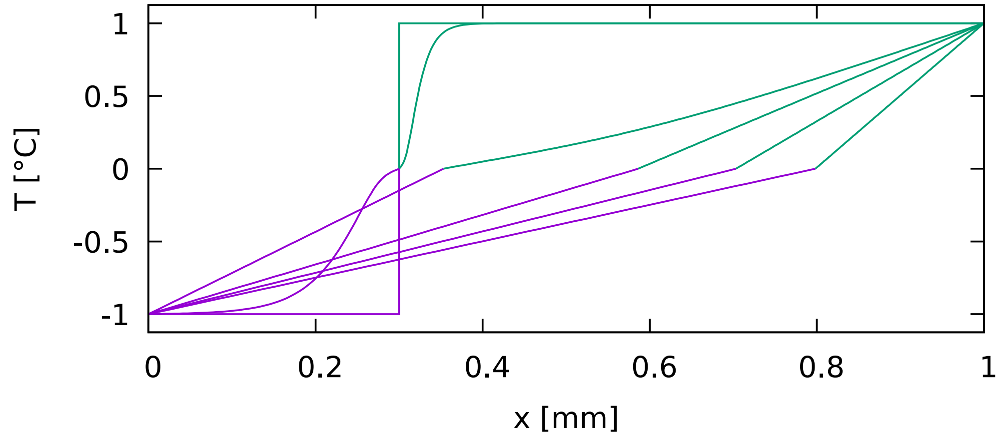

Fig. 7.2: Propagation of the front between solid ice (left) and liquid water (right) due to a temperature gradient at different times.

```
from temperature_conduction_propagation import * # Get some equations from the␣
↪previous example
from pyoomph.equations.navier_stokes import * # We also need a Navier-Stokes equation

# We are lazy and use the Gmsh approach here instead of adding the elements by hand
class TwoDomainMeshAxi(GmshTemplate):
    def __init__(self,R1,R2,H,resolution):
        super(TwoDomainMeshAxi, self).__init__()
        self.R1,self.R2,self.H=R1,R2,H
        self.default_resolution=resolution

    def define_geometry(self):
        p00, p0H=self.point(0,0), self.point(0,self.H)
        pR10, pR1H=self.point(self.R1,0), self.point(self.R1,self.H)
        pR20, pR2H = self.point(self.R2, 0), self.point(self.R2, self.H)
        self.create_lines(pR10,"ice_bottom",p00,"axisymm",p0H,"ice_top",pR1H,
↪"interface",pR10,"liquid_bottom",pR20,"liquid_side",pR2H,"liquid_top",pR1H)
        self.plane_surface("ice_bottom","axisymm","interface","ice_top",name="ice")
        self.plane_surface("liquid_bottom", "liquid_side", "interface", "liquid_top",␣
↪name="liquid")
```

If you have read Section 4.3.4, nothing spectacular is happening here.

Next, we do not only solve thermal conduction, but also thermal convection in the liquid domain. We therefore must add the term $\vec{u}\cdot\nabla T$ to our previously developed `ThermalConductionEquation`. This is easiest by inheriting from the latter class:

```
# Augment the conduction equation by advection for the liquid phase
class ThermalAdvectionConductionEquation(ThermalConductionEquation):
    def __init__(self,k,rho,c_p,wind=var("velocity")):
        super(ThermalAdvectionConductionEquation, self).__init__(k=k,rho=rho,c_p=c_p)
        self.wind=wind

    def define_residuals(self):
        super(ThermalAdvectionConductionEquation, self).define_residuals() # define␣
↪the conduction equation
        T,Ttest=var_and_test("T")
        self.add_residual(weak(self.rho*self.c_p*dot(self.wind,grad(T)),Ttest)) #␣
```

(continues on next page)

(continued from previous page)

```
↪Just add the advection term
```

Next, the problem will be defined. Although the problem is entire different from the previous one, let us re-use it to copy the physical parameters as e.g. the conductivities and the latent heat:

```
# We inherit from the IceFrontProblem to take over the physical parameters. The␣
↪problem will be quite different:
class IceConvectionProblem(IceFrontProblem):
    def __init__(self):
        super(IceConvectionProblem, self).__init__() # this will set all properties␣
↪from the parent class
        self.L=4*centi*meter # L is now the cylinder height
        self.R1=1*centi*meter # radius of the ice cylinder
        self.R2=3*centi*meter # outer radius of the liquid cylinder

        self.T_ice=-1*celsius # initial ice cylinder temperature
        self.T_liq=6*celsius

        self.mu_liq=1*milli*pascal*second
        # Water density for buoyancy calculations
        Trel = (var("T") - self.T_eq) / kelvin # bind the relative temperature␣
↪(measured in Kelvin)
        # Fit for the density anomaly
        self.rho_grav = (0.999849 + 5.77393e-05 * Trel - 7.18258e-06 * Trel ** 2) *␣
↪gram / (centi * meter )** 3
        self.gravity=9.81*meter/second**2 * vector(0,-1) # gravity direction and␣
↪strength

        self.resolution=0.05 # mesh resolution
```

Besides copying the parameters from the previous problem, where `L` is now used for the height of the cylinder, we need two radii, the viscosity of water and a fit for the density anomaly as function of the temperature. Therefore, we normalize the actual relative temperature $T - T_{\mathrm{eq}}$ by the unit K and plug this into a fit for the liquid water mass density between 0 °C and 20 °C.

The `define_problem()` method starts again by adding the mesh, but this time we have a two-dimensional mesh and switch to an `"axisymmetric"` coordinate system. The scales are set as in the previous problem, but we require additional scales for the `"velocity"` and `"pressure"` fields:

```
def define_problem(self):
    # Two-dimensional mesh
    self.add_mesh(TwoDomainMeshAxi(self.R1,self.R2,self.L,self.resolution))
    self.set_coordinate_system("axisymmetric") # axisymmetric coordinate system

    # Similar to the previous problem, scales for nondimensionalization
    self.set_scaling(spatial=self.R1,temporal=1*second)
    self.set_scaling(T=kelvin)
    self.set_scaling(thermal_equation=scale_factor("T") * self.k_ice / scale_factor(
↪"spatial") ** 2)
    self.set_scaling(velocity=scale_factor("spatial")/scale_factor("temporal"))
    self.set_scaling(pressure=self.mu_liq*scale_factor("velocity")/scale_factor(
↪"spatial"))
```

Next, the ice equations are assembled. It is essentially the same, except that we must add a few extra `DirichletBC` terms to fix the mesh at the top and bottom boundary. Furthermore, there is no `DirichletBC` for the temperature here, except the equilibrium temperature at the `"interface"`. The ice cylinder will just warm up to 0 °C over the course of the simulation:

```
self.add_mesh(TwoDomainMeshAxi(self.R1,self.R2,self.L,self.resolution))
self.set_coordinate_system("axisymmetric") # axisymmetric coordinate system

# Similar to the previous problem, scales for nondimensionalization
self.set_scaling(spatial=self.R1,temporal=1*second)
self.set_scaling(T=kelvin)
self.set_scaling(thermal_equation=scale_factor("T") * self.k_ice / scale_factor(
↪"spatial") ** 2)
self.set_scaling(velocity=scale_factor("spatial")/scale_factor("temporal"))
self.set_scaling(pressure=self.mu_liq*scale_factor("velocity")/scale_factor("spatial
↪"))

# Equations for the ice domain
ice_eqs=MeshFileOutput() # Output
ice_eqs+=ThermalConductionEquation(self.k_ice,self.rho_ice,self.cp_ice) # thermal␣
↪conduction
ice_eqs +=InitialCondition(T=self.T_ice) # initially at ice temperature
ice_eqs+=PseudoElasticMesh() # Mesh motion
ice_eqs+=DirichletBC(mesh_x=0)@"axisymm" # fix mesh at axis of symmetry
ice_eqs += DirichletBC(mesh_y=0) @ "ice_bottom" # and at the bottom
ice_eqs += DirichletBC(mesh_y=self.L) @ "ice_top" # and the top
ice_eqs += DirichletBC(T=self.T_eq)@"interface" # melting temperature at interface
```

The liquid equations are analogous, except that we use our new class `ThermalAdvectionConductionEquation` for the convection term and also add a `NavierStokesEquations` for the flow, together with no-slip boundary conditions at all interfaces. In reality, the density difference between ice and liquid would give rise to a non-zero normal velocity, when ice melts or solidifies, but this is not considered here, since this contribution is tiny. One could enforce this velocity jump via a Lagrange multiplier, but then, we also would allow for outflow somewhere in the domain to compensate for the gained/lost volume, i.e. to be able to satisfy the continuity equation. Since we do not allow any outflow, also one pressure degree of freedom must be fixed to remove the nullspace of the pressure (cf. Section 4.4.4):

```
# Equations for the liquid domain
liq_eqs=MeshFileOutput() # output
liq_eqs+=ThermalAdvectionConductionEquation(self.k_liq,self.rho_liq,self.cp_liq) #␣
↪thermal conduction + advection
# Navier-Stokes including Boussinesq-like gravity term
liq_eqs+=NavierStokesEquations(mass_density=self.rho_liq,dynamic_viscosity=self.mu_
↪liq,bulkforce=self.gravity*self.rho_grav)
liq_eqs+=InitialCondition(T=self.T_liq) # Liquid temperature as initial condition
liq_eqs+=PseudoElasticMesh() # Mesh motion
liq_eqs+=DirichletBC(mesh_y=0,velocity_x=0,velocity_y=0)@"liquid_bottom" # no-slip␣
↪and fixed mesh at all boundaries
liq_eqs += DirichletBC(mesh_y=self.L,velocity_x=0,velocity_y=0) @ "liquid_top"
liq_eqs += DirichletBC(mesh_x=self.R2,T=self.T_liq,velocity_x=0,velocity_y=0)@"liquid_
↪side"
liq_eqs+=DirichletBC(T=self.T_eq,velocity_x=0,velocity_y=0)@"interface" # here the␣
↪mesh is not fixed, but the temperature is
liq_eqs += DirichletBC(pressure=0) @ "liquid_top/liquid_side" # For pure DirichletBCs,
↪ we must fix one pressure degree
```

Optionally, we can simplify the problem: Since the front will mainly move in radial direction, we can remove all degrees of freedom associated with the y-coordinate of the mesh:

```
# Since we know that the mesh mainly moves in y-direction, we can speed up the␣
↪calculation by removing the motion in y-direction
ice_eqs += DirichletBC(mesh_y=True)
liq_eqs += DirichletBC(mesh_y=True)
```

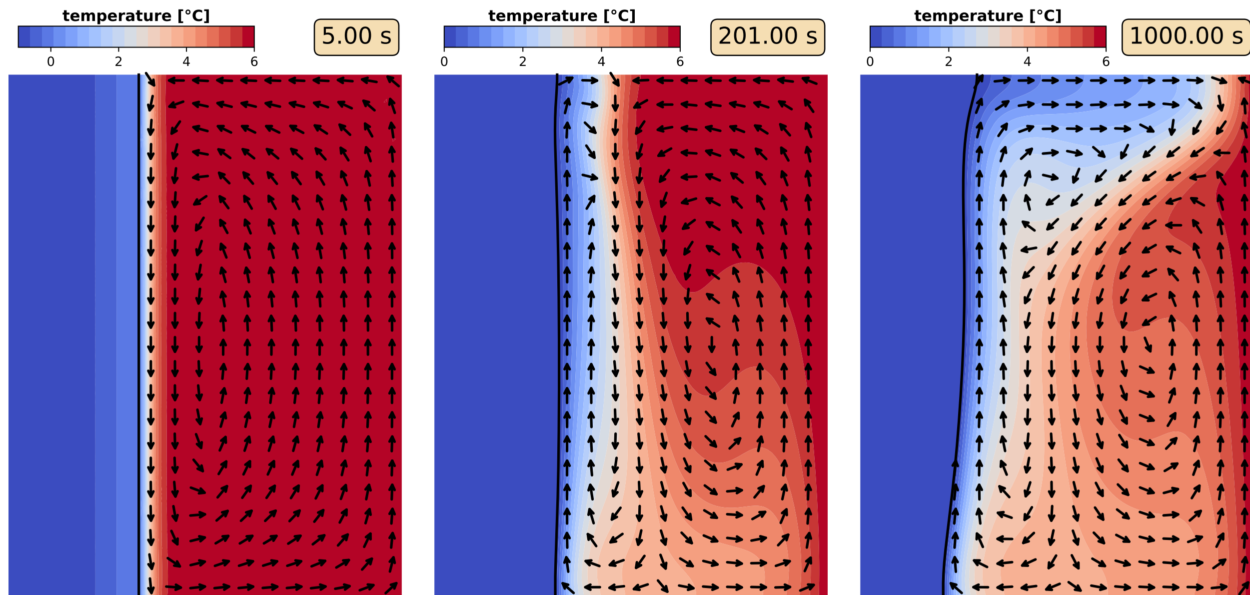

Fig. 7.3: Dynamics of an ice cylinder melting in a liquid bath along with natural convection due to the density anomaly.

Thereby, we have less degrees of freedom in our system and the computation will speed up.

Finally, the interface equations are added as in the previous example and all equations are added to the problem:

```
# Interface: Connect the mesh position and impose the front motion
interf_eqs=ConnectMeshAtInterface()
interf_eqs+=IceFrontSpeed(self.latent_heat)

# Add it to the ice side of the interface
ice_eqs+=interf_eqs@"interface"

# and add all equations to the problem
self.add_equations(ice_eqs@"ice"+liq_eqs@"liquid")
```

The code for execution is trivial:

```
if __name__=="__main__":
    with IceConvectionProblem() as problem:
        problem.run(400*second,outstep=True,startstep=1*second,maxstep=20*second,
↪temporal_error=1)
```

The corresponding results are shown in Fig. 7.3. Obviously, the interface indeed does not recede in a straight manner, but is deformed due to the natural convection. Based on the results, one can obviously simplify the problem by neglecting the ice phase, since it becomes isothermal at $T_{\mathrm{eq}} = 0$ very quickly. This would involve the modification of the `IceFrontSpeed`, but since this chapter is on multi-domain problems, it will not be addressed here.

7.4 Connecting two fluid domains

In Section 6.5, we have developed the equations of a free liquid surface. However, so far we have not considered the presence of the opposite side of the free surface, which is usually also a fluid. So let's do this now…

Let us consider two phases `"inside"` and `"outside"` with respect of the free surface, which we will address by `"interface"` or `"inside/interface"`, since we will add all necessary `InterfaceEquations` on the `"interface"` from the domain `"inside"`. Actually, the kinematic boundary condition (6.4) has to be fulfilled on both sides. When we use again a phase superscript ϕ, we can state

$$\vec{n}^{\mathrm{i}} \cdot \left(\vec{u}^{\phi} - \dot{\vec{x}}\right) = 0 \quad \text{for} \quad \phi = \text{i,o for inside and outside}\,, \tag{7.6}$$

where the normal $\vec{n}^{\mathrm{i}}$ is pointing from `"inside"` to `"outside"` (actually, which direction does not matter here). However, if one just adds one `KinematicBC` object per side, i.e. on `"inside/interface"` and `"outside/interface"`, the kinematic boundary condition would be fulfilled on both sides, but once the mesh is connected with a `ConnectMeshAtInterface` object, the normal mesh motion would be over-constrained.

Both kinematic boundary conditions together can be also formulated by the kinematic boundary condition for the `"inside"` domain and the requirement that the normal velocity is continuous across the `"interface"`:

$$\begin{aligned} \vec{n}^{\mathrm{i}} \cdot \left(\vec{u}^{\mathrm{i}} - \dot{\vec{x}}\right) &= 0 \\ \vec{n}^{\mathrm{i}} \cdot \left(\vec{u}^{\mathrm{i}} - \vec{u}^{\mathrm{o}}\right) &= 0 \end{aligned} \tag{7.7}$$

When we further demand that the tangential velocity should be also continuous (which is a reasonable assumption), (7.7) reads

$$\vec{u}^{\mathrm{i}} - \vec{u}^{\mathrm{o}} = \vec{0} \tag{7.8}$$

which can be enforced by a vectorial Lagrange multiplier field $\vec{\lambda}$ (with test function $\vec{\eta}$) as usual

$$\left\langle \vec{u}^{\mathrm{i}} - \vec{u}^{\mathrm{o}}, \vec{\eta} \right\rangle + \left\langle \vec{\lambda}, \vec{v}^{\mathrm{i}} \right\rangle + \left\langle -\vec{\lambda}, \vec{v}^{\mathrm{o}} \right\rangle\,. \tag{7.9}$$

The dynamic boundary condition (6.6) must be generalized to

$$\vec{n}^{\mathrm{i}} \cdot \left[-p^{\mathrm{i}}\mathbf{1} + \mu^{\mathrm{i}}\left(\nabla\vec{u}^{\mathrm{i}} + (\nabla\vec{u}^{\mathrm{i}})^{\mathrm{t}}\right)\right] + \vec{n}^{\mathrm{o}} \cdot \left[-p^{\mathrm{o}}\mathbf{1} + \mu^{\mathrm{o}}\left(\nabla\vec{u}^{\mathrm{o}} + (\nabla\vec{u}^{\mathrm{o}})^{\mathrm{t}}\right)\right] = \sigma\kappa\vec{n}^{\mathrm{i}} + \nabla_S\sigma\,, \tag{7.10}$$

where the lhs can be simplified due to $\vec{n}^{\mathrm{i}} = -\vec{n}^{\mathrm{o}}$. Indeed, analogous to the heat flux in Section 7.1, this equation is automatically fulfilled if we add a `DynamicBC` to the `"inside/interface"` and couple the velocities on both sides via (7.9): On the `"inside/interface"`, we then have the Neumann contribution

$$\vec{n}^{\mathrm{i}} \cdot \left[-p^{\mathrm{i}}\mathbf{1} + \mu^{\mathrm{i}}\left(\nabla\vec{u}^{\mathrm{i}} + (\nabla\vec{u}^{\mathrm{i}})^{\mathrm{t}}\right)\right] = \sigma\kappa\vec{n}^{\mathrm{i}} + \nabla_S\sigma + \vec{\lambda}$$

and on the `"outside/interface"`

$$\vec{n}^{\mathrm{o}} \cdot \left[-p^{\mathrm{o}}\mathbf{1} + \mu^{\mathrm{o}}\left(\nabla\vec{u}^{\mathrm{o}} + (\nabla\vec{u}^{\mathrm{o}})^{\mathrm{t}}\right)\right] = -\vec{\lambda}\,.$$

It is apparent, that the sum of the latter two equations indeed gives the dynamic boundary condition (7.10).

So the only additional work we have to do is to couple the velocities by a Lagrange multiplier, which can be implemented in pyoomph as

```
from pyoomph import *
from pyoomph.expressions import *
from pyoomph.equations.navier_stokes import *
from pyoomph.equations.ALE import *
```

(continues on next page)

(continued from previous page)

```
class EnforceContinuousVelocity(InterfaceEquations):
	def define_fields(self):
		self.define_vector_field("_couple_velo","C2")

	def define_residuals(self):
		l,ltest=var_and_test("_couple_velo")
		ui,uitest=var_and_test("velocity") # inner velocity at the interface
		uo,uotest=var_and_test("velocity",domain=self.get_opposite_side_of_
↪interface()) # outer velocity
		self.add_residual(weak(ui-uo,ltest)+weak(l,uitest)-weak(l,uotest))

	def before_assigning_equations_postorder(self, mesh):
		# pin Lagrange multiplier if both velocities are pinned
		# we have to iterate over the directions x,y,z (if present)
		for d in ["x","y","z"][0:self.get_nodal_dimension()]:
			self.pin_redundant_lagrange_multipliers(mesh,"_couple_velo_"+d,
↪"velocity_"+d,opposite_interface="velocity_"+d)
```

Again, we have to tell `var()` with `domain=self.get_opposite_side_of_interface()` that we want to have the outer velocity field, whereas without this argument, the inner velocity is meant. When both velocities are prescribed with a `DirichletBC`, i.e. pinned, the Lagrange multiplier would either lead to a null space (if the strongly imposed velocities matching) or to the absence of any solution (if the strongly imposed velocities are mismatching). We have to do this per component, which is done in the `for` loop. Here, only the components are considered, which are actually present in the actual nodal dimension of the mesh via `get_nodal_dimension()`. We also use the argument `opposite_interface=...` to tell `pin_redundant_lagrange_multipliers()` that each component of the Lagrange multiplier $\vec{\lambda}$ is only redundant if both the inside and the outside velocity component is pinned. Note that the predefined `InterfaceEquations` class `ConnectMeshAtInterface` does exactly the same but on the mesh positions.

The rest of the code is rather straight-forward, however, we use the `RectangularQuadMesh` with a `lambda callable` as argument for `name`:

```
class TwoLayerFlowProblem(Problem):
	def __init__(self):
		super(TwoLayerFlowProblem, self).__init__()
		self.W=1
		self.H1=0.1
		self.H2=0.1
		self.quad_size=0.01

	def define_problem(self):
		domain_names=lambda x,y: "lower" if y<self.H1 else "upper" # Name lower␣
↪half lower, upper half upper
		self.add_mesh(RectangularQuadMesh(N=[math.ceil(self.W/self.quad_size),␣
↪math.ceil((self.H1+self.H2)/self.quad_size)], size=[self.W, self.H1+self.H2],
↪name=domain_names,boundary_names={"lower_upper":"interface"}))
```

With this argument, we can split the `RectangularQuadMesh` into multiple domains. The `callable` passed to `name` receives nondimensional x, y coordinates of the element centers and is expected to return the name of the domain. Interfaces between the different domains are automatically marked by `"domain1_domain2"` with the adjacent domain names `"domain1"` and `"domain2"` (in alphabetic order). Here, we rename this interface `"lower_upper"` via the `boundary_names dict` to `"interface"`.

The equations are assembled and added:

```
# Add the same required equations to both domains
for dom in ["lower","upper"]:
        eqs=LaplaceSmoothedMesh()
        eqs+=MeshFileOutput()
        eqs+=DirichletBC(mesh_x=True)
        eqs += DirichletBC(velocity_x=0) @ "left"  # no in/outflow at the sides
        eqs += DirichletBC(velocity_x=0) @ "right"
        self.add_equations(eqs@dom)

# Different fluids
l_eqs = NavierStokesEquations(mass_density=0.01, dynamic_viscosity=1)  # NS equations
u_eqs = NavierStokesEquations(mass_density=0.01, dynamic_viscosity=0.01)  # NS␣
↪equations

# no slip at top and bottom
l_eqs += DirichletBC(velocity_x=0, velocity_y=0, mesh_y=0) @ "bottom"  # no slip at␣
↪bottom and fix the mesh there
u_eqs += DirichletBC(velocity_x=0, velocity_y=0, mesh_y=self.H1+self.H2) @ "top"  #␣
↪no slip at bottom and fix the mesh there
l_eqs += DirichletBC(pressure=0) @"bottom/left" # pin one pressure degree

# Free surface, mesh connection and velocity connection
l_eqs += NavierStokesFreeSurface(surface_tension=1) @ "interface"  # free surface at␣
↪the top
l_eqs += ConnectMeshAtInterface()@"interface"
l_eqs += EnforceContinuousVelocity()@"interface"

# Deform the initial mesh
X, Y = var(["lagrangian_x", "lagrangian_y"])
l_eqs += InitialCondition(mesh_y=Y * (1 + 0.25 * cos(2 * pi * X)))  # small height␣
↪with a modulation
u_eqs += InitialCondition(mesh_y=Y+ (self.H1+self.H2-Y)*(0.25 * cos(2 * pi * X)))  #␣
↪small height with a modulation
self.add_equations(l_eqs @ "lower" + u_eqs @ "upper")  # adding it to the system
```

We use the predefined `NavierStokesFreeSurface` instead of our free surface consisting of `KinematicBC` and `DynamicBC` developed in Section 6.5, but it does essentially the same. With the `EnforceContinuousVelocity`, the velocities are enforced to be continuous, whereas the Lagrange multiplier λ_x in x-direction will be pinned to 0 automatically on the `"left"` and `"right"`, since both inside and outside velocity are prescribed by a `DirichletBC`.

The run code reads

```
if __name__=="__main__":
    with TwoLayerFlowProblem() as problem:
        problem.run(50,outstep=True,startstep=0.25)
```

and the results are depicted in Fig. 7.4.

Tip: There is a similar example case in oomph-lib at https://oomph-lib.github.io/oomph-lib/doc/navier_stokes/two_layer_interface/html/index.html. However, in their case, a single mesh (i.e. domain) is used, but with varying viscosity and mass densities per elements. The free surface is just added at an interior interface. Thereby, the continuity of the velocity field and the mesh position across the interface is automatically fulfilled, i.e. no Lagrange multipliers to connect the velocity and mesh are necessary. However, since the pressure has a jump at the interface due to the Laplace pressure, the pressure space must be discontinuous, i.e. in the oomph-lib example, Crouzeix-Raviart instead of Taylor-Hood elements are used. While it is possible to follow the same approach in pyoomph, it is not discussed here. The moment, mass transfer between both phases is considered, the normal velocity has a jump at the interface as well, provided the mass densities in both phases are different. Then, Lagrange multipliers are definitely required.

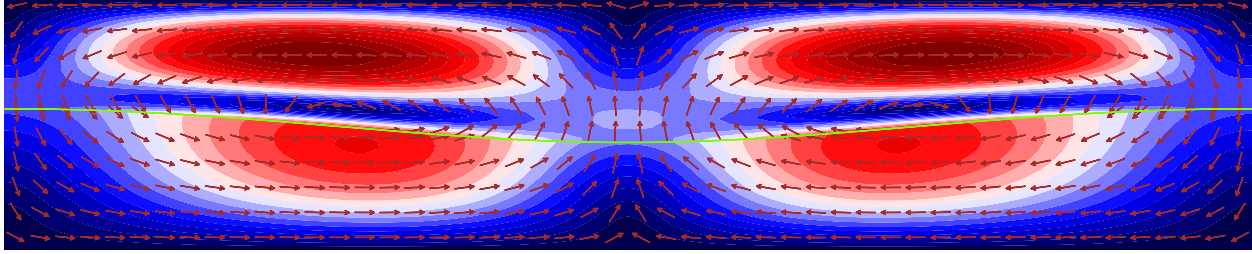

Fig. 7.4: Two-layer flow with connected velocity at the interface. By the velocity coupling, the stress is correctly distributed between both domains.

If you are interested in a pyoomph version of oomph-lib's way of implementing it on a single domain, you find the corresponding code here: `two_layer_flow_single_domain.py`

7.5 Stokes' law for a falling droplet with insoluble surfactants

In Section 5.3.2, a transient Stokes law is solved. We will now slightly modify this problem and exchange the rigid spherical object by a droplet. For simplicity, we still assume that the capillary number is small, i.e. the droplet does not deform when falling. Thereby, we do not have to consider moving meshes. The necessary modifications are first in the mesh class, so that also a droplet mesh is generated:

```
# Modification (besides renaming some interfaces): Add a droplet mesh
self.line(pSnorth,pSsouth,name="droplet_axisymm")
self.plane_surface("droplet_axisymm", "droplet_outside",name="droplet")  # droplet␣
↪domain
```

Then, in the constructor `__init__` of the problem class, we add e.g. the viscosity for the droplet as well:

```
# Modifications: Also define viscosity of the droplet
self.droplet_radius = 0.25 * milli * meter  # radius of the spherical object
self.droplet_density = 1200 * kilogram / meter ** 3  # density of the sphere
self.outer_density = 1000 * kilogram / meter ** 3  # density of the liquid
self.droplet_viscosity = 5 * milli * pascal * second  # viscosity
self.outer_viscosity = 1 * milli * pascal * second  # viscosity
self.surface_tension=50*milli*newton/meter # Also define a surface tension (although␣
↪it does not matter on a static mesh)
```

In the `define_problem()` method, we remove the no-slip boundary condition and add the equations of the droplet:

```
        # Do not add a no slip
        # eqs += DirichletBC(velocity_x=0, velocity_y=0) @ "droplet_outside"  # and␣
↪no-slip on the object

        # droplet equations
        inertia_correction = self.droplet_density * vector([0, 1]) * partial_t(U)
        deqs=MeshFileOutput()
        deqs+=NavierStokesEquations(dynamic_viscosity=self.droplet_viscosity,mass_
↪density=self.droplet_density,bulkforce=inertia_correction)
        deqs+=DirichletBC(velocity_x=0)@"droplet_axisymm"
        deqs+=NavierStokesFreeSurface(surface_tension=self.surface_tension,static_
↪interface=True)@"droplet_outside"
        deqs+=ConnectVelocityAtInterface()@"droplet_outside"
```

(continues on next page)

(continued from previous page)

```
        deqs += DragContribution(U) @ "droplet_outside"  # Also add the tractions␣
↪from the inside to the drag
        self.add_equations(deqs@"droplet")


if __name__ == "__main__":
    with FallingDropletProblem() as problem:
        problem.run(0.5*second,startstep=0.05*second,outstep=True)  # solve and output
```

When we have static meshes (i.e. no equations for the mesh positions are added), we must add `static_interface=True` to the `NavierStokesFreeSurface`. With that, the action of the Lagrange multiplier of the kinematic boundary condition (6.5) will be added to the velocity, not the mesh motion. Since the latter is not allowed to move, only an adjustment of the velocity can guarantee the kinematic boundary condition to hold. Thereby, the kinematic boundary condition is effectively replaced by a zero normal flow condition, i.e. (4.14).

The results of this modifications are shown in Fig. 7.5. The value of the surface tension does not matter here, since on static meshes, the droplet will remain perfectly spherical, i.e. corresponding to a zero capillary number. However, surface tension gradients can still drive Marangoni flow, as we will see next.

To obtain surface tension gradients, we add an insoluble surfactant to the droplet-outside interface. The corresponding transport equation for a surfactant interface concentration Γ with surface diffusivity D_S reads

$$\partial_t\Gamma + \nabla_S \cdot (\vec{u}\Gamma) = \nabla_S \cdot (D_S \nabla_S \Gamma) \qquad (7.11)$$

In pyoomph, it is again trivial to implement this:

```
from falling_droplet import *
from pyoomph.expressions.phys_consts import *


class SurfactantTransportEquation(InterfaceEquations):
    def __init__(self):
        super(SurfactantTransportEquation, self).__init__()
        self.D=1e-9*meter**2/second # diffusivity

    def define_fields(self):
        self.define_scalar_field("Gamma","C2",testscale=scale_factor("temporal")/
↪scale_factor("Gamma"))

    def define_residuals(self):
        u=var("velocity") # velocity at the interface
        G,Gtest=var_and_test("Gamma")
        self.add_residual(weak(partial_t(G)+div(u*G),Gtest))
        self.add_residual(weak(self.D*grad(G), grad(Gtest)))
```

Again, if we solve it on an interface, i.e. a manifold with co-dimension, `div()` and `grad()` will expand to their surface counterparts as discussed in Section 4.3.2.

> **Warning:** This transport equation is only valid without mass transfer. When mass transfer is considered, the normal interface motion would not coincide with the normal interface velocity. Thereby, this equation must be changed.

The only thing we have to do is adding this equation to the interface, setting a suitable initial condition and scaling and modifying the surface tension so that it depends on Γ. The simplest surface tension relation with surfactants is just $\sigma(\Gamma) = \sigma_0 - RT\Gamma$ with the gas constant R and temperature T. Instead of modifying the problem class directly, we just add the `additional_equations` and modify the `surface_tension` in the run script of the simulation:

```
if __name__ == "__main__":
    with FallingDropletProblem() as problem:
        Gamma0=1*micro*mol/meter**2
        problem.set_scaling(Gamma=Gamma0)

        add_
↪ieqs=SurfactantTransportEquation()+InitialCondition(Gamma=Gamma0)+MeshFileOutput()
        problem.additional_equations+=add_ieqs@"droplet/droplet_outside"

        T=20*celsius
        problem.surface_tension=50*milli*newton/meter-gas_constant*T*var("Gamma")

        problem.run(0.5*second,startstep=0.05*second,outstep=True)  # solve and output
```

The surfactants get advected to the top of the droplet and hamper the flow in the droplet, cf. Fig. 7.5.

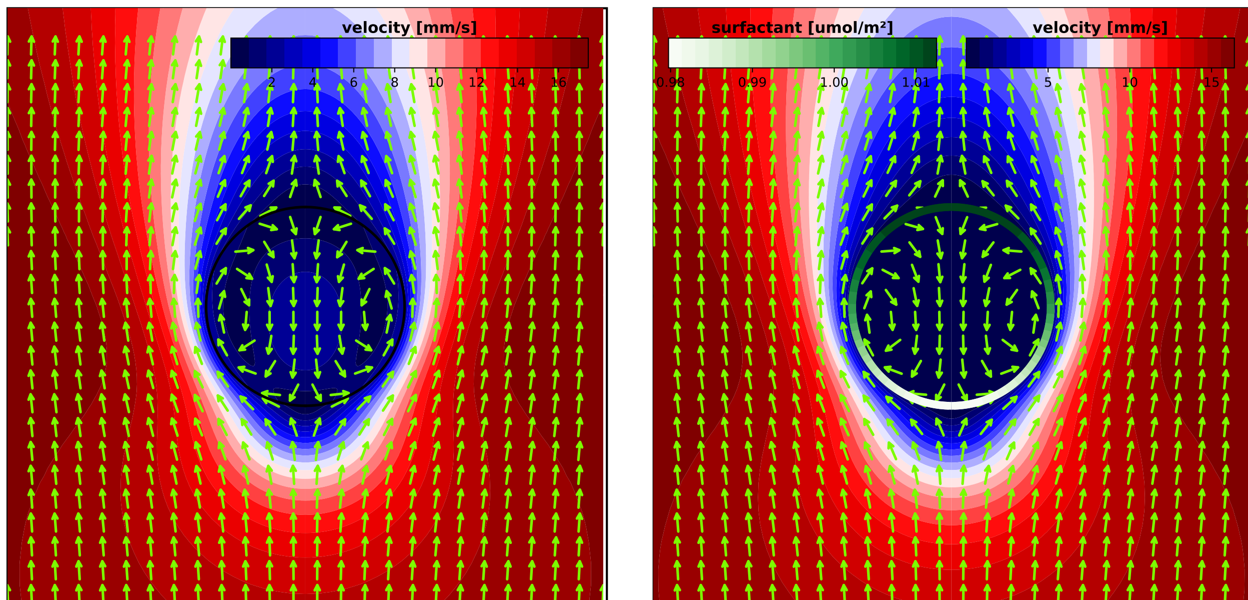

Fig. 7.5: Droplet falling due due gravitiy without (left) and with surfactants (right).

7.6 Evaporation of a sessile droplet

Evaporation, as mass transfer between phases in general, always requires more than a single phase. We have seen how we can couple different equations on individual domains and hence we can solve evaporating sessile droplets. To that end, we have to solve the Navier-Stokes equation for the droplet, the interface dynamics and for the gas phase, the diffusion equation of the vapor field. While there is actually also advective vapor transport, it usually negligible for temperatures below the boiling point.

First of all, we must have a mesh that comprises a sessile droplet and a surrounding gas domain:

```
from pyoomph import *
from pyoomph.equations.ALE import * # Moving mesh
from pyoomph.equations.navier_stokes import * # Flow
from pyoomph.expressions.units import * # units
from pyoomph.utils.dropgeom import *  # utils to calculate droplet contact angle from␣
```

(continues on next page)

(continued from previous page)

```
↪height and radius, etc.
from pyoomph.equations.advection_diffusion import * # for the gas diffusion
from pyoomph.meshes.remesher import * # to remesh at large distortions


# A mesh of an axisymmetric droplet surrounded by gas
class DropletWithGasMesh(GmshTemplate):
    def __init__(self,droplet_radius,droplet_height,gas_radius,cl_resolution_factor=0.
↪1,gas_resolution_factor=50):
        super(DropletWithGasMesh, self).__init__()
        self.droplet_radius,self.droplet_height,self.gas_radius=droplet_radius,
↪droplet_height,gas_radius
        self.cl_resolution_factor,self.gas_resolution_factor=cl_resolution_factor,gas_
↪resolution_factor
        self.default_resolution=0.025
        #self.mesh_mode="tris"

    def define_geometry(self):
        # finer resolution at the contact line, lower at the gas far field
        res_cl=self.cl_resolution_factor*self.default_resolution
        res_gas=self.gas_resolution_factor*self.default_resolution
        # droplet
        p00=self.point(0,0) # origin
        pr0 = self.point(self.droplet_radius, 0,size=res_cl) # contact line
        p0h=self.point(0,self.droplet_height) # zenith
        self.circle_arc(pr0,p0h,through_point=self.point(-self.droplet_radius,0),name=
↪"droplet_gas") # curved interface
        self.create_lines(p0h,"droplet_axisymm",p00,"droplet_substrate",pr0)
        self.plane_surface("droplet_gas","droplet_axisymm","droplet_substrate",name=
↪"droplet") # droplet domain
        # gas dome
        pR0=self.point(self.gas_radius,0,size=res_gas)
        p0R = self.point(0,self.gas_radius,size=res_gas)
        self.circle_arc(pR0,p0R,center=p00,name="gas_infinity")
        self.line(p0h,p0R,name="gas_axisymm")
        self.line(pr0,pR0,name="gas_substrate")
        self.plane_surface("gas_substrate","gas_infinity","gas_axisymm","droplet_gas",
↪name="gas") # gas domain
```

Again, we use the `GmshTemplate` for that to create a mesh. We refine it near the contact line and make it coarser in the far field of the gas domain by adding the `size` keyword to the `point()` calls. The droplet-gas interface is made by a `circle_arc()` starting at the contact line and ending at the droplet apex. Instead of passing the `center` of the circle, we can also pass `through_point`, i.e. a third point which is also located on the circle (but potentially outside the segment). Here, we just use the mirrored contact line.

Next, for the problem class, we must define a few default properties:

```
class EvaporatingDroplet(Problem):
    def __init__(self):
        super(EvaporatingDroplet, self).__init__()
        # Droplet properties
        self.droplet_radius=0.5*milli*meter # base radius
        self.droplet_height=0.2*milli*meter # apex height
        self.droplet_density=1000*kilogram/meter**3 # mass density
        self.droplet_viscosity=1*milli*pascal*second # dyn. viscosity
        self.sliplength = 1 * micro * meter # slip length
```

(continues on next page)

(continued from previous page)

```
        # Gas properties
        self.gas_radius=5*milli*meter
        self.vapor_diffusivity=25.55e-6*meter**2/second
        self.c_sat=17.3*gram/meter**3 # saturated partial density of vapor
        self.c_infty=0.5*self.c_sat # partial vapor density far away

        # Interface and contact line properties
        self.surface_tension=72*milli*newton/meter # surface tension
        self.pinned_contact_line=True
        self.contact_angle=None # will be calculated from the height and radius if␣
↪not set
```

In the gas phase, we will solve the vapor diffusion equation for the partial vapor mass density c (`"c_vap"` in python) with units $\mathrm{kg/m^3}$ with saturated vapor $c = c_\mathrm{sat}$ at the liquid-gas interface and ambient vapor c_∞ far away. Then, the diffusive flux at the interface, i.e. $j = -\nabla c \cdot \vec{n}$ is the evaporation rate, i.e. the mass transfer per area and time, i.e. in $\mathrm{kg/m^2 \cdot s}$). Therefore, we bind it:

```
# Bind the evaporation rate
c_vap = var("c_vap", domain="gas")
n = var("normal")
self.evap_rate = -self.vapor_diffusivity * dot(grad(c_vap), n)
```

It is again important to tell `var()` that we want to evaluate `"c_vap"` in the `domain="gas"`, i.e. in the bulk domain. Otherwise, we might get the wrong gradient (i.e. the surface gradient instead the bulk gradient).

In the `define_problem()`, we first set some reasonable scales for non-dimensionalization and add the mesh:

```
def define_problem(self):
    # Settings: Axisymmetric and typical scales
    self.set_coordinate_system("axisymmetric")
    self.set_scaling(temporal=1*second,spatial=self.droplet_radius)
    self.set_scaling(velocity=scale_factor("spatial")/scale_factor("temporal"))
    self.set_scaling(pressure=self.surface_tension/scale_factor("spatial"))
    self.set_scaling(c_vap=self.c_sat)

    # Add the mesh
    mesh=DropletWithGasMesh(self.droplet_radius,self.droplet_height,self.gas_radius)
    mesh.remesher=Remesher2d(mesh) # add remeshing possibility
    self.add_mesh(mesh)

    # Calculate the contact angle if not set
    if self.contact_angle is None:
        self.contact_angle= DropletGeometry(base_radius=self.droplet_radius, apex_
↪height=self.droplet_height).contact_angle
```

We also calculate the equilibrium `contact_angle` if not set explicitly. This is used only if `pinned_contact_line` is `False`.

The droplet bulk equations are just Navier-Stokes with a moving mesh along with a free surface at the liquid-gas interface, a slip length condition at the substrate and a few `DirichletBC` terms:

```
# Droplet equations
d_eqs=MeshFileOutput() # Output
d_eqs+=PseudoElasticMesh() # Mesh motion
d_eqs+=NavierStokesEquations(mass_density=self.droplet_density,dynamic_viscosity=self.
↪droplet_viscosity) # flow
d_eqs+=DirichletBC(mesh_x=0,velocity_x=0)@"droplet_axisymm" # symmetry axis
```

(continues on next page)

(continued from previous page)

```
d_eqs += DirichletBC(mesh_y=0, velocity_y=0) @ "droplet_substrate" # allow slip, but␣
↪fix mesh
d_eqs += NavierStokesSlipLength(self.sliplength)@ "droplet_substrate" # limit slip by␣
↪slip length
d_eqs+=NavierStokesFreeSurface(surface_tension=self.surface_tension,mass_transfer_
↪rate=self.evap_rate)@"droplet_gas" # Free surface equation
d_eqs += ConnectMeshAtInterface() @ "droplet_gas" # connect the gas mesh to co-move
```

Note that we pass `mass_transfer_rate=evap_rate` to the `NavierStokesFreeSurface`. This augments the kinematic boundary condition (6.4) as follows:

$$\vec{n} \cdot \left(\vec{u} - \dot{\vec{x}}\right) = \frac{j}{\rho} . \tag{7.12}$$

Here, j is the `mass_transfer_rate`, i.e. bound to $j = -\nabla c \cdot \vec{n}$ in this particular problem and ρ is the density of the `NavierStokesEquations`, i.e. of the droplet. Thereby, the relative velocity between the liquid and the interface in normal direction is given by the lost mass.

For the contact line dynamics, we have two options, depending on the value of `pinned_contact_line`. If it is `False`, i.e. a free contact line, we just use the `NavierStokesContactAngle` as discussed in Section 6.6. If the contact line is pinned, i.e. `pinned_contact_line=True`, we have a problem: The free surface solves the augmented kinematic boundary condition (7.12) by adjusting the mesh positions (cf. (6.5)). However, if the contact line is pinned, the mesh should not move directly at the contact line. In order to fulfill both, the kinematic boundary condition with evaporation and the fixed position of the contact line, we adjust the radial velocity at the contact line so that the mesh does not move:

```
# Different contact line dynamics
if self.pinned_contact_line: # if pinned
    # Pinned contact line means mesh_x is fixed.
    # We enforce partial_t(mesh_x,ALE=False)=0 by adjusting the radial velocity at␣
↪the contact line
    cl_constraint=mesh_velocity()[0]-0
    d_eqs+=EnforcedBC(velocity_x=cl_constraint)@"droplet_gas/droplet_substrate"
else:
    d_eqs += NavierStokesContactAngle(contact_angle=self.contact_angle) @ "droplet_
↪gas/droplet_substrate"  # and constant contact angle
```

With the `EnforcedBC`, the radial velocity is adjusted so that `partial_t(var("mesh_x"), ALE=False)=mesh_velocity()[0]=0` holds, i.e. the contact line is stationary. Intrinsically, this is again done by a Lagrange multiplier within the `EnforcedBC`. Of course, this only works with a slip length boundary condition at the substrate, not with a no-slip condition. A no-slip condition would remove the possibility to add a traction to the radial velocity here. Both contact line models, i.e. the pinned and the freely moving constant contact angle condition do essentially the same: They impose a traction at the contact line. However, the `NavierStokesContactAngle` adds exactly the weak term that is required to attain the prescribed contact angle (cf. Section 6.6). With the `EnforcedBC`, we essentially enforce exactly that contact angle for which the contact line remains stationary.

The gas equations are just a diffusion equation, i.e. an `AdvectionDiffusionEquations` without `wind`, i.e. without any advection:

```
# Gas equations
g_eqs=MeshFileOutput() # output
g_eqs+=PseudoElasticMesh() # mesh motion
g_eqs+=AdvectionDiffusionEquations(fieldnames="c_vap",diffusivity=self.vapor_
↪diffusivity,wind=0) # diffusion equation
g_eqs += InitialCondition(c_vap=self.c_infty)
g_eqs+=DirichletBC(mesh_x=0)@"gas_axisymm" # fixed mesh coordinates at the boundaries
```

(continues on next page)

(continued from previous page)

```
g_eqs+=DirichletBC(mesh_y=0)@"gas_substrate"
g_eqs+=DirichletBC(mesh_x=True,mesh_y=True)@"gas_infinity"
g_eqs+=DirichletBC(c_vap=self.c_sat)@"droplet_gas"
g_eqs+=AdvectionDiffusionInfinity(c_vap=self.c_infty)@"gas_infinity"
```

Note how we impose saturated vapor strongly at the liquid-gas interface, whereas the ambient vapor is imposed by a `AdvectionDiffusionInfinity` equation. This equations mimics an infinite mesh by a Robin boundary condition. This condition can be derived by knowing that in three-dimensions, the diffusion equation will follow a $1/r$ behavior in the far field (with r being the distance from the droplet). Far away from the droplet, the vapor field will hence read $c = c_\infty + (c(R) - c_\infty)R/r$ for any reasonably large distance R and for $r > R$. Deriving this with respect to r and plugging it again into the expression gives the Robin condition

$$c(R) + R\partial_r c(R) = c_\infty \,.$$

This condition is implemented by the `AdvectionDiffusionInfinity` class as weak Neumann contribution.

Finally, we also add some remeshing options which invoke a mesh reconstruction whenever the mesh deforms to strongly and also output the volume and the interface data with evaporation to files:

```
# Control remeshing
d_eqs += RemeshWhen(RemeshingOptions(max_expansion=1.5, min_expansion=0.7))
g_eqs+=RemeshWhen(RemeshingOptions(max_expansion=1.5,min_expansion=0.7))

# Output of the volume evolution
d_eqs+=IntegralObservables(volume=1)
d_eqs+=IntegralObservableOutput(filename="EVO_droplet")

# Also output the interface data, along with the evaporation rate
d_eqs+=(LocalExpressions(evap_rate=self.evap_rate)+MeshFileOutput())@"droplet_gas"

self.add_equations(d_eqs@"droplet"+g_eqs@"gas")
```

The run script is trivial and the results are shown in Fig. 7.6:

```
if __name__=="__main__":
    with EvaporatingDroplet() as problem:
        problem.run(500*second,startstep=10*second,outstep=True,temporal_error=1)
```

7.7 Fluid-Structure Interaction

Having deformable solids (Section 6.7) and the Navier-Stokes equations on moving domains (Section 6.5) available, obviously, both can be combined for fluid-structure interaction scenarios. We consider a 2d channel with two leaflets that will deform by the flow and thereby change the flow as well. The problem case is rather short, just combining the Navier-Stokes equations in the liquid domain and the solid equations in the deformable leaflets:

```
from pyoomph import *
from pyoomph.equations.navier_stokes import *
from pyoomph.equations.ALE import *
from pyoomph.equations.solid import *

class SimpleFSIProblem(Problem):
    def __init__(self):
        super().__init__()
```

(continues on next page)

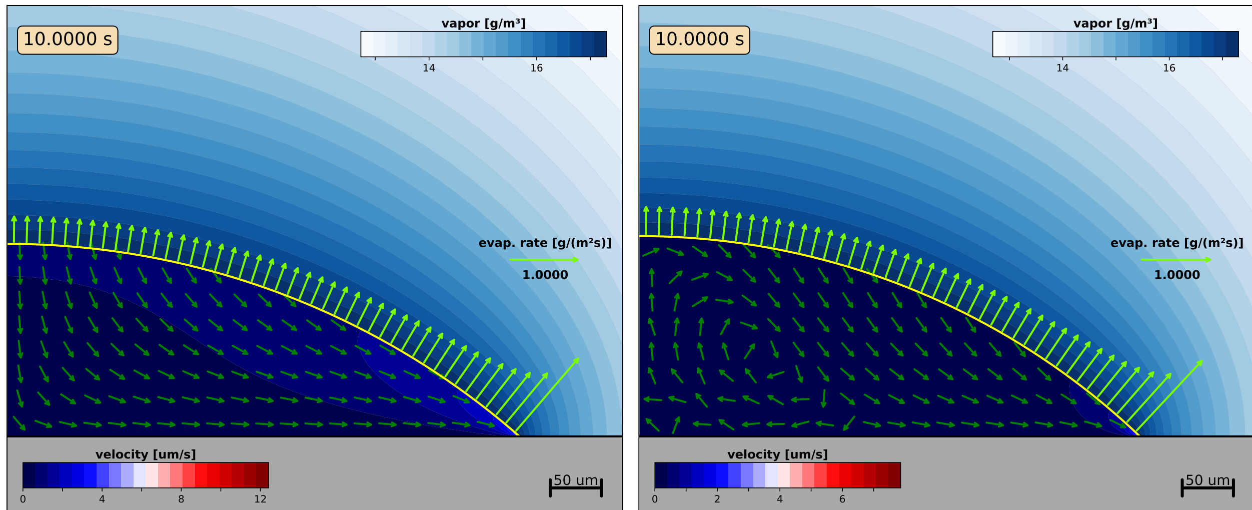

Fig. 7.6: Evaporating droplet with a pinned contact line (left) and with a constant contact angle (right).

(continued from previous page)

```
        self.Pinlet=15 # Inlet pressure
        self.claw=GeneralizedHookeanSolidConstitutiveLaw(E=40000,nu=0.3) # Solid␣
↪dynamics
        self.rho, self.mu=1000 , 1 # Liquid density and dynamic viscosity
        self.max_refinement_level=2 # Adaptivity level

    def define_problem(self):

        # Mark individual domains in the RectangularQuadMesh
        def domain_name(x,y):
            if (x>1.8 and x<2 and y<1.5) or (x>2.8 and x<3 and y>0.5):
                return "solid"
            else:
                return "liquid"
        self+=RectangularQuadMesh(size=[5,2],name=domain_name,N=[50,20])

        # Liquid equations
        leqs=MeshFileOutput()
        leqs+=NavierStokesEquations(mass_density=self.rho,dynamic_viscosity=self.mu)
        leqs+=PseudoElasticMesh()
        leqs+=PinMeshCoordinates()@["left","right"]
        leqs+=DirichletBC(mesh_y=0)@"bottom"
        leqs+=DirichletBC(mesh_y=2)@"top"
        leqs+=NoSlipBC()@["top","bottom"]
        leqs+=(DirichletBC(velocity_y=0)+NeumannBC(velocity_x=-self.Pinlet))@"left"
        leqs+=DirichletBC(velocity_y=0)@"right"

        # Solid equations
        seqs=MeshFileOutput()
        seqs+=DeformableSolidEquations(self.claw,mass_density=2,coordinate_space="C2",
↪scale_for_FSI=True)
        seqs+=PinMeshCoordinates()@"bottom"
        seqs+=PinMeshCoordinates()@"top"

        # Fluid-structure interaction at the mutual interface
```

(continues on next page)

(continued from previous page)

```
        leqs+=FSIConnection()@"liquid_solid"

        # Adaptivity
        leqs+=SpatialErrorEstimator(velocity=1)
        leqs+=RefineToLevel()@"liquid_solid"
        seqs+=RefineToLevel()@"liquid_solid"

        self+=leqs@"liquid"+seqs@"solid"


with SimpleFSIProblem() as problem:
    problem.run(50,outstep=0.5,temporal_error=1,spatial_adapt=1)
```

The real main part is the `FSIConnection`, which must be added to the liquid side of the mutual interface. In Section 7.4, we discussed how the enforcing of continous velociy between two liquid domains via Lagrange multipliers actually ensures the balance of tractions. The same idea is used in the `pyoomph.equations.solid.FSIConnection` interface. We enforce that the liquid velocity agrees with the solid velocity and thereby also ensure the balance of the tractions at the shared interface. Moreover, the fluid mesh is moved with the solid mesh. As discussed in Section 7.1, it is important to use the same scale of the test function on both sides to balance the tractions. Therefore, one has to set `scale_for_FSI=True` in the `DeformableSolidEquations`

Opposed to the `ConnectMeshAtInterface` class, which moves the nodes of the meshes on both sides, the `FSIConnection` only moves the nodes of the liquid mesh to match those of the solid mesh. Otherwise, the particular moving mesh dynamics of the fluid domain, which does not reflect any physics, would add additional unphysical tractions to the system.

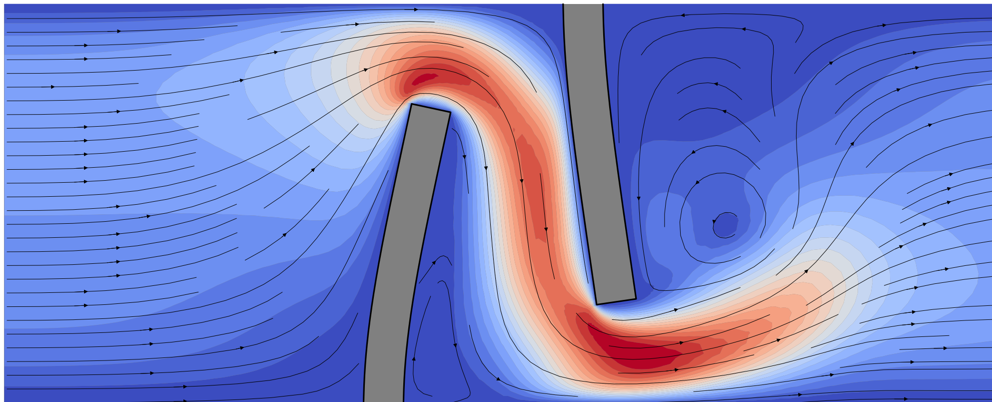

Fig. 7.7: Fluid-Structure Interaction

CHAPTER

EIGHT

MULTI-COMPONENT FLOW

pyoomph has a set of predefined equations, that allows to easily add flow of multi-component mixtures, potentially with mass transfer between different phases, surfactants and thermal effects. All these equations are just `Equations` or `InterfaceEquations` and all required knowledge to develop these by hand yourself was already introduced throughout this manual. Therefore, we will not go into the details of the implementation of these predefined equations, but feel free to have a look at the code `pyoomph.equations.multi_component`.

8.1 Bulk equations

In multi-component flow, we have a mixture of $\alpha = 1, \ldots, n$ components in each phase ϕ. In the bulk, the fluids properties, i.e. the density ρ, viscosity μ and the diffusion coefficients $D_{\alpha\beta}$ may depend on the local composition. Since the Navier-Stokes equations gives the mass-averaged velocity $\vec{u}$, it is beneficial to express the composition in terms of mass fractions $w_1, \ldots, w_n$. Each of these mass fractions range from 0 to 1 and furthermore, the sum $\sum_\alpha w_\alpha = 1$ holds. Thereby, we do not have to explicitly consider w_n, since $w_n = 1 - \sum_{\alpha=1}^{n-1} w_i$ holds.

The bulk equations for multi-component flow are hence a combination of the Navier-Stokes equations and $n-1$ advection-diffusion equations for the fluid composition:

$$\begin{aligned}
\rho\left(\partial_t \vec{u} + \nabla\vec{u}\cdot\vec{u}\right) &= -\nabla p + \nabla\cdot\left[\mu\left(\nabla\vec{u} + \nabla\vec{u}^{t}\right)\right] + \rho\vec{g} + \vec{f} \\
\partial_t \rho + \nabla\cdot(\rho\vec{u}) &= 0 \\
\rho\left(\partial_t w_\alpha + \vec{u}\cdot\nabla w_\alpha\right) &= \nabla\cdot\left(\rho\sum_\beta D_{\alpha\beta}\nabla w_\beta\right) \quad \text{for} \quad \alpha = 1, \ldots, n-1 \\
w_n &= 1 - \sum_\alpha^{n-1} w_\alpha
\end{aligned} \tag{8.1}$$

Here, $\vec{g}$ is a gravity vector and $\vec{f}$ is an additional arbitrary bulk force. Furthermore, if the thermal fluctuations are relevant, one can add a temperature equation

$$\rho c_p\left(\partial_t T + \vec{u}\cdot\nabla T\right) = \nabla\cdot(\lambda\nabla T) \tag{8.2}$$

All properties, the mass density ρ, the dynamic viscosity μ, the diffusion matrix $D_{\alpha\beta}$, the specific heat capacity c_p and the thermal conductivity λ may depend on the local composition w_α and the temperature T. In particular for gases, the properties also depends on the absolute pressure p_abs. Note that this is usually not the pressure p from the Navier-Stokes equations, since one usually choses the latter to be zero at stress free boundaries. For the properties of gases, it is often sufficient to approximate the absolute pressure by the ambient pressure p_amb (e.g. $p_\text{amb} = 1\,\text{atm}$). If pressure fluctuations are strong (i.e. p/p_amb is considerable), one could also cout set $p_\text{abs} = p_\text{amb} + p$.

Pyoomph has generates this entire system by multiple equations via `CompositionFlowEquations()`. If the flow shall not be considered, i.e. $\vec{u} = 0$ is a reasonable assumption (e.g. in the gas phase of an evaporating droplet), one can

remove the Navier-Stokes equations by using the `CompositionDiffusionEquations()` function instead. Both will require a fluid properties object as first argument, which comprises the (potentially varying) mass density, viscosity, diffusion matrix, etc. but also the initial composition of the mixture. Before addressing these classes in detail, let us first discuss how to create properties for fluid mixtures in the next section.

8.2 The material library and defining bulk properties

pyoomph has a material library that allows to specify material properties for liquids, gases, solids and surfactants. You can also specify composition-dependent properties for liquid or gas mixtures, which is relevant for multi-component flow problems.

8.2.1 Definition of a pure gaseous substance

Let us start by defining the parameters of air, where we consider - despite of knowing better - that air is just a pure substance. This is acceptable, if we want to mix water vapor with air to solve e.g. a vapor diffusion problem. But of course, we can also mix nitrogen with oxygen and possibly argon and more pure species if we want to resolve the local composition of the air, but this will be rarely necessary. To define air as a pure gaseous substance, we do the following:

```
from pyoomph.materials import * # Import the material API

# The following line will register this material to the material library
@MaterialProperties.register()
class PureGasAir(PureGasProperties): # Inherit from the PureGasProperties, which will␣
↪set the state of matter to gas
    name="air" # We must set the name here to identify the substance
    def __init__(self):
        super().__init__() # Call the parent constructor
        self.molar_mass = 28.9645 * gram / mol # Setting the molar mass
        self.dynamic_viscosity=0.01813 *milli*pascal*second # dynamic viscosity␣
↪(here a constant)
        self.mass_density=1.225*kilogram/meter**3 # Mass density
```

As you can see, we first have to import `pyoomph.materials` to get access to the material library and the base classes, as e.g. `PureGasProperties`. With the decorator `register()` of the generic `MaterialProperties` material class, pyoomph is instructed to register this material to the library. This registration is not persistent, i.e. you have to declare the class `PureGasAir` with the decorator in every code you want to use this material. However, you can put all your material definitions in a separate python file and `import` it. Some example materials are already defined in the file `pyoomph.materials.default_materials`.

We set the `name` of our substance to `"air"`, which is used as identifier of this gas.

> **Warning:** The `name` of materials may not contain any spaces or symbols (except the underscore and numeric characters). So `name="1,2-hexanediol"` is not possible, but `name="12_hexanediol"` is. The reason is that the name of the materials may also occur in the generated C code and C does not allow for arbitrary variable names.

Afterwards, we set the default properties in the constructor. For a gas, the molar mass (to convert e.g. from mole to mass fractions) and the dynamic viscosity and mass density (to solve flow problems) are relevant. Here, we set the values to constants, which of course limits the applicability of our material definition. At ambient pressures or temperatures deviating from the standard room conditions, this is hence not very accurate. We will improve that very soon by allowing pressure and temperature dependence.

Once the gas is defined, we can access this as follows

```
# Since the material is registered as pure gaseous material, we can load it as follows
air=get_pure_gas("air")

print("Dynamic viscosity:",air.dynamic_viscosity) # Printing some properties

air.mass_density=2*kilogram/meter**3 # Changing the properties by hand

air2=get_pure_gas("air") # Loading another instance of air
print("Mass densities",air.mass_density,air2.mass_density) # Compare the densities
```

You can hence create an instance of the pure gas `"air"` by calling `get_pure_gas()`. You can just access all the properties, but you can also change them. This can be useful to e.g. investigate the influence of some parameters. Note that in the above example, the second instance of the gas, i.e. `air2`, will not be affected by this change, i.e. the mass density will still be `1.225*kilogram/meter**3`. Thereby, you can e.g. modify the properties by hand only for one physical domain while keeping the other occurences of the very same substance untouched.

8.2.2 Properties as function of temperature and pressure

As already mentioned, using constant values for the properties is not very useful, as these constants are only valid within a small temperature and pressure range. Furthermore, any effects as e.g. natural convection or Marangoni flow requires that properties changes with e.g. the temperature or the pressure. For a pure substance, only the pressure and the temperature can influence the physical properties, so we will focus on these first. The (local) temperature can be accessed with `var("temperature")`, whereas the pressure is obtained by `var("absolute_pressure")`. As mentioned before, `var("pressure")` is usually used for the pressure in (Navier-)Stokes problems, which can be quite different, e.g. zero or even negative. `var("absolute_pressure")` is used for the thermodynamic pressure, which must be positive.

We can e.g. use these variables to utilize the ideal gas law for the mass density, $\rho = pM/(RT)$, by defining our air as follows

```
# Load the universal gas constant
from pyoomph.expressions.phys_consts import gas_constant

# Redefine our gas. Since a gas with name "gas" is already registered, we must pass␣
↪override=True here
@MaterialProperties.register(override=True)
class PureGasAir(PureGasProperties):
    name="air"
    def __init__(self):
        super().__init__()
        self.molar_mass = 28.9645 * gram / mol # Same as before, always a␣
↪constant

        # Use ideal gas law to get the density
        self.mass_density=var("absolute_pressure")*self.molar_mass/ (gas_
↪constant * var("temperature"))
        # Alternatively, we can just call self.set_mass_density_from_ideal_gas_
↪law() for that

        TKelvin = var("temperature") / kelvin # bind the temperature in kelvin␣
↪for the following fit expressions
        # Varying dynamic viscosity
        self.dynamic_viscosity=(0.0409424 + 0.00725803 * TKelvin - 4.12727e-06 *␣
↪(TKelvin) ** 2)* 1e-5 * pascal * second
```

(continues on next page)

(continued from previous page)

```
            # For thermal problems, also the following must be set. If only␣
↪isothermal problems are considered, it is not required
            self.thermal_conductivity=(-0.0217506 + 0.00984373 * TKelvin - 3.4318e-
↪06 * TKelvin * TKelvin)*1e-5 * kilo * watt / (meter * kelvin)
            self.specific_heat_capacity=1.005* kilo * joule / (kilogram * kelvin)
```

Obviously, we can just use the dimensional variables `var("temperature")` and `var("absolute_pressure")` to directly calculate the mass density by the ideal gas law, which is now valid for all temperatures and pressures (as long as air can be considered as an ideal gas). There is a method, `set_mass_density_from_ideal_gas_law()`, which does exactly this as a shortcut. For other properties, e.g. the dynamic viscosity, we use fitted data from experimental results. The fitted expression for the viscosity used here reads

$$\mu[10^{-5}\,\mathrm{Pa}\cdot\mathrm{s}] = 0.049424 + 0.00724803T[\,\mathrm{K}] - 4.12727{\times}10^{-6}T^2[\,\mathrm{K}^2]$$

To use the temperature as numeric value in K, we can just divide `var("temperature")/kelvin`. In the same way, we just multiply the fit result by 10^{-5} Pa · s to cast it into the unit required for the viscosity according to this fit.

When the temperature is solved, one usually also requires the thermal conductivity and the specific heat capacity, which are set to the properties `thermal_conductivity` and `specific_heat_capacity`. Note that the latter requires the heat capacity per mass, not per mole, but these can be easily converted using the molar mass.

Of course, these expressions are more complicated than a simple constant and it might cause additional computational time to consider the variations of the properties with the temperature and pressure (and fluid composition/surfactant concentration for mixtures). However, if we solve e.g. an isothermal and isobaric problem, pyoomph only evaluates these expressions at the selected temperature and pressure once, namely when generating the C code. Hence, in that case, one eventually ends up using constants again.

To evaluate a property at particular conditions, we can do the following:

```
air=get_pure_gas("air") # Load the new definition of "air"
print("VARIABLE DENSITY",air.mass_density) # Print the functional expression rho(p,T)

# Evaluate at a particular condition
rho_std=air.evaluate_at_condition("mass_density",temperature=18*celsius,absolute_
↪pressure=1*atm)
print("EVALUATED DENSITY",rho_std)
# To convert to a float (e.g. to write to a file), we just have to cancel out the␣
↪desired unit and call float(...)
print("EVALUATED DENSITY in (kg/m**3)",float(rho_std/(kilogram/meter**3)))
```

Note that `evaluate_at_condition()` can also take the expression `air.mass_density` (or `air.dynamic_viscosity`) instead of the string `"mass_density"` (or `"dynamic_viscosity"`). To cast a constant dimensional value into a `float`, you just cancel out the desired unit (e.g. `kilogram/meter**3`) and wrap it into a `float()` call. Of course, if we divide by e.g. `gram/(centi*meter)**3`, we get the numerical float value in $\mathrm{g/m^3}$ instead. You cannot convert a dimensional quantity into a float without dividing by the desired unit first and likewise, you cannot cast expressions to a float that still depend on the temperature or the pressure, i.e. having terms containing `var("...")` in it.

However, you can easily make a table of float values by scanning ranges of the temperature or the absolute pressure, e.g. by:

```
# Loop over Celsius values
for T_in_Celsius in [10,15,20,25,30]:
    # Evaluate at 1 atm and this temperature
```

(continues on next page)

(continued from previous page)

```
    rho_at_T=air.evaluate_at_condition(air.mass_density,temperature=T_in_
↪Celsius*celsius,absolute_pressure=1*atm)
    print(f"DENSITY AT T[C]= {T_in_Celsius} is {float(rho_at_T/(kilogram/meter**3))}␣
↪kg/m^3") #Print it (can also write to file)
```

Likewise, you can write this data to a file. Thereby, you can easily check whether the implemented expression indeed agrees with the experimental data. Note that we have made use of Python's *Literal String Interpolation (PEP 498)* here.

8.2.3 Definition of gaseous mixtures

If you want to solve e.g. the evaporation of a water droplet, you must account for water vapor (i.e. gaseous water) which can diffuse through the ambient gas, typically air. Before creating a gaseous mixture of these two substances, we first must ensure that both pure gaseous substances are defined. Therefore, we first load our old script, where we defined the air and subsequently define pure water in its gaseous phase:

```
from pyoomph.materials import *


# Pure air: see before (we took the simple constant expressions here, but it also␣
↪works for expressions depending on pressure and temperature)
@MaterialProperties.register()
class PureGasAir(PureGasProperties):
    name="air"
    def __init__(self):
        super().__init__()
        self.molar_mass = 28.9645 * gram / mol
        self.dynamic_viscosity=0.01813 *milli*pascal*second
        self.mass_density=1.225*kilogram/meter**3


# Create the pure gaseous water
@MaterialProperties.register()
class PureGasWater(PureGasProperties):
    name="water" # name it "water"
    def __init__(self):
        super().__init__()
        self.molar_mass = 18.01528*gram/mol # Molar mass is important to convert␣
↪to e.g. mole fractions

        # If only used in mixtures, we do not require the mass density and␣
↪dynamic viscosity of the pure substance here
```

Note that we skipped the definition of the mass density and the dynamic viscosity (as well at the thermal properties) for the `PureGasWater`. When we intend to only use it within a mixture, e.g. with air, the properties of the pure substance are not required. If we want to solve, however, a flow problem of pure gaseous water, i.e. above the boiling point where pure water can exist in the gas phase, we would require the mass density and dynamic viscosity of the pure substance for the Navier-Stokes equation. The molar mass, on the other hand, is always required - for each substance. It is important to convert mass into molar fractions and it is used for e.g. Raoult's law or the calculation of the mass density of mixtures by the ideal gas law.

Next, we move on defining the properties of a mixture of air and water in the gas phase:

```
# Create mixture properties that will apply if you mix gaseous water with gaseous air
@MaterialProperties.register()
class MixtureGasWaterAir(MixtureGasProperties): # MixtureGasProperties is the base␣
```

(continues on next page)

(continued from previous page)

```
↪class for gas mixtures
    components={"water","air"} # This class applied when mixing "water" and "air"
    # We can specify a passive component: We have to solve n-1 advection diffusion␣
↪equations for the mass fractions
    # The nth follows from 1 minus the others. We can select, which component is not␣
↪explicitly solved for
    passive_field="air"  # we choose air here

    # The constructor gets the pure properties as a dict {"water":PureGasWater␣
↪instance, "air":PureGasAir instance}
    def __init__(self,pure_properties):
            super().__init__(pure_properties) # pass it to the parent constuctor

            self.set_mass_density_from_ideal_gas_law() # Density from ideal gas law␣
↪also works for mixtures
            self.dynamic_viscosity=self.pure_properties["air"].dynamic_viscosity #␣
↪Just take the dynamic viscosity from the air

            # In a binary mixture, it is sufficient to specify a single diffusion␣
↪coefficent
            # This may of course also be a function fo the composition, temperature␣
↪and pressure
            self.set_diffusion_coefficient(2.42e-5*meter**2/second)
```

The registration to the library is done - as for pure substances - with the decorator `@MaterialProperties.register()`. Gas mixtures must inherit from the base class `MixtureGasProperties` and it is important that the constructor accepts a single argument, namely the a `dict` which containts the instances of the pure properties (i.e. instances of sub-classes of the `PureGasProperties` class), index by their name. This `dict` of pure properties must be passed to the `super` constructor. Then, it is important to specify the class property `components`, which is a `set` of the component names within this mixture. This `set` is used to find the correct mixture property class, when e.g. mixing the pure substances `"air"` and `"water"` in a second.

We also should set a passive field. This does not really change a lot, so you can basically pick any of the elements of `components` here. It is just required to simplify the advection-diffusion equations for the mixture: When a mixture of n components is considered, only $n-1$ advection-diffusion equations for the mass fractions must be solved. The last one follows from 1 minus the sum of the other $n-1$ mass fractions. This last field, which is not explicitly solved, is identified by the `passive_field`.

> **Warning:** The choice of the `passive_field` has one important consequence: If you want to set an `InitialCondition` or a `DirichletBC`, you cannot set the mass fraction of the component specified by the `passive_field`. In our example here, you cannot explicitly set initial conditions or Dirichlet boundary conditions for the air, but you can set it for water vapor. You still can impose Neumann fluxes, i.e. in/outflux of the passive component, though. This is possible since the corresponding test functions are internally substituted accordingly.

The rest of the constructor follows the definition of the pure substances. You still can access the method `set_mass_density_from_ideal_gas_law()`, which now will evaluate the mass density according to the local composition. Since at room temperature, the vapor concentration is usually small, we just copy the dynamic viscosity from the pure substance air here. Finally, we have to set a diffusion coefficient. For a binary mixture, a single diffusion coefficient is sufficient, which is set by `set_diffusion_coefficient()` with the single diffusion coefficient as argument. Also this coefficient can be a function of the composition, temperature and absolute pressure.

Let us now see how to create a specific mixture:

```
# Get the pure properties
air=get_pure_gas("air")
water_vapor=get_pure_gas("water")

# Mix in terms of mass fraction. One quantifier (here 0.98 for air) can be omitted
mix_gas=Mixture(air+0.02*water_vapor)

# We can access the initial condition, which will result in {'massfrac_air': 0.98,
↪'massfrac_water': 0.02, 'temperature': None}
print(mix_gas.initial_condition)

# To evaluate e.g. the mass density at the initial condition, we can just pass the␣
↪initial condition, but we also have to add information on the pressure and␣
↪temperature to get a single value
print(mix_gas.evaluate_at_condition("mass_density",mix_gas.initial_condition,
↪temperature=20*celsius,absolute_pressure=1*atm))
```

To mix in terms of mass fractions, we can just add multiple pure substances and wrap it into a `Mixture()` call. We have to specify also the initial mass fractions, e.g. here 2 % air in terms of mass fraction. Since the corresponding mass fraction of air, 98 %, follows from the requirement that all mass fractions have to sum to unity, the quantification of one pure substance can be omitted.

The initial condition can be accessed by the `initial_condition` property, which is `dict` containing the mass fractions. Again, we can evaluate properties by calling `evaluate_at_condition()`, but in order to evaluate at the initial condition, we have to pass `initial_condition` as first argument. Furthermore, since the ideal gas law also requires a temperature and a pressure, we have to pass these as keyword arguments to obtain a single dimensional value for the mass density at the end.

Let us now move on to a ternary gas mixture of water, ethanol and air. Again, first pure ethanol as gaseous component is required, followed by a definition of the mixture properties:

```
# Create the pure gaseous ethanol (analogous to water)
@MaterialProperties.register()
class PureGasEthanol(PureGasProperties):
    name="ethanol"
    def __init__(self):
        super().__init__()
        self.molar_mass = 0.4607E-01*kilogram/mol
        # Again we skip any further definitions



# Defining ternary mixture properties is similar to binary mixtures:
@MaterialProperties.register()
class MixtureGasWaterAirEthanol(MixtureGasProperties):
    components={"ethanol","water","air"} # Now three components
    passive_field="air"  # we choose again air as passive field

    def __init__(self,pure_properties):
        super().__init__(pure_properties) # Pure properties now has three entries

        self.set_mass_density_from_ideal_gas_law() # Again assuming ideal gas law

        # However, we now want to (artificially) increase the viscosity slightly␣
↪with the mass fraction of ethanol:
        mu_air=self.pure_properties["air"].dynamic_viscosity # Get the viscosity␣
↪of pure air
```

(continues on next page)

(continued from previous page)

```
            massfrac_ethanol=var("massfrac_ethanol")  # Get the variable ethanol␣
↪mass fraction
            self.dynamic_viscosity=mu_air*(1+0.2*massfrac_ethanol) # With increasing␣
↪ethanol, the gas gets more viscous

            # We now have three components, so effectively have a 2x2 diffusion␣
↪matrix. We only assume diagonal terms:
            self.set_diffusion_coefficient("water",2.42e-5*meter**2/second)
            self.set_diffusion_coefficient("ethanol",1.35e-5* meter**2/second)
```

As apparent, things work exactly the same as for binary mixtures and also higher order mixtures are defined the same way. What we have done additionally here, is explicitly defining the dynamic viscosity to be a function of the ethanol mass fraction. The used expression was chosen arbitrarily and not supported by any experimental data. The mass fraction of ethanol can be obtained by `var("massfrac_ethanol")`, likewise `var("massfrac_water")` and `var("massfrac_air")` can be used. The latter, since air is the passive field, is implicitly replaced by `1-var("massfrac_ethanol")-var("massfrac_water")` when the C code is generated. Additionally, e.g. `var("molefrac_ethanol")` can be used for the molar fractions, but where possible, mass fractions are the better choice when e.g. fitting experimental data of some property. This is due to the fact that the composition is solved in terms of mass fractions and molar fractions must be calculated first from the former.

In a ternary or higher order mixture, also the diffusion matrix becomes more complicated. The general diffusive flux of component α reads

$$\mathbf{J}_\alpha = -\rho \sum_{\beta=1}^{n} D_{\alpha\beta} \nabla w_\beta ,$$

where w_β are the mass fraction fields and $D_{\alpha\beta}$ are the entries of the diffusion matrix. In the binary mixture, we used `set_diffusion_coefficient()` with only a single argument, namely a diffusion coefficient. Calling that method in this way will just set the entire diagonal, i.e. *all* entries D_{ii}, to the supported argument. If `set_diffusion_coefficient()` is called with two parameters, we only set the diagonal diffusion coefficient D_{ii} for a single i, namely the index i which corresponds to the name of the component supported by first argument. All unset diffusion coefficients defaults to zero. This means, in the above example, the diffusion fluxes are set to

$$\mathbf{J}_\text{w} = -\rho D_\text{ww} \nabla w_\text{w}, \qquad \mathbf{J}_\text{e} = -\rho D_\text{ee} \nabla w_\text{e}$$

for water (w) and ethanol (e), respectively. The diffusion flux of air has not been defined propertly, but since it is the passive component, it is not required.

For ternary and higher mixtures, one also might have to set off-diagonal coefficients, which can be done by e.g. calling `set_diffusion_coefficient("water","ethanol",...)` to set the coefficient D_we. Note that off-diagonal diffusion coefficients should not be a constant, but depend on the composition. These off-diagonal entries also can be negative.

Of course, also the thermal properties `thermal_conductivity` and `specific_heat_capacity` must be set in the gas mixture definition class, when thermal dynamics are desired.

8.2.4 Pure liquids

The definition of pure liquids and liquids mixtures are very similar to the definition of gases, except that pure liquids must inherit from the `PureLiquidProperties` and mixtures from the `MixtureLiquidProperties` base classes. To define e.g. the pure liquid water, we proceed as follows:

```
from pyoomph.materials import *

# Define pure water
@MaterialProperties.register()
class PureLiquidWater(PureLiquidProperties):
    name="water"
    def __init__(self):
        super().__init__()
        self.molar_mass=18.01528*gram/mol # Molar mass

        # Density and viscosity (assuming constants here)
        self.mass_density=998*kilogram/meter**3
        self.dynamic_viscosity=1*milli* pascal * second

        # Thermal properties (assuming constants here)
        self.specific_heat_capacity=4.187* kilo * joule / (kilogram * kelvin)
        self.thermal_conductivity=0.597* watt / (meter * kelvin)
        self.latent_heat_of_evaporation=2437.69081321*kilo*joule/kilogram #␣
↪Liquids also have a latent heat of evaporation

        # Default surface tension against air as function of the temperature
        TKelvin=var("temperature")/kelvin
        self.default_surface_tension["gas"]=0.07275*(1.0-0.002*(TKelvin-291.0))␣
↪* newton/meter

        # Vapor pressure can be set by Antoine coefficients (in mmHg, C␣
↪convention)
        # You can also add e.g. bar and kelvin as fourth and fifth argument to␣
↪use the [bar,K] convention
        self.set_vapor_pressure_by_Antoine_coeffs(8.07131,1730.63 ,233.426)
        #Alternatively, you can set the vapor pressure by hand by setting self.
↪vapor_pressure= ...

        #For UNIFAC calculations of activity coefficients in mixtures, we also␣
↪need the UNIFAC groups
        self.set_unifac_groups({"H2O":1}) #Just one H2O group here
```

The `dynamic_viscosity` and `mass_density` properties are the same as in case of gases. Also the thermal properties `specific_heat_capacity` and `thermal_conductivity` must be set when thermal effects should be considered in the simulation.

Pure liquids also have some additional properties, which can be set. First of all, there is the thermal property `latent_heat_of_evaporation` (measured per mass, not per mole), which must be set when evaporative cooling should be considered. Then, pure liquids have a `vapor_pressure()` property, which can either be set by hand or by the method `set_vapor_pressure_by_Antoine_coeffs()`. The coefficients A, B and C of the Antoine equation are often given in the $(\mathrm{mmHg}, {}^\circ\mathrm{C})$ convention in literature, but you can also change the convention by supplying e.g. `bar, kelvin` as additional arguments, if the Antoine parameters A, B and C in the literature are given in that convention. The definition of the vapor pressure is important if mass transfer should be considered, e.g. evaporation. Pure liquids without a vapor pressure will be non-volatile when the default mass transfer model is used.

Also, we can specify a default surface tension of the liquid against the gas phase. Usually, the particular composition of the gas phase does not alter the surface tension strongly, whereas the liquid composition does. Thus, irrespectively of

the composition of the gas phase, we can set a typical surface tension which is always used, when a liquid-gas interface is considered with this particular liquid, unless it is explicitly override by a definition of particular liquid-gas interface properties (cf. Section 8.4.)

Finally, if you want to use UNIFAC models to calculate the activity coefficients of mixtures, you must specify the particular UNIFAC subgroups of each pure component. This can be done with the `set_unifac_groups()` method. More details on this are provided later in Section 8.8.

Loading a pure liquid from the material library works exactly as loading a pure gas, but with the routine `get_pure_liquid()` instead of `get_pure_gas()`.

8.2.5 Liquid mixtures

Properties of liquid mixtures are defined similarly to gas mixtures. Again, we define the required components for this particular mixture and can select one species as passive field, i.e. the composition field which is not explicitly solved for. When we define a pure liquid name `"glycerol"` analogous to the pure liquid `"water"`, we can define the mixture properties e.g. as follows:

```
@MaterialProperties.register()
class MixtureLiquidGlycerolWater(MixtureLiquidProperties):
    components={"water","glycerol"}
    passive_field="water"
    def __init__(self,pure_properties):
        super().__init__(pure_properties)
        self.set_by_weighted_average("mass_density") # realistic assumption␣
↪here: rho=rho_water*w_water+rho_glyc*w_glyc
        self.set_by_weighted_average("thermal_conductivity")
        self.set_by_weighted_average("specific_heat_capacity")

        yG=self.get_mass_fraction_field("glycerol") # will just expand to var(
↪"massfrac_glycerol")

        # Model for the dynamic viscosity
        TCelsius = subexpression(var("temperature") / kelvin-273.15)
        a=0.705 - 0.0017 * TCelsius
        b = (4.9 + 0.036 * TCelsius) * a ** 2.5
        muG=12100 * exp((-1233 + TCelsius) * TCelsius / (9900 + 70 * TCelsius))
        muW =1.790 * exp((-1230 - TCelsius) * TCelsius / (36100 + 360 *␣
↪TCelsius))
        alpha = subexpression(1 - yG + a * b * yG * (1 - yG) / (a * yG + b * (1 -
↪ yG)))
        self.dynamic_viscosity= subexpression(muW* (muG/muW) ** (1-alpha)* 0.
↪001*pascal * second)

        # Surface tension function
        self.default_surface_tension["gas"]=subexpression(72.45e-3 * ((1.0 - 0.
↪1214690683 * yG + 0.4874796412 * yG ** 2 - 2.208295376 * yG ** 3 + 3.412242927 * yG␣
↪** 4 - 1.698619738 * yG ** 5) - (0.0001455 * (1 - yG) + 0.00008845 * yG) *␣
↪(TCelsius))* newton / meter)

        # Diffusion coefficient fit
        D=1.024e-11 * (-0.721 * yG + 0.7368) / (0.49311e-2 * yG + 0.7368e-
↪2)*meter ** 2 / second
        self.set_diffusion_coefficient(D)

        # Set activity coefficients by AIOMFAC
        self.set_activity_coefficients_by_unifac("AIOMFAC")
```

Again, as in the case of gas mixtures, the components and passive_field must be set. The constructor takes again a dict of the pure properties.

If one does not know details on the particular change of the liquid properties with the composition, one always can use set_by_weighted_average() to calculate the average of the pure properties weighted by the local mass fractions. This makes at least sure that the properties are correct when taking the pure limits. One can also modify the optional argument fraction_type to "mole_fraction" to blend between the pure properties weighted by the mole fractions instead the mass fractions. The local mass and mole fractions of each component can be obtained by get_mass_fraction_field() and get_mole_fraction_field(), respectively. Alternatively, one can directly use e.g. var("massfrac_water") or var("molefrac_glycerol") to bind these fields to form arbitrary expressions.

As shown in the above example for the dynamic_viscosity(), one can assemble functions of the composition and temperature easily. Here, we have used a viscosity model developed by Cheng [6], while the surface tension was obtained by a fit of experimental data [41]. The same holds true for the diffusion coefficient based on the data of D'Errico *et al.* [9].

We can set the activity coefficients either directly by setting the dict values of the member activity_coefficients, e.g. activity_coefficients["water"]=.... If the vapor pressure shall be calculated by Raoult's law (cf. (8.4) later on), one has to call set_vapor_pressure_by_raoults_law() afterwards. Alternatively, the vapor pressure of each component can be set directly by the dict vapor_pressure_for, e.g. vapor_pressure_for["water"]=.... One can also invoke the various UNIFAC models to calculate the activity coefficients and set the vapor pressure according to Raoult's law with these activity coefficients. To that end, a simple call of set_activity_coefficients_by_unifac() will do the trick. One has to select a particular UNIFAC model ("Original", "Dortmund" or "AIOMFAC"). Of course, to use these models, one has to set the group contributions in the pure liquids (cf. Section 8.8 later on).

8.2.6 Solids

Compared to gases and liquids, solids are trivial. At the moment, only pure solids are allowed and we only can set thermal properties. Since solids are not fluids, we cannot use them with the CompositionFlowEquations(), but we still can solve the temperature field. In the future, also elasticity and porosity might be added. Pure solids must inherit from the PureSolidProperties class and can be decorated with a @MaterialProperties.register() to register the solid to the library based on the name, i.e. analogous to pure gases or pure liquids. As properties, only the molar_mass, mass_density, thermal_conductivity and specific_heat_capacity can be set.

Mixtures of solids are not implemented yet, so for e.g. an alloy, one has to define it as a pure solid with the properties of the alloy.

8.3 Example: Rayleigh-Taylor instability

Similar to the problem discussed in Section 5.3.3, we now want to setup the same problem with the CompositionFlowEquations():

```
from pyoomph import *
from pyoomph.equations.multi_component import *
from pyoomph.materials import *
import pyoomph.materials.default_materials # Alternatively, define the materials by␣
↪hand

class RayleighTaylorProblem(Problem):
    def __init__(self):
        super(RayleighTaylorProblem, self).__init__()
```

(continues on next page)

(continued from previous page)

```
        self.box_height,self.box_width=1*milli*meter,0.25*milli*meter # box size
        self.Nx=5 # Num elements in x direction
        self.mixture=Mixture(get_pure_liquid("water")+0.5*get_pure_liquid("glycerol
↪")) # Default mixture
        self.temperature=20*celsius # Temperature : Required for some properties
        self.gravity=9.81*vector(0,-1)*meter/second**2
```

We import the material library and also the default materials. Alternatively, we could write our own material library file or define custom fluids and mixtures thereof in the very same python script of the code. Then, we initialize the property `Mixture()` by a default mixture, but in the run script, the user can change it easily.

For the `define_problem()` method, we first have to set the `spatial` and the `temporal` scale. All other required scales can be set to reasonable values by using the `set_reference_scaling_to_problem()` of the liquid:

```
def define_problem(self):
    # Mesh
    self.add_mesh(RectangularQuadMesh(size=[self.box_width,self.box_height],N=[self.
↪Nx,int(self.Nx*self.box_height/self.box_width)]))
    # Spatial and temporal scales must be set by hand
    self.set_scaling(spatial=self.box_width,temporal=1*second)
    # Set remaining scales by the liquid properties
    self.mixture.set_reference_scaling_to_problem(self,temperature=self.temperature)
    # define global constants "temperature" and "absolute_pressure". It might be␣
↪required by the fluid properties
    self.define_named_var(temperature=self.temperature,absolute_pressure=1*atm)
```

It is important to pass the `temperature` variable here since `set_reference_scaling_to_problem()` internally evaluates e.g. the `dynamic_viscosity` and `mass_density` to find a good scaling. Since these properties might depend on the temperature, we must supply some `temperature` for that. Furthermore, the initial composition is substituted in these expressions so that eventually only constant values appear for the reference density and viscosity used for the non-dimensionalization.

Furthermore, since the problem is isothermal and also pressure fluctuations are not allowed to have an effect on the fluid properties, we must tell pyoomph what to use for the fields `"temperature"` and `"absolute_pressure"`, when any fluid property depend on these. Therefore, we set these variables globally to constants by the `define_named_var()`. Whenever pyoomph finds any occurrence of e.g. `var("temperature")` and there is no such field defined in the current domain, these values will be used instead. Thereby, all temperature-dependence will just be evaluated at this constant value. Since this substitution and successive simplification of the properties happens before the C code generation, a potential temperature-dependence of the fluid properties does not slow down the simulation.

The code is now just a `CompositionFlowEquations()` of the `Mixture()` with the desired `gravity`:

```
eqs=MeshFileOutput()
eqs+=CompositionFlowEquations(self.mixture,spatial_errors=True,gravity=self.gravity)
for side in ["left","right","bottom"]:
    eqs+=DirichletBC(velocity_x=0,velocity_y=True)@side
    # Top side must be open: The density is not constant and hence we require in/
↪outflow somewhere!

self.add_equations(eqs@"domain")
```

For the boundary conditions, however, we have to be careful: Since the mass density is in general not constant the total volume will not be conserved. Hence, one side of the domain must allow for in-/outflow. Alternatively, we could also apply a no-slip condition at the `"top"` interface, but pass the argument `boussinesq=True` to the `CompositionFlowEquations()`. In that case, the Boussinesq approximation is applied, i.e. the continuity equation simplifies to $\nabla \cdot \vec{u} = 0$. Then, the volume remains conserved, but we have to fix again one degree of the pressure (cf. Section 4.4.4).

Finally, the user can change the `Mixture()` and prescribe any suitable `InitialCondition` for the liquid:

```
with RayleighTaylorProblem() as problem:
    # Let the user select any mixture
    problem.mixture = Mixture(get_pure_liquid("water") + 0.5 * get_pure_liquid(
↪"glycerol"))

    # And also formulate the initial condition
    x,y=var(["coordinate_x","coordinate_y"])
    xrel, yrel = var("coordinate_x") / problem.box_width, var("coordinate_y") /␣
↪problem.box_height - 0.5
    wg_init = 0.5*(1+tanh(100 * (yrel - 0.0125 * cos(2 * pi * xrel))))
    problem.additional_equations+=InitialCondition(massfrac_glycerol=wg_init)@"domain"

    problem.max_refinement_level=4
    problem.run(10*second,startstep=0.1*second,maxstep=0.5*second,outstep=True,
↪spatial_adapt=1,temporal_error=1)
```

The results are shown in Fig. 8.1. Due to the high viscosity and low diffusivity in the limit of pure glycerol, the flow and diffusion dynamics mostly happen in the lower half.

The benefit of using the material library is that it is now trivial to exchange it to e.g. an ethanol-water mixture or even ternary mixtures with just a single line of code now (provided the desired fluids and mixtures thereof are already in the material library).

8.4 Free surfaces and evaporation

The previous example had just a single liquid domain. When a liquid and a gas domain are in contact, a free surface emerges where evaporation can happen. Fluid-fluid interfaces in general (i.e. liquid-gas or liquid-liquid) are both covered by the `MultiComponentNavierStokesInterface` class in the multi-component flow framework of pyoomph. It takes the properties of the interface as required argument. Interface properties can be obtained from the material library by using the operator `|` on the bulk phase properties. When we hence have e.g. multi-component flow via the `CompositionFlowEquations()` in a `"liquid"` domain and either `CompositionFlowEquations()` (or `CompositionDiffusionEquations()` for pure diffusive transport) in a `"gas"` phase, the assembly of the equations could look like

```
temperature=20*celsius

# Bulk properites (pure water liquid and air with some vapor in the gas phase)
liquid=get_pure_liquid("water")
gas=Mixture(get_pure_gas("air")+20*percent*get_pure_gas("water"),quantity="relative_
↪humidity",temperature=temperature)

# Get the interface properties
interface_props=liquid | gas

# Assembly of an equation system
eqs=CompositionFlowEquations(liquid)@"liquid"
eqs+=CompositionDiffusionEquations(gas)@"gas" # (alternatively␣
↪CompositionFlowEquations)
# free surface should be added to the liquid side of the liquid-gas interface:
eqs+=MultiComponentNavierStokesInterface(interface_props)@"liquid/liquid_gas"
```

The properties of liquid-gas interfaces can be defined by hand, but it is not necessary: If there are no particular properties of an interface between a liquid and a gas phase are present in the material library, default properties will be constructed.

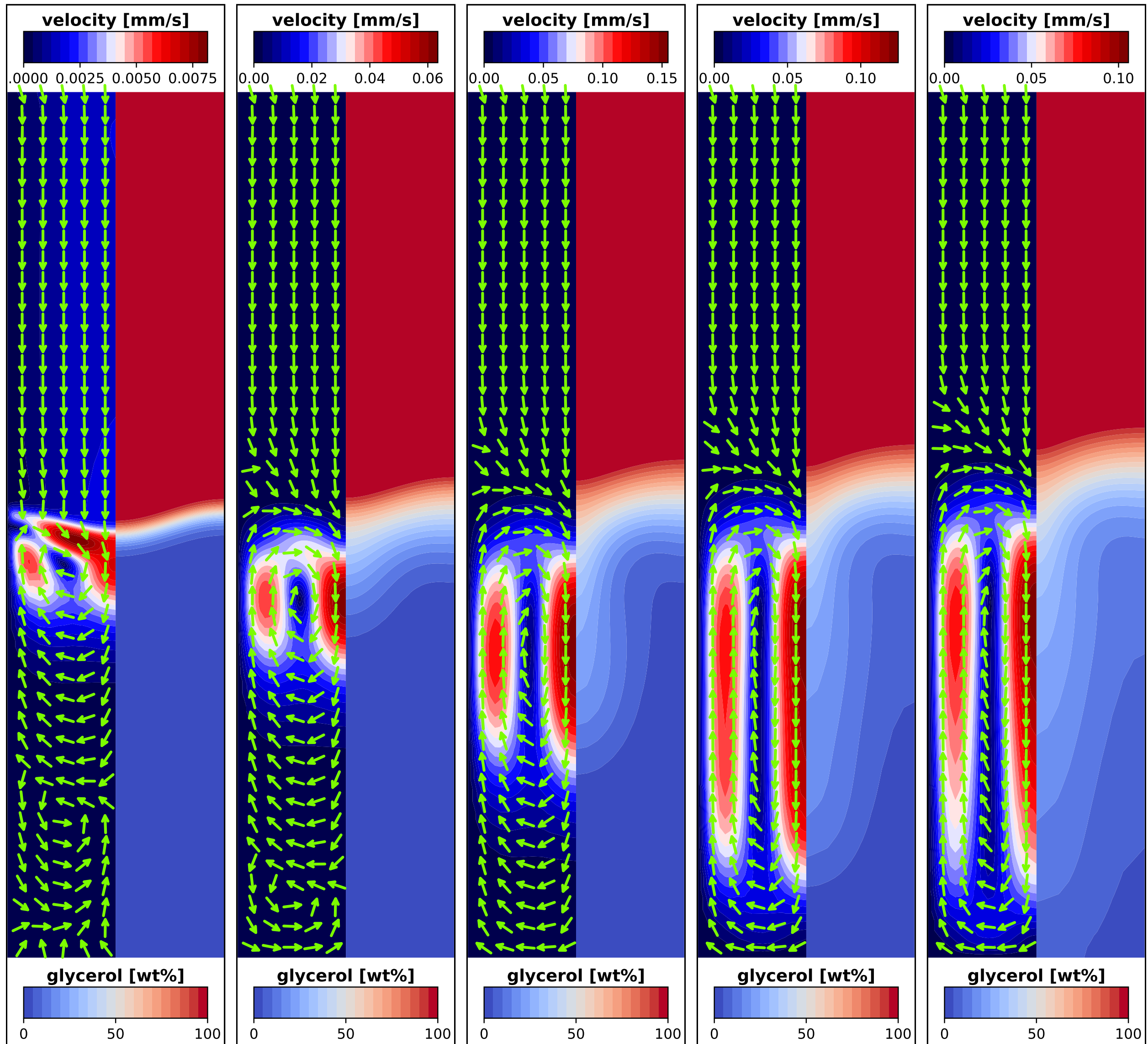

Fig. 8.1: Rayleigh-Taylor instability in the glycerol-water system.

To that end the `default_surface_tension` dict entry `["gas"]` of the liquid bulk properties will be used as `surface_tension`:

```
# Access the surface tension
print("Surface tension function:",interface_props.surface_tension) # same as liquid.
↪default_surface_tension["gas"]
sigma_at_T=liquid.evaluate_at_condition(interface_props.surface_tension,liquid.
↪initial_condition,temperature=temperature)
print("Surface tension at temperature T",sigma_at_T)
```

If one, for whatever reason, want to modify properties of a particular liquid-gas interface in the material library, one can register a particular class for that to the library by inheriting from the `LiquidGasInterfaceProperties` class:

```
# Register a particular liquid-gas interface to the library
@MaterialProperties.register()
class CustomInterfacePropertiesWaterVsVaporAir(LiquidGasInterfaceProperties):
    liquid_components = {"water"} # Components in the liquid phase
    gas_components = {"water","air"} # components in the gas phase
    def __init__(self,phaseA,phaseB,surfactants):
        super(CustomInterfacePropertiesWaterVsVaporAir, self).__init__(phaseA,phaseB,
↪surfactants)
        self.surface_tension=50*milli*newton/meter # set a custom surface tension

# Get the interface properties
new_interface_props=liquid | gas
print("New surface tension",new_interface_props.surface_tension)
```

When one tries to determine the interface properties via the `|` operator, these properties will be used when one has a liquid phase consisting of `"water"` and a gas phase with the components `"water"` and `"air"`. This is identified by the sets `liquid_components` and `gas_components` of the class.

However, if one only wants to change e.g. the surface tension, it is easier to just set it directly after getting the default interface properties. I.e. without defining and registering the particular class, but just overriding the property directly:

```
# Override interface properties by hand
interface_props.surface_tension=50*milli*newton/meter
```

8.4.1 What the MultiComponentNavierStokesInterface does

Once a `MultiComponentNavierStokesInterface` with the passed interface properties is added to the problem's equations, it will add all relevant contributions of a free surface to the system. If the gas phase has just diffusion, i.e. no flow, by a `CompositionDiffusionEquations()` class, it will add both the kinematic and the dynamic boundary condition to the system

$$\rho^{\mathrm{l}}\left(\vec{u}^{\mathrm{l}}-\vec{u}_{\mathrm{I}}\right)\cdot\vec{n}=\sum_{\alpha}j_{\alpha}$$

$$\vec{n}\cdot\left[-p^{\mathrm{l}}\mathbf{1}+\mu^{\mathrm{l}}\left(\nabla\vec{u}^{\mathrm{l}}+(\nabla\vec{u}^{\mathrm{l}})^{\mathrm{t}}\right)\right]=\sigma\kappa\vec{n}+\nabla_{S}\sigma$$

Here, the superscript l refers to the liquid phase. For the details of the weak form implementation, please refer to Section 6.5. j_{α} are potential mass transfer rates, which will be calculated by a mass transfer model (cf. next section). In absence of any mass transfer ($j_{\alpha}=0$ for all components α), it is again the normal kinematic and dynamic boundary condition, where the gas phase is not considered due to the disregarded flow.

If gas flow is considered, i.e. when the gas phase has `CompositionFlowEquations()` instead of `CompositionDiffusionEquations()`, also the gas velocity and pressure will also couple into the system and the `Multi-`

`ComponentNavierStokesInterface` solves the following boundary conditions

$$\rho^{\mathrm{l}}\left(\vec{u}^{\mathrm{l}}-\vec{u}_{\mathrm{I}}\right)\cdot\vec{n}=\sum_{\alpha}j_{\alpha}$$
$$\rho^{\mathrm{g}}\left(\vec{u}^{\mathrm{g}}-\vec{u}_{\mathrm{I}}\right)\cdot\vec{n}=\sum_{\alpha}j_{\alpha}$$
$$\left(\vec{u}^{\mathrm{l}}-\vec{u}^{\mathrm{g}}\right)\cdot\vec{t}=0$$
$$\vec{n}\cdot\left[-(p^{\mathrm{l}}-p^{\mathrm{g}})\mathbf{1}+\mu^{\mathrm{l}}\left(\nabla\vec{u}^{\mathrm{l}}+(\nabla\vec{u}^{\mathrm{l}})^{\mathrm{t}}\right)-\mu^{\mathrm{g}}\left(\nabla\vec{u}^{\mathrm{g}}+(\nabla\vec{u}^{\mathrm{g}})^{\mathrm{t}}\right)\right]=\sigma\kappa\vec{n}+\nabla_{S}\sigma$$

The tangential velocity is hence continuous whereas the normal velocity will have a jump (Stefan flow) when evaporation is considered and the mass densities differs. Also the stress and the pressure are now coupled. The *vapor recoil pressure* is not considered in these terms, since it is usually a tiny contribution. Such additional contributions must be added manually.

On a moving mesh, the kinematic boundary conditions are enforced by moving the mesh accordingly. If the mesh is static, i.e. no equations for the mesh motion are added, the kinematic boundary condition is enforced on the velocity instead (with interface velocity $\vec{u}_{\mathrm{I}}=0$). Optionally, one can also use a static interface on a moving mesh with the keyword argument `static=True`.

Finally, if the mass transfer rates j_α are non-zero, the fluid composition on both sides changes as well. Therefore, let the species velocity of component α in phase $\phi=$ l,g be given by $\vec{u}_{\alpha}^{\phi}$. The species velocity is not the same as the mass-averaged velocity $\vec{u}^{\phi}$ since in a mixture, different species may move into different directions (e.g. in case of diffusion). The mass-averaged velocity is connected to the species velocities via

$$\vec{u}^{\phi}=\sum_{\alpha}w_{\alpha}\vec{u}_{\alpha}^{\phi}\,.$$

A transfer rate of j_α (mass per interface area and time) of component α crossing the interface means that indeed this particle flux is crossing the interface, i.e.

$$j_{\alpha}=\rho^{\phi}w_{\alpha}^{\phi}\left(\vec{u}_{\alpha}^{\phi}-\vec{u}_{\mathrm{I}}\right)\cdot\vec{n}$$

Summing over all α gives the kinematic boundary conditions due to $\sum_{\alpha}w_{\alpha}=1$ and the definition of the mass-averaged velocity. Since the mass has to be conserved when crossing the interface, both sides are the same, i.e. $j_{\alpha}=j_{\alpha}^{\mathrm{l}}=j_{\alpha}^{\mathrm{g}}$.

As mentioned, diffusion can lead to a relative motion of species with respect to the mass-averaged velocity. In fact, the diffusive flux mass flux $\vec{J}_{\alpha}$ is connected to the relative motion of species α with respect to the averaged velocity by

$$\vec{J}_{\alpha}^{\phi}=\rho^{\phi}w_{\alpha}^{\phi}\left(\vec{u}_{\alpha}^{\phi}-\vec{u}^{\phi}\right)\,.$$

Thereby, the species velocity $\vec{u}_{\alpha}^{\phi}$ can be replaced by the diffusive fluxes in the mass transfer rate

$$j_{\alpha}=\rho^{\phi}w_{\alpha}^{\phi}\left(\vec{u}^{\phi}-\vec{u}_{\mathrm{I}}\right)\cdot\vec{n}+\vec{J}_{\alpha}^{\phi}\cdot\vec{n}=w_{\alpha}\sum_{\beta}j_{\beta}+\vec{J}_{\alpha}^{\phi}\cdot\vec{n}\,.$$

Thereby, the normal diffusive flux at the interface, which is exactly the Neumann term which can be imposed for the mass fraction advection diffusion equations (8.1), can be written as

$$\vec{J}_{\alpha}^{\phi}\cdot\vec{n}=j_{\alpha}-w_{\alpha}^{\phi}\sum_{\beta}j_{\beta}\,.$$

If the phase ϕ consider the flow, i.e. `CompositionFlowEquations()` are used, these Neumann terms are applied to the equations (8.1) for $\alpha=1,\dots,n-1$, where the sum over β includes also the nth term to account for transfer of the selected `passive_field`, which is not explicitly solved for.

If flow is disregarded in phase ϕ, i.e. using the `CompositionDiffusionEquations()`, the advection term is also disregarded since it stems from the relative velocity of the fluid to the interface. For domains with `CompositionDiffusionEquations()`, the mass transfer rates are imposed directly as diffusive fluxes:

$$\vec{J}_{\alpha}^{\phi}\cdot\vec{n}=j_{\alpha}\,. \tag{8.3}$$

8.4.2 Evaporation model

Still missing so far are the actual mass transfer rates j_α. We have elaborated how to conserve mass when it crosses the interface, i.e. what Neumann conditions for the advection-diffusion equations must be imposed and also considered for the mass transfer in the kinematic boundary condition. However, up till now, j_α can still be arbitrary.

A common way for evaporating droplets is to impose the saturated vapor at the gas side of the interface and use (8.3), i.e. just evaluating the normal diffusive flux of the vapor at the interface in the gas phase. This works fine if the gas phase has air as host medium. A pure water droplet can hence easily evaporate into the air. However, if both the liquid and the gas phase consist only of water - i.e. no air is present has host component - one cannot impose any saturated vapor. In fact, there is no mass fraction field to be solved and thus there are no diffusive fluxes. Still, mass transfer happens if the temperature at the interface deviates from the boiling point.

To overcome this issue and provide a most general evaporation model, the default evaporation model of pyoomph does not impose any saturated vapor directly. Instead, the ideal gas law and Dalton's law is used to state that mass transfer rate j_α is proportional to the difference of the equilibrium vapor mole fraction, which is $p_\alpha^\text{sat}/p_\text{amb}$, and the actual vapor mole fraction x_α^g in the gas phase:

$$j_\alpha = j_0 \cdot \left(\frac{p_\alpha^\text{sat}}{p_\text{abs}} - x_\alpha^\text{g}\right) .$$

Here, j_0 is an intense rate factor, by default $j_0 = 100\,\mathrm{kg}/(\mathrm{m}^2\cdot\mathrm{s})$, so that it quickly attains equilibrium. However, with this model also the mass transfer between pure water in both liquid and gas phase behaves correctly: the saturation pressure p_α^sat depends on the temperature T and at the boiling point, $p_\alpha^\text{sat}(T_\text{boil}) = p_\text{abs}$ holds. Since the mole fraction $x_\alpha = 1$ in a single component system, it will indeed lead to the correct dynamics. In this setting, of course, the latent heat of evaporation must be considered. Thereby, the process will be limited by the thermal transport to the interface, provided j_0 is chosen sufficiently high.

The mole fractions x_α^ϕ can be accessed via e.g. `var("molefrac_water")` and these fields are calculated based on the `molar_mass` values of the pure components in the mixture from the mass fractions w_α^ϕ internally. The saturation pressure p_α^sat can be accessed by e.g. `get_vapor_pressure_for("water")` of the liquid phase properties. For pure liquids, it is set directly with the property `vapor_pressure` of by Antoine coefficients via `set_vapor_pressure_by_Antoine_coeffs()` (cf. Section 8.2.4). In case of a liquid mixture, Raoult's law is used:

$$p_\alpha^\text{sat} = x_\alpha^\text{l} \gamma_\alpha p_\alpha^\text{sat,pure} \tag{8.4}$$

Here, x_α^l is the mole fraction of the component α in the liquid phase and $p_\alpha^\text{sat,pure}$ is the saturation pressure of the pure substance α. The activity coefficients γ_α can be either set directly or are calculated by UNIFAC. If they are not set, they default to unity, i.e. the ideal Raoult's law. The activity coefficients will be discussed later in Section 8.8.

Warning: Only the components that are present in both phases are allowed to evaporate! If you have e.g. a liquid phase of pure water and a gas phase of pure air, no evaporation will happen. You must add gaseous water to the gas phase. If there is initially no water vapor in the gas phase, you still must provide it in order to allow water to evaporation, e.g. by setting the gas mixture as

```
gas_props=Mixture(get_pure_gas("air")+0*get_pure_gas("water"))
```

Finally, note that the evaporation model can be customized by defining custom interface properties. One can set the mass transfer model by `set_mass_transfer_model` of the `LiquidGasInterfaceProperties` to a custom evaporation model (cf. `pyoomph.materials.mass_transfer`). If one just want to modify j_0, one can do so by setting `default_mass_flux_coefficient` of the default mass transfer model `DifferenceDrivenMassTransferModelLiquidGas`, which is accessed by `get_mass_transfer_model()` of the interface properties.

8.5 Example: Marangoni instability in a Hele-Shaw cell

We want to investigate the Marangoni instability (cf. Section 5.3.4) of an evaporating ethanol-water mixture which is confined between two plates at the top and bottom, i.e. by a *Hele-Shaw cell*. The flow in such a cell is usually three-dimensional, but when the plate distance δ is small compared to the flow structures, one can assume that the flow in z-direction (i.e. the direction between both plates) is parabolic due to the no-slip boundary conditions at the top and bottom plate, i.e. at $z = 0$ and $z = \delta$. The velocity $\vec{u}_{3\mathrm{d}}(x, y, z, t)$ is then just given by the average flow $\vec{u}(x, y, t)$ by $\vec{u}_{3\mathrm{d}} = 6z(\delta - z)\vec{u}/\delta^2$. The presence of the no-slip boundary conditions modify the Navier-Stokes equations for the projected two-dimensional by a factor $6/5$ for the nonlinear term and an additional Brinkman term of $-12\mu\vec{u}/\delta^2$:

$$\rho\left(\partial_t\vec{u} + \frac{6}{5}\nabla\vec{u}\cdot\vec{u}\right) = -\nabla p + \nabla\cdot\left[\mu\left(\nabla\vec{u} + \nabla\vec{u}^t\right)\right] - 12\frac{\mu}{\delta^2}\vec{u} \tag{8.5}$$

In pyoomph, we can just pass the plate distance δ via the `hele_shaw_thickness` argument to the `CompositionFlowEquations()` to automatically account for these modifications of the Navier-Stokes equations.

Experimentally, numerically and analytically, this setting has been investigated in Ref. [10]. Here, we will use the multi-component equations of pyoomph to reprocude the problem by simulation:

```
from pyoomph import *
from pyoomph.equations.multi_component import *
from pyoomph.expressions.utils import * # for the random perturbation
from pyoomph.materials import *
import pyoomph.materials.default_materials # Alternatively, define the materials by␣
↪hand


class MarangoniHeleShawProblem(Problem):
    def __init__(self):
        super(MarangoniHeleShawProblem, self).__init__()
        # domain size: Gas size is the same as domain_length
        self.domain_length,self.domain_width=0.5*milli*meter,0.5*milli*meter
        self.cell_thickness=20*micro*meter # Hele-Shaw plate distance
        self.Nx=18 # Num elements in x direction
        self.max_refinement_level=3 # max refinement level to refine near the␣
↪interface
        self.temperature=20*celsius # Temperature : Required for some properties
        self.liquid_mixture = Mixture(get_pure_liquid("water") + 0.5 * get_pure_
↪liquid("ethanol"))  # Default liquid mixture
        # The gas mixture must be adjusted: In the experiment, the evaporation␣
↪happens in 3d space
        # Here, it is just two-dimensional so the Green's function of the Poisson␣
↪equation for diffusion is not bounded!
        # Therefore, we pin the vapor concentration at the far right to this␣
↪composition
        self.gas_mixture = Mixture(get_pure_gas("air") + 20*percent * get_pure_gas(
↪"ethanol") + 40*percent * get_pure_gas("water"),quantity="relative_humidity",
↪temperature=self.temperature)
        self.interface_props=None # Interface properties, are determined␣
↪automatically if not set
```

In the experiments, the evaporation happens in open space. Here, we only have a two-dimensional setting. While in 3d the ambient conditions could be imposed at infinite distances due to the far field decay of $1/r$ with the distance r of the vapor concentration from the interface, in 2d it does not work: The corresponding Green's function is $\ln(r)$ and hence it is not bounded at infinity. Therefore, we must strongly impose the vapor concentration at a finite distance. This artificial vapor concentration can be set with by the composition of the `gas_mixture` and must be adjusted so that the typical evaporation rates match with the experiment.

Again, in the `define_problem()`, we setup the `spatial` and `temporal` scale by hand and let the remaining scales be determined automatically from the properties of the `liquid_mixture`. However, we adjust the `velocity` and `pressure` scale by hand afterwards to better match the typical orders of magnitude for this particular problem (e.g. by measurements in the experiments). Since properties may depend on the `temperature` and potentially the `absolute_pressure`, we must again set these on a global (i.e. `Problem`-wide) level with the `define_named_var()`:

```
def define_problem(self):
    # Spatial and temporal scales must be set by hand
    self.set_scaling(spatial=self.domain_length, temporal=1 * second)
    # Set remaining scales by the liquid properties
    self.liquid_mixture.set_reference_scaling_to_problem(self, temperature=self.
↪temperature)
    # Adjust pressure and velocity a bit to the problem
    self.set_scaling(pressure=10 * pascal, velocity=1e-4 * meter / second)
    # define global constants "temperature" and "absolute_pressure". It might be␣
↪required by the fluid properties
    self.define_named_var(temperature=self.temperature, absolute_pressure=1 * atm)
```

The mesh is just a `RectangularQuadMesh`, but it has to be separated into two domains. This is possible if we pass a function to the argument `name`. Pyoomph will evaluate this function in the center of each element (in non-dimensional coordinates, i.e. measured in the `spatial` scale) and add these elements to the domain by this name. Here, we mark all elements that are on the left half as `"liquid"`, whereas the elements on the right half are in the `"gas"` domain. If an internal facet is between two elements of different domains, it will be automatically added to the interface named by the two domains (in alphabetical order) separated by an underscore, i.e. here the liquid-gas interface will be automatically named `"gas_liquid"`:

```
# Mesh: All elements with center further away than 1*domain_length (measured in␣
↪spatial scale) will be gas, otherwise liquid
domain_func=lambda x,y: "gas" if x>1 else "liquid"
mesh=RectangularQuadMesh(size=[2*self.domain_length,self.domain_width],N=[2*self.Nx,
↪int(self.Nx*self.domain_width/self.domain_length)],name=domain_func)
self.add_mesh(mesh)
```

Then, the equations have to be assembled. If the user does not explicitly selects the `interface_props` by hand, it will be determined from the material library:

```
# We can either set the interface properties by hand, e.g. to modify the surface␣
↪tension
# if not, we must find it from the material library
if self.interface_props is None:
    # To get the interface properties, we can just use the | operator
    self.interface_props=self.liquid_mixture | self.gas_mixture
    # When a particular liquid-gas interface is not defined, it will use a default␣
↪interface
    # This one will use a reasonable mass transfer model and the default_surface_
↪tension["gas"] of the liquid properties

liq_eqs=MeshFileOutput()
# Flow with Hele-Shaw confinement and use second order for the composition
liq_eqs+=CompositionFlowEquations(self.liquid_mixture,hele_shaw_thickness=self.cell_
↪thickness,compo_space="C2",spatial_errors=True)
liq_eqs+=DirichletBC(velocity_y=0)@"bottom"
liq_eqs += DirichletBC(velocity_y=0) @ "top"
liq_eqs+=MultiComponentNavierStokesInterface(self.interface_props)@"gas_liquid"
liq_eqs+=RefineToLevel()@"gas_liquid" # And refine it to max_refinement_level
```

The liquid equations mainly consist of the `CompositionFlowEquations()` with the `liquid_mixture` properties and the given `hele_shaw_thickness` along with a few boundary conditions and a static `MultiCompo-`

nentNavierStokesInterface with the interface_props. As discussed in the section before, the latter will automatically impose a free surface (static here, since no equations for mesh motion are added) with the surface_tension property of the interface_props. Also the evaporation model is considered and it will couple automatically to the "gas" domain. Note that we switch the space of the advection-diffusion equations for the required mass fraction fields to "C2", i.e. second order fields and also add spatial_errors for the spatial adaptivity. The free interface is always refined to the maximum level by the RefineToLevel object.

The gas equations are now just CompositionDiffusionEquations() with a prescribed far field DirichletBC based on the initial composition of the gas_mixture:

```
# Gas
gas_eqs=MeshFileOutput()
gas_eqs+=CompositionDiffusionEquations(self.gas_mixture) # just diffusion
# And fix the far boundary to the initial condition by iterating over all advection␣
↪diffusion fields for the mass fractions
gas_eqs+=DirichletBC(**{"massfrac_"+c:True for c in self.gas_mixture.required_adv_
↪diff_fields})@"right"

self.add_equations(liq_eqs@"liquid"+gas_eqs@"gas")
```

To run the simulation, we first slightly perturb the initial condition directly at the interface with random numbers. Thereby, the instability kicks in earlier, whereas otherwise, due to perfect symmetry of the mesh, it would start rather late just by the accumulation of tiny numerical errors of the Newton solver:

```
if __name__=="__main__":
    with MarangoniHeleShawProblem() as problem:
        # Slightly perturb the interface
        # 10 random numbers with a small amplitude linearily interpolated on the␣
↪interval 0:1
        randpert=DeterministicRandomField(min_x=[0],max_x=[1],amplitude=0.002,
↪Nresolution=10)
        yn=var("coordinate_y")/problem.domain_width # normalized coordinate
        randpert=randpert(yn) # interpolated random fields
        # Perturb the interface composition slightly
        problem.additional_equations+=InitialCondition(massfrac_ethanol=problem.
↪liquid_mixture.initial_condition["massfrac_ethanol"]+randpert)@"liquid/gas_liquid"
        problem.run(10*second,startstep=0.01*second,maxstep=0.5*second,outstep=True,
↪temporal_error=1,spatial_adapt=1)
```

The results are depicted in Fig. 8.2 and indeed show the experimentally observed coarsening and merging arch-like patterns. For a smaller plate distance, the growing of the arches can be suppressed due to the stronger damping of the Brinkman term in (8.5), whereas without the hele_shaw_thickness argument (e.g. by setting problem.cell_thickness=None) for the CompositionFlowEquations() (i.e. just the normal 2d Navier-Stokes), a violent chaotic flow would emerge.

8.6 Contact line models

In the previous example, we had a static interface and free flow at the side walls "bottom"/"top" towards the liquid-gas interface. For e.g. an evaporating droplet, as already discussed in Section 7.6, the consideration of contact line dynamics is required. Contact lines can either be pinned or moving and of course this also can change during the drying of the droplet, causing a stick-slip behavior of the contact line. Moreover, when the surface tensions of the three interfaces in contact is changing at the contact line, usually also the equilibrium contact angle in case of a moving contact angle will change.

For multi-component flow, pyoomph has a single equation class, the DynamicContactLineEquations, which can be added to the contact line domain to handle all considerable cases. The particular dynamics is implemented in versatile

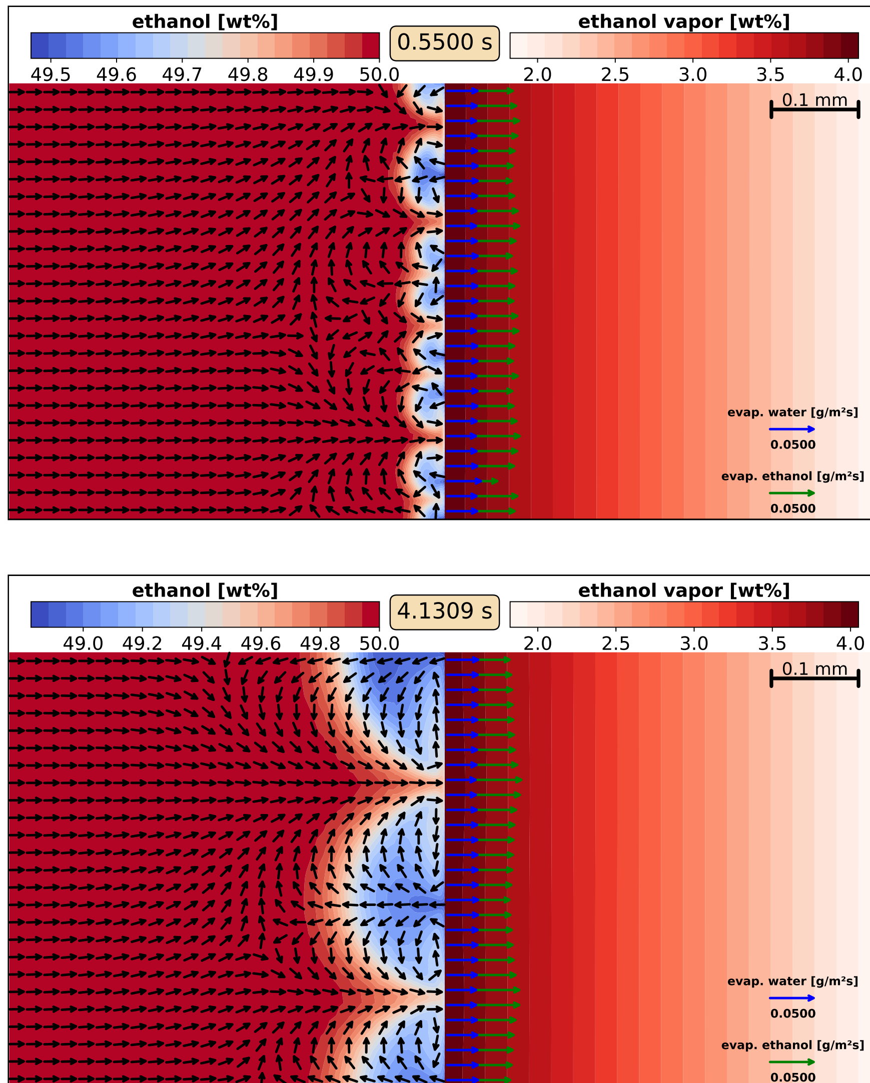

Fig. 8.2: Marangoni instability of an ethanol-water mixture evaporating in a Hele-Shaw cell.

contact angle models, which are passed to the `DynamicContactLineEquations`.

Both the versatile contact line models and the `DynamicContactLineEquations` are defined in the file `pyoomph.equations.contact_angle`.

8.6.1 PinnedContactLine

As we have seen in Section 7.6, a pinned contact line can be realized by enforcing `partial_t(var("mesh"))` $= 0$ on the test space $\vec{v}$ of the velocity. Hence, if an instance of the `PinnedContactLine` model is passed to the `DynamicContactLineEquations`, it will add the following weak contribution

$$\left[\partial_t \vec{X} \cdot \vec{t}_\text{w}, \mu\right] + \left[\lambda, \vec{v} \cdot \vec{t}_\text{w}\right]$$

with the position-enforcing Lagrange multiplier λ with test function μ. $\vec{X}$ is the position of the contact line and $\vec{t}_\text{w}$ is here the tangent of the wall (i.e. the substrate in the case of an evaporating droplet). The wall tangent can be set by the `wall_tangent` keyword argument of the `DynamicContactLineEquations` class. It should be tangential to the wall, having a magnitude of unity and pointing inward (i.e. towards the droplet domain in the evaporating droplet case). Since we must have the possibility to add a contribution to the velocity test space in direction of the wall tangent, it is necessary to allow for slip by using e.g. a slip length boundary condition on the substrate.

8.6.2 UnpinnedContactLine

When the contact angle freely moves, we have seen how to impose an equilibrium contact angle θ_eq in Section 6.6. More general, when the contact angle is measured with respect to the wall tangent $\vec{t}_\text{w}$, we add

$$\left[\sigma\left(\sin(\theta_\text{eq})\vec{n}_\text{w} + \cos(\theta_\text{eq})\vec{t}_\text{w}\right), \vec{v}\right] \,, \tag{8.6}$$

where $\vec{n}_\text{w}$ is the wall normal, pointing into the droplet domain. With these vectors, the contact angle θ approaches zero when the droplet is very flat, $\theta = 90\,°$ holds for a hemi-spherical droplet and for droplets on hydrophobic substrates, $\theta > 90\,°$ can be observed.

However, as also discussed in Section 6.6, the speed of the contact line motion by what θ approaches the equilibrium θ_eq is just controlled by the slip length. This is hard to control and to adjust with e.g. experimental observations. Instead, the `UnpinnedContactLine` let you control the motion velocity by a relation

$$U_\text{cl} = \partial_t \vec{X} \cdot \vec{t}_\text{w} = -U_\text{cl}^0 \left(\theta - \theta_\text{eq}\right) \tag{8.7}$$

with some typical velocity scale U_cl^0. In the spirit of the `PinnedContactLine`, which is just given by $U_\text{cl}^0 = 0$, the `UnpinnedContactLine` class therefore adds the weak contribution

$$\left[\sigma\left(\sin(\theta_\text{eq})\vec{n}_\text{w} + \cos(\theta_\text{eq})\vec{t}_\text{w}\right), \vec{v}\right] + \left[\partial_t \vec{X} \cdot \vec{t}_\text{w} - U_\text{cl}, \mu\right] + \left[\lambda, \vec{v} \cdot \vec{t}_\text{w}\right] \,, \tag{8.8}$$

The wall normal $\vec{n}_\text{w}$ can be passed by the `wall_normal` argument to the `DynamicContactLineEquations` class, whereas the speed scale U_cl^0 and the equilibrium contact angle can be passed via `cl_speed_scale` (defaults to 10^{-5} m/s) and `theta_eq` (defaults to the initial contact angle) of the `UnpinnedContactLine`. If you want to modify the relation (8.7), you can do so by inheriting a custom contact line model from the `UnpinnedContactLine` and override the method `get_unpinned_motion_velocity_expression()` according to your demands.

To adjust the contact angle at infinite speed, i.e. instantaneously to the equilibrium contact angle, you can set `cl_speed_scale=None`. Thereby, the droplet will always be at $\theta = \theta_\text{eq}$. This can be used e.g. to impose exactly some experimental dynamics, if you set `theta_eq` to a function of time that gives the experimental data.

8.6.3 StickSlipContactLine

We have seen similarities between the pinned and the unpinned contact line, where the former is just the latter, but with $U_{\mathrm{cl}} = 0$. The additional Neumann term (8.6) imposing the equilibrium contact angle in (8.8) will be compensated anyhow by the Lagrange multiplier λ in the `PinnedContactLine`. Therefore, a stick slip motion can be realized by toggling between both contact angle modes with an indicator function ψ, which is 0 if the contact line is pinned and 1 if the contact line moves according to (8.7). Hence, the relaxation velocity is just augmented by the factor ψ in the weak form:

$$\left[\sigma\left(\sin(\theta_{\mathrm{eq}})\vec{n}_{\mathrm{w}} + \cos(\theta_{\mathrm{eq}})\vec{t}_{\mathrm{w}}\right), \vec{v}\right] + \left[\partial_t\vec{X}\cdot\vec{t}_{\mathrm{w}} - \psi U_{\mathrm{cl}}, \mu\right] + \left[\lambda, \vec{v}\cdot\vec{t}_{\mathrm{w}}\right], \tag{8.9}$$

The factor ψ can be controlled by the `pin()` and `unpin()` methods of the `StickSlipContactLine`. Initially, it is 1, i.e. the contact line could freely move, unless you call `pin()` before running the simulation. The speed of the contact line and the equilibrium contact angle can be set analogous to the `UnpinnedContactLine`.

However, it is cumbersome to switch the contact line dynamics by hand during the simulation. In reality, the contact line of an evaporating droplet starts usually pinned and switches to an unpinned motion once the contact angle θ falls below some receding unpinning contact angle $\theta^{\mathrm{rec}}_{\mathrm{unpin}}$. It can then re-pin, when the contact line recedes so that the contact angle has risen above some value $\theta^{\mathrm{rec}}_{\mathrm{pin}}$, where $\theta^{\mathrm{rec}}_{\mathrm{unpin}} < \theta^{\mathrm{rec}}_{\mathrm{pin}} < \theta_{\mathrm{eq}}$ must hold for an alternating stick-slip motion during evaporation. If the droplet growth, e.g. due to condensation, we can have similar effects. Here, the contact line will unpin once the contact angle θ raises above $\theta^{\mathrm{adv}}_{\mathrm{unpin}}$ and pins again once the contact angle has fallen again below $\theta^{\mathrm{adv}}_{\mathrm{pin}}$ due to the outward motion of the contact line. For stick-slip behavior, therefore $\theta_{\mathrm{eq}} < \theta^{\mathrm{adv}}_{\mathrm{pin}} < \theta^{\mathrm{adv}}_{\mathrm{unpin}}$ must hold. To set these contact angles, you can use the following methods:

Desired event	relevant quantity	method to use
unpin if below and contact angle decreases	$\theta^{\mathrm{rec}}_{\mathrm{unpin}}$	`set_receding_unpin_below_angle()`
pin if above and contact angle increases	$\theta^{\mathrm{rec}}_{\mathrm{pin}}$	`set_receding_pin_above_angle()`
pin if above and contact angle decreases	$\theta^{\mathrm{adv}}_{\mathrm{pin}}$	`set_advancing_pin_below_angle()`
unpin if above and contact angle increases	$\theta^{\mathrm{adv}}_{\mathrm{unpin}}$	`set_advancing_unpin_above_angle()`

The first argument `angle` passed to these methods is the contact angle threshold, where this transition should happen. Note that the angles must be in increasing order following the table to to bottom and the equilibrium contact angle must be in between the receding angles and the advancing angles. If one passes `by_factor=True`, the first argument `angle` is not interpreted as angle, but as factor and the resulting threshold angle is this factor times the equilibrium angle. This means, that e.g. `set_receding_unpin_below_angle(0.9,by_factor=True)` sets $\theta^{\mathrm{rec}}_{\mathrm{unpin}} = 0.9\theta_{\mathrm{eq}}$. Pass `angle=None` to remove a previously set angle. Further arguments are `only_if_decaying`/`only_if_growing` (default `True`) to indeed check whether the actual contact angle grows or shrinks when crossing the theshold. `explicit` (defaults to `True`) tells pyoomph to evaluate the actual contact angle θ from the previous time step. If set to `False`, the contact line dynamics are fully implicitly considered. However, since the transitions from pinned and unpinned are discontinuous, it is likely not converging well in the Newton solver. One can improve it a bit by smearing out the transition angles with the `heaviside_smoothing` argument. Thereby, the transitions, which are implemented by heaviside step functions Θ, e.g. $\Theta(\theta^{\mathrm{rec}}_{\mathrm{unpin}} - \theta)$, will be smoothed out by $\mathrm{atan}((\theta^{\mathrm{rec}}_{\mathrm{unpin}} - \theta)/S)/\pi + 1/2$ with S given by `heaviside_smoothing`. If `explicit=True`, `heaviside_smoothing` does not improve the convergence.

Once such contact angle threshold are set, the methods `pin()` and `unpin()` will not fix or freely move the contact line, unless it is allowed by the contact angle thresholds (i.e. in the hysteretic regions, e.g. if θ is between $\theta^{\mathrm{rec}}_{\mathrm{unpin}}$ and $\theta^{\mathrm{rec}}_{\mathrm{pin}}$). If we are outside these ranges, `unpin()` and `pin()` will have no effect. If we want to override this, i.e. force the contact line to be either pinned or unpinned, irrespectively of the set threshold angles, we must pass the `forced=True` to `pin()` or `unpin()`. The contact line will then remain is this state until one calls `pin()` or `unpin()` again. If this time the `forced` argument is `False` or omitted, the contact line dynamics based on the set angles will take over again.

8.6.4 YoungDupreContactLine

So far, we had to set the equilibrium contact angle θ_{eq} by hand or it will default to the initial contact angle. However, when the surface tension of the liquid-gas interface changes, e.g. due to preferential evaporation of a multi-component droplet, the equilibrium contact angle will change as well. According to the Young-Dupré equation, the equilibrium contact angle is given by

$$\cos\left(\theta_{\mathrm{eq}}\right)=\frac{\sigma_{\mathrm{sg}}-\sigma_{\mathrm{sl}}}{\sigma} \tag{8.10}$$

where σ, σ_{sg} and σ_{sl} are the surface tensions of the liquid-gas, solid-gas and liquid-solid interfaces at the contact line. The `YoungDupreContactLine` inherits from the `StickSlipContactLine`, i.e. all the stick-slip dynamics works as well. In the constructor, one can pass (besides the `cl_speed_scale`) the surface tensions `sigma_sg` and `sigma_sl`. If one do not pass these, its difference in the numerator of (8.10) will be determined by the initial contact angle and initial surface tension, so that the contact angle is initially at equilibrium.

8.6.5 KwokNeumannContactLine

The Kwok-Neumann contact line model was developed semi-empirical by measuring the contact angle of versatile liquid-substrate combinations [33]. The equilibrium contact angle reads

$$\cos\left(\theta_{\mathrm{eq}}\right)=-1+2\sqrt{\frac{\sigma_{\mathrm{sg}}^{0}}{\sigma}}\exp\left(-\beta\left(\sigma_{\mathrm{sg}}^{0}-\sigma\right)^{2}\right)$$

where the coefficient β (set via `beta` in the constructor) defaults to the value of Kwok and Neumann, $\beta=124.7\,\mathrm{m}^4/\mathrm{J}^2$. The solid-gas surface tension σ_{sg}^{0} can be set by `sigma_sg_0` via the constructor. If not set, it will be calculated so that the initial contact angle is in equilibrium given the initial surface tension. Since the `KwokNeumannContactLine` inherits from the `StickSlipContactLine`, also the `cl_speed_scale` can be set and the stick-slip methods are available.

8.7 Surfactants

Opposed to normal mixture components, surfactants are allowed to have an excess concentration at the very interface. In pyoomph, surfactants at the interface are represented by fields Γ_α (measured in $\mathrm{mol}/\mathrm{m}^2$), where α enumerates all present surfactants.

8.7.1 Insoluble surfactant transport equation in presence of mass transfer

In Section 7.5, we already discussed the surfactant transport equation for insoluble surfactants, however, in absence of evaporation. When there is no mass transfer, (7.11) holds. However, if there is mass transfer, we have to modify it to

$$\partial_t\Gamma+\nabla_S\cdot\left(\vec{u}_{\mathrm{P}}\Gamma\right)=\nabla_S\cdot\left(D_S\nabla_S\Gamma\right) \tag{8.11}$$

The only modification is the exchange of the fluid velocity $\vec{u}$ to the velocity $\vec{u}_{\mathrm{P}}$, which is the fluid velocity in tangential direction, but the interface velocity in normal direction, i.e.

$$\vec{u}_{\mathrm{P}}=(\mathbf{1}-\vec{n}\vec{n})\vec{u}+\left(\vec{u}_{\mathrm{I}}\cdot\vec{n}\right)\vec{n}\,. \tag{8.12}$$

In absence of mass transfer, the kinematic boundary conditions dictates that the normal interface velocity and the normal fluid velocity are equal and thus $\vec{u}_{\mathrm{P}}=\vec{u}$. If there is mass transfer, this does not hold. However, the normal velocity in (8.11) must follow the interface velocity, not the fluid velocity. This can be understood by the example of a levitating

spherical droplet evaporating in free space. In the droplet, the fluid velocity $\vec{u}$ will be zero, but the interface velocity will be not due to evaporation. When initially a homogeneous insoluble surfactant concentration Γ_0 is on the droplet (with initial droplet radius R_0), we can simplify the equation to

$$\partial_t\Gamma = -\nabla_S \cdot \left(\left(\vec{u}_\mathrm{I} \cdot \vec{n} \right) \vec{n}\Gamma \right) \,,$$

where the diffusion term has been disregarded since the problem will remain isotropic, i.e. Γ will only depend on time, but not on the location of the interface. Likewise, $\vec{u}_\mathrm{I}$ will point in negative normal direction and is magnitude is constant along the interface due to isotropy. After switching to a radial-symmetric spherical coordinate system, we can carry out the surface divergence, leading to

$$\partial_t\Gamma(t) = -\nabla_S \cdot \left(\vec{n} \right) \vec{u}_\mathrm{I}\Gamma = \frac{2}{R(t)}\vec{u}_\mathrm{I}(t)\Gamma(t) = \frac{2\dot{R}}{R}\Gamma(t) \,.$$

Since the surfactants are insoluble, the total moles of surfactants, i.e. the integral of Γ over the droplet surface, must be conserved. This is given by $\Gamma(t) = \Gamma_0 R_0^2/R^2(t)$. Plugging this in the lhs indeed shows that the surfactant equation conserves the total moles of surfactant. When the fluid velocity $\vec{u} = 0$ would have been used instead of $\vec{u}_\mathrm{P}$, the total amount of surfactants would not be conserved.

In pyoomph, the surfactant transport equation with $\vec{u}_\mathrm{P}$ as advection velocity is already implemented as part of the `MultiComponentNavierStokesInterface`. The velocity $\vec{u}_\mathrm{P}$ is internally calculated according to (8.12) and the rest of the implementation is analogous to Section 6.3. Furthermore, not only one surfactant may be added, but an arbitrary number.

8.7.2 Defining insoluble surfactants

To register an insoluble surfactant to the material library, one has to inherit from the `SurfactantProperties` class and again decorate it with the `@MaterialProperties.register()`. For an insoluble surfactant, it is sufficient to just set a default value for the `surface_diffusivity`, i.e. D_S in (8.11):

```
from pyoomph import *

from pyoomph.materials import *  # materials
import pyoomph.materials.default_materials  # and the default materiasl

from pyoomph.expressions.units import *  # units
from pyoomph.expressions.phys_consts import gas_constant  # and the gas constant


# Register an insoluble surfactant
@MaterialProperties.register()
class MyInsolubleSurfactant(SurfactantProperties):
    name = "my_insoluble_surfactant"

    def __init__(self):
        super(MyInsolubleSurfactant, self).__init__()
        self.surface_diffusivity = 1e-9 * meter ** 2 / second  # default surface␣
↪diffusivity
```

An insoluble surfactant will reduce the surface tension according to some relation. Therefore, we must define interface properties in the material library for each combination of the liquid composition and the surfactants on the interface (and potentially also for different gas compositions). This can be done similar to the specification of liquid-gas interface properties as discussed in Section 8.4. We just inherit from the class `DefaultLiquidGasInterface`, which automatically sets the surface tension to the liquid-gas surface tension from the `default_surface_tension`. We can then modify the surface tension and potentially also the surface diffusivity for this particular interface.

```
# Register the interface properties of a water liquid phase in contact with a gaseous␣
↪phase
# with the surfactant "my_insoluble_surfactant" on the interface
# It is best to inherit from the DefaultLiquidGasInterface to setup all properties to␣
↪reasonable defaults
@MaterialProperties.register()
class InterfaceWaterVsVaporAirWithMyInsolubleSurfactant(DefaultLiquidGasInterface):
    liquid_components = {"water"} # Pure water must be the liquid phase
    # If we uncomment this, it will only be used if the gas phase consist of air and␣
↪water vapor
    # If not set, it is valid for arbitrary gas phases
    #       gas_components = {"air","water"}
    surfactants = {"my_insoluble_surfactant"} # This surfactant must be present

    def __init__(self, phaseA, phaseB, surfactants):
        super(InterfaceWaterVsVaporAirWithMyInsolubleSurfactant, self).__init__
↪(phaseA, phaseB, surfactants)
        # set the surface tension sigma(Gamma)=sigma_0 - R*T*Gamma
        Gamma = var("surfconc_my_insoluble_surfactant")  # surface concentration␣
↪Gamma of the surfactant "my_surfactant"
        T = var("temperature")
        self.surface_tension = self.surface_tension - gas_constant * T * Gamma
        # We could also modify the surface diffusivity for this particular interface
        # self.set_surface_diffusivity("my_surfactant",1e-10*meter**2/second)
```

When the gas phase may be an arbitrary pure gas or gaseous mixture, we can just leave out the `gas` specification. Then, this property will hold for all gases. The `surface_tension` is now just altered by $\sigma(\Gamma,T)=\sigma_0(T)-RT\Gamma$, with R being the gas constant. This relation is the simplest effect of surfactants and belongs to the *Henry isotherm*. It just considers the thermodynamic effect of the presence of other molecules at the interface without any interaction terms.

Additionally, we can change the surface diffusivity D_S to give a different value on each interface, i.e. overriding the default `surface_diffusivity` set in the `SurfactantProperties` class.

When the surfactant is defined, we can obtain the interface properties by a combination of liquid properties, gas properties and a surfactant `dict`. We cannot use the operator `|` anymore, i.e. `liquid | gas` to get the properties, since the surfactant table must be passed as third argument. Therefore, one must call the `get_interface_properties()` function, which finds the right interface properties based on the passed phases and the `surfactants`. The latter is just a `dict`, where the keys are either surfactant names (strings) or surfactant properties loaded by `get_surfactant()`. The values of the `dict surfactants` are the initial concentrations:

```
liquid = get_pure_liquid("water")
gas = get_pure_gas("air")
surfactants = {"my_insoluble_surfactant": 1 * micro * mol / meter ** 2} # Dict␣
↪stating the initial concentration
# alternatively, load the surfactant:
#       my_surfactant=get_surfactant("my_insoluble_surfactant")
#       surfactants = {my_surfactant: 1 * micro * mol / meter ** 2} #

# Gettting interface properties with surfactants
interface = get_interface_properties(liquid, gas, surfactants=surfactants)
```

Again, we can just get the properties, as e.g. the `surface_tension` directly from the properties. But usually, these are functions of the liquid composition and the temperature. If one wants to get the initial surface tension, e.g. to set a reasonable pressure scale based on the initial surface tension, one can again plug in the initial mixture composition of the liquid into the expression to evaluate the expression at the initial liquid composition and temperature:

```
# Getting e.g. the surface tension
sigma=interface.surface_tension
print(sigma) # Rather complicated, since it depends on T and Gamma

# First plug in by the liquid initial composition and the temperature
sigma1=liquid.evaluate_at_condition(sigma,liquid.initial_condition,
↪temperature=20*celsius)
print(sigma1) # Still a function of Gamma

# Now also evaluate at the initial surfactant concentration
sigma2=interface.evaluate_at_initial_surfactant_concentrations(sigma1) # plug in␣
↪initial surfactant concentration
print(sigma2)
```

However, `sigma1` will still depend on the surface concentration Γ of the surfactant `"my_insoluble_surfactant"`. To plug in the initial surface concentrations, one to call the `evaluate_at_initial_surfactant_concentrations()` method of the interface properties. Thereby, `sigma2` will be just a constant value, corresponding to the initial surface tension of this interface.

8.7.3 Soluble surfactants and surfactant isotherms

Soluble surfactants are allowed to move from the liquid bulk phase to the interface and vice versa. Hence, in order for a surfactants to be soluble, we must have it in the liquid phase as well as surfactant concentration at the interface. In fact, the `SurfactantProperties` class we have used so far to define insoluble surfactants inherits from the `PureLiquidProperties` class, i.e. each surfactant is automatically also a pure liquid and can hence be mixed with other liquids. However, before doing so, we must at least set the `molar_mass` so that the mole fractions in the liquid mixture can be calculated. This is e.g. relevant for Raoult's law for the evaporation (cf. (8.4)).

```
# Register an soluble surfactant
@MaterialProperties.register()
class MySolubleSurfactant(SurfactantProperties):  # It is automatically also a pure␣
↪liquid
    name = "my_soluble_surfactant"

    def __init__(self):
        super(MySolubleSurfactant, self).__init__()
        self.molar_mass = 100 * gram / mol  # required so that we can mix it with␣
↪other liquids
        self.surface_diffusivity = 0.5e-9 * meter ** 2 / second  # default surface␣
↪diffusivity
```

Since the surfactant is now also in the liquid phase, we must define the properties of the bulk liquid mixture we want to use. In particular, the presence of the surfactant could influence the `dynamic_viscosity` or `mass_density`. However, for low concentrations, it is reasonable to disregard this effect and just copy the values of e.g. pure water:

```
# Define how the liquid mixture should behave in the bulk
@MaterialProperties.register()
class MixLiquidWaterMySolubleSurfactant(MixtureLiquidProperties):
    components = {"water", "my_soluble_surfactant"}

    def __init__(self, pure_props):
        super(MixLiquidWaterMySolubleSurfactant, self).__init__(pure_props)
        # Copy the relevant properties from the water. We assume that the surfactant␣
↪concentration is small
        # so that all properties are close to these of water
```

(continues on next page)

(continued from previous page)

```
        self.mass_density = self.pure_properties["water"].mass_density
        self.dynamic_viscosity = self.pure_properties["water"].dynamic_viscosity
        self.default_surface_tension["gas"] = self.pure_properties["water"].default_
↪surface_tension["gas"]
        # However, we must set a diffusivity
        self.set_diffusion_coefficient(1e-9 * meter ** 2 / second)
```

However, we must specify the diffusivity in the bulk. This may be different from the diffusivity at the interface.

Of course, also the properties of the interface are relevant, i.e. how the surfactant influences the surface tension. For soluble surfactants, there is another relevant property to set, namely how the surfactant moves between the bulk and the interface. Therefore, the surfactant transport equation (8.11) is augmented by a sink/source term S_Γ:

$$\partial_t\Gamma + \nabla_S \cdot (\vec{u}_\mathrm{P}\Gamma) = \nabla_S \cdot (D_S \nabla_S \Gamma) + S_\Gamma \tag{8.13}$$

S_Γ is now the flux (in $\mathrm{mol}/\mathrm{m}^2 \cdot \mathrm{s}$) from the bulk to the interface. This flux is constituted by adsorption of surfactants to the interface (positive contribution to S_Γ) and desorption from the interface to the bulk (negative contribution to S_Γ). Of course, this transfer has to be compensated by the bulk in order to conserve the total mass of the surfactants, i.e. the sum in the liquid bulk and the interface. The molar flux S_Γ can be converted to a mass flux by multiplying it with the molar mass M of the surfactant and this flux can be applied as Neumann condition, i.e. as diffusive mass flux, for the corresponding compositional advection-diffusion equation (8.1). It does not contribute to the mass transfer flux rate j_α, though, since the surfactant does not cross the interface. Of course, all this is subject to a few assumptions, since a molecule requires volume in the bulk phase, but will occupy zero volume at the interface. The flux S_Γ is automatically considered in the `MultiComponentNavierStokesInterface`, so there is nothing to be done.

For the adsorption/desorption rates, there are plenty of models in the literature. To that end, pyoomph offers the most common *surfactant isotherms* in the module `pyoomph.materials.surfactant_isotherms`. The isotherms are usually expressed in terms of the surface concentration Γ and the *molar concentration* C in the bulk, where the latter can be calculated from the bulk mass fraction w via $C = \rho w/M$. The molar concentrations can be accessed in pyoomph via the prefix `"molarconc_"`, e.g. `var("molarconc_my_soluble_surfactant")`. All surfactant isoterms contain expressions for the adsorption flux S_Γ^ads, S_Γ^des and the surface pressure Π, where the latter is just the decrease of the surface tension due to the presence of the surfactant, i.e. $\sigma = \sigma_0 - \Pi$. The total flux is just $S_\Gamma = S_\Gamma^\mathrm{ads} - S_\Gamma^\mathrm{des}$. The equilibrium relation between C and Γ, where the surfactants in the bulk and the interface are at equilibrium, is given by $S_\Gamma = 0$, i.e. $S_\Gamma^\mathrm{ads} = S_\Gamma^\mathrm{des}$. These are listed for all predefined isotherms in Table 8.1.

Table 8.1: Predefined surfactant isotherms stating the adsorption and desorption rates and the surface pressure.

isotherm	S_Γ^ads	S_Γ^des	Π
`HenryIsotherm`	$k_\mathrm{ads}C$	$k_\mathrm{des}\Gamma$	$RT\Gamma$
`LangmuirIsotherm`	$k_\mathrm{ads}C\frac{\Gamma_\infty-\Gamma}{\Gamma_\infty}$	$k_\mathrm{des}\Gamma$	$-RT\Gamma_\infty \ln\left(1-\frac{\Gamma}{\Gamma_\infty}\right)$
`VolmerIsotherm`	$k_\mathrm{ads}C\frac{\Gamma_\infty-\Gamma}{\Gamma_\infty}$	$k_\mathrm{des}\Gamma\exp\left(\frac{\Gamma}{\Gamma_\infty-\Gamma}\right)$	$\frac{RT\Gamma_\infty}{1-\Gamma\Gamma_\infty}$
`FrumkinIsotherm`	$k_\mathrm{ads}C\frac{\Gamma_\infty-\Gamma}{\Gamma_\infty}$	$k_\mathrm{des}\Gamma\exp\left(-\frac{\beta\Gamma}{RT}\right)$	$-RT\Gamma_\infty \ln(1-\frac{\Gamma}{\Gamma_\infty})$
`VanDerWaalsIsotherm`	$k_\mathrm{ads}C\frac{\Gamma_\infty-\Gamma}{\Gamma_\infty}$	$k_\mathrm{des}\Gamma\exp\left(\frac{\Gamma}{\Gamma_\infty-\Gamma}-\frac{\beta\Gamma}{RT}\right)$	$\frac{RT\Gamma}{1-\Gamma/\Gamma_\infty}-\frac{\beta\Gamma^2}{2}$

To constuct an isotherm, we just have to pass the surfactant name and the parameters `k_ads` and `k_des`, as well as potential further parameters `GammaInfty` and `beta` to the constructor. Sometimes in the literature, you will find a value K, which is just $K = k_\mathrm{ads}/k_\mathrm{des}$. Moreover, some literature define k_ads as product of $k_\mathrm{ads}\Gamma_\infty$. Here, the convention was chosen that k_ads always has the units m/s, whereas k_des has always the unit $1/\mathrm{s}$. If required for the isotherm, the infinity concentration Γ_∞ has the unit $\mathrm{mol}/\mathrm{m}^2$ and the interaction parameter β is associated with the units $\mathrm{m}^4/(\mathrm{mol}^2\cdot\mathrm{s}^2)$. Hence, when using values from the literature, always make sure that you cast the isotherms and parameters accordingly.

The typical time scale of the surfactant equilibration is given by both k_{ads} and k_{des}, whereas the ratio of these and the further parameters control the equilibrium and the surface tension reduction.

To use the isotherms on an interface, we just construct it and apply the its method `apply_on_interface()`. This will set the `surface_tension` of this liquid-gas interface to the passed `pure_surface_tension` minus the surface pressure Π. Furthermore, it will set the transfer rate S_Γ according to the particular isotherm. S_Γ can alternatively be set by hand with the `surfactant_adsorption_rate` dict:

```
@MaterialProperties.register()
class InterfaceWaterMySolubleSurfactantVSGas(DefaultLiquidGasInterface):
    liquid_components = {"water", "my_soluble_surfactant"}  # Water and the␣
↪surfactant are in the liquid phase
    # gas_components = {"air","water"} # do not specify any particular gas phase␣
↪here: Hold for all gas mixtures
    surfactants = {"my_soluble_surfactant"}  # The soluble surfactant may also be on␣
↪the interface

    def __init__(self, phaseA, phaseB, surfactants):
        super(InterfaceWaterMySolubleSurfactantVSGas, self).__init__(phaseA, phaseB,␣
↪surfactants)
        # Create a LangmuirIsotherm for my_soluble_surfactant
        isotherm = LangmuirIsotherm("my_soluble_surfactant", k_ads=5e-6 * meter /␣
↪second, k_des=9.5 / second,
                                     GammaInfty=5 * micro * mol / meter ** 2)
        # And apply it to this interface. This will modify self.surface_tension by␣
↪substracting the surface pressure
        # and furthermore it will set self.surfactant_adsorption_rate["my_soluble_
↪surfactant"] to the total ad-/desorption flux
        isotherm.apply_on_interface(self, pure_surface_tension=self.surface_tension,
↪min_surface_tension=20*milli*newton/meter)
```

Since some isotherms have an unbounded surface pressure, the surface tension might become negative once the surfactant concentration exceeds the validity range of the isotherm. Therefore, you can pass a `min_surface_tension` to the `apply_on_interface()` call to make sure the surface tension never becomes negative. This can help to prevent crashes of the simulation, when the surfactant leaves the valid bounds.

As for the insoluble surfactants, the interface properties of for an interface with soluble surfactants is obtained by `get_interface_properties()`. However, in order for the surfactant to be indeed soluble, the surfactant must be present in both the liquid bulk properties and the interface `surfactants`.

```
# For soluble surfactants, we also must have it in the bulk (potentially at zero␣
↪concentration)
liquid = Mixture(get_pure_liquid("water")+0.001*get_pure_liquid("my_soluble_surfactant
↪"))
gas = get_pure_gas("air")
# Dict stating the initial surface concentration
surfactants = {"my_soluble_surfactant": 1 * micro * mol / meter ** 2}

# Getting interface properties with surfactants.
# For a soluble surfactant, it must be present in both the liquid phase and in the␣
↪surfactants dict
# Any of them may be present at zero concentration, but it must be specified to be␣
↪present at all
interface = get_interface_properties(liquid, gas, surfactants=surfactants)
```

8.8 UNIFAC models

For the calculation of activity coefficients, group contribution methods provide a promising approach. Of course, these models cannot yield the exact activity coefficients for arbitrary mixtures, but for a widespread range of mixtures they give quite reasonable results.

pyoomph has three different parameter sets of UNIFAC implemented, namely the original UNIFAC (`"UNIFAC"`) [22, 23], Dortmund modified UNIFAC (`"Dortmund"`) [39] and AIOMFAC (`"AIOMFAC"`) [47, 48].

Group contribution methods are based on the assumption that each molecule can be split into functional groups of atoms. Water consists of just a single group, namely a `"H2O"` group, whereas most other molecules are separated into several subgroups. Ethanol, for example, is split in the original UNIFAC into three functional groups, namely `"CH3"`, `"CH2"` and `"OH"`, all appearing once in the ethanol molecule. Groups can also appear multiple times in each molecule, e.g. 1×`"CH3"`, 4×`"CH2"`, 1×`"CH"` and 2×`"OH"` is the group decomposition of 1,2-hexanediol in the original UNIFAC model. To set these for the `"Original"` UNIFAC model, one has to use the `set_unifac_groups()` method, i.e. add the line

```
self.set_unifac_groups({"CH3": 1, "CH2": 4, "CH": 1, "OH": 2}, only_for="Original")
```

to the definition of the pure liquid 1,2-hexanediol. One can add more lines with different `only_for` arguments to set the groups for `"Dortmund"` UNIFAC and `"AIOMFAC"`.

Details on UNIFAC-like models are not given here, but can be found e.g. in the references [22, 23, 39, 47, 48]. For just the equations, *Wikipedia* provides a brief overview at https://en.wikipedia.org/wiki/UNIFAC.

The possible groups and their parameters as well as the interaction table of different groups can be found in the python files in `pyoomph.materials.UNIFAC`, e.g. in `pyoomph.materials.UNIFAC.aiomfac`. To use these models in a liquid mixture for the activity coefficients, the groups of all components in the mixture have to be defined by the `set_unifac_groups()` for the desired model as described above. In the mixture itself, you can assemble the activity coefficients with the `set_activity_coefficients_by_unifac()` method of the liquid mixture class. By default, it will also set the vapor pressures of all components according to the non-ideal Raoult's law. If the vapor pressure of a pure component is not set, this component will be considered as nonvolatile also in the mixture.

Important: When publishing results based on any of these UNIFAC-like models, please cite the corresponding papers:

The published parameters of the original and modified (Dortmund) UNIFAC model were taken with kind permission from the *DDBST* website https://www.ddbst.com. Please cite the papers listed at https://www.ddbst.com/published-parameters-unifac.html for original UNIFAC and https://www.ddbst.com/PublishedParametersUNIFACDO.html for modified (Dortmund) UNIFAC.

Also note that the *UNIFAC Consortium* provides *updated and revised parameters*, which will increase the accuracy of the predicted activity coefficients. Please refer to the https://unifac.ddbst.com/unifac_.html for more information. These updated parameters are not included in pyoomph.

When using the AIOMFAC model, please cite the papers listed here https://aiomfac.lab.mcgill.ca/citation.html.

CHAPTER

NINE

DISCONTINUOUS GALERKIN METHODS

So far, the considered solutions were always approximated by shape functions which are continuous in space. One exception are the Crouxeiz-Raviart elements in Section 4.4.8, where the pressure was allowed to discontinuously jump between two elements. However, in this particular case, the pressure is just an auxiliary field enforcing the incompressibility.

In general, any field can be approximated by a discontinuous discretization, which is can be helpful if the solution itself become (close to) discontinuous, e.g., in the case of shock waves. In this case, the standard finite element method (*Continuous Galerkin method*, CG) is not well suited to capture the solution accurately. The *Discontinuous Galerkin* (DG) method is a generalization which allows for such discontinuous solutions. The idea is to consider the solution in each element separately and to allow for discontinuities at the element interfaces. The solution is then approximated by a piecewise polynomial function which is continuous in each element but can have jumps at the element interfaces.

In that case, however, it is important to incorporate these jumps into the weak formulation. This can be done by introducing so-called *numerical fluxes* at the element interfaces. These fluxes are used to enforce the continuity of the solution across the element interfaces in a weaker sense.

One can also think of the DG method as a generalization of finite volume methods. In the latter, the approximations are usually constant in each element, whereas in the DG method, the approximations can be piecewise polynomial.

For each of the continuous spaces, first (`"C1"`) and second order (`"C2"`) and the corresponding spaces `"C1TB"` and `"C2TB"` enriched by a bubble on triangular/tetrahedral elements, pyoomph provides discontinuous versions, namely `"D1"`, `"D2"`, `"D1TB"` and `"D2TB"`. The discontinuous spaces can be used in the same way as the continuous spaces, but the weak formulation will have to be modified to incorporate the jumps at the element interfaces.

Opposed to the pure elemental discontinuous spaces `"D0"` and `"DL"` discussed so far, the values of `"D1"`, `"D2"`, `"D1TB"` and `"D2TB"` can be accessed directly at interfaces (i.e. without specifying `domain=".."` when binding them via `var()` or `var_and_test()`). You can also set strong Dirichlet boundary conditions on these spaces, which is also not possible for `"D0"` and `"DL"`.

9.1 Advection-diffusion equation with an upwind scheme

To illustrate how to define the neccessary facet terms in the weak formulation of DG methods, we will refer back to the example of the advection-diffusion equation from Section 5.2.2 where we discussed the SUPG stabilization.

In strong formulation, we want to solve

$$\partial_t c + \nabla\cdot(\vec{u}c) - \nabla\cdot(D\nabla c) = 0$$

Upon multiplication with a test function d and subsequent spatial integration, we get

$$(\partial_t c, d) + (\nabla\cdot(\vec{u}c), d) - (\nabla\cdot(D\nabla c), d) = 0$$

Our first focus is now the advection term, $(\nabla\cdot(\vec{u}c),d)$, which be split into the contributions of each element E, i.e.

$$
\begin{aligned}
(\nabla\cdot(uc),d) &= \sum_E (\nabla\cdot(\vec{u}c),d)_E \\
&= \sum_E \left\{-(\vec{u}c,\nabla d)_E + \langle \vec{n}\cdot\vec{u}c,d\rangle_{\partial E}\right\}
\end{aligned}
$$

Here, $\vec{n}$ is the normal pointing outward at each point on the boundary ∂E of each element E. Opposed to continuous Galerkin methods, $\vec{u}$ and c may be different on both sides of the boundary in discontinuous Galerkin methods. This means that the integrations along the interior element boundaries ∂E do not cancel out by the corresponding interior boundary contribution of the adjacent element.

Nevertheless, the boundary integrals can be rearranged into the contributions of each facet F, shared by two elements E^+ and E^-. Each facet is only defined once in the mesh, even if it is shared by two elements. The facet normal $\vec{n}$ is pointing from the element E^+ to the element E^-. The particular order, i.e. which element is assigned as E^+ and E^- viewed from each facet F is irrelevant here. We can now write it as bulk integration over the full domain Ω, its exterior boundary $\partial\Omega$, and the interior facets F:

$$
\begin{aligned}
(\nabla\cdot(uc),d) = &-(\vec{u}c,\nabla d)_\Omega + \langle \vec{n}\cdot\vec{u}c,d\rangle_{\partial\Omega} \\
&+ \sum_F \left\{\left\langle \vec{n}^+\cdot\vec{u}^+c^+,d^+\right\rangle_F + \left\langle \vec{n}^-\cdot\vec{u}^-c^-,d^-\right\rangle_F\right\}
\end{aligned}
$$

Note that $\vec{n}^- = -\vec{n}^+$ holds, since the nodal coordinates are always continuous in pyoomph (however, it might not hold on curved manifold with a co-dimension). Also, we assume a continuous given velocity field, so that $\vec{u}^- = -\vec{u}^+$ holds as well. In CG methods, also $c^- = c^+$ and $d^- = d^+$ holds, so that the facet terms indeed cancel out. In DG methods, however, this is not the case. In fact, the interaction of c^+ and c^- across the facet must be explicitly implemented in a reasonable way that reflects the nature of the equation. Here, we have to make sure that whatever is transported from one element to the other is also removed from the first element. To that end, we apply here the *upwind scheme* by transporting c^+ if $\vec{n}^+\cdot\vec{u}^+ > 0$ and c^- otherwise. To that end, we define an auxiliary velocity normal to the facet, which makes sure to account only for upwind transport and vanishes otherwise

$$
u_n^{\mathrm{up},\pm} = \frac{1}{2}\left(\vec{u}^\pm\cdot\vec{n}^\pm + |\vec{u}^\pm\cdot\vec{n}^\pm|\right)
$$

and rewrite the facet terms of the advection by

$$
\sum_F \left\langle u_n^{\mathrm{up},+}c^+ - u_n^{\mathrm{up},-}c^-, d^+ - d^-\right\rangle_F
$$

Obviously, either $u_n^{\mathrm{up},+}$ or $u_n^{\mathrm{up},-}$ is positive and the other one vanishes. Thereby, we either transport c^+ or c^-, depending on the direction of the velocity, by removing this flux from one element and adding it to the other, i.e. ensure conservation.

Since such *jump* terms, i.e. the difference of an expression on the two sides of a facet, are common in DG methods, pyoomph offers the function `jump()`, which lets you write the above expression in pyoomph in a shorter notation. This will be addressed later in the implementation.

For the diffusion term, similar facet terms must be derived. Doing the same splitting in bulk contribution, exterior boundary, and interior facets, we get

$$
-\left(D\nabla^2 c,d\right) = (D\nabla c,\nabla d)_\Omega - \langle D\vec{n}\cdot\nabla c,d\rangle_{\partial\Omega} + \sum_F \left\{\left\langle J_n^+,d^+\right\rangle_F + \left\langle J_n^-,d^-\right\rangle_F\right\}
$$

where we abbreviated the diffusive fluxes through to the facet by

$$
\vec{J}^\pm = -D\cdot\nabla c^\pm
$$

Opposed to the upwind scheme for advection, the diffusive fluxes should be considered in a symmetric way, i.e. we can define the average on both sides

$$
\vec{J}_n^{\mathrm{avg}} = \frac{1}{2}\left(\vec{J}_n^+ + \vec{J}_n^-\right)
$$

For such a term, we can use the `avg()` function in pyoomph, which calculates the average of the expression on both sides of the facet. This will be addressed later in the implementation as well. Similar to the transport by advection, the diffusive fluxes must be removed from one element and added to the other, i.e. we must ensure conservation, which can be done by replacing the facet terms stemming from diffusion by

$$\left\langle J_n^+, d^+\right\rangle_F + \left\langle J_n^-, d^-\right\rangle_F = \left\langle \vec{J}_n^{\text{avg}} \cdot \vec{n}^+, d^+ - d^-\right\rangle_F = -\left\langle D\,\text{avg}(\nabla c)\cdot\vec{n}^+, \text{jump}(d)\right\rangle_F$$

While this considers a conservative and reasonable diffusive transport across the facet, this form neither penalizes huge jumps in the concentraction c (which should be smoothed by diffusion) nor is it symmetric. For a symmetric, consistent and coercive formulation we can weakly enforce continuity of c in the sense of *Nitsche's method* by adding a symmetric and a penalty term to the facet terms, i.e. we write

$$\begin{aligned}\left\langle J_n^+, d^+\right\rangle_F + \left\langle J_n^-, d^-\right\rangle_F = &-\left\langle D\,\text{avg}(\nabla c)\cdot\vec{n}^+, \text{jump}(d)\right\rangle_F \\ &-\left\langle D\,\text{jump}(c)\cdot\vec{n}^+, \text{avg}(\nabla d)\right\rangle_F \\ &+\left\langle D\frac{\alpha}{h}\,\text{jump}(c), \text{jump}(d)\right\rangle_F\end{aligned}$$

Here, α is a penalty parameter and h is the average element size of both elements attached to the facet. The penalty term ensures coercivity of the problem, i.e. the existence of a unique solution. The penalty parameter α should be chosen large enough to ensure coercivity, but small enough to not dominate the solution. The average element size h is calculated by the average of the element sizes of both elements attached to the facet.

For the implementation, the equation starts as usual. We allow to pass and arbitrary finite element space, continuous or discontinuous:

```
class ConvectionDiffusionEquation(Equations):
   def __init__(self, u, D ,space="C2",alpha_DG=5):
      super(ConvectionDiffusionEquation, self).__init__()
      self.u = u  # advection velocity
      self.D = D  # diffusivity
      self.space=space
      # Activate interior facet terms if the space is discontinuous
      self.requires_interior_facet_terms=is_DG_space(self.space)
      self.alpha_DG=alpha_DG # penalty parameter for DG

   def define_fields(self):
      self.define_scalar_field("c", self.space)
```

Whenever facet terms must be added, as here, we must set the property `requires_interior_facet_terms` to `True`. This will trigger the generation of an interior skeleton mesh which includes all interior facets of the mesh. Exterior boundaries are not part of this skeleton mesh. Each facet appears only once in the interior skeleton mesh, although each interior facet is shared by two elements. As an auxiliary method, the function `is_DG_space()` is used, which automatically detects if the space is discontinuous.

The definition of the weak form now also tests whether we have an continuous space or must add additional facet terms:

```
def define_residuals(self):
    c, ctest = var_and_test("c")
    # Conventional form, used for CG spaces
    self.add_weak(partial_t(c), ctest)
    self.add_weak(self.D * grad(c) -self.u*c, grad(ctest))

    if is_DG_space(self.space):
        # Additional facet terms for DG spaces
        h_avg=avg(var("cartesian_element_length_h")) # length of an element:
        n=var("normal") # in facet terms, the normal vector is the facet normal. For␣
↪the element normal, var("normal",domain="..") can be used.
```

(continues on next page)

(continued from previous page)

```
        # if used without any restriction, i.e. outside from jump or average, it will␣
↪default to n^+

        # Upwind scheme. See whether the velocity is in the direction of the normal␣
↪vector, otherwise, it will be zero
        un_upwind=(dot(self.u, n) + absolute(dot(self.u, n)))/2

        # Assemble the facet terms:
        facet_terms=weak(self.D*(self.alpha_DG/h_avg)*jump(c),jump(ctest))
        facet_terms=-weak(self.D*jump(c)*n,avg(grad(ctest)))
        facet_terms=-weak(self.D*avg(grad(c)),jump(ctest)*n)
        facet_terms+=weak( jump(un_upwind*c,at_facet=True) ,jump(ctest))

        # And add them to the skeleton mesh of the facets
        self.add_interior_facet_residual(facet_terms)
```

Again, we make use of `is_DG_space()` to check whether the space is discontinuous. If this is the case, we define the facet terms. For these terms, we first define the upwind convection, which is only giving a contribution if the velocity is advecting in the same direction as the facet normal, which can be bound by `var("normal")`. The normal of the element itself, i.e. not the facet normal, can be obtained by `var("normal",domain="..")`. This is a consequence from the expansions of the variables at the domain where you add the residual terms, which are here the facets. For the elements, you hence must go one level up in the domain hierarchy.

We furthermore define the average element length h, which is calculated by the average `avg()` of both elements sizes attached to the facet. The element size can be obtained by the special variable name `var("cartesian_element_length_h")`. You can also use `var("element_length_h")` instead, which is identical for Cartesian coordinate systems, however, for other coordinate systems, both will differ. In fact, these lengths are calculated by integrating the elemental length/area/volume (depending on the element dimension N_{el}) and subsequently taking the N_{el}-th root of the result. The length/area/volume calculation is done in a Cartesian coordinate system for `var("cartesian_element_length_h")` and in the coordinate system of the current equations for `var("element_length_h")`. In dimensional problems, both consider the scaling `scale_factor("spatial")`, i.e. h will be measured in meters if the problem is dimensional.

Also, we use `jump()` to calculate the jumps of the variables across the facets. By default, both `avg()` and `jump()` expands all contained variables at the bulk element domains on both sides. Both attached bulk elements can also be accessed directly via the domains `"+"` and `"-"`. Here, we could e.g. replace `avg(var("cartesian_element_length_h"))` by `(var("cartesian_element_length_h", domain="+"))+var("cartesian_element_length_h",domain="-"))/2`, which is exactly what `avg()` does.

However, in the last `jump()`, we want to calculate the jump of the upwind convection term. Since this term requires the facet normal, we must make sure that `var("normal")` is indeed evaluated at both sides of the facet, not the attached bulk elements. Therefore, we must use the `at_facet=True` flag to indicate that the variables should be expanded at the facet domain. It is also possible to explicitly on both sides of the facet domain by using the domains `"+|"` and `"|-"`, respectively.

Note that one cannot know a priori which element is on the `"+"` and which is on the `"-"` side of the facet. The order of the elements is not guaranteed. However, in reasonable formulations of such facet terms, the order of the elements should not matter.

Finally, we add the facet terms to the skeleton mesh of the facets by the `add_interior_facet_residual()` method.

The problem code reads as before, but we use the keyword argument `discontinuous=True` to obtain a better output to include the discontinuities. Thereby, each element writes its nodes individually to the output, i.e. overlapping nodes are multiple times present in the output. The same also works for the `MeshFileOutput` class to write VTU files in higher dimensions. The default `discontinuous=False` will just take the average value at overlapping nodes for the

output.

```
class OneDimAdvectionDiffusionProblem(Problem):
   def __init__(self):
      super(OneDimAdvectionDiffusionProblem, self).__init__()
      self.u=vector(1,0)
      self.D=0.0001
      self.space="D1"

   def define_problem(self):
      self.add_mesh(LineMesh(N=100,size=100,minimum=-20)) # coarse mesh from [-20:80]

      eqs=TextFileOutput(discontinuous=True)
      eqs+=ConvectionDiffusionEquation(u=self.u,D=self.D,space=self.space)

      x=var("coordinate_x")
      cinit=exp(-x**2*0.25)
      eqs+=InitialCondition(c=cinit)

      self.add_equations(eqs@"domain")


if __name__=="__main__":
   with OneDimAdvectionDiffusionProblem() as problem:
      problem.run(50,outstep=1,maxstep=0.1)
```

The discontinuous output is plotted in Fig. 9.1.

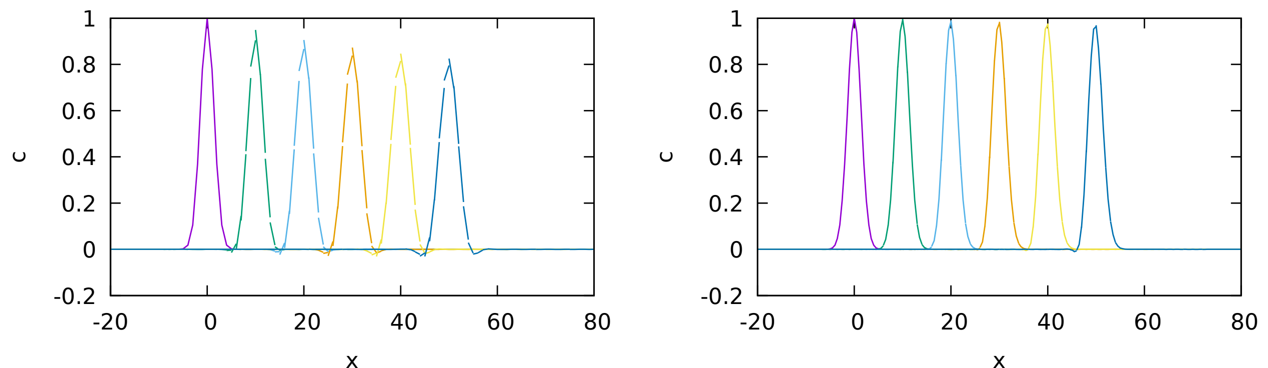

Fig. 9.1: Discontinuous Galerkin method for the advection-diffusion equation on spaces "D1" (left) and "D2" (right). The solutions is shown at times steps of 10. The discontinuities are clearly visible. Also, "D1" has too few degrees of freedom, which leads to severe numerical diffusion. This is typical for underresolved upwind schemes, as also seen in the in the SUPG implementation, cf. Fig. 5.5.

Note: It is easily possible to switch the upwind scheme by a central scheme by replacing `un_upwind` by `un_central=dot(self.u, n)/2`

9.2 Weakly imposing Dirichlet boundary conditions

The last example illustrated how the continuity of the concentration field c across the facets can be enforced in a weak sense by adding facet terms. If you use the conventional `DirichletBC` on the exterior boundaries, it will enforce the conditions strongly, also in discontinuous Galerkin methods. However, in some cases, you may want to enforce the Dirichlet boundary conditions weakly, i.e. in the same way as the facet terms. In order to easily switch between continuous and discontinuous Galerkin spaces, one can still use the `DirichletBC` class, but one has to add the implementation of the weak Dirichlet boundary conditions manually to the equation class. This will be discussed in the following.

We consider a simple Poisson equation, again implemented for both continuous and discontinuous Galerkin spaces. The facet terms can be directly copied from the diffusion term of the advection-diffusion equation. The only modification we do is adding the keyword argument `allow_DL_and_D0=True` to the test for discontinuous spaces via the `is_DG_space()` function. Thereby, we can use the discontinous spaces without degrees of freedom at the nodes, i.e. `"DL"` and `"D0"`, as well.

```
class PoissonEquations(Equations):
            def __init__(self,f,space,alpha_DG=4):
                super().__init__()
                self.f=f
                self.space=space
                self.requires_interior_facet_terms=is_DG_space(self.space, allow_DL_
↪and_D0=True)
                self.alpha_DG=alpha_DG

            def define_fields(self):
                 self.define_scalar_field("u",self.space)

            def define_residuals(self):
                u,v=var_and_test("u")
                # Both continuous and discontinuous spaces
                self.add_residual(weak(grad(u),grad(v))-weak(self.f,v))
                if is_DG_space(self.space, allow_DL_and_D0=True):
                  # Discontinuous penalization
                  h_avg=avg(var("cartesian_element_length_h"))
                  n=var("normal") # will default to n^+ if used without any␣
↪restriction in facets

                  facet_terms= weak(self.alpha_DG/h_avg*jump(u),jump(v))
                  facet_terms-=weak(jump(u)*n,avg(grad(v)))
                  facet_terms-=weak(avg(grad(u)),jump(v)*n)
                  self.add_interior_facet_residual(facet_terms)
```

However, we now also add a special function which gives the correct weak terms for weakly imposed Dirichlet boundary conditions. This function must return the weak terms that are necessary to enforce some particular Dirichlet value. These are essentially the same as the facet terms, however, instead of `jump()`, we just take the current value on the boundary minus the prescribed value. The function `avg()` is just replaced by the evaluation of the variable on the boundary. In case we do not provide such a field or the field is continuous, we just return `None`, advising the `DirichletBC` to impose the value strongy:

```
def get_weak_dirichlet_terms_for_DG(self, fieldname, value):
   if fieldname!="u" or not is_DG_space(self.space, allow_DL_and_D0=True):
      return None
   else:
      u,v=var_and_test("u",domain="..") # bind the bulk field to get bulk gradients
      n=var("normal") # exterior normal
      h=var("cartesian_element_length_h",domain="..") # element size of the bulk␣
```

(continues on next page)

(continued from previous page)

```
↪element
      facet_terms=weak(self.alpha_DG/h*(u-value),v)
      facet_terms-=weak((u-value)*n,grad(v))
      facet_terms-=weak(grad(u),v*n)
      return facet_terms
```

The problem is as usual, but we allow to select the space and whether we want to use the weak Dirichlet boundary conditions when possible:

```
class PoissonProblem(Problem):
   def __init__(self):
      super().__init__()
      x=var("coordinate")
      self.f=500.0*exp(-((x[0] - 0.5)** 2 + (x[1] )**2)/ 0.02)
      self.space="D1"
      self.prefer_weak_dirichlet=True
      self.alpha_DG=4
      self.N=8

   def define_problem(self):
      self+=RectangularQuadMesh(N=self.N)
      eqs=MeshFileOutput(discontinuous=True)
      eqs+=PoissonEquations(self.f,self.space,self.alpha_DG)
      eqs+=DirichletBC(u=0,prefer_weak_for_DG=self.prefer_weak_dirichlet)@["left",
↪"right","top","bottom"]
      self+=eqs@"domain"

with PoissonProblem() as problem:
   problem.solve()
   problem.output()
```

By default, `DirichletBC` will impose the conditions weakly whenever the equations in the bulk provide corresponding facet terms by the method `get_weak_dirichlet_terms_for_DG()`. If this function returns `None` or if the keyword argument `prefer_weak_for_DG` in `DirichletBC` is set to `False`, the conditions will be imposed strongly. The output is shown in Fig. 9.2.

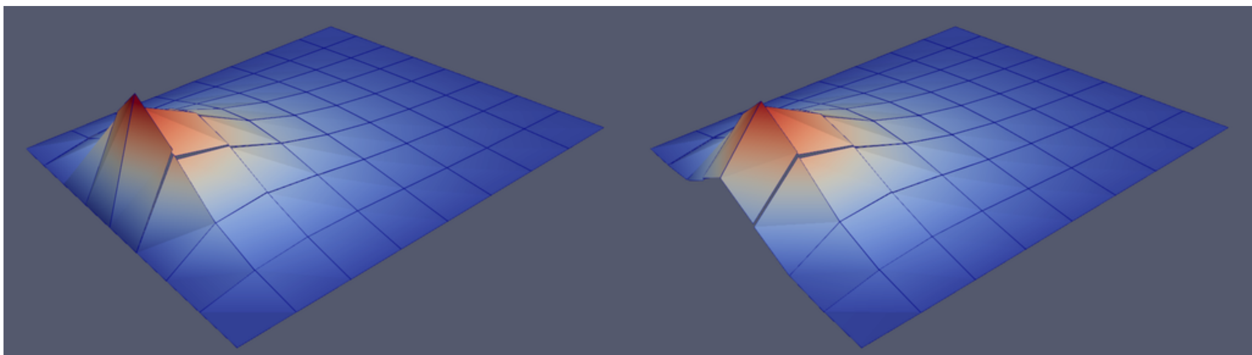

Fig. 9.2: Strongly (left) and weakly (right) imposed Dirichlet boundary conditions. While the strongly imposed conditions exactly fulfill the relation at the boundaries, they induce stronger discontinuities in the bulk. Upon refinement, both approaches converge to the same correct solution.

Finally, we want to address that the discontinous Galerkin implementation can easily be switched to a *finite volume method*. In such methods, quantities are usually element-wise constant, i.e. approximated on the space `"D0"`. If setting `space="D0"`, all terms involving `grad(u)` and `grad(v)` will vanish, since the gradients are zero in the element-wise

constant space. The weak formulation will hence only read

$$-(f,v)_\Omega + \sum_F \left\langle \frac{\alpha}{h_\text{avg}}(u^+ - u^-), v^+ - v^- \right\rangle_F = 0\,,$$

i.e. we only integate over the source f in the bulk and the fluxes are just represented by the jumps of the field values. The flux terms can be understood as finite difference approximation of the fluxes, i.e. we just take the difference of the values at the cell centers, divided by the distance h_avg. While α is a penalty parameter for higher order polynominal approximations, it is now crucial to set the penalty parameter to $\alpha = 1$ to recover the correct finite difference approximations when using the `"D0"` space. Upon setting `space="D0"` and `alpha_DG=1` and increasing the number of elements to `N=40`, we obtain the solution plotted on the left side of Fig. 9.3.

Discontinous Galerkin methods can hence be understood as generalization of finite volume methods by allowing for higher order polynominal approximations inside the elements. However, it is important to note that if we use e.g. a second order space, `"D2"`, as depicted on the right side of Fig. 9.3, it is necessary to increase the penalty parameter (here we used $\alpha = 6$) to obtain stable solution.

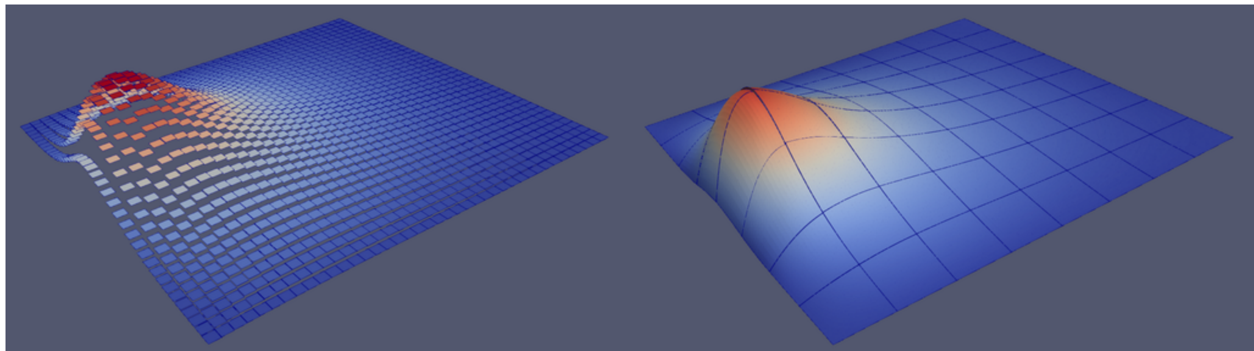

Fig. 9.3: (left) Using the space `"D0"` requires to set $\alpha = 1$ and resembles a finite volume method. (right) for the `"D2"` space it is necessary to increase the penalty parameter.

CHAPTER TEN

ADVANCED STABILITY ANALYSIS

In this section, we cover some advanced topic on linear stability analysis, i.e. stability analysis of moving meshes and stability analysis of axisymmetric base states subject to azimuthal perturbations. Also the linear response the periodic driving is discussed.

10.1 Linear response to periodic driving

Periodic driving of a system will eventually lead to a response that attains the same frequency as the driving, since all transients are damped out (provided there is damping in the system). In general, one has to integrate the system over time and extract the response amplitude and the phase shift from the long-term solution. However, for linear equations or for small driving amplitudes, one can directly solve a linear eigenvalue problem to obtain the complex-valued response. pyoomph comes with a utility for this, which can be applied to arbitrary problems.

10.1.1 Damped harmonic oscillator

Consider a damped harmonic oscillator with driving, i.e.

$$\partial_t^2 y + \delta \partial_t y + \omega_0^2 y = F \cos(\omega t)\,.$$

Due to the damping, all transients will decay and after some time, the system will converge to the response to the driving frequency ω, i.e. to

$$y = A \cos(\omega t + \varphi)$$

Here A is the reponse amplitude and φ is the phase shift with respect to the driving. pyoomph can calculate this response automatically for arbitrary problems, i.e. also complex PDEs on moving meshes. To that end, any potential nonlinearities will be linearized around the stationary solution (here, $y = 0$). First, we define the harmonic oscillator with arbitrary driving and assemble it in a problem. The oscillator must be written as first order system in time, because eventually, an eigenproblem will be solved:

```
from pyoomph import *
from pyoomph.expressions import *
from pyoomph.expressions.units import *
# Load tools for periodic driving response and text file output
from pyoomph.utils.periodic_driving_response import *
from pyoomph.utils.num_text_out import *

# Driven damped harmonic oscillator
class DampedHarmonicOscillatorEquations(ODEEquations):
    def __init__(self,omega0,delta,driving):
```

(continues on next page)

(continued from previous page)

```
        super().__init__()
        self.omega0,self.delta,self.driving=omega0,delta,driving

    def define_fields(self):
        # Must be formulated first order here
        self.define_ode_variable("y",testscale=scale_factor("temporal")**2/scale_
↪factor("y"))
        self.define_ode_variable("ydot",scale=scale_factor("y")/scale_factor("temporal
↪"),testscale=scale_factor("temporal")/scale_factor("y"))

    def define_residuals(self):
        y,y_test=var_and_test("y")
        ydot,ydot_test=var_and_test("ydot")
        self.add_weak(partial_t(y)-ydot,ydot_test)
        self.add_weak(partial_t(ydot)+self.delta*ydot +self.omega0**2*y-self.driving,
↪y_test)


class DampedHarmonicOscillatorProblem(Problem):
    def __init__(self):
        super().__init__()
        self.omega0=1/second
        self.delta=0.1/second
        # Default driving
        self.driving=meter/second**2 *cos(0.3/second*var("time"))

    def define_problem(self):
        self.set_scaling(y=2*meter,temporal=1*second)
        eqs=DampedHarmonicOscillatorEquations(self.omega0,self.delta,self.driving)
        eqs+=ODEFileOutput()
        self+=eqs@"oscillator"
```

The trivial way of getting the response is to just integrate over sufficient long time and extract the response A and the angle φ from the output, i.e.

```
with DampedHarmonicOscillatorProblem() as problem:
     # Trivial, but long way: integrate in time, extract response manually from the␣
↪output
     problem.run(100*second,outstep=0.1*second)
```

However, with the periodic driving response tool, you can scan the linear response quickly:

```
# Quick way of scanning
# Create the PeriodicDrivingResponse before the problem is initialized
pdr=PeriodicDrivingResponse(problem)

F=1*meter/second**2 # Driving amplitude, does not really matter, will cancel out in␣
↪the normalized response
problem.driving=F*pdr.get_driving_mode() # means F*cos(omega*t)

# solve for a stationary state
problem.solve()

# Get the equation index to y
dofindices,dofnames=problem.get_dof_description()
yindex=numpy.argwhere(dofindices==dofnames.index("oscillator/y"))[0,0]
```

(continues on next page)

(continued from previous page)

```
# Factor to absorb the dimensions. We want to have response amplitude divided by␣
↪driving at the end
response_dim_factor=F/meter

# Scan the frequency and write output
outfile=NumericalTextOutputFile(problem.get_output_directory("response.txt"))
outfile.header("omega[1/s]","(A/F)_num[m/(m/s^2)]","phi_num","(A/F)_ana[m/(m/s^2)]")

omegas=numpy.linspace(0.01,3,300)
for response in pdr.iterate_over_driving_frequencies(omegas=omegas,unit=1/second):
    response_ampl,phi=pdr.split_response_amplitude_and_phase() # nondimensional␣
↪response amplitude and angle
    omega=pdr.get_driving_omega() # current omega
    # redimensionalize the response amplitude and divide by the driving, afterwards␣
↪nondimensionalize
    A_num=response_ampl[yindex]*problem.get_scaling("y")/F*response_dim_factor
    # Analytic solution
    A_analytic=1/square_root((problem.omega0**2-omega**2)**2+(problem.
↪delta*omega)**2)*response_dim_factor
    phi_analytic=atan2(-problem.delta*omega,problem.omega0**2-omega**2)
    # outpuf
    outfile.add_row(omega*second,A_num,phi[yindex],A_analytic,phi_analytic)
```

Before the problem is initialized, we must create a `PeriodicDrivingResponse` object and attach it to the problem. This one will introduce a nondimensional, undamped harmonic oscillator with a variable angular frequency ω, i.e. $\partial_t z + \omega^2 z = 0$ to the problem. The method `get_driving_mode()` gives back z, i.e. internally, we use this auxiliary harmonic oscillator to impose the driving to the harmonic oscillator equation for y. To obtain the response, we first must find the equation number corresponding to y, which can be done by describing the degrees of freedom of the problem with the `get_dof_description()` and finding the correct index in the degrees of freedom. By `iterate_over_driving_frequencies()`, we can scan a full range of driving frequencies ω in a loop. The `response` is a complex eigenvector, but it can be split into amplitude and phase by `split_response_amplitude_and_phase()`. By extracting the right component corresponding to y, we get the amplitude and phase directly, correctly account for any physical dimensions and compare it with the analytical solution in the output. The result is plotted in Fig. 10.1.

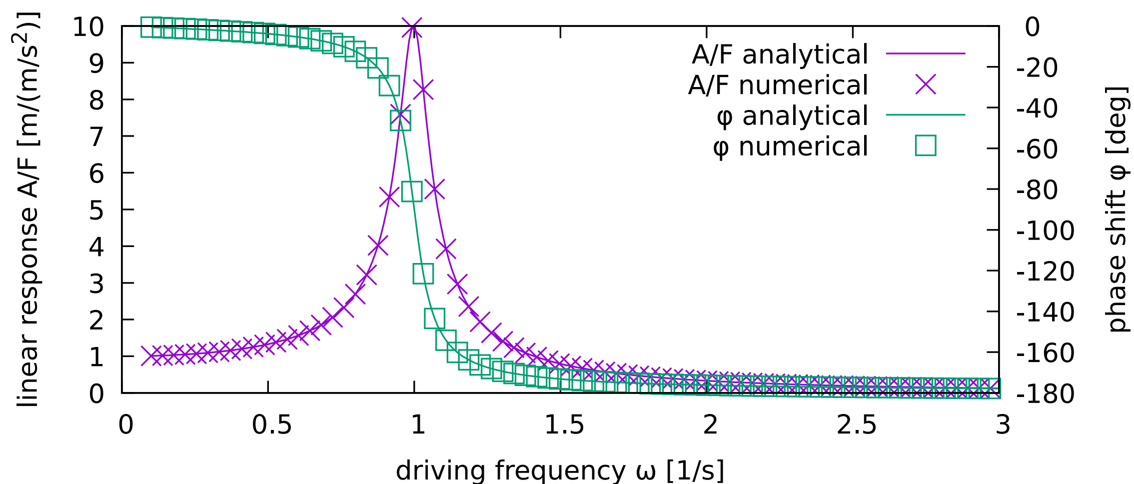

Fig. 10.1: Numerical linear response to a periodic driving with $F\cos(\omega t)$ of a harmonic oscillator with $\omega_0 = 1$ and damping $\delta = 0.1$. The analytical result is also plotted.

10.1.2 Drums getting exited by a guitar

A band has a rest during rehearsal. As usual, the guitar guy cannot resist to play. By the incident sound wave, the drums are put into motion. We solve the drum exitation height h by a driven damped wave equation on a circular membrane, where we assume axisymmetric modes only, i.e.

$$\partial_t^2 h + \delta \partial_t h = c^2 \nabla^2 h + F \cos \omega t$$

where $\omega = 2\pi f$ is the frequency of the guitar sound incidenting as planar wave with a forcing amplitude of F. c depends on the drum and the surrounding gas and δ is a damping coefficient. We demand that $h(r = R) = 0$ at the radius R of the drum.

As usual, we start by the drum equation in a coordinate system indepedent weak formulation:

```
from pyoomph import *
from pyoomph.expressions import *
from pyoomph.expressions.units import *
# Load tools for periodic driving response and text file output
from pyoomph.utils.periodic_driving_response import *
from pyoomph.utils.num_text_out import *

# Driven damped wave equation
class DrumEquation(Equations):
    def __init__(self,c,damping,driving):
        super().__init__()
        self.c,self.damping,self.driving=c,damping,driving

    def define_fields(self):
        # Must be formulated first order here
        self.define_scalar_field("h","C2",testscale=scale_factor("temporal")**2/scale_
↪factor("h"))
        self.define_scalar_field("hdot","C2",scale=scale_factor("h")/scale_factor(
↪"temporal"),testscale=scale_factor("temporal")/scale_factor("h"))

    def define_residuals(self):
        h,h_test=var_and_test("h")
        hdot,hdot_test=var_and_test("hdot")
        self.add_weak(partial_t(h)-hdot,hdot_test)
        self.add_weak(partial_t(hdot)+self.damping*hdot-self.driving,h_test)
        self.add_weak(self.c**2*grad(h),grad(h_test))
```

For the problem, we just define it on a line mesh in axisymmetric coordinates, which are indeed axisymmetric polar coordinates. Thereby, only axisymmetric modes are allowed. Alternatively, one could use a 2d circular Cartesian domain to also allow for nonaxisymmetric modes.

```
class DrumProblem(Problem):
    def __init__(self):
        super().__init__()
        self.c=60*meter/second
        self.damping=50/second
        self.R_drum=30*centi*meter
        self.driving=meter/second**2 *cos(2*pi*440*hertz*var("time"))

    def define_problem(self):
        self.set_coordinate_system("axisymmetric")
        self+=LineMesh(N=100,size=self.R_drum,name="drum",left_name="center",right_
↪name="rim")
        self.set_scaling(h=1*centi*meter,temporal=0.001*second,spatial=self.R_drum)
```

(continues on next page)

(continued from previous page)

```
        eqs=DrumEquation(self.c,self.damping,self.driving)
        eqs+=DirichletBC(h=0)@"rim"+AxisymmetryBC()@"center"
        eqs+=TextFileOutput()
        self+=eqs@"drum"
```

The general procedure is the same as before for a simple harmonic oscillator. However, here, we project the response on different Bessel modes:

```
with DrumProblem() as problem:
        # Create the PeriodicDrivingResponse before the problem is initialized
        pdr=PeriodicDrivingResponse(problem)

        F=10*meter/second**2 # Driving amplitude, does not really matter, will cancel␣
↪out in the normalized response
        problem.driving=F*pdr.get_driving_mode() # means F*cos(omega*t)

        # solve for a stationary state
        problem.solve()

        # Scan the frequency and write output
        numbessel=10 # Number of Bessel modes to project
        bessel_roots=scipy.special.jn_zeros(0,numbessel)

        outfile=NumericalTextOutputFile(problem.get_output_directory("response.txt"))
        reson_freqs=[float(root*problem.c/problem.R_drum/hertz/(2*pi)) for root in␣
↪bessel_roots]
        print("RESONANT UNDAMPED FREQS",reson_freqs)
        outfile.header("freq[Hz]",*["mode_"+str(i)+"[mm/(m/s^2)](f={:.4f})".
↪format(reson_freqs[i]) for i in range(numbessel)])

        # Output the resonant undamped frequencies
        freqs=numpy.linspace(1,1000,1000) # Use frequencies f instead of omega
        for response in pdr.iterate_over_driving_frequencies(freqs=freqs,unit=hertz):
            # Get the response as nondimensional data. The response is stored as␣
↪eigenvector, so we split it in real and imaginary part
            nd_resp_real=problem.get_cached_mesh_data("drum",eigenmode="real",
↪eigenvector=0,nondimensional=True)
            nd_resp_imag=problem.get_cached_mesh_data("drum",eigenmode="imag",
↪eigenvector=0,nondimensional=True)
            # Add interpolators to perform the Bessel projection
            interr=scipy.interpolate.UnivariateSpline(nd_resp_real.get_data(
↪"coordinate_x"),nd_resp_real.get_data("h"),k=3,s=0)
            interi=scipy.interpolate.UnivariateSpline(nd_resp_imag.get_data(
↪"coordinate_x"),nd_resp_imag.get_data("h"),k=3,s=0)
            # Calculate the Bessel decomposition of the response
            bessel_data=[]
            for i in range(numbessel):
                bess_proj=lambda r : r*scipy.special.j0(bessel_
↪roots[i]*r)*(interr(r)+1j*interi(r))
                numer=scipy.integrate.quad(bess_proj,0,1,complex_func=True)[0] #␣
↪Integrate BBessel projection
                denom=1/2*scipy.special.j1(bessel_roots[i])**2 # Denominator of the␣
↪Fourier-Bessel transform
                nd_response_ampl=numpy.absolute(numer/denom)
                dim_response_ampl_by_F=(nd_response_ampl*problem.get_scaling("h")/
↪(milli*meter))/(F/(meter/second**2))
                bessel_data.append(dim_response_ampl_by_F)
```

(continues on next page)

(continued from previous page)

```
        # add a row to the output
        outfile.add_row(pdr.get_driving_frequency()/hertz,*bessel_data)
```

To obtain the full response data, we can access the eigenvector of the problem. Here, the response is stored in the `eigenvector=0`. The nondimensional eigenfunction is stored as real and imaginary contribution, which can be accessed by `get_cached_mesh_data()` for each mesh. Then, cubic interpolators are generated and in the loop, the individual Bessel modes are projected. The result is plotted in Fig. 10.2.

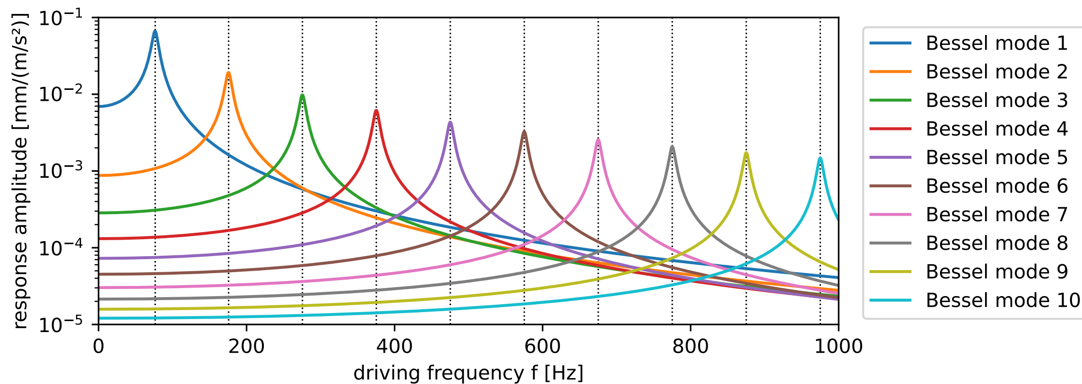

Fig. 10.2: Response of the drum separated into different Bessel modes that fulfill $h(r = R) = 0$. The dashed lines indicate the analytical eigenfrequencies of the undamped wave equation.

10.2 Stability analysis involving the shape, i.e. on a moving mesh

Performing linear stability analysis on a moving mesh is a cumbersome task, since the mass and Jacobian matrices of the generalized eigenproblem will contain plenty of terms stemming from the mesh motion, i.e. the feedback of a change in a nodal position on the discretized equations. Pyoomph can handle this automatically by its full symbolical differentiation. This is discussed in detail in our article [16].

10.2.1 Droplet detaching by gravity

Consider a droplet hanging at the bottom of a horizontal wall, where the contact line is pinned at some fixed radius R. What is the maximum volume V_c the droplet can assume before it pinches off due to gravity? Obviously, if the volume is less, $V < V_\mathrm{c}$, the droplet will assume some stationary state which can be calculated by the Young-Laplace equation. If the volume exceeds $V > V_\mathrm{c}$, this stationary state of a hanging droplet will either lose its stability or this solution will even cease to exist (fold bifurcation). At $V = V_\mathrm{c}$ an eigenvalue of the stationary solution must cross 0, at least the real part. Since the stationary solution has vanishing velocity and also there is no instrinsic time scale, the critical volume V_c will be independent on the Reynolds number, but of course the balance of gravity and surface tension will enter, i.e. the Bond number

$$\mathrm{Bo} = \frac{\rho g R^2}{\sigma}$$

Even if inertia does not enter for the critical volume $V_\mathrm{c}(\mathrm{Bo})$, we require the time derivative of the velocity to appear in the mass matrix for the eigenvalue determination. And, of course, the dynamical behavior of the pinch-off is strongly dependent on the liquid properties, expressed e.g. in the Reynolds, Ohnesorge or capillary number, but this is irrelevant for the determination of $V_\mathrm{c}(\mathrm{Bo})$.

For more details please refer to [16].

We start by a hemispherical droplet with $R = 1$ at $\mathrm{Bo} = 0$. To that end, we create the following mesh:

```
from pyoomph import *
from pyoomph.equations.navier_stokes import * # Navier-Stokes for the flow
from pyoomph.equations.ALE import * # Moving mesh equations
from pyoomph.meshes.remesher import Remesher2d # Remeshing
from pyoomph.utils.num_text_out import NumericalTextOutputFile # Tool to write a file␣
↪with numbers

# Make a mesh of a hanging hemispherical droplet with radius 1
class HangingDropletMesh(GmshTemplate):
    def define_geometry(self):
        self.default_resolution=0.05
        self.mesh_mode="tris"
        self.create_lines((0,-1),"axis",(0,0),"wall",(1,0))
        self.circle_arc((0,-1),(1,0),center=(0,0),name="interface")
        self.plane_surface("axis","wall","interface",name="droplet")
        self.remesher=Remesher2d(self) # attach a remesher
```

It is rather trivial, just three points, an axis of symmetry, a wall at the top and a circle segment as interface. We also add a remesher, since remeshing is required due to large mesh deformations.

The problem class itself starts also quite simple, we just add the two parameters Bo and the volume V with good starting values, i.e. without gravity and with the volume of the hemispherical initial mesh (in axisymmetry), so that the parameters agree with the initial stationary solution. We then add `NavierStokesEquations` with the correct `bulkforce`. For the mass density and dynamic viscosity, any numerical value can be taken (> 0). It does not change the result since the stability is determined by a balance of surface tension and gravity alone. Also the boundary conditions are added, i.e. axisymmetry at the axis, no-slip and fixed coordinates at the wall, as well as a free surface (with unity surface tension due to non-dimensionalization with the Bond number). Also, the detection of required remeshing is added with default parameters:

```
class HangingDropletProblem(Problem):
    def __init__(self):
        super().__init__()
        # Initially no gravity
        self.Bo=self.define_global_parameter(Bo=0)
        # Initial volume of a hemispherical droplet with radius R=1
        self.V=self.define_global_parameter(V=2*pi/3)

    def define_problem(self):
        self.set_coordinate_system("axisymmetric") # axisymmetry
        self+=HangingDropletMesh() # Add the mesh

        # Bulk equations: Navier-Stokes + Moving Mesh
        eqs=MeshFileOutput() # Paraview output
        # Mass density can be set arbitrarly with the same results. But must be >0 to␣
↪get a du/dt -> mass matrix
        eqs+=NavierStokesEquations(dynamic_viscosity=1,mass_density=1,bulkforce=self.
↪Bo*vector(0,-1))
        eqs+=LaplaceSmoothedMesh()

        # Boundary conditions:
        eqs+=(DirichletBC(mesh_y=0,mesh_x=True)+NoSlipBC())@"wall" # static no-slip␣
↪wall
        eqs+=AxisymmetryBC()@"axis"
        eqs+=NavierStokesFreeSurface(surface_tension=1)@"interface" # free surface
```

(continues on next page)

(continued from previous page)

```
        # Remeshing
        eqs+=RemeshWhen(RemeshingOptions())
```

As already pointed out in Section 6.6.4, stationary solutions on a moving mesh are problematic, since there are plenty of stationary solutions that do not necessary have the desired volume V. Again, we have to enforce the desired volume V by adding a Lagrange multiplier which then acts on the pressure:

```
# For stationary solutions, we must enforce the droplet volume to be the given one
# Add a global Lagrange multiplier -> something like the gas pressure that acts to␣
↪ensure the volume
# we want to solve p_gas by integral_droplet(1*dx)-Paramter_V=0, so we subtract the␣
↪symbolic volume parameter
self+=GlobalLagrangeMultiplier(p_gas=-self.V,only_for_stationary_solve=True)@"globals"
p_gas=var("p_gas",domain="globals") # bind the gas pressure
# And rewrite 1*dx=div(x)/3*dx=1/3*x*n*dS, add it to the p_gas equation to complete it
eqs+=WeakContribution(1/3*var("mesh"),var("normal")*testfunction(p_gas))@"interface"
# p_gas must now act somewhere. We just let it act on the contact line, and only if␣
↪solved stationary
# The kinematic BC at the rest of the interface will adjust accordingly
eqs+=EnforcedBC(pressure=p_gas,only_for_stationary_solve=True)@"interface/wall"

# Add the equation system to the droplet
self+=eqs@"droplet"
```

Note how we use the trick mentioned in the info box in Section 6.6.4 here to calculate the actual volume $\int 1\mathrm{d}V$ by a surface integral. The other surfaces (axis of symmetry and wall) are not required, since $\vec{x}\cdot\vec{n}=0$. With the Lagrange multiplier, the pressure at the contact line is enforced to the required pressure for the given volume. All other contributions, i.e. the values of the Lagrange multiplier field that enforces the kinematic boundary condition, will adjust accordingly. Also note that we pass the keyword-argument `only_for_stationary_solve=True` to the `GlobalLagrangeMultiplier`. With this, both the global Lagrange multiplier and the enforced contact line pressure, will be disabled during any transient solve, where volume enforcing is not required. Thereby, the problem can be used for both stationary `solve()` and transient `run()` statements.

The run code to determine the curve is quite simple. We start with:

```
if __name__=="__main__":
    with HangingDropletProblem() as problem:
        # Calculate the Hessian symbolically/analytically -> faster and more accurate␣
↪than finite differences
        problem.setup_for_stability_analysis()
        problem.do_call_remeshing_when_necessary=False # Don't auto-remesh. Can be␣
↪problematic during continuation

        # Increase Bond number towards the bifurcation, do remeshing if necessary␣
↪during that
        problem.go_to_param(Bo=2.8, call_after_step=lambda a : problem.remesh_handler_
↪during_continuation())
        problem.force_remesh() # Force a new mesh
        problem.solve() # and resolve (mainly for the hydrostatic pressure and small␣
↪shape adjustments of the new mesh)

        # Solve the eigenvalues to get a good guess for the critical eigenfunction
        problem.solve_eigenproblem(5)
        # Looking for a fold bifurcation when the droplet pinches off
        problem.activate_bifurcation_tracking("Bo","fold")
```

(continues on next page)

(continued from previous page)

```
        # and solve for it. Bo will be adjusted to the critical Bond number at the␣
↪initial volume
        problem.solve()
```

First the problem is created and we demand the code generation for an analytical Hessian tensor. The Hessian is required for the bifurcation tracking of the fold bifurcation later on. If it is set to be analytical, the C code to be generated is considerably larger and more complicated, i.e. take longer to compile. Therefore, by default, Hessians are calculated by finite differences. However, an analytical Hessian is more accurate and its assembly is also considerably faster. As optional argument to `setup_for_stability_analysis()`, we can pass `use_hessian_symmetry=True` (default) if we want to use the symmetry of the Hessian tensor $H_{ijk} = H_{ikj}$ according to Schwarz's theorem. This can speed up the Hessian calculation even more.

We then deactivate the automatic remeshing, since it can give problems in continuation and bifurcation tracking. Instead, we will manually check whether remeshing is required after each continuation step, e.g. by invoking `remesh_handler_during_continuation()` passed as kwarg `call_after_step` in the `go_to_param()`. There, we crank up the Bond number, i.e. the gravity. The droplet will deform and after each step in increasing the Bond number, remeshing is invoked when necessary. Without this procedure, the mesh would deform too much. We have to wrap it in a `lambda`, since the `call_after_step` gets the current parameter (here, the Bond number) passed as argument, which we have to discard.

Then bifurcation tracking is activated. But in order to work well, we must be rather close to the bifurcation and calculate the eigenvalues and -vectors in beforehand, since a good guess for the critical eigenvector is required in order for the bifurcation tracking to converge. The `solve()` command will now solve for the shape of the droplet, but also for the critical Bond number Bo_c at the initial volume.

Afterwards, we can continue in the volume V to create the critical curve $\mathrm{Bo}_c(V)$:

```
# Create an output file for the curve Bo_c(V)
critical_curve_out=NumericalTextOutputFile(problem.get_output_directory("critical_
↪curve.txt"))
critical_curve_out.header("V","Bo_c") # header line
critical_curve_out.add_row(problem.V.value,problem.Bo.value) # V and Bo_c -> file
problem.output_at_increased_time() # also Paraview output

# Increase the volume, still tracking for the critical Bond number:
dV=0.1*problem.V.value
while problem.V.value<50:
    dV=problem.arclength_continuation("V",dV,max_ds=0.1*problem.V.value)
    problem.remesh_handler_during_continuation()
    critical_curve_out.add_row(problem.V.value,problem.Bo.value)
    problem.output_at_increased_time()
```

We write a text output file containing V and Bo in the output directory. Arclength continuation in V is made in steps that may not exceed 10% of the current volume. Since the bifurcation tracking is still active, Bo will be adjusted automatically. Since the droplet inflates quite a lot, remeshing is of course occasionally required - again with the `remesh_handler_during_continuation()`, which remeshes whenever necessary, but does not break the bifurcation tracking and the continuation in V.

It is easy to perform the continuation also in the direction of smaller volumes. In total, one then gets the results depicted in Fig. 10.3.

Of course, the choice to nondimensionalize the system by the contact line radius R, not by e.g. the volume, is questionable. But once the curve is obtained, it is trivial to rescale the curve in a more natural way. Likewise, it is trivial to replace the pinned contact line e.g. by a freely moving contact line or testing the influence of e.g. insoluble surfactants on this problem.

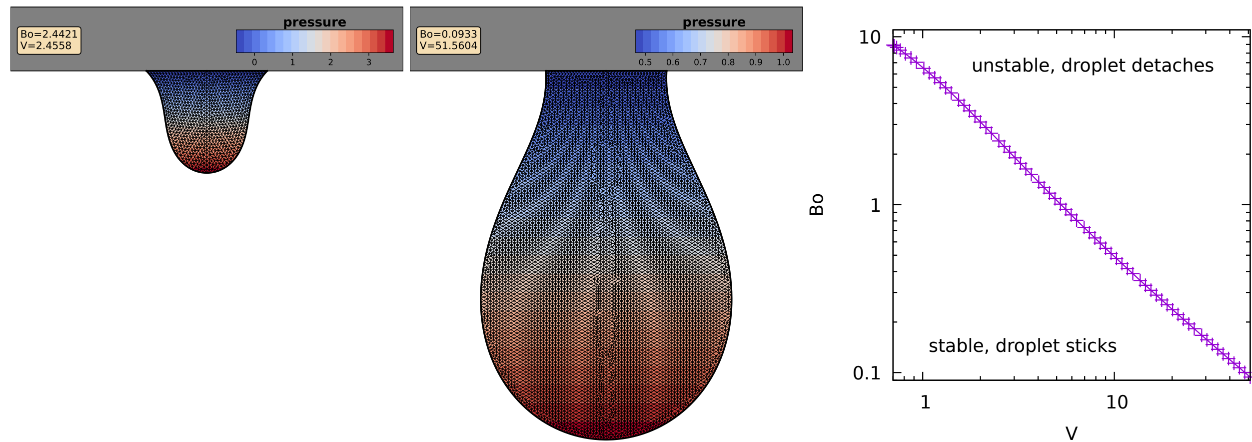

Fig. 10.3: Hanging droplets directly at the threshold to pinch-off due to gravity. Also the curve $\mathrm{Bo}(V)$ is shown.

10.3 Azimuthal stability analysis

This section covers a powerful tool to investigate full three-dimensional stability analysis by considering just a two-dimensional axisymmetric mesh. The requirement is hence, that the base solution, which should be tested for stability, is perfectly axisymmetric, but then undergoes a symmetry breaking with respect to azimuthal modes. The general idea the folllowing: We have some stationary solution of our unknown vector $\vec{U}^{(0)}$, which is axisymmetric, i.e. $\vec{U}^{(0)}=\vec{U}^{(0)}(r,z)$. Components of this solution vector could be a velocity and pressure field, for instance. We want to investigate what happens, if this solution is perturbed by some azimuthal mode m with tiny amplitude, i.e. some mode $\epsilon\vec{U}^{(m)}\exp(im\phi+\lambda_m t)$. Here, the amplitude ϵ should be sufficiently small that only linear terms enter. $\vec{U}^{(m)}(r,z)$ is the pertubation vector of this mode (i.e. an eigenvector) and λ_m the corresponding eigenvalue. When plugging into the system of equations and truncating at linear order in ϵ, the linear dynamics of this pertubation is given by the generalized eigenvalue problem:

$$\lambda_m\mathbf{M}^{(m)}\vec{U}^{(m)}=\mathbf{J}^{(m)}(\vec{U}^{(0)})\vec{U}^{(m)}$$

Even if the original system is entirely real, the mass matrix $\mathbf{M}^{(m)}$ and the Jacobian $\mathbf{J}^{(m)}$ are usually complex-valued. The reason is that any even spatial derivatives with respect to the azimuthal coordinate, i.e. ∂_ϕ, will produce an im.

When activated, pyoomph can derive this particular eigensystem automatically. To that end, all unknown scalar (s) and vectorial ($\vec{v}$) fields will be expanded like

$$\begin{aligned} s(r,z,\phi,t)&=s^{(0)}(r,z)+\epsilon s^{(m)}(r,z)\exp(im\phi+\lambda_m t)\\ \vec{v}(r,z,\phi,t)&=\vec{v}^{(0)}(r,z)+\epsilon\vec{v}^{(m)}(r,z)\exp(im\phi+\lambda_m t)\end{aligned}$$

before actually being plugged into the equations. Furthermore, all vectors $\vec{v}$ will automatically get a ϕ-component when defined by the `define_vector_field()`, which is expanded the same way. All coordinate-system-independent spatial derivatives, i.e. `grad()` and `div()`, will consider also the derivative with respect to ϕ and include the additional vector components v_ϕ. The test functions will be augmented by a $\exp(-im\phi)$ and also a test functions for the v_ϕ-component is considered in all vector fields. Global, i.e. ODE quantities, will not be expanded that way.

After plugging the augmented fields and test functions into the system of weak formulations and carrying out the spatial derivatives, we obtain the residual vector $\vec{R}$ including the augmented mode expansions. The basic residual vector $\vec{R}^{(0)}$, i.e. the residual for the axisymmetric base state, is obtained by setting $\epsilon=0$ and $m=0$. This corresponds to the conventional residual and is real-valued. It is in fact the same residual as if just a normal axisymmetric coordinate system is used. Besides $\vec{R}^{(0)}$, also $\mathrm{Re}(\vec{R}^{(m)})$ and $\mathrm{Im}(\vec{R}^{(m)})$ are determined. We just take the first order ϵ term here, i.e. $\vec{R}^{(m)}=\partial_\epsilon\vec{R}|_{\epsilon=0}$. Thereby, azimuthal modes can appear in linear order only, but any non-linear coupling with $\vec{U}^{(0)}$ contributions appear. The linear contributions proportional $\exp(im\phi)$ will cancel out with the corresponding $\exp(-im\phi)$

of the test functions. Thereby, the complex-valued azimuthal residual $\vec{R}^{(m)}$ emerges, which does not depend explictly of ϕ, but has the mode number m in it, provided that spatial derivatives have produced these terms.

As usual, the base Jacobian $\mathbf{J}^{(0)}$, mass matrix $\mathbf{M}^{(0)}$ and, if desired, the Hessian $\mathbf{H}^{(0)}$ by deriving $\vec{R}^{(0)}$ with respect to $\vec{U}^{(0)}$. The azimuthal Jacobian and mass matrix, $\mathbf{J}^{(m)}$ and $\mathbf{M}^{(m)}$, is obtained by deriving $\vec{R}^{(m)}$ with respect to $\vec{U}^{(m)}$. If the Hessian $\mathbf{H}^{(m)}$ is desired, it is calculated by deriving $\mathbf{J}^{(m)}$ with respect to $\vec{U}^{(0)}$, not $\vec{U}^{(m)}$, as one might naively expect. The reason is that $\mathbf{J}^{(m)}$ is indepedent on $\vec{U}^{(m)}$, since $\vec{U}^{(m)}$ enters $\vec{R}^{(m)}$ only in linear order. For bifurcation tracking, however, it is required to get the variations of $\mathbf{J}^{(m)}$ with respect to the base mode $\vec{U}^{(0)}$. Therefore, the Hessian is derived that way.

Lastly, the boundary conditions at the axis of symmetry are of fundamental importance. For conventional axisymmetry, one demands that $\partial_r s^{(0)} = \partial_r v_z^{(0)} = 0$ and $v_r^{(0)} = v_\phi^{(0)} = 0$, since one otherwise get a singularity at the axis. However, for azimuthal modes, this can be different. For $m = 1$, we get $s^{(m)} = v_z^{(m)} = 0$, since otherwise these values are not well defined at $r = 0$ when varying ϕ. However, $v_r^{(m)}$ and $v_\phi^{(m)}$ may be non-zero for $m = 1$, since these basis vectors of these components exactly rotate with the ϕ in exactly the same manner as the $m = 1$-mode. There, however, $\partial_r v_r^{(m)} = \partial_r v_\phi^{(m)} = 0$ is required. For all other m, all vector components and scalars have to vanish, i.e. $v_r^{(m)} = v_\phi^{(m)} = v_z^{(m)} = s^{(m)} = 0$.

These m-dependent boundary conditions are automatically taken care of in the `AxisymmetryBC` object. It will impose exactly the correct boundary conditions for all vector and scalar fields.

For more details on this, we refer to our article [16].

10.3.1 Rayleigh-Benard convection in a cylindrical container

To illustrate the quite longish preface of this section by an example, let us consider a Rayleigh-Benard setting in a cylinder, which is heated from below and cooled from above, with no-slip boundary conditions at all walls. At some specific temperature difference (or better: Rayleigh number), convection will set in. We want to use the azimuthal stability framework to obtain the critical Rayleigh number Ra as function of the aspect ratio $\Gamma = R/H$ of the cylinder. We have to do it individually for each mode m.

We start by the problem definition, analogous to the same case discussed in [16]:

```
from pyoomph import *
from pyoomph.equations.navier_stokes import * # Navier-Stokes for the flow
from pyoomph.equations.advection_diffusion import * # Advection-diffusion for the␣
↪temperature
from pyoomph.utils.num_text_out import * # Output for the critical Rayleigh as␣
↪function of the aspect ratio


class RBConvectionProblem(Problem):
    def __init__(self):
        super().__init__()
        # Aspect ratio, Rayleigh and Prandtl number with defaults
        self.Gamma = self.define_global_parameter(Gamma=1)
        self.Ra = self.define_global_parameter(Ra=1)
        self.Pr =self.define_global_parameter(Pr=1)

    def define_problem(self):
        # Axisymmetric coordinate system
        self.set_coordinate_system(axisymmetric)
        # Scale radial coordinate with aspect ratio parameter
        self.set_scaling(coordinate_x=self.Gamma)
        # Axisymmetric cross-section as mesh.
        # We use R=1 and H=1, but due to the radial scaling, we can modify the␣
```

(continues on next page)

(continued from previous page)

```
↪effective radius
        self+=RectangularQuadMesh(size=[1, 1], N=20)
```

By setting the spatial scale of `"coordinate_x"` in the axisymmetric coordinate system, we effectively scale the radial coordinate $r \to \Gamma r$, so that we can modify the cylinder radius without changing the mesh at all. Of course, one should not go to extreme aspect ratios ($\Gamma \ll 1$ or $\Gamma \gg 1$) by this, since the solution won't be captured well then.

The rest starts trivial, just adding Navier-Stokes with body force given by the nondimensional temperature, which is solved by a corresponding advection-diffusion equation:

```
RaPr=self.Ra*self.Pr # Shortcut for Ra*Pr
# Equations: Navier-Stokes. We scale the pressure also with RaPr,
# so that the hydrostatic pressure due to the bulk-force is independent on the value␣
↪of Ra*Pr
NS=NavierStokesEquations(mass_density=1, dynamic_viscosity=self.Pr,bulkforce=RaPr*var(
↪"T")*vector(0, 1), pressure_factor=RaPr)
# Since u*n is set at all walls, we have a nullspace in the pressure
# This offset is fixed by an integral constraint <p>=0
# One could also set it via a DirichletBC(pressure=0) at e.g. a single corner,
# but this yields problems in the azimuthal stability analysis then
# The pressure integral constraint is automatically deactivated when m!=0, since <p>=0
# holds automatically when p = p^(m)*exp(I*m*phi) for m!=0
eqs = NS.with_pressure_integral_constraint(self,integral_value=0,set_zero_on_normal_
↪mode_eigensolve=True)

# And advection-diffusion for temperature
eqs += AdvectionDiffusionEquations(fieldnames="T",diffusivity=1, space="C1")
```

With `pressure_factor` in the `NavierStokesEquations`, we scale the pressure with the product of the Rayleigh and Prandtl number. This product is entering the bulk force, i.e. the buoyancy. When scaling the pressure the same way, the stationary pressure field is independent on $\mathrm{Ra}\,\mathrm{Pr}$. Thereby, one can solve the stationary conductive solution (mainly pressure and temperature field) for any Rayleigh number and change the Rayleigh number afterwards.

Furthermore, we have to fix the null space of the pressure, originating from the fact that only no-slip boundary conditions are used. Usually, we just pin e.g. a single corner to some pressure value. However, this is problematic, since it will also pin the corresponding eigenfunction value to zero there. A typical Dirichlet condition would just remove the pressure value at this corner from the unknowns and hence also from the pressure eigenfunction. Therefore, the volume average of the pressure is enforced to be zero instead. All pressure values remain unpinned and will have a degree of freedom in the eigenvector as well. However, when considering modes $m \neq 0$, the average pressure of the eigenfunction will be automatically zero, since $\exp(im\phi)$ averages to zero. In that case, we must deactivate this constraint to prevent overconstrainment. This is achieved by the keyword argument `set_zero_on_normal_mode_eigensolve` in the pressure null space removal `with_pressure_integral_constraint()`.

The boundary conditions are straightforward:

```
# Boundary conditions
eqs += DirichletBC(T=0)@"bottom"
eqs += DirichletBC(T=-1)@"top"
# The NoSlipBC will actually also set velocity_phi=0 automatically
eqs += NoSlipBC()@["top", "right", "bottom"]
# Here, the magic happens regarding the m-dependent boundary conditions
eqs += AxisymmetryBC()@"left"

# Output
eqs+=MeshFileOutput()
```

(continues on next page)

(continued from previous page)

```
# Add the system to the problem
self+=eqs@"domain"
```

Note that the `NoSlipBC` will also set the ϕ-component of the velocity to zero automatically. Also, note the `AxisymmetryBC`, which will set the correct boundary conditions for the azimuthal stability analysis, as outline before. Also normal output is added, before the equation system is added to the problem. One last thing which has to be done when running the problem is to activate the azimuthal stability analysis. This is done by passing `azimuthal_stability=True` to the `setup_for_stability_analysis()` call.

```
# Activating azimuthal stability: It will perform all necessary adjustments, i.e.
#   -expand fields and test functions with exp(i*m*phi)
#   -consider phi-components in vector fields, i.e. here velocity
#   -incorporate phi-derivatives in grad and div
#   -generate the base residual, Jacobian, mass matrix and Hessian, but also
#    the corresponding versions for the azimuthal mode m!=0
problem.setup_for_stability_analysis(azimuthal_stability=True)
```

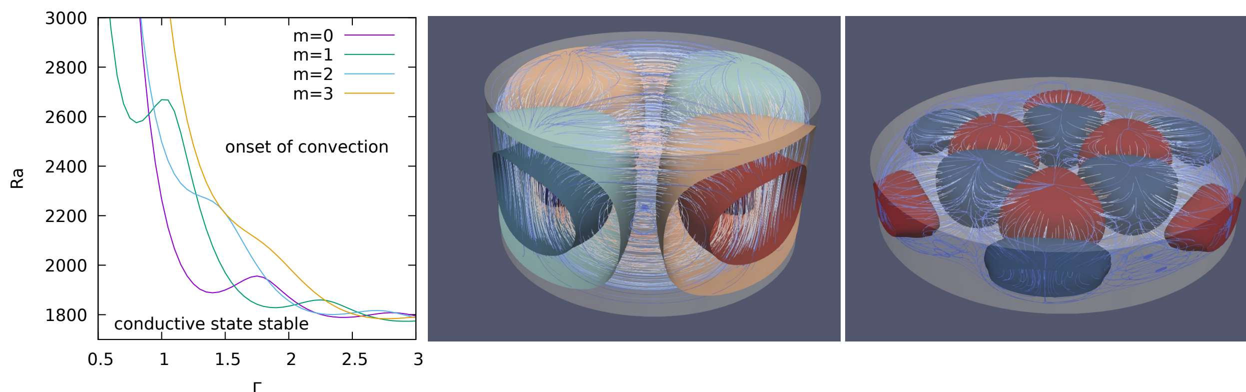

Fig. 10.4: Critical Rayleigh number for the onset of convection as function of the aspect ratio Γ and the critical eigenfunction for aspect ratio $\Gamma = 1$ and azimuthal mode $m = 2$ and $\Gamma = m = 3$, respectively.

10.3.2 Path instability of a rising bubble

One particular powerful feature is the possibility to tackle azimuthal instabilities of rather arbitrary problems defined on moving meshes. Such numerical approaches have been developed only quite recently to investigate e.g. the instability of a rising bubble [4, 29]. Pyoomph does all the cumbersome work of deriving the azimuthal eigenproblem automatically and fully symbolically and generates a corresponding C codes to fill the mass and Jacobian matrices for these eigenproblems. In the following, we will reproduce the results of [4] in pyoomph.

First of all, we start by defining a rectangular mesh with circular hole in the center. This hole later represents the bubble. While we could just use the `GmshTemplate` class to create such a mesh, we prefer to make a structured mesh by manually placing all elements of a coarse mesh, which will be refined during the solution procedure. Read Section 4.3.1 to learn how to create meshes that way. The mesh class itself is skipped here for brevity, but is part of the example code you can dowload below.

As in [4], we neglect the viscosity and the mass density inside the bubble and nondimensionalize the equations in terms of a Bond number $Bo = \rho g D^2/\sigma$ and a Galilei number $Ga = \rho\sqrt{gD^3}/\mu$ ($D = 2R$ is the droplet diameter), where we express Galilei number by the Bond number via a Morton number $Mo = g\mu^4/(\rho\sigma^3)$. The Morton number in independent of the bubble size and only depends on the liquid properties and the gravity. In particular, $Mo = 6.2 \times 10^{-7}$ holds for DMS-T05 [4], which we consider here. Keeping the Morton number fixed, we can calculate the corresponding Galilei number from the Bond number by $Ga = (Bo^3/Mo)^{1/4}$. However, the latter expression is problematic if the

Bond number becomes negative. By default, pyoomph's global parameters may attain positive and negative values and therefore, the 4th root will be rather problematic. We therefore inform pyoomph that the Bond number is always a positive parameter. This information will be used in the code generation later on for a good code generation of the 4th root:

```
class RisingBubbleProblem(Problem):
    def __init__(self):
        super().__init__()
        self.R=0.5
        self.Mo=6.2e-7 # Morton number selects the fluid
        self.Bo=self.define_global_parameter(Bo=0.4) # Bond number, effectively␣
↪selects the bubble size
         # This helps a lot in reducing the code size: We calculate Ga from Bo with␣
↪an rational exponent.
         # Since we must separate the real and imaginary part of the azimuthal mode,␣
↪this would generate a lot of code if Bo could be negative, meaning that Ga could␣
↪become complex according to the definition
        self.Bo.restrict_to_positive_values()

        self.L_top=15/4 # Far sizes. These are considerably smaller than in the␣
↪literature
        self.L_bottom=30/4
        self.W=15/4

        self.max_refinement_level=4 # Do not refine more than 4 times (we want to␣
↪have it fast, not perfectly accurate)
```

The nondimensionalized equations read

$$\begin{aligned}
\partial_t\vec{u}+\nabla\vec{u}\cdot\vec{u}&=-\nabla p+\frac{1}{Ga}\nabla\cdot\left[\left(\nabla\vec{u}+\nabla\vec{u}^{t}\right)\right]-\dot{U}e_y\\
\nabla\cdot\vec{u}&=0\\
\left(\vec{u}-\dot{X}\right)\cdot\vec{n}&=0\\
\left(-p\mathbf{1}+\frac{1}{Ga}\left(\nabla\vec{u}+\nabla\vec{u}^{t}\right)\right)\cdot\vec{n}&=\left(\frac{1}{Bo}\kappa+y+P\right)\vec{n}
\end{aligned}\tag{10.1}$$

Here, U is the velocity of the bubble, which is determined by enforcing that the center of mass of the bubble does not move. So we transform into the coordinate system comoving with the bubble as in Section 4.4.7. Also, we have absorbed the hydrostatic pressure in the pressure field p, which leads to the additional axial coordinate y in the rhs of the pressure acting on the surface. The unknown P is the bubble pressure which is determined by enforcing a constant nondimensional volume $4/3\pi R^3$ of the bubble (with $R=0.5$, i.e. a nondimensional diameter of $D=1$). For the volume constraint, we use the divergence theorem trick described in the box in Section 6.6.4. In a similar fashion, the center of mass is calculated by an integration over the surface:

```
def define_problem(self):
     Ga=(self.Bo**3/self.Mo)**rational_num(1,4) # Galilei number

     self.set_coordinate_system("axisymmetric")

     self+=StructuredBubbleMesh()  # Add the mesh

     # Assemble the equations: First, output with eigenfunction included
     eqs=MeshFileOutput(operator=MeshDataCombineWithEigenfunction(0))

     # Unknown bubble velocity and bubble pressure (global degrees)
     U,Utest=self.add_global_dof("U")
     P,Ptest=self.add_global_dof("P",equation_contribution=-4/3*pi*self.R**3,initial_
```

(continues on next page)

(continued from previous page)

```
↪condition=8/self.Bo)

    # Bulk equations: Navier-Stokes in the co-moving frame with inertia correction␣
↪of a potentially accelerating frame
    eqs+=NavierStokesEquations(dynamic_viscosity=1/Ga ,mass_density=1,
↪gravity=vector(0,-1)*partial_t(U),mode="CR")

    # Free surface with the additional pressure of the bubble and the absorbed␣
↪hydrostatic pressure
    eqs+=NavierStokesFreeSurface(surface_tension=1/self.Bo,additional_normal_
↪traction=P+var("coordinate_y"))@"interface"

    # Constraints fixing the bubble velocity U and the bubble pressure P
    eqs+=WeakContribution(1/2*var("coordinate_y")**2*var("normal_y"),Utest)@
↪"interface"
    eqs+=WeakContribution(-dot(var("coordinate"),var("normal"))/3,Ptest)@"interface"
```

We still have to add moving mesh equations and some missing boundary conditions. The `AxisymmetryBC` ensures again to toggle the m-dependent boundary conditions for the eigenfunction at $r = 0$. It automatically transfers to e.g. the intersection `"interface/axis"`, where we have to modify e.g. the Lagrange multiplier for the kinematic boundary condition.

```
# Boundary conditions
eqs+=AxisymmetryBC()@"axis"
eqs+=DirichletBC(mesh_x=self.W,velocity_x=0,velocity_phi=0)@"side"
eqs+=DirichletBC(mesh_y=-self.L_bottom)@"bottom"
eqs+=DirichletBC(mesh_y=self.L_top,velocity_x=0,velocity_phi=0)@"top"
eqs+=EnforcedDirichlet(velocity_y=-U)@"top" # Adjust the far field velocity

# Add a moving mesh
eqs+=PseudoElasticMesh()
# But pin in further away from the bubble to save degrees of freedom
eqs+=PinWhere(mesh_x=True,mesh_y=True,where=lambda x,y : x**2+y**2>4)

# Refinement strategy: Max level at the interface
eqs+=RefineToLevel()@"interface"
# And also, refine according velocity gradients, both for the base solution and the␣
↪eigenfunction
eqs+=SpatialErrorEstimator(velocity=1)

self+=eqs@"domain"
```

Optionally, we can process all calculated eigenvectors. Here, we make sure that the average of the mesh displacement at the interface has a zero complex angle. This is possible since eingenvectors can have an arbitrary nonzero multiplicative factor. In particular, it can be complex to rotate the eigenvector with respect to real and imaginary parts. The method `process_eigenvectors()` is called whenever eigenvectors are calculated. Here, we just call `rotate_eigenvectors()` to ensure it is rotated the way mentioned above:

```
def process_eigenvectors(self, eigenvectors):
    # This function is called whenever the eigenvectors are calculated.
    # Eigenvectors are arbitrary up to a scalar constant.
    # We can multiply it by such a constant that the average x-displacement of the␣
↪interface mesh has positive real part and zero imaginary part (on average)
    # This is optional, but makes the results more consistent, since the␣
↪multiplicative constant is otherwise arbitrary
    return self.rotate_eigenvectors(eigenvectors,"domain/interface/mesh_x",normalize_
```

(continues on next page)

(continued from previous page)

```
↪amplitude=0.2,normalize_dofs=True)
```

The driver code now mainly sets up the problem. In particular, we have to activate again the azimuthal stability analysis. We need a robust complex eigensolver. For that, you have to install a complex variant of the package SLEPc (see Section 2.4).

We then start at some Bond number, relax to the initial state by some transient steps followed by a stationary solve. Then, we create an output file to write the eigenvalues and scan over the Bond number. We solve the eigenproblem using first an initial guess for the eigenvalue (using the `shift` and `target` kwargs of `solve_eigenproblem()`). After the first step, we just use the previously calculated eigenvalue as guess for the next Bond number. We can adapt the mesh based on the eigenfunction using `refine_eigenfunction()`. It will use the `SpatialErrorEstimator` added to the problem to refine with respect to jumps in velocity gradients across the elements. Thereby, strong changes in the eigenfunction are better captured:

```
with RisingBubbleProblem() as problem:

    # Make sure to get the most optimized code available
    problem.set_c_compiler("system").optimize_for_max_speed()
    # Use SLEPc for the eigenvalue problem, use MUMPS as linear solver, since we have␣
↪constraints.
    # These have a zero diagonal and give problems in the default LU decomposition of␣
↪PETSc
    problem.set_eigensolver("slepc").use_mumps()

    # Setup the problem for azimuthal stability analysis. We don't use the analytic␣
↪Hessian, since we don't do any bifurcation tracking
    # This saves some code generation and compilations time
    problem.setup_for_stability_analysis(azimuthal_stability=True,analytic_
↪hessian=False)

    # Settings
    problem.Mo=6.2e-7 # Morton number selects the fluid
    problem.Bo.value=3 # Start at Bo=3
    BoMax=10 # Maximum Bond number
    dBond=0.25 # Step size in Bond number
    m=1 # Azimuthal mode number
    lambd=-0.1+0.75j # Guess for the eigenvalue

    # Relax to the base state, then solve for the stationary solution
    problem.run(10,startstep=0.1,outstep=False,temporal_error=1)
    problem.solve(max_newton_iterations=20,spatial_adapt=4)

    # Now we can start the eigenanalysis
    outfile=problem.create_text_file_output("m1_instability.txt",header=["Bo",
↪"ReLambda","ImLambda"])

    # Scan the branch
    while problem.Bo.value<BoMax:
        # Solve it with a shift-inverted method close to the guess
        problem.solve_eigenproblem(1,azimuthal_m=m,shift=lambd,target=lambd)
        # Refine the mesh according to the eigenfunction and recalculate the␣
↪eigenproblem
        problem.refine_eigenfunction(use_startvector=True)
        # And update the eigenvalue and the eigenvector guess
        lambd=problem.get_last_eigenvalues()[0] # Update the eigenvalue for the next␣
↪iteration
```

(continues on next page)

(continued from previous page)

```
        # Store it to the text file
        outfile.add_row(problem.Bo,numpy.real(lambd),numpy.imag(lambd))
        # Output the solution with eigenfunction
        problem.output_at_increased_time()
        # And continue in Bo
        problem.go_to_param(Bo=problem.Bo.value+dBond)
```

Eventually, we get the eigenvalues shown below, which agree decently with the data of [4]. We can do the same for other liquids and branches described in [4]. Note that our mesh is quite coarse and small in terms of the far field, so one might have to take a finer mesh (using the `max_refinement_level`) and a larger domain with the properties `L_top`, `L_bottom` and `W` of our problem class. Also note that the plots of the solutions in [4] apparently scales the nondimensional radius, not the diameter to unity. Therefore, the fields have different amplitudes.

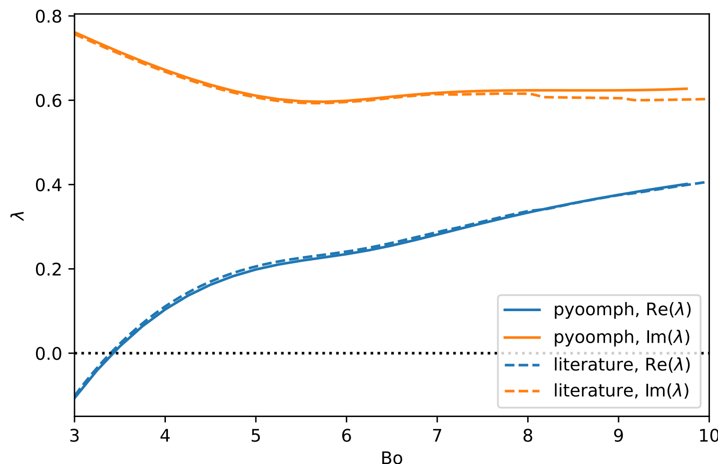

Fig. 10.5: Eigenvalues of the first $m = 1$ instability of a rising bubble with $Mo = 6.2 \times 10^{-7}$ (DMS-T05), agreeing well with the literature data. We thank Javier Sierra-Ausin and Jacques Magnaudet for providing the data of their paper.

We can also generate a movie of the instability. Please refer to Section 11.4 for a tutorial on this.

10.4 Cartesian normal mode stability analysis

Similar to the azimuthal stability analysis, it is possible to expand a N-dimensional Cartesian problem to an $N + 1$-dimensional problem for stability analysis. Instead of the azimuthal angle ϕ, now a wavenumber k in the $N+1$-th direction is introduced. Again, we have a stationary solution of our unknown vector $\vec{U}^{(0)}(x_1, \ldots, x_N)$, which is independent of x_{N+1}. We then investigate perturbations like $\epsilon\vec{U}^{(k)}\exp(ikx_{N+1} + \lambda_k t)$.

Analogous to the azimuthal component in azimuthal stability analysis, all vector fields $\vec{v}$ will get an additional component, i.e. a contribution $v_{N+1}\vec{e}_{N+1}$. Main differences are that k is real-valued (opposed to the integer-valued azimuthal mode m) and that there are no specific boundary conditions for the eigenfunction at an axis of symmetry. However, global degrees of freedom, e.g. a global pressure enforcing a volume, will be deactivate by default when $k \neq 0$.

10.4.1 Numerically obtaining the dispersion relation of a Turing instability

The additional Cartesian mode can be even used if $N = 0$ holds, i.e. we can expand a single point to an interval of length $[0 : 2\pi/k)$ (with periodic boundary conditions) and determine the eigenvalues and eigenfunctions here. This allows to quickly scan over k to numerically extract a dispersion relation. We will discuss this here for a simple Turing instability (see [26, 40, 45] for more details).

We begin by the equation class, which takes the diffusivity ratio and the reaction terms of the activator and inhibitor as arguments:

```
class TuringEquations(Equations):
    def __init__(self,d,f,g):
        super().__init__()
        self.d=d # Diffusivity ratio
        self.f,self.g=f,g # Reaction terms

    def define_fields(self):
        self.define_scalar_field("u","C2") # activator
        self.define_scalar_field("v","C2") # inhibitor

    def define_residuals(self):
        u,utest=var_and_test("u")
        v,vtest=var_and_test("v")
        self.add_weak(partial_t(u)-self.f,utest).add_weak(grad(u),grad(utest))
        self.add_weak(partial_t(v)-self.g,vtest).add_weak(self.d*grad(v),grad(vtest))
```

The problem class now just defines the reaction terms according to the Gierer-Meinhardt model [26, 40]. We then add a `PointMesh`, which is 0-dimensional mesh consisting only of a single point and define the just developed equations on this mesh. Of course, on such a mesh, we could only get the eigendynamics of the ODEs $\dot{u} = f, \dot{v} = g$, arising in absence of the diffusion terms. However, we will soon use the additional Cartesian normal mode to add a single x-coordinate to quickly and exactly (i.e. without any spatial discretization) calculate the eigenvalues of the corresponding one-dimensional system, i.e. with diffusion terms.

```
class TuringProblem(Problem):
    def __init__(self):
        super().__init__()
        # Parameters, see https://dx.doi.org/10.1007/s10994-023-06334-9
        self.d,self.a,self.b,self.c=self.define_global_parameter(d=40,a=0.01,b=1.2,c␣
↪= 0.7)
        # Reaction terms
        self.f=self.a-self.b*var("u")+var("u")**2/(var("v")*(1+self.c*var("u")**2))
        self.g=var("u")**2-var("v")

    def define_problem(self):
        # We add a 0d mesh here. So we do not have any spatial information
        from pyoomph.meshes.simplemeshes import PointMesh
        self+=PointMesh()
        eqs=TuringEquations(self.d,self.f,self.g)
        # Some reasonable guess here (not the exact solution)
        eqs+=InitialCondition(u=(self.a + 1)/self.b, v=(self.a + 1)**2/self.b**2)
        self+=eqs@"domain"
```

In the driver code, we call `setup_for_stability_analysis()` with `additional_cartesian_mode=True` to add the additional Cartesian normal mode. Supplying `normal_mode_k=k` during the calls of `solve_eigenproblem()` will select the wavenumber k for such a one-dimensional case. We can thereby quickly scan the full dispersion relation:

```
with TuringProblem() as problem:
    # Take a quick C compiler, to speed up the code generation
    problem.set_c_compiler("tcc")
    # Important part: This will add the N+1 dimension allowing for perturbations like␣
↪exp(i*k*x+lambda*t)
    problem.setup_for_stability_analysis(additional_cartesian_mode=True)
    # Solve for the flat stationary solution
    problem.solve()
    # And scan over the k values, solve the normal mode eigenvalue problem and write␣
↪the results to a file
    output=problem.create_text_file_output("dispersion.txt",header=["k","ReLamba1",
↪"ReLambda2","ImLambda1","ImLambda2"])
    for k in numpy.linspace(0,1,400):
        problem.solve_eigenproblem(2,normal_mode_k=k)
        evs=problem.get_last_eigenvalues()
        output.add_row(k,numpy.real(evs[0]),numpy.real(evs[1]),numpy.imag(evs[0]),
↪numpy.imag(evs[1]))
```

Of course, the results depicted in Fig. 10.6 can also be calculated analytically. However, it is already rather complicated to find the stationary base solution. Also, it is only a few lines of code and we have already implemented our equation class to be used in arbitrary dimensions and coordinate systems.

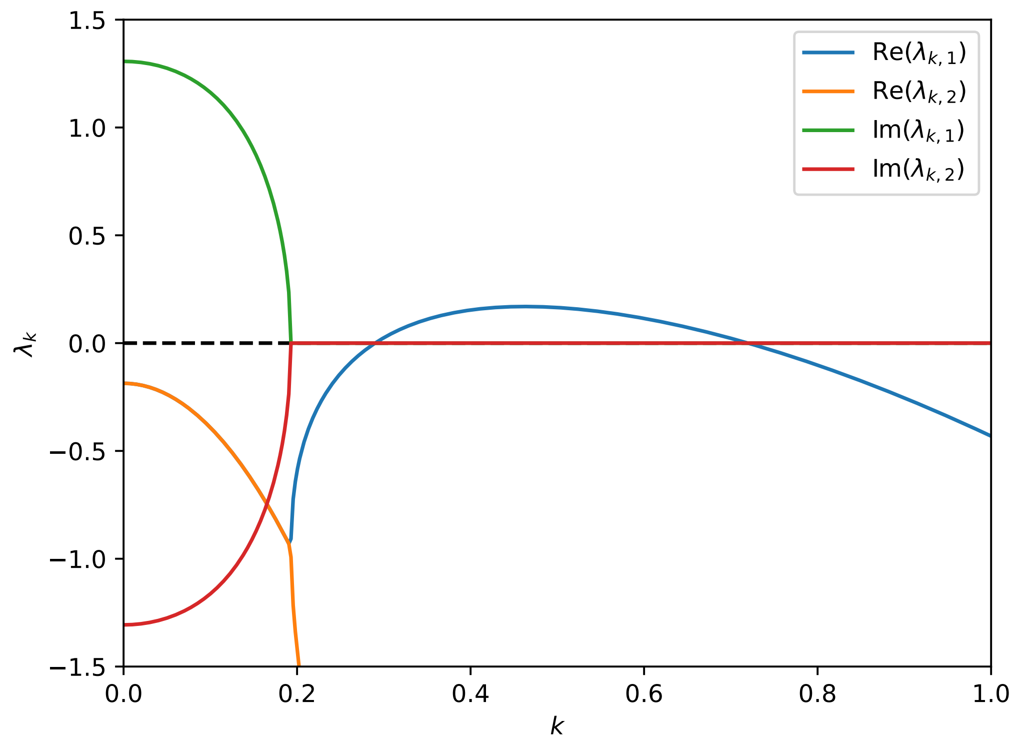

Fig. 10.6: Numerically calculated dispersion relation of the Turing instability.

In particular, we can quickly get a good guess for the dominant wavenumber, here around $k = 0.46$. This can be used to estimate a reasonable domain size for e.g. a two-dimensional transient simulation, where we can reuse the existing equation and problem class to obtain Turing patterns as depicted in Fig. 10.7. The corresponding script can be found here: `turing_transient.py`.

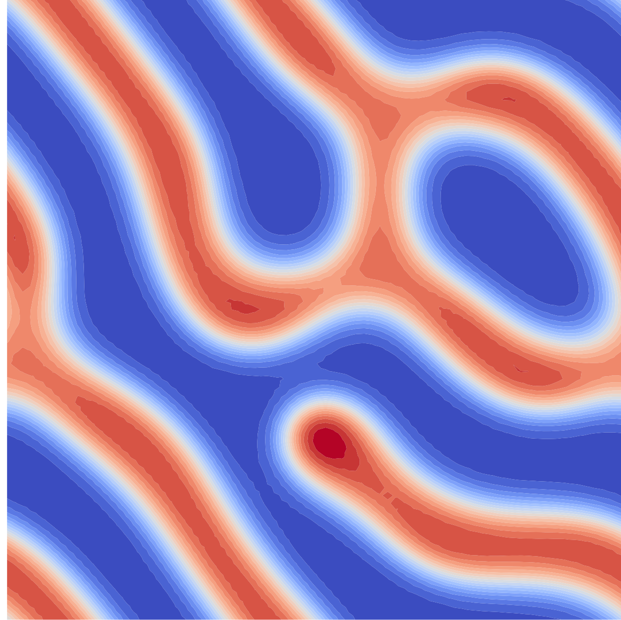

Fig. 10.7: After finding the dominant wave number, we can run transient 2d simulations with a suitable domain size.

10.4.2 Rayleigh-Plateau instability in presence of a substrate

The previous example was quite simple. In particular, the calculation of the dispersion relation could also have been done analyically by pen and paper. However, the stability analysis in an additional Cartesian direction can be used to quickly investigate considerably more intricate problems.

Here, we want to consider a long printed line of a liquid on a substrate, also called rivulet. We assume that this rivulet is infinitely long in the z-direction and its shape is independent on z. Alternatively, we can also think of a such a rivulet confined between two plates with distance $L = 2\pi/k$, free slip conditions and an 90° equilibrium contact angle with respect to the plate tangent.

In absence of a substrate, i.e. for a cylindrically shaped liquid bridge, it is well-known that it undergoes a Rayleigh-Plateau instability when $L/R > 2\pi$. However, how does the dynamics change if we consider a rivulet on a substrate instead? Of course, there must be a specific length (or wavenumber k in z-direction) when this printed line also undergoes a Rayleigh-Plateau instability, provided that we allow for some slip at the substrate. However, how does the critical wavenumber k and the time scale of the instability depend on e.g. the contact angle with respect to the substrate and/or the slip length at the substrate?

We can easily find all the answers by expressing the shape of base solution only in the x-y-plane and let pyoomph calculate the stability of such a solution with respect to perturbations in the third direction z. We will use Stokes flow for the liquid and combine it with a moving mesh to allow for shape deformations. Moreover, we only consider the right half (i.e. $x \geq 0$) and thereby only selects modes which are symmetric with respect to x in the following. Therefore, our mesh just creates half of the domain:

```
class RivuletMesh(GmshTemplate):
    def define_geometry(self):
        self.default_resolution=0.1
        self.mesh_mode="tris"
        cl_factor=0.5 # Make it finer at the contact line
        pr=cast(RivuletProblem,self.get_problem())
        geom=DropletGeometry(volume=pi/2,rivulet_instead=True,contact_angle=pr.theta)
        p00=self.point(0,0)
```

(continues on next page)

(continued from previous page)

```
        prl=self.point(-geom.base_radius,0,size=cl_factor*self.default_resolution) #␣
↪Mirrored point for the circle_arc
        prr=self.point(geom.base_radius,0,size=cl_factor*self.default_resolution) #␣
↪contact line
        p0h=self.point(0,geom.apex_height) # Top of the droplet

        self.circle_arc(prr,p0h,through_point=prl,name="interface")
        self.line(prr,p00,name="substrate")

        self.line(p00,p0h,name="axis")
        self.plane_surface("interface","substrate","axis",name="liquid")
```

Note how we use `DropletGeometry` with the kwarg `rivulet_instead=True` to convert the volume and the contact angle (which we obtain from the problem defined later on) to the base radius and apex height. Using `rivulet_instead=True` actually converts the value shipped by `volume` as surface area of the circle segment. In particular, using the volume of $\pi/2$, it gives a radius of curvature of unity for a contact angle $\theta = 90°$.

For the problem class, we just define the two parameters (slip length and contact angle) and add all equations to the system. Since we will have an effectively three-dimensional dimensional problem, it is important again to pass the `wall_tangent` to the `NavierStokesContactAngle`. This is a vector pointing inward to the bulk domain tangentially along the substrate, i.e. orthogonal to the `wall_normal` which is just the axially upward pointing normal. In a pure axisymmetric or 2d Cartesian case `wall_tangent=vector(-1,0)` is fine, but it is not true once the free surface deforms in a third direction (cf. Section 6.6.3). If again $\vec{n}$ is the normal of the free surface and $\vec{n}_\text{w}$ is the wall normal, we can use the double cross product $\vec{n}_\text{w} \times (\vec{n}_\text{w} \times \vec{n})$ to obtain such a (non-normalized) vector. We can use the known bac-cab identity along with $\|\vec{n}_\text{w}\| = 1$ to calculate it via $\vec{n}_\text{w}(\vec{n}_\text{w} \cdot \vec{n}) - \vec{n}$ and subsequently normalize the result, which is valid for all contact angles $0 < \theta < 180°$.

```
class RivuletProblem(Problem):
    def __init__(self):
        super().__init__()
        # Contact angle and slip length
        self.theta,self.sliplength=self.define_global_parameter(theta=90*degree,
↪sliplength=1)

    def define_problem(self):
        self+=RivuletMesh() # Add a 2d mesh

        # Assemble the equation system
        eqs=HyperelasticSmoothedMesh() # Moving mesh, Hyperelastic mesh is quite␣
↪robust, since we do not remesh in this particular tutorial
        eqs+=NavierStokesEquations(dynamic_viscosity=1) # bulk flow
        # Boundary conditions:
        # Navier-slip and no penetration at the substrate
        eqs+=( NavierStokesSlipLength(sliplength=self.sliplength) +␣
↪DirichletBC(velocity_y=0,mesh_y=0) )@"substrate"
        # Free surface at the interface
        eqs+=NavierStokesFreeSurface(surface_tension=1)@"interface"
        # Impose a contact angle at the contact line
        wall_normal=vector(0,1,0) # The substrate has an upwards normal (to be read␣
↪as 0*e_r+1*e_z+0*e_phi)
        ninter=var("normal",domain="..") # The normal at the free surface (one domain␣
↪up when evaluated at the contact line)
        wall_tangent=wall_normal*dot(wall_normal,ninter)-ninter # double cross␣
↪product bac-cab rule (with dot(wall_normal,wall_normal)=1)
        wall_tangent=wall_tangent/square_root(dot(wall_tangent,wall_tangent)) #␣
↪Normalize it
```

(continues on next page)

(continued from previous page)

```
        eqs+=NavierStokesContactAngle(contact_angle=self.theta,wall_normal=wall_
↪normal,wall_tangent=wall_tangent)@"interface/substrate"
        # Symmetry at the axis
        eqs+=DirichletBC(mesh_x=0,velocity_x=0)@"axis"
        # Enforce the volume/area of the liquid by a pressure constraint
        eqs+=EnforceVolumeByPressure(volume=pi/4)

        eqs+=MeshFileOutput()
         # Apply the equation system to the liquid domain
        self+=eqs@"liquid"
```

This only sets up the two-dimensional problem. The eigenanalysis with the additional normal mode is activated in the driver code:

```
problem=RivuletProblem() # Create the problem
# Setup the problem for k-stability analysis, we do not need an analytic Hessian,␣
↪since we don't do any bifurcation tracking
problem.setup_for_stability_analysis(additional_cartesian_mode=True,analytic_
↪hessian=False)
# Use the SLEPc eigensolver with MUMPS
problem.set_eigensolver("slepc").use_mumps()
problem.solve() # Solve the base state
problem.save_state("start.dump") # Save the start case at 90°


# Scan the contact angle
for theta_deg in [60,90,120]:
    problem.load_state("start.dump",ignore_outstep=True)
    problem.go_to_param(theta=theta_deg*degree)
    # Scan the slip length (either essentially free slip or quite low slip length)
    for sl in [10000,0.01]:
        problem.go_to_param(sliplength=sl)

        outf=problem.create_text_file_output("for_"+str(round(float(problem.theta/
↪degree)))+"_deg_SL_"+str(sl)+".txt",header=["k","Lambda"])

        for k in numpy.linspace(0.01,1.5,50):
            problem.solve_eigenproblem(1,normal_mode_k=k) # Solve the k-dependent␣
↪eigenproblem
            evs=problem.get_last_eigenvalues()
            outf.add_row(k,numpy.real(evs[0]))
```

Again, it just takes the call of `setup_for_stability_analysis()` with `additional_cartesian_mode=True` to activate this feature and shipping `normal_mode_k=k` to the call of `solve_eigenproblem()`.

The eigenvalues are plotted in Fig. 10.8. It is apparent that, indepedently of the slip length, the critical wavenumber is at $k = 1$ for $\theta = 90°$, which is reasonable, since the problem can be essentially mirrored at both axis to get the conventional Rayleigh-Plateu instability (at least for high slip lengths). A smaller slip influences the magnitude of the eigenvalues, which is reasonable, since it damps the motion of the contact line. For other contact angles, it is essentially the same, but the cricial wave number shifts. Due to the fixed cross-sectional area of the rivulet, a change in contact angle influences the radius of curvature, therefore the critical wave number shifts. Some plots of the eigendynamics are shown in Fig. 10.9, from which the influence of the slip length is clearly apparent.

To visualize the eigenmodes, it is beneficial to modify the problem code above by adding some operators to the `MeshFileOutput`:

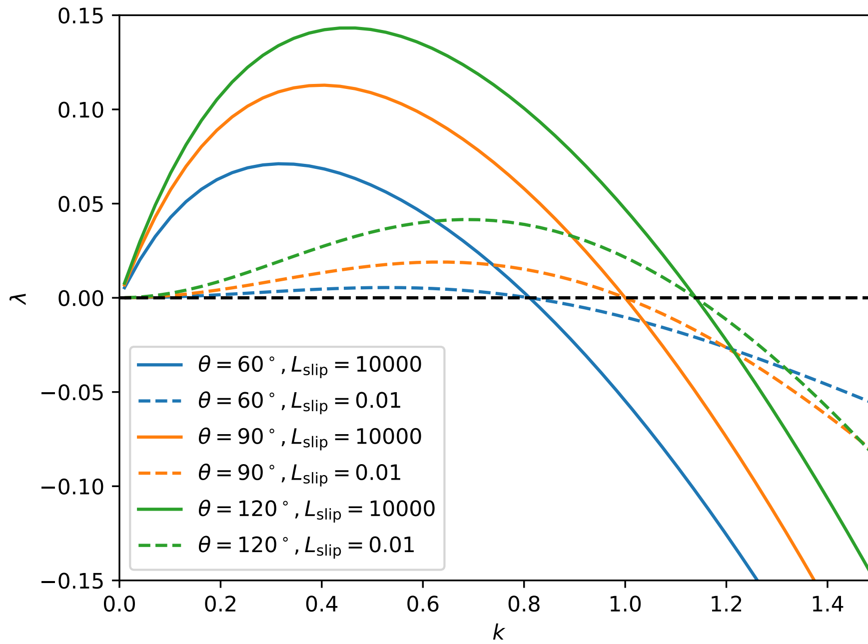

Fig. 10.8: Eigenvalues of the rivulet with different contact angles and slip lengths plotted against the wave number k.

```
from pyoomph.meshes.meshdatacache import MeshDataCombineWithEigenfunction,
↪MeshDataCartesianExtrusion
eqs+=MeshFileOutput(operator=MeshDataCombineWithEigenfunction(0)+MeshDataCartesianExtrusion(50))
```

Here `MeshDataCombineWithEigenfunction` will combine the base state with the eigenfunction at index 0, so that both the base solution and the eigenfunction are written to the file for Paraview. `MeshDataCartesianExtrusion` will apply the extrusion in the z-direction, respecting the oscillation of the eigenmode with $\exp(ikz)$. To write this output, add the `output()` method of the `Problem` to the driver code where you want to have output, however, after an `solve_eigenproblem()`, so that the eigensolution is available. Afterwards, you can load the files in Paraview, use the `Calculator` filter with an expression `iHat*Eigen_coordinate_x+jHat*Eigen_coordinate_y` to cast the mesh perturbation to a vector, combine it with `Wrap by Vector` and `Reflect` filters and you obtain plots like shown in Fig. 10.9.

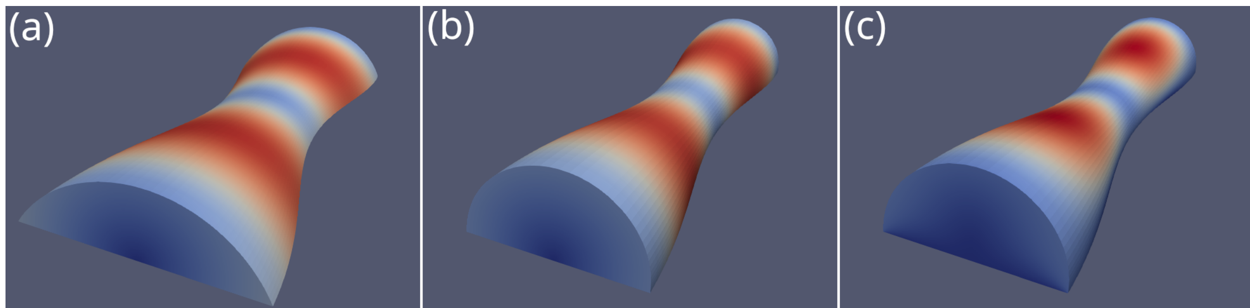

Fig. 10.9: Eigendynamics at $k = 0.6$ of the rivulet with (a) $\theta = 60°$, $L_{\text{slip}} = 10000$, (b) $\theta = 90°$, $L_{\text{slip}} = 10000$ and (c) $\theta = 90°$, $L_{\text{slip}} = 0.01$. Color-coded is the velocity magnitude.

10.5 Continuation of eigenbranches

While `solve_eigenproblem()` can solve for eigenvalues, it is sometimes hard to order them when varying a parameter. All eigenvalues can change, cross each other and it is then cumbersome to disentangle what is happening to each eigenbranch. Similar to bifurcation tracking, it is also possible to track a specific eigenfunction. To that end, we first have to solve for an eigenvalue/-vector pair as initial guess. Then, we solve an augmented system for the base state and this particular eigenvalue/-vector pair. Upon continuation, we will follow this particular eigenbranch.

We will discuss this feature on the basis of a liquid bridge with gravity. In absence of gravity, it is well known that the system undergoes a Rayleigh-Plateau instability at $L/R = 2\pi$. But what happens if we add gravity in axial direction to the system? Since the Rayleigh-Plateau instability sets in at rest, we will ignore the inertia term, i.e. going for Stokes flow. The problem class is quite simple and reads:

```
class LiquidBridgeProblem(Problem):
    def __init__(self):
        super().__init__()
        # Length of the domain
        self.L=self.define_global_parameter(L=2*pi)
        self.Bo=self.define_global_parameter(Bo=0)
        self.R=1 # Radius of the cylinder
        self.Nr=6 # Number of elements in the radial direction

    def define_problem(self):
        # Axisymmetric problem
        self.set_coordinate_system("axisymmetric")

        # Calculate the number of elements in the axial direction
        aspect0=float(self.L/self.R)
        Nl=round(aspect0*self.Nr)

        # Add the mesh
        self+=RectangularQuadMesh(size=[self.R,self.L],N=[self.Nr,Nl])

        # Bulk equations are: Stokes equations, pseudo-elastic mesh, mesh file output
        eqs=StokesEquations(dynamic_viscosity=1,bulkforce=self.Bo*vector(0,-1))
        eqs+=PseudoElasticMesh()
        eqs+=MeshFileOutput(operator=MeshDataCombineWithEigenfunction(0)) # Add also␣
↪the zeroth eigenfunction to the output

        # Boundary conditions: Axisymmetry, no-slip at the bottom, no-slip at the top,
↪ no-slip at the right, free surface at the left
        eqs+=AxisymmetryBC()@"left"
        eqs+=DirichletBC(velocity_x=0,velocity_y=0,mesh_y=0,mesh_x=True)@"bottom"
        eqs+=DirichletBC(velocity_x=0,velocity_y=0,mesh_x=True)@"top"
        # However, since we want to vary the length, we must trick a bit
        # First, we enforce that mesh_y=L at the top (i.e. we adjust mesh_y so that␣
↪var("mesh_y")-self.L=0)
        eqs+=EnforcedDirichlet(mesh_y=self.L)@"top"
        # However, thereby, the Lagrange multiplier for the kinematic boundary␣
↪condition is not automatically pinned to zero
        # Since mesh_y is a degree of freedom now at the right/top corner, the␣
↪kinematic BC constraint is not pinned automatically
        # So we must pin it manually
        eqs+=DirichletBC(_kin_bc=0)@"right/top"

        # Free surface at the left
        eqs+=NavierStokesFreeSurface(surface_tension=1)@"right"
```

(continues on next page)

(continued from previous page)

```
        # Volume constraint for the pressure to fix the volume
        Vdest=pi*self.R**2*self.L
        P,Ptest=self.add_global_dof("P",equation_contribution=-Vdest) # Subtract the␣
↪desired volume
        eqs+=WeakContribution(1,Ptest) # Integrate the actually present volume, P is␣
↪now determined by V_act-V_desired=0
        #eqs+=WeakContribution(P,"pressure") # And this pressure is added to the␣
↪pressure field
        eqs+=AverageConstraint(_kin_bc=P)@"right" # Average the normal traction to␣
↪agree with the gas pressure

        # Add the equations to the problem
        self+=eqs@"domain"
```

Note how we fix the top of the domain to the parameter L via an `EnforcedDirichlet`. Unlike a conventional `DirichletBC`, the `mesh_y` positions are now still degrees of freedom of the system, but enforced via Lagrange multipliers to L. This helps for continuation in L, since the mesh positions are now considered in the arclength tangent as well. However, since the axial mesh position at the contact line is now a degree of freedom, pyoomph does not automatically pin the Lagrange multiplier for the kinematic boundary condition (see Section 6.5, where we pin the Lagrange multiplier of the kinematic boundary condition only if all mesh positions are pinned). Therefore, we manually have to pin it, since the system is otherwise overconstrained.

The volume is again enforced by varying the gas pressure, which is added as normal traction by enforcing the average of the kinamatic boundary condition Lagrange multipliers (which are normal tractions) to the gas pressure.

Once set up, we can use this problem and solve for the stationary state at the minimum considered length L. We store this state, so that we can load it after each branch. To scan an eigenbranch, we first load the start point, solve the eigenproblem for the initial guess and then activate eigenbranch tracking with the desired index of the eigenvalue. By continuation of the length, we can follow this particular eigensolution. At the end of the scan for $\mathrm{Bo} = 0$, we again store the base solution. This is then used as a start for other branches with $\mathrm{Bo} \neq 0$. As you can see in the figure below, the presence of gravity leads to a fold bifurcation before the conventional Rayleigh-Plateau instability actually happens. With our approach, we find the other eigenbranches easily.

```
with LiquidBridgeProblem() as problem:

    # Generate analytically derived C code for the Hessian (for the eigenbranch␣
↪tracking)
    problem.setup_for_stability_analysis(analytic_hessian=True)
    problem.set_c_compiler("system").optimize_for_max_speed()
    # Solve the base problem
    problem.solve()
    L0=float(problem.L) # Store the initial length
    minL=0.8*L0
    maxL=1.2*L0
    problem.go_to_param(L=minL) # Go to the stable length
    problem.save_state("start.dump") # Save the initial state

    neigen=2
    def create_Bond_curve(Bo,eigenindex,startfile, postfix,start_high_L=False):
        problem.load_state(startfile,ignore_outstep=True) # Load the initial state
        # Go to the desired Bond number and length
        problem.go_to_param(Bo=Bo)
        problem.go_to_param(L=(maxL if start_high_L else minL))
        # Create and output file for this Bond number
        curve=NumericalTextOutputFile(problem.get_output_directory("curve_Bo_
```

(continues on next page)

(continued from previous page)

```
↪"+str(Bo)+"_"+str(eigenindex)+"_"+postfix+".txt"),header=["L","ReLambda","ImLambda
↪"])
        # Solve the eigenproblem and add the first eigenvalue to the curve
        problem.solve_eigenproblem(neigen)
        # We need to solve one eigenproblem only
        # Now we activate eigenbranch tracking
        problem.activate_eigenbranch_tracking(eigenvector=eigenindex)
        problem.solve() # And solve for it

        # Scan the curve
        curve.add_row(problem.L,numpy.real(problem.get_last_eigenvalues()[0]),numpy.
↪imag(problem.get_last_eigenvalues()[0]))
        dL0=(maxL-minL)/20*(-1 if start_high_L else 1) # Initial step size
        dL=dL0 # Current step size
        while problem.L.value<=maxL and problem.L.value>=minL:
            # We must use arclength continuation here, since we hit fold bifurcations␣
↪if Bo!=0
            dL=problem.arclength_continuation("L",dL,max_ds=dL0)
            curve.add_row(problem.L,numpy.real(problem.get_last_eigenvalues()[0]),
↪numpy.imag(problem.get_last_eigenvalues()[0]))
        problem.deactivate_bifurcation_tracking() # Stop the bifurcation tracking␣
↪(here, eigenbranch tracking)


    # Create the Bond curve for Bo=0
    create_Bond_curve(0,0,"start.dump","std")
    # Save the end state for later (high L)
    problem.save_state("end.dump")
    # Create the Bond curve for Bo=1
    create_Bond_curve(0,1,"start.dump","std")

    # Now create the lower L curves for Bo=0.0025
    create_Bond_curve(0.0025,0,"start.dump","fold")
    create_Bond_curve(0.0025,1,"start.dump","fold")

    # And also the higher L curves for Bo=0.0025
    create_Bond_curve(0.0025,0,"end.dump","unstab",start_high_L=True)
    # Save the end state for later, when going back to Bo=0
    problem.save_state("end2.dump")
    create_Bond_curve(0.0025,1,"end.dump","unstab",start_high_L=True)

    # Now we have found a rather strange unstable branch, where the interface is not␣
↪straight despite of Bo=0.
    # Here, the two curvatures cancel each other out, but it is not stable.
    create_Bond_curve(0,0,"end2.dump","unstab")
```

Note: If you want to find the pitchfork bifurcation using the bifurcation tracking tools (cf. Section 3.11.7), you will get some issues here. Since symmetry broken by the pitchfork bifurcation is not centered around the x-axis, the pitchfork won't be found. To overcome this issue, you can just enforce the `"top"` boundary to be at `mesh_y=self.L/2` and do it the same way with the `"bottom"` to `mesh_y=-self.L/2` with an `EnforcedDirichlet` including the pinning of the `_kinbc` at `"right/bottom"`. If the symmetry broken by the pitchfork is symmetric with respect to the x-axis, it works fine.

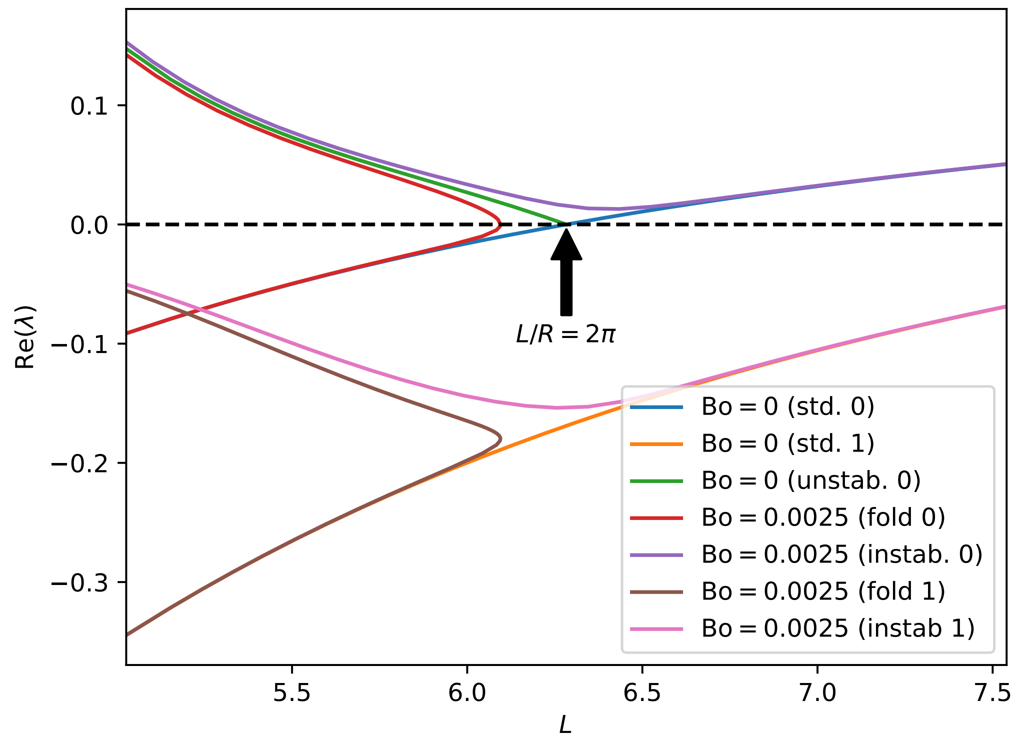

Fig. 10.10: Eigenbranches of a liquid bridge with gravity. The original Rayleigh-Plateau instability is broken by the presence of gravity. The subcritical pitchfork bifurcation becomes imperfect when gravity is considered.

CHAPTER

ELEVEN

PLOTTING INTERFACE

Since meshes are a quite complicated data structure (at least compared to simple grids), pyoomph has a built-in feature to plot meshes and fields. Currently, this only works for two-dimensional meshes, but it is envisioned to also support this for one- and three-dimensional domains in future.

To activate plotting, one has to set the `plotter` property of the `Problem` class to either an instance or a `list` of instances of the class `MatplotlibPlotter`, which is defined in the module `pyoomph.output.plotting`. After each `output()` call, plots will be generated automatically.

11.1 Two-dimensional plotting example

As an example case, let us take the evaporating droplet case from Section 7.6, i.e. we refer to the script `evaporating_water_droplet.py`. To add plotting, first a specialization of the `MatplotlibPlotter` class must be implemented. The entire plot is defined in the method `define_plot()`, in which the one first have to define the field of view, i.e. the area which should be covered by the plot. Since this area often depends on the problem settings, one can access the problem with the `get_problem()` method. Here, the considered area depends on the radius of the droplet:

```
from evaporating_water_droplet import * # Import the problem
from pyoomph.output.plotting import *


# Specialize the MatplotlibPlotter for the droplet problem
class DropPlotter(MatplotlibPlotter):
    # This method defines the entire plot
    def define_plot(self):
        p=self.get_problem() # we can get access to the problem by get_problem()
        r=p.droplet_radius # and hence can access the droplet base radius
        xrange=1.5*r # in x-direction, the image will be from [-1.5*r : 1.5*r ]
        self.background_color="darkgrey" # background color

        # First step: Set the field of view (xmin,ymin,xmax,ymax)
        self.set_view(-xrange,-0.165*xrange,xrange,0.9*xrange)
```

Also the `background_color` can be set, where either hex-codes for the color or predefined colors from the python package `matplotlib` can be used.

Once the desired plot area is selected by the `set_view()` method (arguments are minimum x, minimum y, maximum x and maximum y, in (potentially dimensional) spatial coordinates), we can start to add parts to the plot. Usually, one starts with color bars with `add_colorbar()`, that can are used later on to plot the fields. This can read e.g. as follows:

```
# Second step: add colorbars with different colormaps at different positions
# with 'factor', you can control the multiplicative factor (i.e. mm/s requires a␣
↪factor of 1000 to convert the m/s to mm/s)
cb_v=self.add_colorbar("velocity [mm/s]",cmap="seismic",position="bottom right",
↪factor=1000)
cb_vap=self.add_colorbar("water vapor [g/m^3]",cmap="Blues",position="top right",
↪factor=1000)
```

Each color bar gets first a title and can also contain LaTeXcode, usually by a `r`-string, e.g. `r"$\phi$"` to obtain ϕ. `cmap` selects the color map, see the documentation of `matplotlib` for a reference. With `position`, we can control the location of the color bar. This can either be a tuple of graph coordinates or a string indicating the position as shown in the example above. By default, all fields are plotted in the normal *SI* units without any prefixes. If a color bar should indicate the range e.g. in mm/s, one must set the `factor` to 1000 to compensate for the milli prefix.

Color bars have additional properties, which can be set, e.g. the `length`, `thickness`, `xshift` and `yshift`, `xmargin` and `ymargin` (all in graph coordinates). For a complete list of settings, read e.g. the output of `print(dir(cb_v))` or have a look at the `MatplotLibColorbar` class in the module `pyoomph.output.plotting`.

Once the color bars are set up, one can plot fields with those. Basically all plots of field data can be done by the `add_plot()` method, e.g.

```
# Now, we can add all kinds of plots
# plot the velocity (magnitude, since it is a vector) of the droplet domain (on both␣
↪sides)
self.add_plot("droplet/velocity",colorbar=cb_v,transform=["mirror_x",None])
# add velocity arrows on both sides
self.add_plot("droplet/velocity", mode="arrows",linecolor="green",transform=["mirror_x
↪",None])
```

Each `add_plot()` call requires to pass the data to plot as string, e.g. `"droplet/velocity"`. When a color bar is supplied by the `colorbar` argument, it will be plotted as color map. Vectorial fields, as e.g. the velocity, will be plotted as magnitude. The color bars will automatically increase in range to comprise the visible data range of all plots with the same color bar.

The argument `transform` (default `None`) will apply a transform on the plot, which can e.g. by `"mirror_x"` to mirror the data (and the vector fields) along the x-axis. You can also supply a list of transforms to plot all transformed data simultaneously. In that case, the return value of `add_plot()` is also a `list` of the individual plots. Further strings indicating transforms are `"rotate_cw"`, `"rotate_ccw"` and `"rotate_ccw_mirror"` for clock-wise, counter-clockwise and counter-clockwise rotation including mirroring, respectively. If a custom transform is required, you can overload the base class `PlotTransform` of `pyoomph.output.plotting` accordingly and pass an instance of your custom transform class as `transform`.

If no `colorbar` is set, you have to specify the plotting `mode`. To plot e.g. arrows indicating the direction of a vector field, you can use `mode="arrows"`. Alternatively, you can also use `mode="streamlines"`. Each `mode` has a different class with different settings creating the desired part of the plot. In the `pyoomph.output.plotting` module, you find all available classes for plot modes. These are decorated by `@MatplotLibPart.register()` and their class string `mode` indicates the plotting mode. You can furthermore see the attributes that you can set from the class definitions.

Again, you can access the problem to select a reasonable field to plot, e.g. the vapor on both sides:

```
# Plot the vapor in the gas phase
self.add_plot("gas/c_vap",colorbar=cb_vap,transform=["mirror_x",None])
```

To plot interface lines, just use `add_plot()` where the first argument indicates an interface mesh. This will automatically plot the interface lines:

```
# at the interface lines
self.add_plot("droplet/droplet_gas",linecolor="yellow",transform=["mirror_x",None])
self.add_plot("droplet/droplet_substrate",transform=["mirror_x",None])
self.add_plot("gas/gas_substrate",transform=["mirror_x",None])
```

You cannot plot an interface if there is no single equation defined on this interface. In that case, just add a dummy equation to this interface when defining the problem. A dummy equation instances can be just e.g. the base class `Equations` (or `InterfaceEquations`), which neither define any fields nor residuals nor doing anything else.

Finally, you can also plot interface fields. These can be either plotted as color maps (`mode="interfacecmap"`, which is selected automatically if you pass a `colorbar` to `add_plot()`) or as `"interfacearrows"`. The latter `mode` will be selected automatically, if you pass a `arrowkey` argument to `add_plot()`. However, therefore, you first have to add the arrow key, similar as done with the color bars above by using `add_arrow_key()`:

```
# For the evaporation rate, we require an arrow key, again with a factor 1000, since␣
↪we convert from kg to g per m^2s
ak_evap=self.add_arrow_key(position="top center",title="evap. rate water [g/(m^2s)]",
↪factor=1000)
# We can hide instances by setting invisible=True:
#       ak_evap.invisible=True
# or we can move it a bit relative to the "position" by the xmargin and ymargin
# or with xshift and yshift. All are in graph coordinates, i.e. 1 means the width/
↪height of the entire image
ak_evap.ymargin+=0.2
ak_evap.xmargin *= 2

# add the evaporation arrows at the interface, both sides
arrs=self.add_plot("droplet/droplet_gas/evap_rate",arrowkey=ak_evap,transform=[
↪"mirror_x",None])
```

Finally, there are few additional global parts (i.e. parts without any field data) you can add. These are e.g. `add_text()`, `add_time_label()` or `add_scale_bar()`. To add e.g. the current time and a scale bar to the plot, you can call

```
# and a time label and a scale bar
 self.add_time_label(position="top left")
 self.add_scale_bar(position="bottom center").textsize*=0.7
```

That is all you have to do in the plotter class. To use it, you just have to create an instance of it to the `plotter` property of the `Problem` class:

```
if __name__=="__main__":
    with EvaporatingDroplet() as problem:
        problem.plotter=DropPlotter(problem) # set the plotter and pass the problem␣
↪itself
        # The rest is the same as before
        # .....
        #
```

Alternatively, you can also set `plotter` to a `list` of multiple plotters.

On each output, the plotter(s) will be invoked to create each a plot in the `_plots` folder of the output directory. For an example of the resulting plot with this plotter class, refer to Fig. 7.6.

11.2 Replotting of existing data

The default file extension of plots are *png* files. You can change this by the `file_ext` to e.g. *pdf* files. *Png* files are more suitable to assemble a movie out of the plots, whereas *pdf* files are better for inclusion in publications.

To change the file extension, you can just set it after `plotter` has been assigned, e.g.

```
# Changing the file extension (also a list works, e.g. ["pdf","png"]
problem.plotter.file_ext = "pdf"
```

It can also be a `list` of file extensions to be created simultaneously.

Furthermore, you can change e.g. the `dpi` of the plots or change the default settings of individual plotting parts here:

```
# Changing e.g. the dpi or default settings of the velocity arrows:
problem.plotter.dpi *= 1.5
# problem.plotter.defaults("arrows").arrowdensity /= 2
# problem.plotter.defaults("arrows").arrowlength *= 1.5
```

However, after having simulated a long simulation, you do not want to run it again to create the new plots. Instead, you can instruct pyoomph to just redo all plots, if you supply the command line argument `--runmode p` to the call (cf. also Section 3.9.2):

```
python evap_droplet_thermal_plot.py --runmode p
```

If you only want to recreate e.g. the plot number 10, you can add the `--where` argument as well

```
python evap_droplet_thermal_plot.py --runmode p --where step==10
```

Alternatively, you can also create something like `--where 'step in [10,11,20]'` or alternative python `bool` expressions involving the output step `step`.

Note that this replotting only works, if `write_states` is `True` (which is default). Replotting, just like continuing a simulation (cf. Section 5.5), relies on the state files which contain all information of the current state of the simulation.

11.3 Plotting of eigenfunctions

When investigating the stability of stationary solutions by bifurcation analysis, one is frequently interested in the eigenfunction corresponding to the critical eigenvalue which real value passes the zero. To plot these functions, we can directly use the very same plotting class, but just pass a few additional arguments to the constructor.

As illustration, let us consider the problem of Section 5.6.3, which depends on the scripts `kuramoto_sivanshinsky_bifurcation.py`, `kuramoto_sivanshinsky_arclength_eigen.py` and `kuramoto_sivanshinsky.py`. The plotting class can be rather short, since we only have to plot the height field. However, we directly add the functionality for the plots of the eigenfunction:

```
class KSEPlotter(MatplotlibPlotter):
    def define_plot(self):
        cb=self.add_colorbar("h",position="top center") # Add a color bar, but do not␣
↪show it
        cb.invisible=True

        self.add_plot("domain/h",colorbar=cb) # plot the height field (or its␣
↪eigenfunction)
```

(continues on next page)

(continued from previous page)

```
        # Parameters as text
        self.add_text(r"$\gamma={:5.4f}, \delta={:5.4f}$".format(self.get_problem().
↪param_gamma.value,self.get_problem().param_delta.value), position="bottom left",
↪textsize=15)

        # Information text, based on whether we plot the eigenfunction or the normal␣
↪solution
        if self.eigenvector is not None: # eigenfunction is plotted
            self.add_text("eigenvalue={:5.4f}".format(self.get_eigenvalue()),position=
↪"bottom right",textsize=15) # eigenvalue (will be 0 anyhow)
            if self.eigenmode=="abs":
                self.add_text("eigenfunction magnitude",textsize=20,position="top␣
↪center") # title
            elif self.eigenmode=="real":
                self.add_text("eigenfunction (real)", textsize=20, position="top␣
↪center") # title
            elif self.eigenmode == "imag":
                self.add_text("eigenfunction (imag.)", textsize=20, position="top␣
↪center") # title
        else:
            self.add_text("height field", textsize=20, position="top center") # title
```

The plotting is done as before, via a color bar (which is hidden here by setting `invisible`) and `add_plot()` of the height field. With the `add_text()`, the parameters are added. Depending on whether we want to plot an eigenfunction or the normal solution, we add additional text to the plot. If an eigenfunction is plotted, `eigenvector` will refer to the index of the eigenfunction, whereas it will be `None` if no eigenfunction (i.e. the normal solution) is about to be plotted. With `get_eigenvalue()` we can get the eigenvalue corresponding to the plotted eigenfunction. `eigenmode` can hold different modes to plot the eigenfunction. Since it is in general complex, we have to choose whether we want to plot the `"real"` part, the `"imag"` part, the `"abs"` magnitude or the `"angle"` (which is the phase).

To use this plotting class for the normal solution and different eigenfunction plots, we just have to create an instance of this plotter for each desired plot:

```
if __name__ == "__main__":
    with KSEBifurcationProblem() as problem:
        # Add a bunch of plotters
        problem.plotter=[KSEPlotter(problem)] # Plot the normal solution
        problem.plotter.append(KSEPlotter(problem,eigenvector=0,eigenmode="real",
↪filetrunk="eigenreal_{:05d}")) # real part of the eigenfunction
        problem.plotter.append(KSEPlotter(problem, eigenvector=0, eigenmode="imag",␣
↪filetrunk="eigenimag_{:05d}")) # imag. part
        problem.plotter.append(KSEPlotter(problem, eigenvector=0, eigenmode="abs",␣
↪filetrunk="eigenabs_{:05d}")) # magnitude

        for p in problem.plotter:
            p.file_ext=["png","pdf"] # plot both png and pdf for all plotters

        # ...
        # the rest is the same
        # ...
```

The `plotter` property is now a `list` containing multiple plotters. Without further arguments, the plotter will plot the normal solution. If `eigenvector` is set, it indexes the desired eigenfunction to be plotted and `eigenmode` selects the desired mode (see above). Since all plots would write to the same file, we also have to specify `filetrunk`, which takes a format string, which is formatted based on the output step. All plotters are instructed to plot both *pdf* and *png* files.

Some plots are depicted in Fig. 11.1. As expected, the critical eigenfunction is only real valued an similar to the solution

itself, i.e. the collapse to the flat solution beyond the fold bifurcation is apparent.

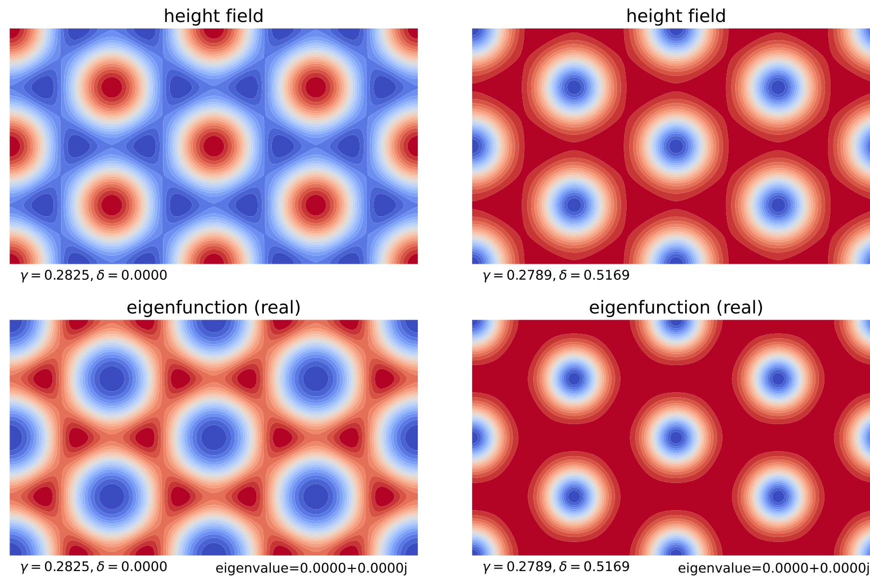

Fig. 11.1: Critical solution and corresponding eigenfunction at the fold bifurcation of the damped Kuramoto-Sivashinsky equation (5.7).

11.4 Generating a movie of eigendynamics

Eigendynamics, i.e. the exponential growth of a perturbation with a potential oscillation can be nicely visualized in a movie. To that end, we can generate a sequence of images with an eigenperturbation that grows (and oscillates) like $exp(\lambda t)$. As an example of such a movie, refer to Section 10.3.2, where the azimuthal instability of a bubble is investigated. Here, due to the azimuthal instability, it is effectively a three-dimensional scenario which is even harder to visualize in any other way.

We first start by importing the problem class from `rising_bubble.py` and define a conventional plotter class. We don't have to do anything regarding the eigendynamics here. This plotter can therefore be also used for the base state, as done in the previous examples.

```
from rising_bubble import *
from pyoomph.output.plotting import MatplotlibPlotter

# Define a plotter as usual. We don't have to care about the eigendynamics here
class RisingBubblePlotter(MatplotlibPlotter):
    def define_plot(self):
        pr=cast("RisingBubbleProblem",self.get_problem())
        self.set_view(-3*pr.R,-5*pr.R,3*pr.R,2*pr.R)
        cb_v=self.add_colorbar("velocity",cmap="viridis",position="top right")
        self.add_plot("domain/velocity",colorbar=cb_v,transform=[None,"mirror_x"])
        self.add_plot("domain/velocity",mode="streamlines",transform=[None,"mirror_x
↪"])
        self.add_plot("domain/interface",transform=[None,"mirror_x"])
```

We can then just find a stationary solution, solve (and refine) the eigensolution:

```
with RisingBubbleProblem() as problem:
    # Same as before in the rising bubble problem, except that we just solve at Bo=4
    problem.set_c_compiler("system").optimize_for_max_speed()
    problem.set_eigensolver("slepc").use_mumps()
    problem.setup_for_stability_analysis(azimuthal_stability=True,analytic_
↪hessian=False)
    problem.Mo=6.2e-7
    problem.Bo.value=4
    m=1
    lambd=0.1+0.67j

    problem.run(10,startstep=0.1,outstep=False,temporal_error=1)
    problem.solve(max_newton_iterations=20,spatial_adapt=4)
    problem.solve_eigenproblem(1,azimuthal_m=m,shift=lambd,target=lambd)
    problem.refine_eigenfunction(use_startvector=True)

    # Create an animation of the eigenfunction using the RisingBubblePlotter class
    problem.create_eigendynamics_animation("eigenanim",RisingBubblePlotter(),max_
↪amplitude=1,numperiods=4,numouts=4*25)
```

The final call of the method `create_eigendynamics_animation()` does the following: It creates a subdirectory (here `eigenanim`) in the output directory. It used the plotter class shipped as argument to perform plots. However, it won't plot the base state, but it will perturb the base state by a perturbation with the eigenfunction which grows (and oscillates) over time like $exp(\lambda t)$. We can specify the maximum amplitude, from which the initial amplitude is calculated. Note that the total amplitude is given by the amplitude of the eigenfunction times the growing amplitude. Therefore, it also depends on the way we implemented `process_eigenvectors()` in `rising_bubble.py`. We can specify the number of periods to consider. In case the eigenvalue is real, i.e. no oscillation amplitude given, this time is given by $2\pi/|\lambda|$. For complex eigenvalues, the period is the reasonable choice $2\pi/|\mathrm{Im}(\lambda)|$. The amount of output steps then controls the time step.

Once done, you can assemble all generated images to a movie. It is noteworthy that for azimuthal instabilities, the transform `"mirror_x"` will also invert the eigenperturbation on the left part of the plot in a reasonable way.

If you do not want to create a movie, but e.g. a sequence of images for a static document, consider adding `file-ext="pdf"` to the constructor of the plotter class.

CHAPTER TWELVE

COUPLING MULTIPLE SIMULATIONS WITH PRECICE

Pyoomph already comes with a adapter for the library preCICE.

This allows to couple pyoomph simulations with other pyoomph simulations or with any other simulation software that is compatible to use preCICE.

To use the preCICE coupling, you require the preCICE library (see installation instructions) and the preCICE python bindings.

12.1 Solving the heat equation on a domain by two simulations

To show how the preCICE adapter in pyoomph works, we cover the preCICE tutorial example Partitioned heat conduction.

A rectangular domain of size 2×1 is separated in a left (Dirichlet) and right (Neumann) participant, each of size 1×1. On the full domain, we solve a heat conduction equation, i.e. we also solve it in both participants. preCICE will take the lead, i.e. controls the time stepping of in both running pyoomph simulations and transfers the data at the mutual coupling interface from one participant to the other.

Via preCICE, The Dirichlet participant (left half) will receive the values of the temperature at the coupling boundary, impose these values as Dirichlet condition, solve the system and feed back the heat flux to the Neumann participant (right half). The latter will solve the system with the received heat flux as Neumann condition and feeds the temperature Dirichlet values again to the Dirichlet participant.

More details can be found in the preCICE tutorial.

To use preCICE in pyoomph, you can just import the module `pyoomph.solvers.precice_adapter`. As mentioned on the previous page, you must have installed preCICE and the preCICE python bindings to import it.

We want to formulate the problem in a way that it can be either run monolithically in a single simulation, which calculates the full domain. Alternatively, we can run either the Dirichlet or the Neumann participant. If both are running simultaneously, they will interact via preCICE.

pyoomph's `Problem` has the attributes `precice_participant` and `precice_config_file`. When using preCICE, you must specify the preCICE config file with the latter and the participant name with the former. Here, the config file `precice-config.xml` of this example defines the participants `"Dirichlet"` and `"Neumann"`.

As usual, we start by importing the required modules and define the heat equation, i.e. besides importing pyoomph's preCICE adapter, nothing spectacular is happening here:

```
from pyoomph import *
from pyoomph.expressions import *

# Import the preCICE adapter of pyoomph
from pyoomph.solvers.precice_adapter import *

```

(continues on next page)

(continued from previous page)

```
# Heat conduction equation with a source term
class HeatEquation(Equations):
    def __init__(self,f):
        super().__init__()
        self.f=f

    def define_fields(self):
        self.define_scalar_field("u","C2")

    def define_residuals(self):
        u,v=var_and_test("u")
        self.add_weak(partial_t(u),v).add_weak(grad(u),grad(v)).add_weak(-self.f,v)
```

The magic happens in the definition of the problem, where we use several classes from the `precice_adapter` module:

```
# Generic heat conduction problem. Can be run without preCICE on the full domain or␣
↪as Dirichlet or Neumann participant
class HeatConductionProblem(Problem):
    def __init__(self):
        super().__init__()
        self.alpha=3 # Parameters
        self.beta=1.2
        # Config file
        self.precice_config_file="precice-config.xml"

    def get_f(self):
        # Source term
        return self.beta-2-2*self.alpha

    def get_u_analytical(self):
        # Analytical solution
        return 1+var("coordinate_x")**2+self.alpha*var("coordinate_y")**2+self.
↪beta*var("time")

    def define_problem(self):
        # Depending on the participant, set up the coupling equations

        # First of all, we must provide the mesh at the coupling interface to preCICE
        # If we run without preCICE, this equation part is not used, so it will be␣
↪just discarded
        coupling_eqs=PreciceProvideMesh(self.precice_participant+"-Mesh")
        if self.precice_participant=="Dirichlet":
            # Dirichlet participant: We take the left part and use the right boundary␣
↪as coupling boundary
            x_offset,box_length=0,1
            coupling_boundary="right"
            # Here we write the heat flux and read the temperature
            # Note that we cannot use PreciceWriteData(Heat-Flux=partial_x(var("u",
↪domain="..")))
            # because "Heat-Flux" is not a valid keyword argument. Therefore, we use␣
↪the **{...} syntax
            # It will calculate the gradient of u in x-direction and write it to the␣
↪preCICE data "Heat-Flux"
            coupling_eqs+=PreciceWriteData(**{"Heat-Flux":partial_x(var("u",domain="..
↪"))})
            # It reads the preCICE field "Temperature" and stores it in the field "uD"
            coupling_eqs+=PreciceReadData(uD="Temperature")
```

(continues on next page)

(continued from previous page)

```
            # We cannot set a Dirichlet boundary condition depending on a variable,␣
↪so we use an enforced boundary condition
            # We adjust u so that u-uD=0 at the coupling boundary. This is equivalent␣
↪to setting u=uD
            coupling_eqs+=EnforcedBC(u=var("u")-var("uD"))
        elif self.precice_participant=="Neumann":
            # The Neuamnn participant is on the right side and uses the left boundary␣
↪as coupling boundary
            x_offset,box_length=1,1
            coupling_boundary="left"
            # It writes the preCICE field "Temperature" by evaluating the variable u
            coupling_eqs+=PreciceWriteData(Temperature=var("u"))
            # It reads the preCICE field "Heat-Flux" and stores it in the field "flux"
            coupling_eqs+=PreciceReadData(flux="Heat-Flux")
            # This flux is used as Neumann boundary condition.
            coupling_eqs+=NeumannBC(u=var("flux"))
        elif self.precice_participant=="":
            # If we run without preCICE, we use the full domain and have no coupling␣
↪boundary
            x_offset=0
            box_length=2
            coupling_boundary=None
        else:
            raise Exception("Unknown participant. Choose 'Dirichlet', 'Neumann' or an␣
↪empty string")

        # Create the corresponding mesh
        N0=11
        self+=RectangularQuadMesh(size=[box_length,1],N=[box_length*N0,N0],lower_
↪left=[x_offset,0])

        # Assemble the base equations
        eqs=MeshFileOutput()
        eqs+=HeatEquation(self.get_f())
        eqs+=InitialCondition(u=self.get_u_analyical())

        # Add the coupling equations and the Dirichlet boundary conditions
        # All potential Dirichlet boundaries
        dirichlet_bounds=set(["bottom","top","left","right"])
        if coupling_boundary:
            # Of course, the coupling boundary must not be set as Dirichlet boundary
            dirichlet_bounds.remove(coupling_boundary)
            eqs+=coupling_eqs@coupling_boundary
        eqs+=DirichletBC(u=self.get_u_analyical())@dirichlet_bounds

        # Calculate the error
        eqs+=IntegralObservables(error=(var("u")-self.get_u_analyical())**2)
        eqs+=IntegralObservableOutput()

        self+=eqs@"domain"
```

If we use preCICE, we only define half of the domain. The mesh of the `"Neumann"` participant is furthermore shifted to the right. Depending on the side we solve, the `coupling_boundary` is either the `"left"``or ``"right"` boundary of the domain. When we set `precice_participant=""` (default value), we just solve the full problem and do not add any coupling.

If we select one of the participant, however, we have to setup the coupling. This happens in multiple steps. First

of all, we must export the interface mesh at the coupling boundary to preCICE, which is done by the class `PreciceProvideMesh`. You must supply a mesh name agreeing with the `provide-mesh` definition in the config file `precice-config.xml`. This will tell preCICE where the nodes are located, so that it can be connected to the other participant.

Then, both participants have to exchange data. For writing data from the current participant to the other, you can use the class `PreciceWriteData`. It takes arguments of the form `PRECICE_NAME = PYOOMPH_EXPRESSION`, where `PRECICE_NAME` must coincide with the name of a `data` declaration in the preCICE config file. Since `Heat-Flux` cannot be used as keyword argument (due to the dash), we instead supply it via a `dict` using the `**{}` syntax in the Dirichlet participant. In pyoomph, we calculate the normal gradient and send it to the `"Heat-Flux"` data of preCICE. In the Neumann participant, we just write the nodal values of `var("u")` to the `"Temperature"` data of preCICE.

For the opposite direction, we can use `PreciceReadData`. It takes arguments like `PYOOMPH_NAME = PRECICE_NAME` and defines a pyoomph variable given by `PYOOMPH_NAME`, which will hold the values of the preCICE data given by `PRECICE_NAME`. Again, all used `PRECICE_NAME` must be declared in the config file to be readable from the mesh.

Thereby, the transfer of data is complete, but you still have to use the data read from the other participant in the current participant. In the Dirichlet case, we use `EnforcedBC`, since a `DirichletBC` may only depend on the time, Lagrangian and Eulerian (for a static mesh only) coordinates. However, in fact the `EnforcedBC` does exactly the same as a `DirichletBC` here. The Neumann part just imposes the read `var("flux")` as `NeumannBC`. Here, one has to be careful with the signs. From the weak form of the heat equation, the Neumann term would require a minus sign, but since the normal is pointing in negative x-direction on the Neumann side of the interface, it cancels out.

Eventually, we just have to attach all coupling equations to the corresponding boundary and make sure to not apply the Dirichlet boundary conditions of the far field here. If we do not use preCICE, we just discard the coupling equations.

The coupling is complete, but for running a coupled simulation, preCICE must take the lead for the time stepping. Typical time steps and the maximum simulation time are given in the config file. Therefore, pyoomph's `run()` method cannot be used. Instead, the method `precice_run()` has to be used:

```
if __name__=="__main__":
    problem=HeatConductionProblem()
    problem.initialise() # After this, precice_participant could have been set via␣
↪command line, e.g. by  -P precice_participant=Neumann
    if problem.precice_participant=="":
        # Just run it manually without preCICE
        problem.run(1,outstep=0.1)
    else:
        # Run it with preCICE. Time stepping is taken from the config file
        problem.precice_run()
```

This completes the simulation. For running with preCICE, you have to run the script two times, passing the participant name as command line parameter (see Section 3.9.2).

```
python partitioned_heat_conduction.py --outdir Dirichlet -P precice_
↪participant=Dirichlet &
python partitioned_heat_conduction.py --outdir Neumann -P precice_participant=Neumann
```

Of course, you must place the config file `precice-config.xml` in the same directory. If you run the scripts without setting `precice_participant`, it will just run the monolithic case without preCICE.

12.2 Non-matching meshes and passing vectorial quantities

The preCICE tutorial also covers a generalization of the previous case.

It solves exactly the same equation as in the previous, but the domains are different. One participant is now a rectangle with a circular hole and the other is the missing circular domain that fits in the hole. Most interestingly, the nodes do not have to match at the coupling boundary. While such non-matching coupling is possible in oomph-lib, it cannot be done in pyoomph directly yet. preCICE, however, does not care about the resolution and node positioning at the coupling boundaries, provided that it can find a corresponding node in the coupled participant.

The changes compared to the previous case are minor. Mainly, we need another mesh:

```
class RectMeshWithCircleHole(GmshTemplate):
    def define_geometry(self):
        pr=self.get_problem()
        y_bottom, y_top = 0, 1
        x_left, x_right = 0, 2
        radius = 0.2
        self.mesh_mode="tris"
        self.default_resolution=0.1
        midpoint = self.point(0.5, 0.5)
        outer_lines=self.create_lines((x_left, y_bottom), "bottom", (x_right,y_
↪bottom),"right",(x_right,y_top),"top",(x_left,y_top),"left")

        # Make non-matching meshes for testing
        circle_res=0.025 if pr.precice_participant=="Neumann" else 0.05

        circle_lines=self.create_circle_lines(midpoint,radius=radius,line_name=None␣
↪if pr.precice_participant=="" else "interface",mesh_size=circle_res)
        if pr.precice_participant!="Neumann":
            self.plane_surface(*outer_lines,holes=[circle_lines],name="domain")
        if pr.precice_participant!="Dirichlet":
            self.plane_surface(*circle_lines,name="domain")
```

We create either the full mesh for the monolithic run, otherwise, we either create a fine circle for the Neumann problem or a box with the corresponding circular hole in the Dirichlet case. The meshes do not match, since the resolution of the Neumann mesh is smaller. For the coupling interface, we define the boundary `"interface"`, which is not required for the monolithic case.

Furthermore, as in the preCICE example, we pass the heat flux as vector now. This means we have to change the `PreciceWriteData` in the Dirichlet participant to

```
coupling_eqs+=PreciceWriteData(**{"Heat-Flux":grad(var("u",domain=".."))},vector_
↪dim=2)
```

Vectorial quantities must be marked with `vector_dim`, otherwise, it is a scalar. However, it also must agree with the definition in the config file `precice-config-circle.xml`.

Likewise, reading and imposing the vectorial flux on the Neumann participant is now different:

```
coupling_eqs+=PreciceReadData(flux="Heat-Flux",vector_dim=2)
coupling_eqs+=NeumannBC(u=-dot(var("flux"),var("normal")))
```

Here, we have to consider the minus sign in the `NeumannBC`, since it has to agree with the weak formulation.

The rest is mainly the same. However, in the config file `precice-config-circle.xml`, we relax a bit the threshold for convergence and use acceleration. Therefore, this problem requires less simulation time than the previous example.

For running with preCICE, you must place the config file `precice-config-circle.xml` in the same directory and run again the script two times simultaneously:

```
python partitioned_heat_conduction_circle.py --outdir Dirichlet -P precice_
↪participant=Dirichlet &
python partitioned_heat_conduction_circle.py --outdir Neumann -P precice_
↪participant=Neumann
```

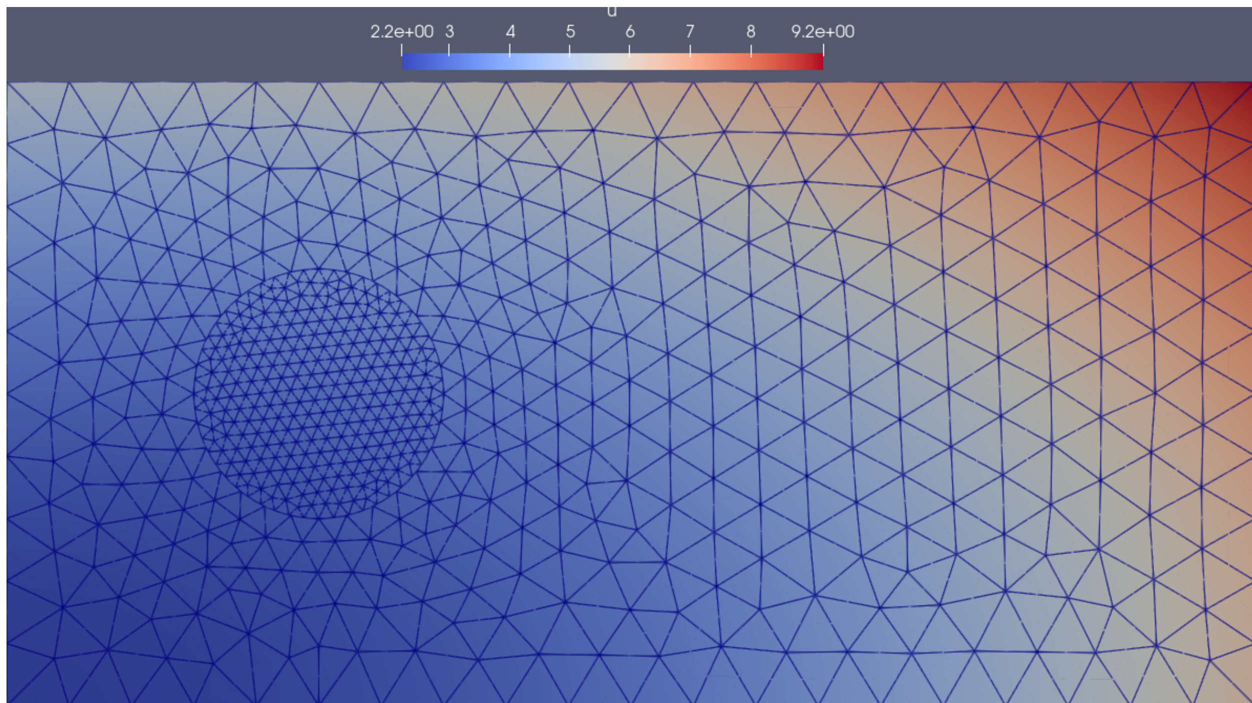

Fig. 12.1: Dirichlet/Neumann coupling with preCICE on non-matching meshes and with vectorial heat fluxes

CHAPTER

THIRTEEN

MISCELLANEOUS SETTINGS

Pyoomph has some further settings, which can be used to control the simulation. These are covered in this section.

13.1 Controlling the spatial integration order

Pyoomph uses the Gauss quadrature to perform the spatial integrals of the weak formulations. These are applied per element, i.e. each element is integrated separately and the result for the residual and Jacobian of all elements is accumulated. By default, the order of the Gauss quadrature depends on the order of the element, i.e. elements with maximum space `"C1"` will be integrated by a quadrature formula that is accurate for at least second order polynomials, whereas if `"C2"` spaces are used, the formula accurate for at least fifth order polynomials will be used. If one forms linear problems, these approximations are always accurate since any combination of shape and test functions cannot exceed these orders. However, for nonlinear problems, the integrated expression can be not even a polynomial at all, so that its Taylor expansion will exceed any polynomial order and small inaccuracies will occur. Usually, it does not contribute a lot to the error, but one can modify the integration order in two ways.

Either, the default integration order can be set globally by the `default_spatial_integration_order` property of the problem class. It takes a positive `int` value to account for the number of nodes per one-dimensional element for which is should be exact in case of a linear expression (e.g. 2 for `"C1"` and 3 for `"C2"`, since a 1d line element has exactly this amount of nodes). The integration orders 2-5 are implemented. Any lower or higher value will select the lower or upper value.

Alternatively, you can set the integration order for a domain - and, if not overridden in the same way, also for all its interfaces - by adding a `SpatialIntegrationOrder` object to the equation tree, where the order 2-5 has to be passed to the constructor.

You can hence easily check whether a higher integration order gives better results. This is rarely a vast improvement and comes at the cost of increased computation time for the assembly of the residual and Jacobian, but one can (and should) give it a try if the problem is nonlinear.

CHAPTER FOURTEEN

MATHEMATICAL EXPRESSIONS

Here, we provide an overview over the defined mathematical functions, differential operators and keyword names of relevant quantities, i.e. how to access the time and the coordinate.

14.1 Elementary functions

The following elementary mathematical functions are implemented:and work on scalar expressions and numbers.

Table 14.1: Elementary mathematical functions

`square_root(x,[d=2])`	d-th root $\sqrt[d]{x}$
`exp(x)`	Exponential function $\exp(x)$
`log(x)`	Natural logarithm $\log(x)$
`sin(x)`	Sine $\sin(x)$
`cos(x)`	Cosine $\cos(x)$
`tan(x)`	Tangent $\tan(x)$
`asin(x)`	Inverse sine $\mathrm{asin}(x)$
`acos(x)`	Inverse cosine $\mathrm{acos}(x)$
`atan(x)`	Inverse tangent $\mathrm{atan}(x)$
`atan2(y,x)`	Inverse tangent with case distinguishment $\mathrm{atan2}(y, x)$
`sinh(x)`	Hyperbolic sine $\sinh(x)$
`cosh(x)`	Hyperbolic cosine $\cosh(x)$
`tanh(x)`	Hyperbolic tangent $\tanh(x)$

Further functions can be implemented using the `CustomMathExpression` class from the module `pyoomph.expressions.cb`, see Section 3.10.

14.2 Keyword variables

The following keyword variables are defined and should not be used for any other field, i.e. not defined in an `Equations` class via e.g. `define_scalar_field()`. You can use these keyword variables as usually with `var()` or `nondim()` for the dimensional or nondimensionalized quantity, respectively.

Note that the time-derivative of the `mesh` variable only gives the mesh velocity if you use it with `ALE=False`, i.e. `partial_t(var("mesh"),ALE=False)`. Therefore, better use `mesh_velocity()` instead and e.g. `mesh_velocity()[0]` for the velocity in x-direction.

Table 14.2: Defined keyword variables to be used with either `var()` or `nondim()`. All mentioned time derivatives must be understood with the `ALE=False` setting.

"time"	Current time
"coordinate"	independent coordinate vector. Time derivatives of this variable are zero
"coordinate_x"	independent x-coordinate. Time derivatives of this variable are zero
"coordinate_y"	independent y-coordinate. Time derivatives of this variable are zero
"coordinate_z"	independent z-coordinate. Time derivatives of this variable are zero
"mesh"	mesh coordinate vector, similar to "coordinate", but the time derivative gives the mesh velocity
"mesh_x"	mesh x-coordinate, similar to "coordinate_x", but the time derivative gives the mesh x-velocity
"mesh_y"	mesh y-coordinate, similar to "coordinate_y", but the time derivative gives the mesh y-velocity
"mesh_z"	mesh z-coordinate, similar to "coordinate_z", but the time derivative gives the mesh z-velocity
"lagrangian"	Lagrangian coordinate vector. By default, initialized with the initial Eulerian "coordinate"
"lagrangian_x"	Lagrangian x-coordinate
"lagrangian_y"	Lagrangian y-coordinate
"lagrangian_z"	Lagrangian z-coordinate
"normal"	Normal vector. To be used at elements with co-dimension, i.e. interface elements
"normal_x"	x-component of the normal
"normal_y"	y-component of the normal
"normal_z"	z-component of the normal
"dx"	Can be used like in FEniCS to express `weak(a,b)` as `a*b*var("dx")`. It does not respect the functional determinant of the coordinate system, though.
"dX"	Same as "dX", but for Lagrangian integrals
"element_size_Eulerian"	Eulerian integration of the volume/area/length of the current element. Uses the coordinate system of the element, i.e. considers e.g. $2\pi r$ in axisymmetry
"cartesian_element_size_Eu	Same as above, but does not consider the coordinate system
"element_size_Lagrangian	Same as "element_size_Eulerian", but by Lagrangian integration
"cartesian_element_size_La	Same as "cartesian_element_size_Eulerian", but by Lagrangian integration
"element_length_h"	Typical length scale of the element, calculated by taking "element_size_Eulerian" to the power of one over the element dimension
"cartesian_element_length_	Typical length scale of the element, but calculated in a cartesian coordinate system

CHAPTER

FIFTEEN

REFERENCES

BIBLIOGRAPHY

[1] Martin Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E Rognes, and Garth N Wells. *The FEniCS Project Version 1.5*. University Library Heidelberg, 2015. doi:10.11588/ans.2015.100.20553.

[2] Nils Anspach and Stefan J Linz. Analysis of a solid-on-solid type block model for particle redeposition in ion-beam erosion processes under normal incidence. *J. Stat. Mech.*, 2012(06):P06012, jun 2012. doi:10.1088/1742-5468/2012/06/p06012.

[3] Pavel B. Bochev, Max D. Gunzburger, and John N. Shadid. Stability of the SUPG finite element method for transient advection-diffusion problems. *Comput. Methods Appl. Mech. Eng.*, 193(23):2301–2323, 2004. doi:10.1016/j.cma.2004.01.026.

[4] Paul Bonnefis, Javier Sierra-Ausin, David Fabre, and Jacques Magnaudet. Path instability of deformable bubbles rising in Newtonian liquids: a linear study. *J. Fluid Mech.*, 980:A19, 2024. doi:10.1017/jfm.2024.19.

[5] Richard A. Cairncross, P. Randall Schunk, Thomas A. Baer, Rekha R. Rao, and Phillip A. Sackinger. A finite element method for free surface flows of incompressible fluids in three dimensions. Part I. Boundary fitted mesh motion. *Int. J. Numer. Meth. Fluids*, 33(3):375–403, 2000. doi:https://doi.org/10.1002/1097-0363(20000615)33:3<375::AID-FLD13>3.0.CO;2-O.

[6] Nian-Sheng Cheng. Formula for the viscosity of a glycerol-water mixture. *Ind. Eng. Chem. Res.*, 47(9):3285–3288, 2008. doi:10.1021/ie071349z.

[7] Sarah Cleve, Christian Diddens, Tim Segers, Guillaume Lajoinie, and Michel Versluis. Time-resolved velocity and pressure field quantification in a flow-focusing device for ultrafast microbubble production. *Phys. Rev. Fluids*, 6(11):114202, 2021. doi:10.1103/PhysRevFluids.6.114202.

[8] M. Crouzeix and P.-A. Raviart. Conforming and nonconforming finite element methods for solving the stationary Stokes equations I. *R.A.I.R.O.*, 7(R3):33–75, 1973. doi:10.1051/m2an/197307R300331.

[9] Gerardino D'Errico, Ornella Ortona, Fabio Capuano, and Vincenzo Vitagliano. Diffusion coefficients for the binary system glycerol + water at 25 °c. a velocity correlation study. *Journal of Chemical & Engineering Data*, 49(6):1665–1670, 2004. doi:10.1021/je049917u.

[10] Ricardo Arturo Lopez de la Cruz, Christian Diddens, Xuehua Zhang, and Detlef Lohse. Marangoni instability triggered by selective evaporation of a binary liquid inside a Hele-Shaw cell. *J. Fluid Mech.*, 2021. doi:10.1017/jfm.2021.555.

[11] T.H.B. Demont, G.J. van Zwieten, C. Diddens, and E.H. van Brummelen. A robust and accurate adaptive approximation method for a diffuse-interface model of binary-fluid flows. 2022. doi:https://doi.org/10.1016/j.cma.2022.115563.

[12] C Diddens, Johannes GM Kuerten, CWM Van der Geld, and HMA Wijshoff. Modeling the evaporation of sessile multi-component droplets. *J. Colloid Interf. Sci.*, 487:426–436, 2017. doi:10.1016/j.jcis.2016.10.030.

[13] Christian Diddens. Detailed finite element method modeling of evaporating multi-component droplets. *J. Comp. Phys.*, 340:670–687, 2017. doi:10.1016/j.jcp.2017.03.049.

[14] Christian Diddens, Yaxing Li, and Detlef Lohse. Competing Marangoni and Rayleigh convection in evaporating binary droplets. *J. Fluid Mech.*, 2021. doi:10.1017/jfm.2020.734.

[15] Christian Diddens and Stefan J. Linz. Continuum modeling of particle redeposition during ion-beam erosion. *EPJ B*, July 2015. URL: https://doi.org/10.1140/epjb/e2015-60468-7, doi:10.1140/epjb/e2015-60468-7.

[16] Christian Diddens and Duarte Rocha. Bifurcation tracking on moving meshes and with consideration of azimuthal symmetry breaking instabilities. *J. Comput. Phys.*, 518:113306, 2024. doi:10.1016/j.jcp.2024.113306.

[17] Christian Diddens, Huanshu Tan, Pengyu Lv, Michel Versluis, JGM Kuerten, Xuehua Zhang, and Detlef Lohse. Evaporating pure, binary and ternary droplets: thermal effects and axial symmetry breaking. *J. Fluid Mech.*, 823:470–497, 2017. doi:/10.1017/jfm.2017.312.

[18] S. Facsko, T. Bobek, A. Stahl, H. Kurz, and T. Dekorsy. Dissipative continuum model for self-organized pattern formation during ion-beam erosion. *Phys. Rev. B*, 69:153412, Apr 2004. doi:10.1103/PhysRevB.69.153412.

[19] Thomas F. Fairgrieve and Allan D. Jepson. O.k. floquet multipliers. *SIAM Journal on Numerical Analysis*, 28(5):1446–1462, 1991. URL: http://www.jstor.org/stable/2157875 (visited on 2025-03-21).

[20] Patrick E. Farrell, Casper H. L. Beentjes, and Ásgeir Birkisson. The computation of disconnected bifurcation diagrams. 2016. URL: https://arxiv.org/abs/1603.00809, arXiv:1603.00809.

[21] Patrick E. Farrell, Ásgeir Birkisson, and Simon W. Funke. Deflation techniques for finding distinct solutions of nonlinear partial differential equations. 2015. URL: https://arxiv.org/abs/1410.5620, arXiv:1410.5620.

[22] Aage Fredenslund, Jurgen Gmehling, and Peter Rasmussen. *Vapor-liquid equilibria using UNIFAC : a group contribution method*. Elsevier Scientific Pub. Co. ; distributors for the U.S. and Canada, Elsevier North-Holland Amsterdam ; New York : New York, 1977. ISBN 0444416218. doi:10.1016/B978-0-444-41621-6.X5001-7.

[23] Aage Fredenslund, Russell L. Jones, and John M. Prausnitz. Group-contribution estimation of activity coefficients in nonideal liquid mixtures. *AIChE Journal*, 21(6):1086–1099, 1975. doi:https://doi.org/10.1002/aic.690210607.

[24] Anaïs Gauthier, Christian Diddens, Rémi Proville, Detlef Lohse, and Devaraj van der Meer. Self-propulsion of inverse Leidenfrost drops on a cryogenic bath. *Proc. Natl. Acad. Sci.*, 116(4):1174–1179, 2019. doi:10.1073/pnas.1812288116.

[25] Artur Gesla, Yohann Duguet, Patrick Le Quéré, and Laurent Martin Witkowski. Stability analysis of periodic orbits in nonlinear dynamical systems using chebyshev polynomials. 2024. URL: https://arxiv.org/abs/2407.18230, arXiv:2407.18230.

[26] A. Gierer and H. Meinhardt. A theory of biological pattern formation. *Kybernetic*, 12:30–39, 1972.

[27] Michiel A Hack, Patrick Vondeling, Menno Cornelissen, Detlef Lohse, Jacco H Snoeijer, Christian Diddens, and Tim Segers. Asymmetric coalescence of two droplets with different surface tensions is caused by capillary waves. *Phys. Rev. Fluids*, 6(10):104002, 2021. doi:10.1103/PhysRevFluids.6.104002.

[28] Matthias Heil and Andrew L. Hazel. oomph-lib - An Object-oriented multi-physics finite-element library. *Lecture Notes in Computational Science and Engineering*, 53:19–49, 2006. doi:10.1007/3-540-34596-5_2.

[29] Miguel A. Herrada and Jens G. Eggers. Path instability of an air bubble rising in water. *Proc. Natl. Acad. Sci.*, 120(4):e2216830120, 2023. doi:10.1073/pnas.2216830120.

[30] H. Hu and R.G. Larson. Analysis of the effects of Marangoni stresses on the microflow in an evaporating sessile droplet. *Langmuir*, 21(9):3972–3980, April 2005. doi:10.1021/la0475270.

[31] Hua Hu and Ronald G. Larson. Analysis of the microfluid flow in an evaporating sessile droplet. *Langmuir*, 21(9):3963–3971, 2005. PMID: 15835962. doi:10.1021/la047528s.

[32] Yuri A Kuznetsov. *Elements of applied bifurcation theory*. Applied mathematical sciences. Springer International Publishing, Cham, 2023.

[33] D.Y Kwok and A.W Neumann. Contact angle interpretation in terms of solid surface tension. *Colloid. Surface. A*, 161(1):31–48, 2000. URL: https://www.sciencedirect.com/science/article/pii/S0927775799003234, doi:https://doi.org/10.1016/S0927-7757(99)00323-4.

[34] Yanshen Li, Christian Diddens, Andrea Prosperetti, Kai Leong Chong, Xuehua Zhang, and Detlef Lohse. Bouncing oil droplet in a stratified liquid and its sudden death. *Phys. Rev. Lett.*, 122(15):154502, 2019. doi:10.1103/PhysRevLett.122.154502.

[35] Yaxing Li, Christian Diddens, Pengyu Lv, Herman Wijshoff, Michel Versluis, and Detlef Lohse. Gravitational effect in evaporating binary microdroplets. *Phys Rev. Lett.*, 122(11):114501, 2019. doi:10.1103/PhysRevLett.122.114501.

[36] Yaxing Li, Christian Diddens, Tim Segers, Herman Wijshoff, Michel Versluis, and Detlef Lohse. Evaporating droplets on oil-wetted surfaces: suppression of the coffee-stain effect. *Proc. Natl. Acad. Sci.*, 117(29):16756–16763, 2020. doi:10.1073/pnas.2006153117.

[37] Yaxing Li, Pengyu Lv, Christian Diddens, Huanshu Tan, Herman Wijshoff, Michel Versluis, and Detlef Lohse. Evaporation-triggered segregation of sessile binary droplets. *Phys Rev. Lett.*, 120(22):224501, 2018. doi:10.1103/PhysRevLett.120.224501.

[38] Anders Logg, Kent-Andre Mardal, and Garth Wells, editors. *Automated solution of differential equations by the finite element method*. Lecture notes in computational science and engineering. Springer, New York, NY, January 2012. doi:10.1007/978-3-642-23099-8.

[39] Jürgen Lohmann, Ralph Joh, and Jürgen Gmehling. From UNIFAC to Modified UNIFAC (Dortmund). *Ind. Eng. Chem. Res.*, 40(3):957–964, 2001. doi:10.1021/ie0005710.

[40] David Schnörr and Christoph Schnörr. Learning system parameters from turing patterns. *Mach. Learn.*, 112(9):3151–3190", 2023. doi:10.1007/s10994-023-06334-9.

[41] Koichi Takamura, Herbert Fischer, and Norman R. Morrow. Physical properties of aqueous glycerol solutions. *Journal of Petroleum Science and Engineering*, 98-99:50–60, 2012. doi:https://doi.org/10.1016/j.petrol.2012.09.003.

[42] Huanshu Tan, Christian Diddens, Pengyu Lv, Johannes GM Kuerten, Xuehua Zhang, and Detlef Lohse. Evaporation-triggered microdroplet nucleation and the four life phases of an evaporating ouzo drop. *Proc. Natl. Acad. Sci.*, 113(31):8642–8647, 2016. doi:10.1073/pnas.1602260113.

[43] Huanshu Tan, Christian Diddens, Michel Versluis, Hans-Jürgen Butt, Detlef Lohse, and Xuehua Zhang. Self-wrapping of an ouzo drop induced by evaporation on a superamphiphobic surface. *Soft Matter*, 13(15):2749–2759, 2017. doi:10.1039/C6SM02860H.

[44] Huanshu Tan, Christian Diddens, Xuehua Zhang, and Detlef Lohse. Evaporation of ternary sessile drops. *Drying of Complex Fluid Drops: Fundamentals and Applications*, 2022. doi:10.1039/9781839161186-00033.

[45] Alan Mathison Turing. The chemical basis of morphogenesis. *Philos. Trans. R. Soc. B*, 237(641):37–72, 1952. doi:10.1098/rstb.1952.0012.

[46] Scott Weady, Joshua Tong, Alexandra Zidovska, and Leif Ristroph. Anomalous convective flows carve pinnacles and scallops in melting ice. *Phys. Rev. Lett.*, 128:044502, Jan 2022. URL: https://link.aps.org/doi/10.1103/PhysRevLett.128.044502, doi:10.1103/PhysRevLett.128.044502.

[47] A. Zuend, C. Marcolli, A. M. Booth, D. M. Lienhard, V. Soonsin, U. K. Krieger, D. O. Topping, G. McFiggans, T. Peter, and J. H. Seinfeld. New and extended parameterization of the thermodynamic model AIOMFAC: calculation of activity coefficients for organic-inorganic mixtures containing carboxyl, hydroxyl, carbonyl, ether, ester, alkenyl, alkyl, and aromatic functional groups. *Atmos. Chem. Phys.*, 11(17):9155–9206, 2011. doi:10.5194/acp-11-9155-2011.

[48] A. Zuend, C. Marcolli, B. P. Luo, and T. Peter. A thermodynamic model of mixed organic-inorganic aerosols to predict activity coefficients. *Atmos. Chem. Phys.*, 8(16):4559–4593, 2008. doi:10.5194/acp-8-4559-2008.