The OOPDE setting of pde2path – a tutorial via some Allen-Cahn models

Jens D.M. Rademacher[1], Hannes Uecker[2]

[1] Fachbereich Mathematik, Universität Bremen, D28359 Bremen, jdmr@uni-bremen.de
[2] Institut für Mathematik, Universität Oldenburg, D26111 Oldenburg, hannes.uecker@uni-oldenburg.de

May 11, 2020

Abstract

We describe some basic pde2path demo directories for the (cubic–quintic) Allen–Cahn (AC) equation in the OOPDE setting. We start with the simplest possible case of a 1D interval with Neumann boundary conditions, and step by step proceed to more complicated problems such as 2D and 3D domains, (linear and nonlinear) Dirichlet boundary conditions, mesh adaptation, fold continuation, and quasilinear and globally coupled problems.
Recent updates: December 2019, HU: added [Uec19c] and hints on trullekrul to Remark 1.2. May 2020, HU: modified 1D mesh–adaptation to ad-hoc coarsening before refinement (background mesh no longer needed).

Contents

1 Introduction

The purpose of this tutorial is to give a soft introduction into the `OOPDE` [Prü16] setting of `pde2path` [Uec20c], and to `pde2path` in general. We consider the (cubic–quintic) Allen–Cahn equation

$$\partial_t u = c\Delta u + \lambda u + u^3 - \gamma u^5, \tag{1}$$

and focus on the associated steady state equation

$$G(u) := -c\Delta u - \lambda u - u^3 + \gamma u^5 \overset{!}{=} 0, \tag{2}$$

with $u = u(x) \in \mathbb{R}$, $x \in \Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$, Ω an interval, a rectangle or cuboid, respectively, and with various boundary conditions (BC). Table 1 gives an overview of the associated demo directories from `pde2path/demos/acsuite` discussed here (see Remark 1.2), where `pde2path/` is the `pde2path` home directory.

Table 1: Overview of the demos in `pde2path/demos/acsuite` discussed here.

directory	remarks
`ac1D_simple`	(2) over $\Omega = (-5, 5)$, with homogeneous Neumann BC; a minimal setup, §2
`ac1D`	extension of `ac1D_simple` to Dirichlet BC, also explains fold point and branch point continuation, mesh adaptation, and "restarts" for imperfect bifurcations; §3.1.
`ac1Dnlbc`	(2) over $\Omega = (-5, 5)$ with nonlinear BC, §3.2
`ac1Dxa/ac1Dxb`	a modification of (2) with x-dependent terms; two equivalent setups, §3.3
`ac2D`	(2) over $\Omega = (-2\pi, 2\pi)\times(-\pi, \pi)$ with Dirichlet BC, §4.1
`ac3D`	(2) over $\Omega = (-2\pi, 2\pi)\times(-3\pi/2, 3\pi/2)\times(-\pi, \pi)$ with Dirichlet BC, §4.2
`acql`	a quasilinear modification of (2) in the form $-\nabla \cdot [h(u)\nabla u] - f(u) = 0$; 1D–3D, §5.
`acgc`	a modification of (2) with a global coupling, i.e., a term $g(u, \lambda, h(\langle u\rangle))$ on the rhs. Because the Jacobian is no longer sparse, this requires some major modifications, i.e., adapted linear system and spectral solvers, based on Sherman Morrison formulas.

In 1D ($d = 1$), we have five demo-directories **ac1d***. We start with **ac1D_simple**, where we use homogeneous Neumann BC and explain a minimal setting and usage, and then step by step proceed to more complicated settings and additional features such as other boundary conditions, fold continuation, and mesh adaptation. In **ac2D*** and **ac3D*** we then explain how to extend this to 2D and 3D, and in **acql** consider quasilinear problems in 1D, 2D and 3D.

The original ('legacy') setup of `pde2path`, based on the `Matlab pdetoolbox`, is described in [UWR14, DRUW14]; the `OOPDE` setup extends `pde2path` to efficiently work in 1,2, and 3 spatial dimensions. See [Uec20c] for download information of the software and a documentation; in particular [dWDR$^+$20] contains a quickstart guide and reference card. Basically, `pde2path` treats PDEs of type

$$M\partial_t u = \nabla \cdot (c \otimes \nabla u) - au + b \otimes \nabla u + f, \tag{3}$$

where $u = u(x, t) \in \mathbb{R}^N$, $t \geq 0$, $x \in \Omega$, $\lambda \in \mathbb{R}^p$ is a parameter vector, $M \in \mathbb{R}^{N\times N}$ is a mass matrix, which may be singular, but for simplicity in the following $M = 1$, and the coefficients c, a, b and f may depend on x, λ and, typically nonlinearly, on u.[1] The boundary conditions (BC) are of the form

$$\mathbf{n} \cdot (c \otimes \nabla u) + qu = g, \tag{4}$$

[1] In (3), $[\nabla \cdot (c \otimes \nabla u)]_i := \sum_{j=1}^N [\partial_x c_{ij11}\partial_x + \partial_x c_{ij12}\partial_y + \partial_y c_{ij21}\partial_x + \partial_y c_{ij22}\partial_y]u_j$ (i^{th} component), and similarly $[au]_i = \sum_{j=1}^N a_{ij}u_j$, $[b \otimes \nabla u]_i := \sum_{j=1}^N [b_{ij1}\partial_x + b_{ij2}\partial_y]u_j$, and $f = (f_1, \dots, f_N)$ as a column vector.

where $\mathbf{n}$ is the outer normal, and extensions to periodic boundaries are available as well.

Our focus here is on the steady state problems

$$G(u,\lambda) := -\nabla\cdot(c\otimes\nabla u) + au - b\otimes\nabla u - f = 0, \tag{5}$$

but for instance Hopf bifurcations and so called canonical paths in optimal control have also been considered [Uec19a, Uec20b, Uec16, GU17, GUU19]. Moreover, there are interfaces to couple (5) with $n_Q \geq 1$ additional equations, here written as

$$Q_j(u,\lambda) = 0, \quad j = 1,\dots,n_Q. \tag{6}$$

Thus, the Allen-Cahn equation (1) is a simple scalar example, with $n_Q = 0$. The basic idea is to convert (5),(4) (and (6), if applicable) by the finite element method (FEM) into algebraic equations of the form (with a slight abuse of notation, see also Remark 1.1)

$$G(\mathrm{u}) := K_{\text{total}}(\mathrm{u})\mathrm{u} - F_{\text{total}}(\mathrm{u}) = 0 \tag{7}$$

for the nodal values $\mathrm{u} \in \mathbb{R}^{n_p}$, where n_p is the number of mesh points. Furthermore, K_{total} is the stiffness matrix, assembled from c, a, b and q in (5),(4), and F_{total} is the discretization of f, g from (5),(4), thus including the BC. Typically we choose $a = 0$ and put all terms without spatial derivatives into f.

In the OOPDE setup, we first need to set up the domain and the FEM discretization, and then employ the routines `assema, assemb` to assemble the required matrices and right–hand sides. In detail,

$$K_{\text{total}} = K + K_{\text{adv}} + sQ_{\text{BC}}, \quad F_{\text{total}} = F - sG_{\text{BC}}, \tag{8}$$

where s is a so called stiff spring factor, which can be used to approximate Dirichlet BC $qu = g$ via (4) by choosing a 'large' s, and where

- K, K_{adv} and F correspond to c, b and f in (5) (assuming as above that $a = 0$),
- $Q_{\text{BC}}, G_{\text{BC}}$ correspond to q, g in (4).

Remark 1.1 The idea of the FEM is to approximate $u : \Omega \to \mathbb{R}$ (for simplicity restricting to a steady scalar ($N = 1$) problem here) by $u_h \in V_h$ where

$$V_h := \left\{ u_h(x) = \sum_{j=1}^{n_p} \mathrm{u}_j\phi_j(x) : \mathrm{u} \in \mathbb{R}^{n_p} \right\} \subset V \subset H^1(\Omega),$$

is called the element space; here V also encodes the BC in appropriate form. In `pde2path` we use piecewise linear continuous ansatz functions ϕ_i, $i = 1,\dots,n_p$, with local supports over a mesh (intervals, triangles, tetrahedra) with gridpoints $x_i, i = 1,\dots,n_p$, such that $\phi_i(x_j) = \delta_{ij}$, i.e., hat–functions, or, more precisely, Lagrangian P_1 elements.

The matrix K and the vector F in (8) are then obtained from the weak form, for simplicity restricting to the scalar equation $-\nabla\cdot(c\nabla u) - f(u) = 0$,

$$a(u,v) := \int_\Omega \langle c\nabla u, \nabla v\rangle \,\mathrm{d}x + \int_{\partial\Omega} quv \,\mathrm{d}s \overset{!}{=} b(v) := \int_\Omega fv \,\mathrm{d}x + \int_{\partial\Omega} gv \,\mathrm{d}s, \quad \text{for all } v \in V, \tag{9}$$

as

$$K_{ij} = \int_\Omega \langle c\nabla\phi_i, \nabla\phi_j\rangle \,\mathrm{d}x, \quad F_i = \int_\Omega f\phi_i \,\mathrm{d}x, \tag{10}$$

and similar for the boundary terms $Q_{\rm BC}$ and $G_{\rm BC}$. If c and f in (10) are general functions, then expressions like in (10) in general can not be evaluated explicitly, and the question is how to approximate them numerically by quadrature formulas. In 1D, two options are the midpoint rule $\int_{x_1}^{x_2} f(x)\,{\rm d}x \approx I_m(f) := hf(m)$, $m = \frac{1}{2}(x_1{+}x_2)$, $h = (x_2 - x_1)$, which corresponds to a Riemann sum, and the trapezoidal rule

$$\int_{x_1}^{x_2} f(x)\,{\rm d}x \approx I_t(f) := \frac{h}{2}(f(x_1) + f(x_2)), \tag{11}$$

which corresponds to replacing f by its linear interpolant. Both rules integrate linear functions exactly, and for general functions $c \in C^2([x_1, x_2])$ the errors are $\left|\int_{x_1}^{x_2} c(x)\,{\rm d}x - I_m(c)\right| \le \frac{1}{24}\|c''\|_\infty h^3$ and $\left|\int_{x_1}^{x_2} c(x)\,{\rm d}x - I_t(c)\right| \le \frac{1}{12}\|c''\|_\infty h^3$, where $h = x_2 - x_1$. Similar rules also exist in 2D and 3D, and the `pdetoolbox` (2D) and `OOPDE` (1D, 2D, 3D) use trapezoidal rules to evaluate integrals as in (10), and like in any FEM package there are fast and efficient routines for such tasks.

In `OOPDE`, if we assume that c and f are given on the element centers, we have for instance

$$[\texttt{K},\texttt{M},\texttt{F}] = \texttt{fem.assema}(\texttt{grid},\texttt{c},\texttt{1},\texttt{f}), \tag{12}$$

where M is the (FEM) mass matrix

$$M_{ij} = \int_\Omega \phi_i\phi_j\,{\rm d}x, \tag{13}$$

which naturally also occurs in the evolutionary FEM formulation $M\dot{\rm u} = -G({\rm u})$. Moreover, if, e.g., c is constant, then K can be assembled exactly, and if $f \in V_h$, i.e., $f(x) = \sum_{j=1}^{n_p} f_j\phi_j(x)$, then we obtain the fast and convenient formula

$$F_i = F_i^M := (M\vec{f})_i. \tag{14}$$

On the other hand, here we are mainly interested in nonlinear[2] problems, e.g. $f = f(u) = u^2$, and if $u \in V_h$ then $u^2 \notin V_h$, and (14) is only an approximation. In detail, instead of $F_i = \int_\Omega f\phi_i\,{\rm d}x$ with $f(x) = f(\sum_j u_j\phi_j(x))$ we use

$$F_i^M = \int_\Omega \tilde{f}\phi_i\,{\rm d}x, \quad \text{i.e. } F^M = M\vec{f}, \tag{15}$$

where $\tilde{f}(x) = \sum_j f(u_j)\phi_j(x) \ne f(x)$ and $\vec{f} = (f(u_1), \dots, f(u_{n_p}))$. However, under mild conditions on f (e.g., $f \in C^2$), we obtain that

$$\|F - F^M\|_\infty \le Ch^2, \tag{16}$$

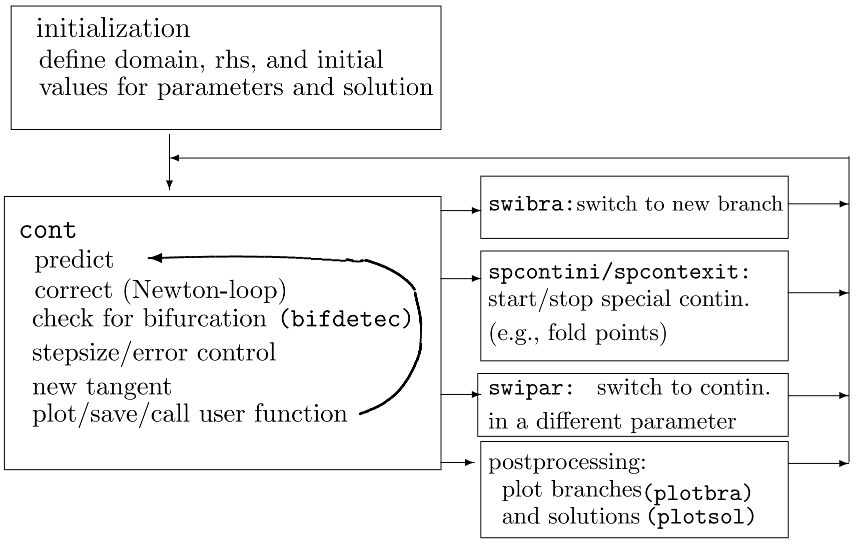
where $C = \sup_{u\in B_\gamma}\|D^2 f(u)\|_\infty$ with B_γ a region of interest of our typical solutions, and (15) is generally good enough for us. In particular, for the problems already considered in [UWR14] in the classic setup based on formulas such as (11) we find that a setup based on (15) yields qualitatively the same and quantitatively almost the same results. See also §4.1 for a quantitative example. This also holds for all other (semilinear) problems we considered, and thus except for §5 we use the simplified setup throughout this tutorial, and also in most of our other work (on semilinear problems), mainly because it is slightly more convenient to code for systems with $N \ge 1$, and because of some speed advantages.

For quasilinear problems, i.e., if $c = c(u)$ in (9), we need to proceed more classically. In this case, the standard procedure are trapezoidal rules via interpolating the nodal values of u to the element centers and then evaluating c. See §5 for an example. ⌋

[2]semilinear, i.e., the highest derivatives Δu appear only linearly, i.e., c does not depend on u;

In the `ac*` demos (except for `ac1Dxb`) we always have $b = 0$, and hence $K_{\text{adv}} = 0$, and in `ac1D_simple` we also choose $q, g = 0$, and hence $Q_{\text{BC}}, G_{\text{BC}} = 0$. Problems with $q, g \neq 0$ are treated subsequently, and for the more advanced case of a quasilinear AC equation in §5 we also need some differentiation matrices `Dx, Dy, Dz`. The demos that illustrate the implementation and use of `pde2path` generally consist of

- a function `acinit.m`, which collects some initialization calls;
- a function `oosetfemops.m` which defines the matrices needed in (7) or (8);
- functions `sG.m` and `sGjac.m`, defining the rhs G and its derivative $\partial_u G$;
- one (or more) script(s) `cmds.m (cmds1.m, cmds2.m, ...)` which collect the `pde2path` commands such as continuation calls, branch switching, plotting, ...
- additional functions used to illustrate additional features of `pde2path`, for instance: adaptive mesh-refinement, which in the `OOPDE` setting requires a function `e2rs` ('elements to refine selector'), typically based on error estimators; spectral continuation (fold points and branch points), which for efficiency needs a function `spjac.m` that computes $\partial_u(\partial_u G(u_*)\phi)$, where ϕ is a kernel eigenvector of $\partial_u G(u_*)$ at a fold or branch point u_* , see (20) and (21).



initialization
define domain, rhs, and initial values for parameters and solution
cont
predict
correct (Newton-loop)
check for bifurcation (bifdetec)
stepsize/error control
new tangent
plot/save/call user function
swibra:switch to new branch
spcontini/spcontexit:
start/stop special contin.
(e.g., fold points)
swipar: switch to contin.
in a different parameter
postprocessing:
plot branches(plotbra)
and solutions (plotsol)

Figure 1: Basic flow diagram of using `pde2path`. The initialization block is typically put into a function `*init`, where `*` is the name of the problem, and it usually starts with a call of `p=stanparam(p)`, setting all `pde2path` parameters to standard values, after which the user should redefine the pertinent problem parameters. The `cont` block gives a schematic overview of the main steps in the function `cont`, where the loop is executed for a number of steps, or until some other criterion is fulfilled, for instance if a parameter leaves a predefined range. To the right there are four typical next steps after an initial (or subsequent) run of `cont`: (a) branch switching to a bifurcating branch; (b) fold point continuation; (c) switching to continuation in a different parameter; (d) post-processing, i.e., mostly plotting. (a), (b) make sense if during `cont` a bifurcation or fold point was found. After each of these commands, `cont` can be called again to continue new branches (or further extend those already given). For convenience, all these commands are typically put into a script file `cmds.m` in "cell mode", i.e., where (groups of) commands are executed individually.

The basic flow of exploring a model with `pde2path` is also sketched in Fig. 1, here repeated from [dWDR+20] for convenience. As already said, this tutorial aims to give a soft introduction to `pde2path` from a users point of view, i.e., to explain the basic structure of the software and thus enable the user to set up own problems. More background on the used algorithms can be found in

[UWR14, DRUW14], while [dWDR+20] gives installation instructions, and a brief function and data structure overview of pde2path for quick reference.

Remark 1.2 For simplicity, here we restrict to steady problems and simple bifurcation points. Hopf bifurcations are considered in [Uec19a, Uec19a], and (steady) bifurcations of higher multiplicity in [Uec20a]. Furthermore, see [RU17] for problems with continuous symmetries which require $n_Q > 0$, cf. (6), and in [DU17] we treat variants of (2) with periodic boundary conditions, in 1D, 2D and 3D, collected in demos/acpbc. Finally, here we restrict to mesh adaptation in 1D ad 2D based on classical error–estimators. pde2path also contains interfaces to the package trullekrul, which provides anisotropic mesh adaptation in 2D and 3D, and these interfaces together with Allen–Cahn examples are described in [Uec19c]. ⌋

2 The simplest setting: ac1D_simple

2.1 Problem setup and implementation

In 1D we consider (2) on the interval $\Omega = (-5, 5)$, first with homogeneous Neumann boundary conditions (BC). For all c, λ, γ we then have the trivial solution $u \equiv 0$, and we can explicitly compute the bifurcation points for continuation in λ, namely

$$\lambda_{j+1} = (j\pi/10)^2, \quad \text{with eigenfunctions } \phi_{j+1}(x) = \cos(j\pi x/10), \quad j = 0, 1, 2, \ldots . \tag{17}$$

The goal is to numerically find these bifurcation points and the bifurcating branches of nontrivial solutions, including possible secondary bifurcations, where we then again can switch to bifurcating branches.[3]

The ac1D_simple demo contains the 5 files acinit.m, oosetfemops.m, sG.m, sGjac.m and cmds.m. The file sGjac.m for fast computation of Jacobians is optional, but recommended as good practice, even though this is not crucial for the present 1D example. Below we give complete listings of these files, Table 2 gives a summary, and in §2.2 and §2.3 we comment on the results of running the script file ac1D_simple/cmds.m.

Table 2: Scripts and functions in ac1D_simple.

script/function	purpose, remarks
cmds.m	main script, to be run cell-by-cell
p=acinit(p,lx,nx,par)	function for initialization of the problem
p=oosetfemops(p)	set FEM matrices (stiffness K and mass M)
r=sG(p,u)	encodes G from 2 (including the BC)
Gu=sGjac(p,u)	Jacobian $\partial_u G(u)$ of G (optional, but highly recommend)

```
function p=acinit(p,lx,nx,par) % init AC as a rather generic scalar PDE
p=stanparam(p); screenlayout(p); % standard settings/screenlayout
p.nc.neq=1; p.sw.sfem=-1; % scalar problem, implementation with OOPDE:
p.fuha.sG=@sG; p.fuha.sGjac=@sGjac; % function handles to rhs and Jacobian
pde=stanpdeo1D(lx,2*lx/nx); % standard PDE object 1D, yields domain and mesh
p.pdeo=pde; p.np=pde.grid.nPoints; p.nu=p.np; p.sol.xi=1/(p.nu);
p.u=zeros(p.np,1); p.u=[p.u; par']; % initial sol, with param appended
```

[3]The spatially homogeneous branch b1 bifurcating at λ_1, and the secondary bifurcation points on b1, can of course also be calculated explicitly; the point is to show how do do this with pde2path in the simplest possible setting.

```
p=oosetfemops(p);     % generate FEM matrices
p.nc.nsteps=20; % # of continuation steps
p.nc.ilam=2; p.nc.lammax=1; % continue in par(2)=lambda, up to lammax
11 p.sol.ds=0.1; p.nc.dsmax=0.2; % starting and max value for contin. step size
p.sw.foldcheck=1; % detect and localize folds
p.plot.auxdict={'c','lambda','gamma'}; % parameter names
```

Listing 1: `ac1D_simple/acinit.m` collects some typical initialization commands. The command `p=stanparam(p)` in line 2 sets the `pde2path` controls, switches and numerical constants to standard values to be possibly adjusted afterwards. In lines 3,4 the sfem switch is set to -1, indicating the use of `OOPDE`, and then the function handles to the rhs and its Jacobian are set. In line 5 a 1D pde-object is created, i.e., the interval (-lx,lx) with mesh width lx/dsc, and in line 6 this object, the number of grid points, and the associated norm weight ξ is stored in the struct `p`. In line 7 the solution vector is initialized (here with the explicitly known trivial solution $u = 0$) and the parameters appended. In line 8 the call to oosetfemops (see Listing 2) sets the FEM matrices. The remainder of acinit sets some additional controls, explained by the comments.

```
function p=oosetfemops(p) % set FEM operators, no BC, i.e., homog. Neumann BC
[p.mat.K,p.mat.M,~]=p.pdeo.fem.assema(p.pdeo.grid,1,1,1); % FEM matrices
```

Listing 2: `ac1D_simple/oosetfemops.m`. We use homogeneous Neumann BC, thus q and g are zero, and we only need to call `assema`. The (Neumann Laplacian) K and the mass matrix M are then saved in `p.mat.K` and `p.mat.M`.

```
function r=sG(p,u) % rhs for AC with Neumann BC, "simple" setting
par=u(p.nu+1:end); u=u(1:p.nu); % split u into parameters and PDE variables
K=par(1)*p.mat.K; f=par(2)*u+u.^3-par(3)*u.^5; % effective stiffness and nonlin.
r=K*u-p.mat.M*f;   % putting together 'Laplacian' K*u and nonlinearity -M*f
```

Listing 3: `ac1D_simple/sG.m`. In line 2 we split the full vector u into the pde-variables and the parameters. In line 3 we set up the effective K and the (nodal) nonlinearity f; this is the only problem specific line of sG.m. In line 4 we then compute G, using the approximation (15) for the nonlinearity.

```
function Gu=sGjac(p,u)  % AC Jacobian
par=u(p.nu+1:end); u=u(1:p.nu);  % split u into parameters and PDE variables
K=par(1)*p.mat.K; fu=par(2)+3*u.^2-5*par(3)*u.^4; % K, and local derivatives
Fu=spdiags(fu,0,p.nu,p.nu);  % put derivatives into (sparse) matrix
Gu=K-p.mat.M*Fu;         % the Jacobian matrix
```

Listing 4: `ac1D_simple/sGjac.m`. Again we first split u into the pde and parameter part (l2). In line 3 we compute the effective K, and the derivative of the nonlinearity f, from which $\partial_u G$ is computed in line 4. Again, line 3 is the only problem specific line.

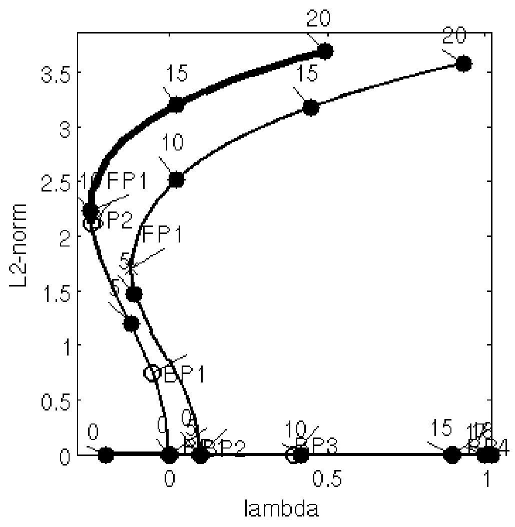
```
%% C1, preparations, close windows, clear workspace (keep only pphome for help)
close all; keep pphome;
%% C2: init (generic), then specific settings (could also be set in init)
p=[]; par=[1 -0.2 1]; p=acinit(p,5,20,par); p=setfn(p,'tr');  % output dir
%% C3: first continuation call  (cont. of trivial branch to find bifpoints)
p=cont(p);
%% C4: switch to first 3 bifurcating branches and continue
p=swibra('tr','bpt1','b1',0.1); p=cont(p);
p=swibra('tr','bpt2','b2',0.1); p=cont(p);
p=swibra('tr','bpt3','b3',0.1); p=cont(p);
%% C5: compute branch bifurcating from first nontrivial branch
p=swibra('b1','bpt1','b1-1',0.1); p.file.smod=3;  p=cont(p,10);
%% C6: minimal syntax bifurcation diagram plotting:
% uses last point in dir, and info from dir
figure(3); clf; plotbra('tr','lsw',2); % trivial branch, label (only) BPs
plotbra('b1','lsw',31);  % branch 1, label everything
plotbra('b2','fplab',1); % branch 2, only label FP1
%% C7: advanced  bifurcation diagram plotting, check fancy=0,1,2
f=3; c=0; figure(f); clf; % f=figure-Nr, c=component number (of branch)
plotbra('tr','pt18',f,c,'cl',[0.5 0.5 0.5],'bplab',[1 2 3 4]);
plotbra('b1','pt20',f,c,'cl','k','bplab',[1,2],'fplab',1,'lab',10);
```

Listing 5: `ac1D_simple/cmds.m` (slightly abbreviated). In cell 1 we close all windows and clear the workspace, except for the root-directory `pphome` for the help-system. In cell 2 we initialize with the selected parameter values by calling `acinit` from Listing 1. Note that with lx=5 and nx=20 we aim at a rather coarse mesh of mesh width $h = 0.5$. Then we set the output dir. Cell 3 then contains the first continuation call; this continues the trivial branch $u \equiv 0$, hence the main purpose is to find bifurcation points. In cell 4 we then continue 3 nontrivial bifurcating branches. On the first nontrivial branch (b1) there are (secondary) bifurcation points, and we compute a bifurcating branch in cell 5 (it turns out, that this branch connects the first and the second bifurcation point on b1). The remaining cells concern plotting, which we explain in §2.2 and §2.3.

2.2 Branch plotting

Plotting the information computed is typically an iterative process: one starts with some easy and minimal syntax as in cell 6 of Listing 5, and then iteratively improves the plots by adding colors and adding (or removing) more branches/points/labels. Cells 6 and 7 of Listing 5 are just a suggestion, with the output in Fig. 2. More details on `plotbra` can for instance be found in [dWDR$^+$20, dW17].

(a) Initial plot (cell 6)



(b) Fancy plot (cell7)

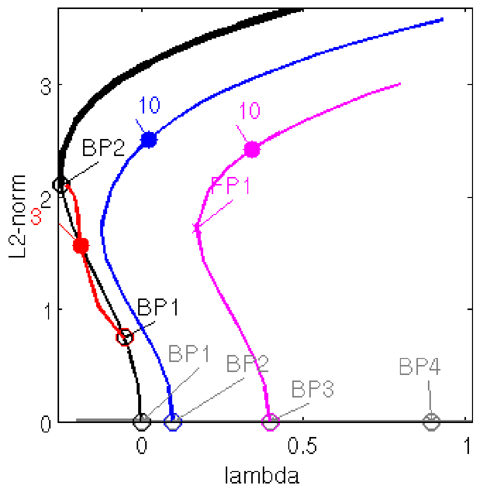


Figure 2: Bifurcation diagram (BD) plots. (a) is from cell 6 where we call `plotbra(dir)` (for dir='tr', 'b1', 'b2'), i.e., without additional arguments. In this case, `plotbra` scans `dir` for saved points, uses the data of the last point in `dir` for the branch plotting, and moreover marks the other points found in `dir` in the BD. As this often gets cluttered, its main purpose is an initial overview of the branches and available points. In cell 7 we use somewhat refined calls of `plotbra` to plot all the branches and some labels, see (b).

2.3 Solution plotting

While in general `plotsol` has many optional settings and arguments, see [Wet17] for a detailed tutorial, in cell 8 of Listing 5 the initial plot is usually done via `plotsol('dir','pt')`, with the directory(=branch) 'dir' and the point 'pt' only; the figure number, the component numbers and the plot styles are then taken from the field `p.plot`. The (verbatim) result of cell 8 is in Fig. 3. The titles are added automatically, and everything can be changed interactively in the `Matlab` plot window, and then saved to disk.

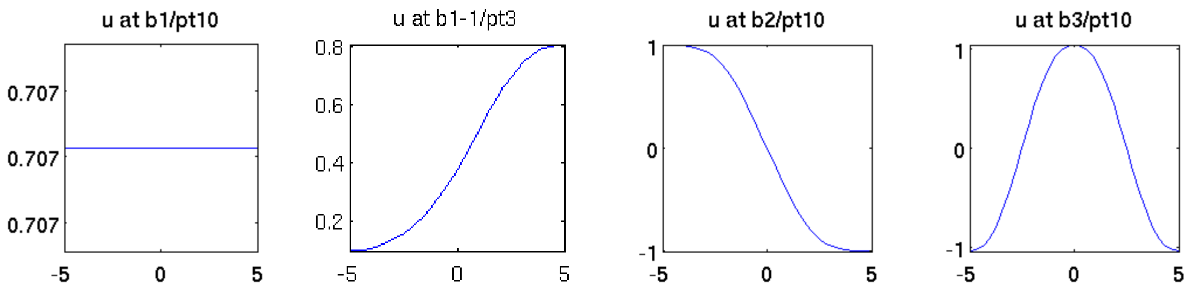


Figure 3: Example solution plots (Cell 8 from `ac1D_simple/cmds.m`).

3 Some 1D extensions: ac1D, ac1Dnlbc, and ac1Dx

We now extend `ac1d_simple` in four directions, namely

- (linear and nonlinear) Dirichlet boundary conditions (BC).
- mesh-refinement based on suitable a posteriori error estimates: although this is not an issue for simple 1D problems, it can greatly improve efficiency and reliability.
- fold-continuation, which is useful to understand parameter dependence of bifurcation diagrams and to detect codimension-two points
- x–dependent terms, which illustrates the flexibility of the software.

Moreover, we give some examples of parameter switching. To explain these issues we only comment on the changes compared to `ac1D_simple`.

3.1 ac1D: Dirichlet BC (linear)

In `acsuite/ac1D` we consider (2) with the BC

$$u|_{x=-5} = 0, \quad u|_{x=5} = d \tag{18}$$

with parameter d added to the parameter list. For $d = 0$ we still have the trivial branch with bifurcation points at

$$\lambda_j = (j\pi/10)^2, \quad \text{eigenfunctions } \phi_j(x) = \sin(j\pi(x+5)/10), \quad j = 1, 2, \dots. \tag{19}$$

There are three new functions in `ac1D`, namely

- `nodalf.m`, with signature `f=nodalf(p,u)`. This outsources the computation of the nonlinearity $f(u) = \lambda u + u^3 - \gamma u^5$, since it is now needed in two places: `sG.m`, as before, and additionally in the error-estimate function `e2rs.m`.
- `e2rs.m` (elements2refine selector), with signature `[p,idx]=e2rs(p,u)`, which besides `p` should return a list `idx` of elements to be refined.
- `spjac.m`, with signature `Guuph=spjac(p,u)`, which computes the derivative $\partial_u(\partial_u G\phi)$ where ϕ is in $\ker\partial_u G(u)$ at a fold point u; see [DRUW14, §2.1.4].

To set up the Dirichlet BC (18) we need to first modify `oosetfemops`, see Listing 6, and subsequently `sG` and `sGjac`, see Listings 7 and 12. Table 3 summarizes all functions used in `ac1D` in a prototypical form for all subsequent demos.

Table 3: Scripts and functions in `ac1D`.

script/function	purpose, remarks
cmds1,cmds2,cmds3, cmds4	scripts: `cmds1` computes primary bifurcations from the trivial branch, and illustrates continuation in other parameters; `cmds2` contains mesh-refinement and fold continuation; `cmds3` gives an example of branch point continuation, and `cmds4` illustrates the use of multiple initial guesses to find different branches
p=acinit(p,lx,nx,par)	initialization
p=oosetfemops(p)	sets FEM matrices (stiffness K, mass M, and matrices/vectors Q,G for the BC)
r=sG(p,u)	encodes G from (2) (including the BC)
f=nodalf(p,u)	nodal nonlinearity f, called in `sG` and `e2rs`
Gu=sGjac(p,u)	Jacobian $\partial_u G(u)$ of G
[p,idx]=e2rs(p,u)	classical elements2refine selector (as in pdejmps of the pde-toolbox)
Guuphi=spjac(p,u)	used to assemble $\partial_u(\partial_u G(u)\phi)$ from (21)

```
function p=oosetfemops(p)
gr=p.pdeo.grid; fem=p.pdeo.fem; % just introduced as shorthands
[K,M,~]=fem.assema(gr,1,1,1); p.mat.K=K; p.mat.M=M; % indep. of BC
bcl=gr.robinBC(1,0); bcr=gr.robinBC(1,1); % 2 different BC (left and right)
gr.makeBoundaryMatrix(bcl,bcr); % intermediate step before assembling BC matr.
[Q,G,~,~]=fem.assemb(gr); p.mat.Q=Q; p.mat.G=G; % the BC matrices
p.nc.sf=1e3; % stiff spring constant for DBC via Robin-BC
```

Listing 6: `ac1D/oosetfemops.m` Lines 2 and 3 are as in `ac1d_simple`, where we compute K, M, which are independent of the BC. The set up of the BC starts in line 4, where we use the `OOPDE` method `robinBC` to prepare the left $u|_{x=-5} = 0$ and right $u|_{x=-5} = d$ BC. Note that the parameter d does not appear here, but in `sG.m` and `sGjac.m`. In lines 5-7 we then assemble and save in `p.mat` the matrix Q and the vector G_{BC}, and choose a stiff spring factor s (see (8))

```
function r=sG(p,u)  % PDE rhs
f=nodalf(p,u); % compute "nonlinearity" (everything but diffusion)
par=u(p.nu+1:end); u=u(1:p.nu); % split in par and PDE u
r=par(1)*p.mat.K*u-p.mat.M*f...   % bulk part of PDE
  +p.nc.sf*(p.mat.Q*u-par(4)*p.mat.G); %  BC
```

Listing 7: `ac1D/sG.m`: line 2 calls `nodalf.m` to compute the nonlinearity, line 3 splits u into the PDE-variables and the parameters. Lines 4,5 compute G: line 4 the part that is independent of the BC, and line 5 the BC-dependent part, where the parameter d is at par(4).

```
function f=nodalf(p,u)  % nodal nonlinearity f, called in sG and e2rs
par=u(p.nu+1:end); u=u(1:p.nu); f=par(2)*u+u.^3-par(3)*u.^5;
```

Listing 8: `ac1D/nodalf.m`: split u into the PDE variables and the parameters, then compute f.

```
function Gu=sGjac(p,u) % Jacobian
par=u(p.nu+1:end); u=u(1:p.nu); % split into parameters and PDE variables
fu=par(2)+3*u.^2-5*par(3)*u.^4; % compute derivative of nonlinearity
Fu=spdiags(fu,0,p.nu,p.nu);     % and convert to matrix
Gu=par(1)*p.mat.K-p.mat.M*Fu+p.nc.sf*p.mat.Q; % build Jac from bulk and BCs
```

Listing 9: `ac1D/sGjac.m`, following from `sG.m` in a straightforward way.

```
function p=acinit(p,lx,nx,par) % init with prep. of mesh-adaption and usrlam
p=stanparam(p); screenlayout(p); p.nc.neq=1;
p.sw.sfem=-1; p.fuha.sG=@sG; p.fuha.sGjac=@sGjac; p.fuha.e2rs=@e2rs;
```

```
pde=stanpdeo1D(lx,2*lx/nx); p.pdeo=pde; % domain and mesh
p.np=pde.grid.nPoints; p.nu=p.np; p.sol.xi=1/(p.nu);
[po,t,e]=getpte(p);  p.mesh.nt=size(t,2); % next: background-mesh for legacy
p.mesh.bp=po; p.mesh.bt=t; p.mesh.be=e; % setup of oomeshadac, now obsolete
p.u=zeros(p.np,1); p.u=[p.u; par']; % initial guess, and parameters
p.usrlam=-0.5:0.5:1; % compute point and write to disk at these par values
```

Listing 10: `ac1D/acinit.m` (lines 1-9). This has minor changes compared to the `ac1d_simple` version only, mainly due to the intended mesh-adaptation. In line 3 we additionally set the `p.fuha.e2rs` function handle, and in line 7 we save the starting mesh to serve as a 'background mesh': in order to keep the number of elements low, the mesh adaptation (if requested by the user, or to fulfill an error bound) via `meshadac` in the legacy setup first interpolates (coarsens) to the background mesh, which is then refined. This is essentially superseded by a different ad-hoc coarsening, see cmds3.m and Remark 3.1. In line 9 we fill `p.usrlam=[-0.5 0 0.5 1]`; whenever the active parameter crosses a value λ_0 from `p.usrlam`, a solution is computed at the exact value λ_0. The remainder of `acinit` is as before.

The tutorial starts with the script file `cmds1.m`, where `cmds1.m` differs significantly from the `ac1d_simple` version in line 4 only: `par=[1 -0.2 1 0 0]`, whose first three entries stand for the parameters c, λ, γ, the fourth for the new parameter d initialized to 0, and the fifth for an additional parameter also initialized to 0 that will be used later. In cell 8 we then illustrate the use of `swiparf.m` which is used to switch to a different active continuation parameter. Similarly, in cell 9 we continue in the boundary value parameter d. Some results of running `cmds1.m` are plotted in Fig. 4(a-d).

```
%% c8: continue in some other param, here in diffusion constant par(1)
% to smaller values, which corresponds to stretching of the domain!
p=swiparf('b1','pt11','b1-cc',1); p.sol.ds=-0.1;
p.nc.lammin=0.15; p.nc.lammax=2; clf(2); p.usrlam=0.3;  p=cont(p,10);
plotsol('b1-cc','pt8',1,1,1); axis([-5 5 -0.1 1.2]); % solution plot
%% c9: continue in boundary coefficient
p=swiparf('b1','pt10','b1-dc',4); p.sol.ds=0.1;
p.nc.lammin=-1; p.nc.lammax=2; p=resetc(p); clf(2); p=cont(p,10);
plotsol('b1-dc','pt0',1,1,1); pause; plotsol('b1-dc','pt10',1,1,1);
```

Listing 11: Selection from `ac1D/cmds1.m`, illustrating the use of `swiparf` to continue in a different parameter, namely the diffusion constant c, which is the first parameter, and the boundary coefficient d, which is the fourth; see Fig. 4(c,d) for example plots.

3.1.1 Mesh adaptation

In the `OOPDE` setting, the setup of an appropriate function for the selection of elements to refine is the responsibility of the user, but there is a standard option: compute an element-wise error estimate similar to the `pdetoolbox` function `pdejmps`, which requires the diffusion tensor c, the 'matrix' a (here always $a = 0$), and the nonlinearity f, hence `nodalf`, and then generate the element list `idx` via the Dörfler algorithm [Dör95]. See Listing 12 for an example. However, the user can also generate `idx` in any other way – as an illustration, we include an ad hoc function `e2rs_ad_hoc` in the `ac1D` demo. Cells 1,2 of `cmds2.m` (see Listing 15) contain basic mesh-adaptation calls. Some output results are shown in Fig. 4(e,f).

```
function [p,idx]=e2rs(p,u)  % classical elements2refine selector as in pdejmps
par=u(p.nu+1:end); c=par(1); a=0; fv=nodalf(p,u); u=u(1:p.nu);
E=p.pdeo.errorInd(u,c,a,fv); p.sol.err=max(max(E));
```

```
idx=p.pdeo.selectElements2Refine(E,p.nc.sig);
```

Listing 12: `ac1D/e2rs.m`. In line 2 we set up the diffusion tensor c, the linear part a, compute f (we choose $a = 0$ and put 'everything but c' into f), and as usual split u into u and the parameters. The `OOPDE` method errorInd in line 3 then computes an error estimate as `pdejmps`. In line 4 we save the max error in `p.sol.err` for later use, and employ `selectElements2Refine` (cf. the `pdetoolbox` routine `pdeadworst`) to select the elements to be refined.

```
%% c1: some mesh-refinement, here just for one fixed solution
p=loadp('b1','pt10','b1'); p.nc.maxt=100;
%p.fuha.e2rs=@e2rs_ad_hoc; % uncomment to use ad hoc element selector
p=meshada(p,'ngen',3,'sig',0.5);
%% c2: continuation with mesh-adaption depending on error bound
p=swibra('tr','bpt1','b1ref',0.1); p.nc.sig=0.95; % large sig gives fine mesh!
p.nc.lammax=2; p.sw.errcheck=2; p.nc.errbound=0.2; p=cont(p,40);
figure(3); clf; plotbra(p,3,-1,'lsw',0,'labi',5); % plot error-est on branch
%% c3: continuation with mesh-adaption each amod-th step
p=swibra('tr','bpt1','b1ref',0.1); p.nc.sig=0.95;
p.sw.errcheck=0; p.nc.amod=5; p.nc.maxt=200; p.nc.lammax=2;  p=cont(p,40);
figure(3); clf; plotbra(p,3,-1,'lsw',0,'labi',3); % plot error-est on branch
```

Listing 13: `ac1D/cmds2.m` (Cells 1-3). Cell 1 illustrates mesh-adaptation on a single solution, while in cell 2 we use continuation with mesh-adaptation whenever the error-estimate exceeds the bound p.nc.errbound; see Fig. 4(e) for the error plot, and (f) for a plot of the solution on the adapted mesh near $\lambda = 0.7$. This mesh-adaptation proceeds by first interpolating the current solution to the background mesh (see l6,7 in Listing (9)), which is then refined.

Remark 3.1 In the context of continuation we do not aim at just repeated refinements, which would quickly lead to a too large number of mesh points. Instead, before a new refinement we need a coarsening step. Originally, this was implemented by interpolating the solution on the current mesh down to a somewhat coarse background mesh, which is basically uniform and kept fixed, and this coarse solution is then used for refinement. This sometimes is problematic as the refinement of the coarse solution may fail, or may lead to jumping to a different branch, which is usually even more problematic as it may go unnoticed. In 2D and 3D, we therefore now usually recommend mesh–adaption by `trullekrul`, see [Uec19c].

In 1D, we changed to a simple ad-hoc coarsening step via `p=simplecoarse(p,aux)`. By default, this coarsens the current mesh by removing every second mesh–point, unless the elements (intervals) adjacent to the point are already coarse, i.e., already have length> `dxmax`, which is either read from `p.nc.dxmax`, or can be passed in `aux.dxmax`, or otherwise defaults to `dxmax=1`. The legacy setup with interpolation to a background mesh can still be used in 1D by setting `p.sw.scoarse=0`. See the last cell of `cmds2.m`. For the simple problem considered here, both methods perform essentially equally, but in general we recommend the new version with `simplecoarse`.

The parameters for the refinement step (in 1D, and 2D if the legacy setup is used instead of `trullekrul`) are as follows.

- `p.nc.sig` $\in (0,1)$: this controls the elements–to–refine selection rule from the error estimator (larger σ means more elements are selected);
- `p.nc.ngen`: this gives the number of refinement generations;
- `p.nc.maxt`: refinement is stopped if the current mesh has more than `p.nc.maxt` elements (intervals in 1D, triangles in 2D).

Additionally, if p.nc.amod > 0, then mesh–adaption is done each amod$^{\text{th}}$ step, and if p.nc.amod $= 0$ but p.nc.errbound > 0, then adaption is done when p.sol.err, which should be estimated in p.fuha.e2rs, exceeds p.nc.errbound. ⌋

3.1.2 Fold and branch point continuation

Fold point continuation is described in [DRUW14, §2.1.4] and requires a second free parameter. Here γ is chosen. Introducing the shorthand $w = (\lambda, \gamma)$ it works via the extended system

$$H(U) = \begin{pmatrix} G(u,w) \\ \partial_u G(u,w)\phi \\ \|\phi\|_{L^2}^2 - 1 \\ p(U) \end{pmatrix} = 0, \quad U = (u, \phi, w), \tag{20}$$

such that ϕ is in the kernel of $\partial_u G$ with L^2-norm constrained to 1 by the third equation, and $p(U) = 0$ is the usual arclength equation. For continuation of (20) we need the Jacobian

$$D_U H(U) = \begin{pmatrix} \partial_u G & 0 & \partial_w G \\ \partial_u(\partial_u G\phi) & \partial_u G & \partial_w(\partial_u G\phi) \\ 0 & 2\phi^T & 0 \\ \partial_u p & \partial_\phi p & \partial_w p \end{pmatrix}. \tag{21}$$

The last line of $D_U H(U)$ is generated automatically, and also $\partial_w G$, $\partial_w(\partial_u G\phi)$ are obtained quickly by finite differences, but for efficiency it is highly recommend to implement a function for $\partial_u(\partial_u G\phi)$. This is quite easy here, see Listing 14, but may become somewhat involved for more complicated systems. Therefore we provide the function `[Ja,Jn]=spjaccheck(p)`, which compares the (user provided) 'analytical' $\partial_u(\partial_u G\phi)$ (Ja) with its finite difference approximation (Jn). In a correct implementation of $\partial_u(\partial_u G\phi)$, the relative error should be on the order of 10^{-6} or smaller.

In order to initialize FP or BP continuation, we need to double the numbers of unknowns to $U = (u, \phi, w)$ and initialize ϕ in (20). For this we use the function `p=spcontini(varargin)`, for which there are the following calling syntaxes:

- `p=spcontini('dir','pt',new_active,'newdir');` where `'dir'`, `'pt'` is the usual directory/point combination, `new_active` is the new active parameter (here γ for `new_active`=1), and `'new_dir'` a new output directory. This also resets counters and the branch data (by default).
- `p=spcontini('dir','pt',new_active,'newdir',resetsw);` with `resetsw`=0 does not reset the branch data, which is useful for BP and FP localization, see Remark 3.2.
- `p=spcontini(p,new_active,'newdir')` does not read a file from disk, and also has the version `p=spcontini(p,new_active,'newdir',resetsw)`

The converse function, switching back to regular continuation, is `spcontexit` and has similar argument lists. An example of fold continuation is given in Cells 3-5 of `cmds2.m` (Listing 15), and example outputs in Fig.4(g,h), and an example of branch point continuation in `cmds3.m` (Listing 16), with plots in Fig. 5.

```
function J=spjac(p,u) % \pa_u (G_u phi), called in getGu if p.sw.spcont=1
par=u(2*p.nu+1:end); phi=u(p.nu+1:2*p.nu); u=u(1:p.nu); % params, Evec, PDE-vars
fuu=6*u-20*par(3)*u.^3; J=-p.mat.M*spdiags(fuu.*phi,0,p.nu,p.nu);
```

Listing 14: `ac1D/spjac.m`, used in fold continuation. Note that fold continuation doubles the numbers of unknowns to (u, ϕ), where ϕ is in the kernel at the fold, hence the parameters now start at `2*p.nu+1`.

```
%% c1: fold-continuation
p=spcontini('b1','fpt1',3,'b1f');    % init fold cont with par 3 new prim. par
p.plot.bpcmp=p.nc.ilam(2); figure(2); clf; % use this new param. for plotting
p.sol.ds=-0.01;                       % new stepsize in new primary parameter
p.sw.spjac=1; p.fuha.spjac=@spjac; % spectral jac
[Ja, Jn]=spjaccheck(p); pause % check impl. of spjac (comment out when fine)
tic; p=cont(p); toc
clf(3); plotbraf('b1f','pt20',3,2,'lab',15); % plot BD for fold-cont
%% c2: switch back to regular continuation from one of the fold points
p=spcontexit('b1f','pt15','b1-a'); p.nc.dsmax=0.2; p.sw.bifcheck=0;
p.plot.bpcmp=0; p.nc.lammin=-5; p.sol.ds=1e-3; clf(2);
p=cont(p,1); p=cont(p,35); % cont. in one direction, 1 initial step for saving
```

Listing 15: `ac1D/cmds3.m` (Cells 1,2). We continue the fold-point on b1 (including it's λ value) in the quintic coefficient γ, see Fig. 4(h). Line 6 gives an example of `spjaccheck`, and should be commented out when (as here) the implementation of spjac is OK. In cell 2 we to switch back to regular continuation in λ. For this we use `spcontexit`. In the given form with three arguments, this deletes all branch data except the last point, resets counters, sets the output to the last argument as a new directory, and saves the fold point to this directory. We then also reset some other data and switches, and first do just one step in one direction, to save the first point, and then a number of further steps. The initial step is needed (if `p.sw.smod`$\neq 1$) because in the next cell we want to load point 1 and continue in the other direction. The remainder of `cmds3.m` deals with plotting.

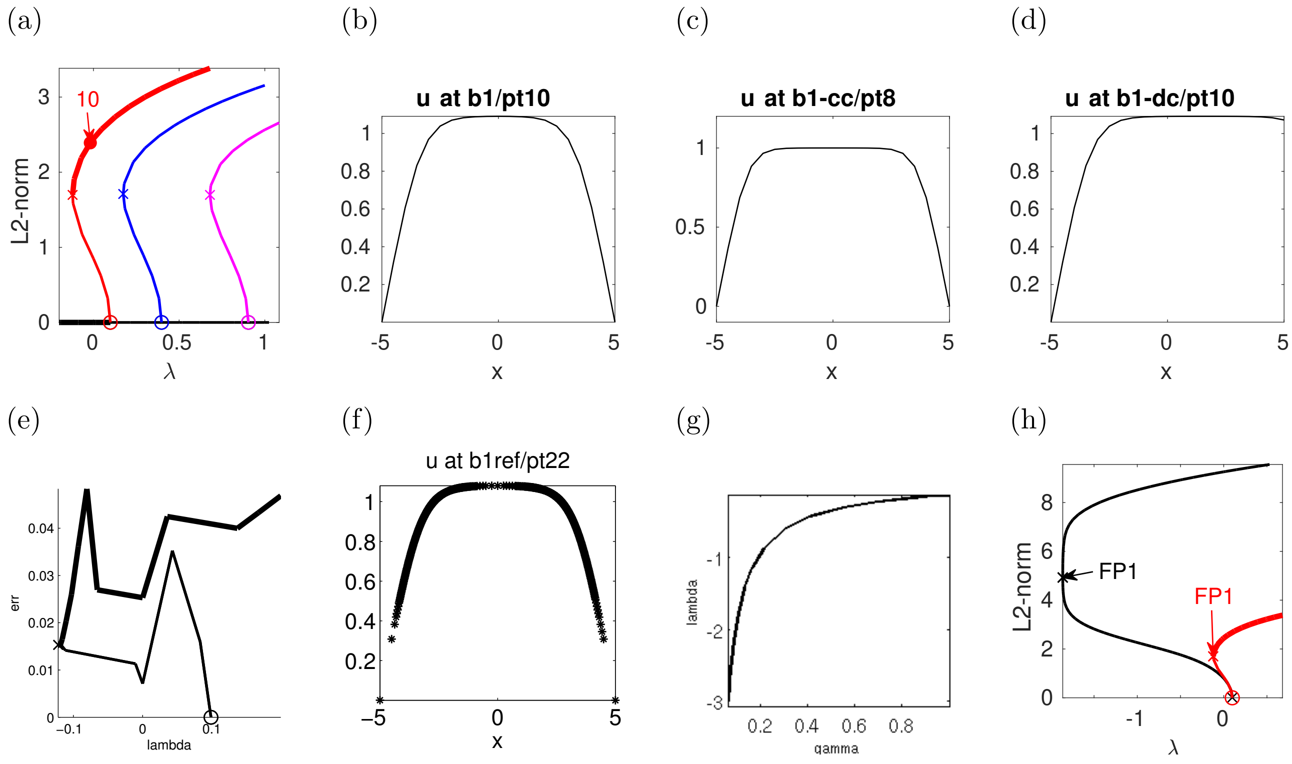


Figure 4: Selection of plots generated in `ac1D/cmds1.m` (a-d) and `ac1D/cmds2.m`. (a),(b) Bifurcation diagram and a selected solution plot for continuation in λ; b1=first nontrivial branch (red). (c),(d) results for continuation in c and d. (e) continuation of the first nontrivial branch with mesh adaptation based on error-bound. (f) Solution after mesh-adaptation near $\lambda = 0.5$ in (e). (g) continuation of the first fold point from (a) (i.e., fold-position λ_f as a function of γ). (h) continuation in λ from the fold point for $\gamma \approx 0.06$ in (g), together with the branch b1 (red) from (a).

Similar to (20), BP continuation can be based on the extended system

$$H(U) = \begin{pmatrix} G(u,\lambda) + \mu M\psi \\ G_u^T(u,w)\psi \\ \|\psi\|_2^2 - 1 \\ \langle \psi, G_\lambda(u,w)\rangle \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad U = (u,\psi,w), \tag{22}$$

where (u,λ) is a (simple) BP (for the continuation in λ), ψ is an adjoint kernel vector, $w = (\lambda,\mu)$ with $w_1 = \lambda$ the primary active parameter and $w_2 = \mu$ as additional active parameter, see `cmds3` and Fig. 5 for example usage and results.

```
%% c5 - branch point continuation, here in c for first BP on trivial branch
p=bpcontini('tr','bpt1',1,'bpcon'); % init branch point cont. in par(1) (c)
p.sw.bifcheck=0; p.plot.bpcmp=2; % plot lam over c now
p.sol.ds=-0.1; p.fuha.spjac=@spjac; p.usrlam=0.3; clf(2); p=cont(p,10);
%% c6 - plot lam-pos over c
figure(3); clf; plotbra('bpcon', 'pt10');
```

Listing 16: Further cells from `ac1D/cmds3.m`, continuation of the first branch-point on b1 in the diffusion coefficient c. To switch back to regular continuation, use `bpcontexit`. See the remainder of `cmds3.m`.

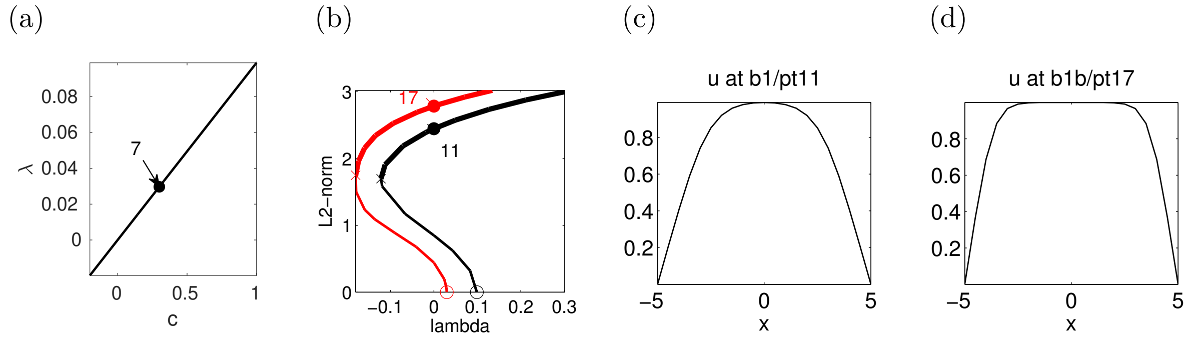


Figure 5: Branch point continuation, plots generated in `ac1D/cmds3.m`. (a) BP position λ over c. (b) old ($c = 1$, black) and new ($c = 0.3$, red) first nontrivial branches. (c,d) solution plots at $\lambda = 0$.

Remark 3.2 The extended system (20) and (22) are regular at simple FPs and BPs, respectively, see [Moo80, Mei89, Mei00], and thus can be used for localization of (simple) FPs or BPs if sufficiently good initial guesses are available. For instance, given a point (u_0, λ_0) presumably near a FP, we can use a Newton loop on

$$H(U) = \begin{pmatrix} G(u,\lambda) \\ \partial_u G(u,\lambda)\phi \\ \|\phi\|_{L^2}^2 - 1 \end{pmatrix} = 0, \quad U = (u,\phi,\lambda), \tag{23}$$

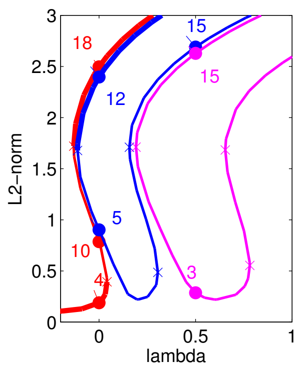
to converge to a solution (u,ϕ,λ). Here we do not regularly use this option, as the FP and BP localization via bisection is very reliable and for most problems sufficiently fast. However, for more extensive problems ($N * n_p$ on the order of 10^5, say), bisection becomes slow, and FP and BP localization via (23) (or (22)) becomes an important alternative. See §4.2 and [UW17] for examples. ⌋

3.1.3 Imperfect bifurcations

So far all our solution branches were connected to a trivial branch, so that continuation of this branch in one ore more parameters, and subsequent switching of branches at bifurcation points yields all solutions of interest. This works well for (2) with homogeneous Dirichlet or Neumann BC due to the presence of the symmetry $G(-u) = G(u)$, which generates pitchfork bifurcations from the trivial branch. However, if such symmetries are broken, the connectedness of branches is typically destroyed, though for weak symmetry breaking branches come close to each other without intersecting, which is called imperfect bifurcation.

As an example we consider (2) with inhomogeneous Dirichlet BC $u(-l_x) = 0$ and $u(l_x) = 0.1$. In this case we have $G(-u) \neq G(u)$ (taking the BC into account), and $u \equiv 0$ is no longer a solution. However, we may still take $u_0 \equiv 0$ as an initial guess for a solution (at different λ), and try a Newton loop for $G(u, \lambda) = 0$ to converge to a solution $u = u(\lambda)$. This is implemented in `cmds4.m`, see Fig.6 for some results.

(a)



(b)

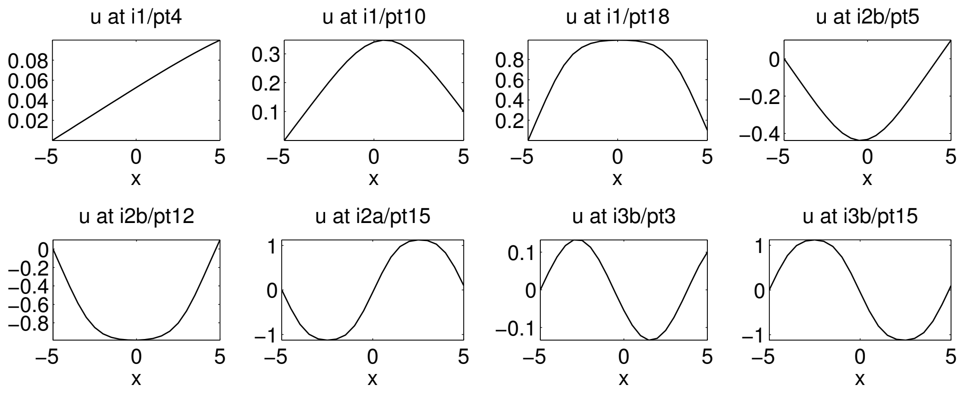


Figure 6: Imperfect bifurcations for (2) with $u(-l_x) = 0$ and $u(l_x) = 0.1$, plots generated in `ac1D/cmds4.m`. To find the 3 non-intersecting branches i1 (red), i2 (blue) and i3 (magenta), we use the initial guess $u \equiv 0$ at $\lambda = -0.2, \lambda = 0.2$ and $\lambda = 0.6$, respectively. The parts not close to $u \equiv 0$ are clearly perturbations of the respective branches with $u(l_x) = 0$ from Fig. 4(a).

3.2 ac1Dnlbc: nonlinear boundary conditions

In principle, the matrix $Q_{\rm BC}$ and the vector $G_{\rm BC}$ encoding the BCs in (8) can depend on u (and, e.g., the spatial variables x, y, see the following sections). Notably, simple nonlinear BCs an also be implemented within the `sG` routine with *fixed* matrices as in §3.1. In general, one generates sets of $Q_{\rm BC}$ and $G_{\rm BC}$ in `oosetfemops.m` as identifiers of the boundary nodes for each part of the desired functional dependency. Since $Q_{\rm BC} \in \mathbb{R}^{n_u \times n_u}$ and $G_{\rm BC} \in \mathbb{R}^{n_u}$ are sparse this is computationally more efficient than reassembling matrices for each evaluation of `sG`.[4]

Here we consider the extension of (18) to

$$u|_{x=-5} = 0, \quad (u + \alpha u^2)|_{x=5} = d, \tag{24}$$

with new parameter α and modify `oosetfemops.m, sG, sGjac` and `spjac` as given in Listings 17–20. In Fig. 7 we plot some results from the short `cmds.m` file, where $\alpha = 2$ and just a continuation in d is done, and a fold continuation.

[4]Examples where the BCs depend on a parameter in a way that requires assembling in each step are give in `ac2Dwspot` and `ac3Dwspot`, see [Uec19c].

```
function p=oosetfemops(p) % for ac1Dnlbc: setting up two sets of (Q,G)
gr=p.pdeo.grid; p.nc.sf=1e3;
[K,M,~]=p.pdeo.fem.assema(gr,1,1,1); p.mat.K=K; p.mat.M=M; % indep. of BC
bcl=gr.robinBC(1,1); bcr=gr.robinBC(0,0); % BCs for the first set (Q1, G1) of
% BC matrices which will control the left BCs, i.e., Q1,G1=0 at the right
gr.makeBoundaryMatrix(bcl,bcr); [p.mat.Q1,p.mat.G1,~,~]=p.pdeo.fem.assemb(gr);
bcl=gr.robinBC(0,0); bcr=gr.robinBC(1,1); % BCs for right. Now Q2,G2=0 at the
% left, such that multipl. of Q2,G2 with param only affects the right
gr.makeBoundaryMatrix(bcl,bcr); [p.mat.Q2,p.mat.G2,~,~]=p.pdeo.fem.assemb(gr);
```

Listing 17: `ac1dnlbc/oosetfemops.m`. Q1, G1 will encode the left BC $u|_{x=-5} = 0$ from (24), while Q2, G2 will encode the right BC $(u + \alpha u^2)|_{x=5} = 0$.

```
function r=sG(p,u)  % rhs for ac1Dnlbc
f=nodalf(p,u); par=u(p.nu+1:end); d=par(4); al=par(5); beta=par(6);
u=u(1:p.nu); K=par(1)*p.mat.K;
r=K*u-p.mat.M*f...    % bulk part of PDE
  +p.mat.Q1*u-beta*p.mat.G1 ... % left BCs, i.e., Q1,G1=0 at right
  +p.nc.sf*(p.mat.Q2*(u+al*u.^2)-par(4)*p.mat.G2); % right BCs
```

Listing 18: `ac1dnlbc/sG.m`, using Q1,G1 and Q2,G2 from Listing 17 to encode the different dependencies on u at the left and right boundaries

```
function Gu=sGjac(p,u)  % Jac for ac1Dnlbc (includes D_u of BC)
par=u(p.nu+1:end); al=par(5); u=u(1:p.nu);
fu=par(2)+3*u.^2-5*par(3)*u.^4; Fu=spdiags(fu,0,p.nu,p.nu);
Gu=par(1)*p.mat.K-p.mat.M*Fu...
   +p.mat.Q1+p.nc.sf*spdiags(1+2*al*u,0,p.nu,p.nu)*p.mat.Q2;
```

Listing 19: `ac1dnlbc/sGjac.m`, which follows straightforwardly from Listing 18.

```
function Guuphi=spjac(p,u) % for ac1Dnlbc
par=u(2*p.nu+1:end); phi=u(p.nu+1:2*p.nu); u=u(1:p.nu); % params, Evec, PDE-vars
fuu=6*u-20*par(3)*u.^3; al=par(5);
Guuphi=-p.mat.M*spdiags(fuu.*phi,0,p.nu,p.nu)...   % from the bulk
       +p.nc.sf*spdiags(2*al*p.mat.Q2*phi,0,p.nu,p.nu); % from the BC
```

Listing 20: `ac1dnlbc/spjac.m`. The right BC contains u^2, thus its 2nd derivative appears in $\partial_u(\partial_u G\phi)$.

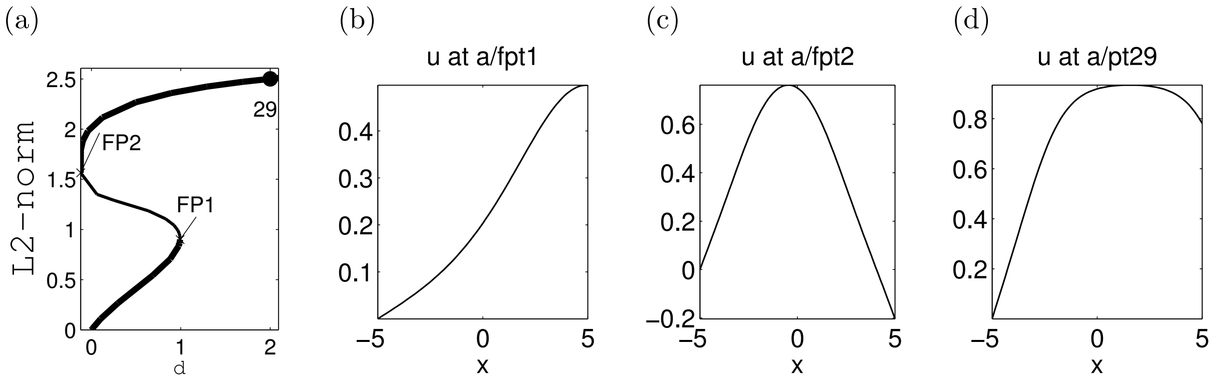


Figure 7: Results for (2) with nonlinear BC (24) ($\alpha = 2$) from `ac1Dnlbc/cmds.m`.

3.3 x-dependent terms

As an example for the setup of x-dependent terms we consider the modified problem

$$G(u) := -\mathrm{div}(c(x)\nabla u) - \lambda u - u^3 + \gamma u^5 - 0.5xu \overset{!}{=} 0, \tag{25}$$

on $\Omega = (-5, 5)$, where $c(x) = 1+0.1x^2$, now again with homogeneous NBC, and where (1D) $\nabla u = \partial_x u$ and div $= \partial_x$.

3.3.1 ac1Dxa

In the (divergence) form (25), there are only three relevant changes compared to `ac1D`: in `oosetfemops.m`, to implement the now x–dependent stiffness matrix K, and in `sG.m` as well as in `sGjac.m`, where we add the x-dependent term $0.5xu$. See demo-dir `ac1Dxa`, and Listings 21-23.

```
function p=oosetfemops(p) % x-dependent stiffness matrix
x=getpte(p); c=1+0.1*x.^2; % extract the point coord. from p and set c
gr=p.pdeo.grid; [K,M,~]=p.pdeo.fem.assema(gr,c,1,1); p.mat.K=K; p.mat.M=M;
```

Listing 21: `ac1Dxa/oosetfemops.m`. In l2 we extract the x coordinates of the grid points from `p`. This is a short syntax for `[points,tr,ed]=getpte(p);` where `points` are the mesh points, `tr` are the 'triangles' (1D: intervals), and `ed` are the edges (no meaning in 1D).

```
function r=sG(p,u)  % AC with x-dependent terms (K(x) from oosetfemops)
par=u(p.nu+1:end);  u=u(1:p.nu); % split u into parameters and PDE vars
x=getpte(p); x=x'; % extract the point coordinates from p and transpose
f=par(2)*u+u.^3-par(3)*u.^5+0.5*x.*u; % f, with x-dependent term
r=p.mat.K*u-p.mat.M*f;  % bulk part of PDE
```

Listing 22: `ac1Dxa/sG.m`. Like in `oosetfemops` we first extract the point-coordinates; after transposition (since coordinates are row vectors) these can be used to set up the nonlinearity as expected.

```
function Gu=sGjac(p,u) % jac for AC with x-depend. terms (K(x) from oosetfemops)
par=u(p.nu+1:end); u=u(1:p.nu);
x=getpte(p); x=x'; fu=par(2)+3*u.^2-5*par(3)*u.^4+0.5*x;
Gu=p.mat.K-p.mat.M*spdiags(fu,0,p.np,p.np);
```

Listing 23: `ac1Dxa/sGjac.m`, following immediately from `sG` in Listing 22.

Because the x–dependent terms in (25) only occur linearly, `spjac.m` stays exactly as in `ac1D`, and similarly the script `cmds.m` works as before, including mesh-adaptation and fold-continuation. Figure 8 shows a basic bifurcation diagram and selected solution plots. One effect of the x-dependent term xu in (25) is that the first bifurcation now occurs at lower λ, and that this bifurcation is, in a heuristic sense, on the "right of the domain".

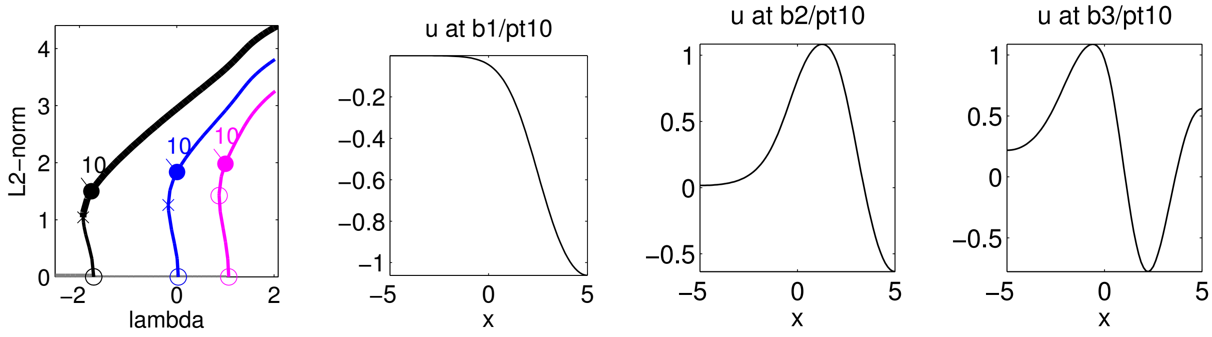


Figure 8: Results for (25) from `ac1Dxa/cmds.m` (homogeneous NBC). Bifurcation diagram, and example plots on first three bifurcating branches.

3.3.2 ac1Dxb

If $c \in C^1(\Omega, \mathbb{R})$, then (25) can alternatively be written in the form

$$G(u) := -c(x)\Delta u - c'(x)\nabla u - \lambda u - u^3 + \gamma u^5 - 0.5xu \overset{!}{=} 0, \tag{26}$$

which in some cases may be more convenient to implement in `pde2path`, for instance, if the function c contains parameters. In this case, in `oosetfemops` we assemble the constant coefficient diffusion matrix `p.mat.K` as before, and additionally a matrix `p.mat.Kx` which corresponds to the convective derivative ∇, and use these two matrices with x–dependent prefactors in `sG.m` (and `sGjac.m`), see Listings 24–25. For (25) the results are then exactly as in `ac1Dxa` (as should be).

```
function p=oosetfemops(p) % for AC1Dxb: x-dep. diffusion in sG and sGjac,
gr=p.pdeo.grid; [K,M,~]=p.pdeo.fem.assema(gr,1,1,1); p.mat.M=M; p.mat.K=K;
b=1; p.mat.Kx=p.pdeo.fem.convection(gr,b); % convection coefficient and matrix
```

Listing 24: `ac1Dxb/oosetfemops.m`. We assemble x-independent matrices K (Neumann Laplacian) and Kx (convective derivative), and then implement the prefactors in `sG`. For completeness we remark than replacing line 5 by `x=getpte(p); b=0.5*x;` or simply `b='0.5*x';` (a string expression) would also be possible (similar to `c=1+0.1*x.^2;` in `ac1Dxa/oosetfemops`), in which case we would change `sG.m` and `sGjac.m` accordingly.

```
function r=sG(p,u) % AC with x-dependent terms; const.coeff. K in oosetfemops,
% thus x-dependent diffusion implemented here
par=u(p.nu+1:end); n=p.nu; u=u(1:n); % split u into parameters and PDE vars
x=getpte(p); x=x'; % extract point coordinates from p
f=par(2)*u+u.^3-par(3)*u.^5+0.5*x.*u; % f, with x-dependent term
F=p.mat.M*f;
r=(spdiags(1+0.1*x.^2,0,n,n)*p.mat.K-spdiags(0.2*x,0,p.np,p.np)*p.mat.Kx)*u-F;
```

Listing 25: `ac1Dxb/sG.m`. Explicit x dependence implemented here instead of in `oosetfemops`.

4 Higher space dimensions

4.1 2D

In `ac2D/cmds1` we consider (2) on the rectangle $\Omega = (-2\pi, 2\pi) \times (-\pi, \pi)$, with the Dirichlet BC

$$u = d\cos(y/2) \text{ on } \{x = 2\pi\}, \text{ parameter } d, \tag{27}$$

and $u = 0$ on the remaining boundary. For $d = 0$ we have the bifurcation points

$$\lambda_{jl} = (j/4)^2 + (l/2)^2, \quad \phi_{jl} = \sin(j(x + 2\pi)/4)\sin(l(y + \pi)/2), \quad j, l = 1, 2, \ldots \tag{28}$$

from the trivial branch. The only two essential changes compared to `ac1D` are

- setting up the 2D domain, which has 4 boundary segments;
- setting up the y–dependent BC (27), on the right boundary segment.

This can be readily done by modification of `acinit.m` and `oosetfemops.m` from **ac1D**, while `sG.m`, `nodalf.m`, `sGjac.m` and `e2rs.m` (for mesh-refinement) as well as `spjac.m` (for fold-continuation) stay exactly as in **ac1D**. Additionally, in `femtest.m` we compare the `sfem=1` solutions with the `sfem=0` (classical FEM) solutions, and therefore in line 8 of `acinit.m` we introduce a switch, see also `res_cl.m` in Listing 29. In the (L^2) bifurcation diagrams, the branches for both methods are indistinguishable, and Figure 9(f) shows how the error between the two methods decays for finer meshes. Also, we can still combine `sGjac` with the classical setup. The script-file `cmds1.m` also differs in plotting from the 1D version since in 2D there are 4 styles for solution plotting: 0=only mesh, 1=3D meshgrid, 2=2D density plot, 3=3D surface; see Fig. 9 (b)-(e) for examples.

```
function p=acinit(p,lx,ly,nx,par,varargin) % ac2D
sw.sym=0; if nargin>5; sw=varargin{1}; end % switch for meshing symmetry
p=stanparam(p); screenlayout(p); p.nc.neq=1; p.sw.sfem=-1;
p.fuha.sG=@sG; p.fuha.sGjac=@sGjac; p.fuha.e2rs=@e2rs;
pde=stanpdeo2D(lx,ly,2*lx/nx,sw); % h as argument
%pde=stanpdeo2D(lx,ly,nx,round(nx*ly/lx)); % alternate syntax with nx and ny
p.cl=0; % (convenience) switch for later comparison with classical FEM
```

Listing 26: `ac2D/acinit.m` (lines 1-6). Compared to ac1D/acinit.m, the main difference is in lines 4,5, where now we set up a 2D PDE object, showing two alternative ways; see also [Uec20a, §3.1.6] for advanced meshing details in 2D and 3D. In line 7 we define a switch later used to switch to the classical FEM for comparison. The (short) remainder of `acinit` is as before.

```
function p=oosetfemops(p) % ac2D
gr=p.pdeo.grid; p.nc.sf=1e3; % stiff spring constant for DBC via Robin-BC
% uncomment the ff line to identify the 4 boundary segments!
%for i=1:4; p.pdeo.grid.identifyBoundarySegment(i); axis tight; pause; end
[p.mat.K,p.mat.M,~]=p.pdeo.fem.assema(gr,1,1,1);  % indep. of BC
bc1=gr.robinBC(1,0); bc2=gr.robinBC(1,'cos(y/2)');
gr.makeBoundaryMatrix(bc1,bc2,bc1,bc1); % bottom, right, top, left
[p.mat.Q,p.mat.G,~,~]=p.pdeo.fem.assemb(gr); % the BC matrices
```

Listing 27: `oosetfemops.m`; uncommenting line 4 invokes the `OOPDE` method `identifyBoundarySegment` to find the one at $x = 2\pi$, which is needed in (27). It is number 2 here. We then again set up 2 types of BCs: `bc1`, which corresponds to homogeneous Dirichlet BC, and `bc2`, which are used on segment 2.

```
function r=sG(p,u)  % ac2D, with optional "classical FEM" evaluation
if p.cl; r=res_cl(p,u); return; end; % to compare to classical FEM
par=u(p.nu+1:end); u=u(1:p.nu); f=par(2)*u+u.^3-par(3)*u.^5;
r=par(1)*p.mat.K*u-p.mat.M*f+p.nc.sf*(p.mat.Q*u-par(4)*p.mat.G);
```

Listing 28: `ac2D/sG.m`. As in 1D, except for line 2 where for `p.cl=1` we switch to the classical FEM.

```
function r=res_cl(p,u) % compute res. of a OOPDE soln in classical way
par=u(p.nu+1:end);u=u(1:p.np); lam=par(2);ga=par(3); % separate u and pars
ut=p.pdeo.grid.point2Center(u); % u interpolated to element (triangle) centers
f=lam*ut+ut.^3-ga*ut.^5; % evaluate f on centers
[K,~,F]=p.pdeo.fem.assema(p.pdeo.grid,par(1),0,f); % assemble K and F (K not
% strictly needed since c=par(1) is const and hence we could use par(1)*p.mat.K)
r=K*u+p.nc.sf*(p.mat.Q*u-par(4)*p.mat.G)-F; % the rhs
```

Listing 29: `ac2D/cl_res.m`, computing the residual in the classical FEM.

Remark 4.1 a) For completeness we remark that `OOPDE` in 2D also supports the assembly of FEM operators K corresponding to $c_1\partial_x^2+c_2\partial_y^2$, in the form `[K,~,~]=p.pdeo.fem.assema(grid,[[c1 0];[0 c2]],1,1)`. See `aCoefficientsMpt` in `OOPDElightNA/grid2D`. Similarly, to assemble the operator associated to $C = b_1\partial_x+b_2\partial_y$, call `C=p.pdeo.fem.convection(grid,[b1; b2])`, where b1, b2$\in \mathbb{R}^{1\times n_p}$ can also be vectors (values at grid-points), or may contain symbolic expressions in x and y. Analogous remarks apply to 3D, see [Prü16].

b) Additional to `cmds1.m`, there is `cmds2.m` which compares the default 2D mesh adaptation with an alternative anisotropic mesh adaptation setup based on `trullekrul`, see [Uec19c]. ⌋

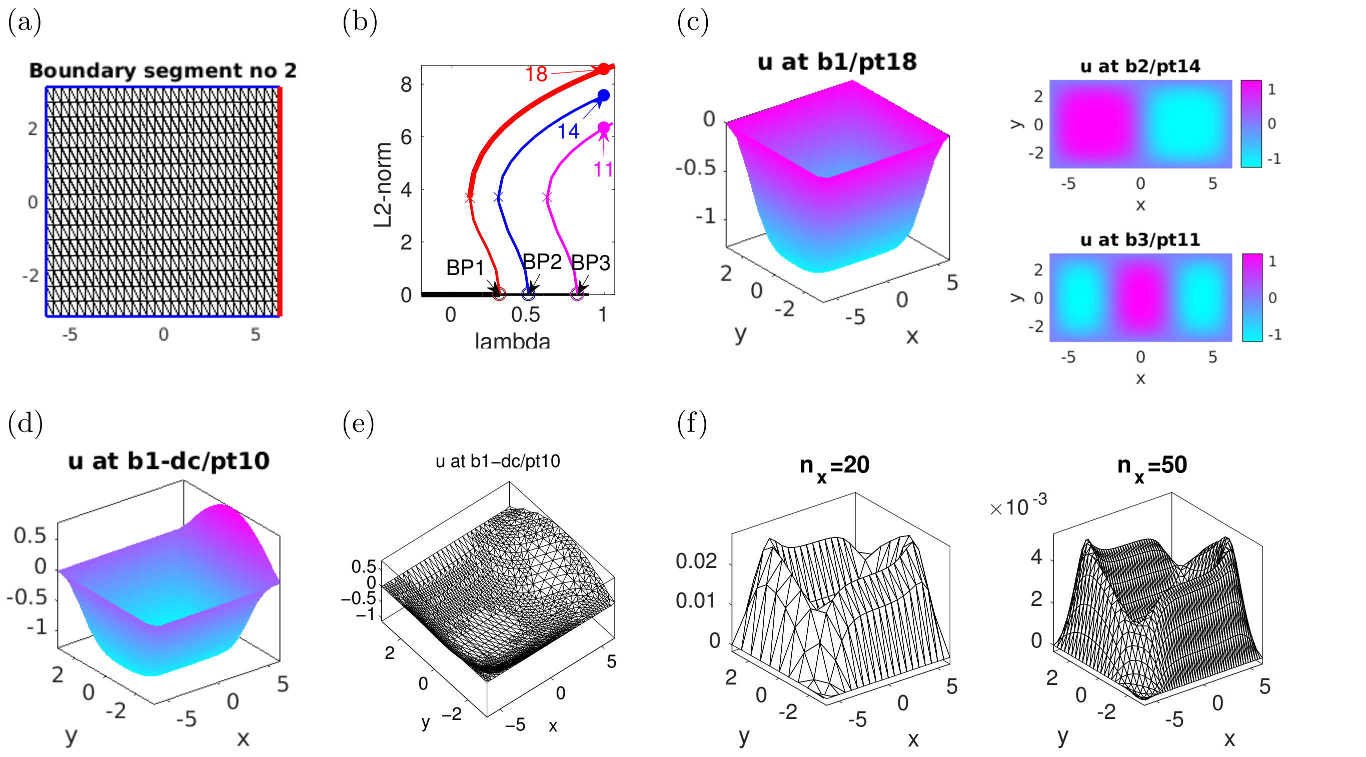


Figure 9: Selection of plots generated in `ac2D/cmds1.m` (a)-(e) and `ac2D/femtest.m` (f). (a) boundary segment 1. (b),(c) continuation in λ and selected solution plots, in plot-styles 3,2, respectively. (d),(e) result of continuation in d, and mesh-refinement of this solution. (f) difference between the `sfem=1` solutions and the `sfem=0` solutions on the first branch at $\lambda = 1$ in dependence of the mesh-size.

4.2 ac3D

In 3D we consider (2) on the cuboid $\Omega = (-2\pi, 2\pi) \times (-3\pi/2, 3\pi/2) \times (-\pi, \pi)$, with BC similar to (27), namely

$$u = d\cos(y/3)\cos(z/2) \text{ on } \Gamma_1 = \{x = -2\pi\}, \text{ parameter } d, \text{ and } u = 0 \text{ on } \partial\Omega \setminus \Gamma_1. \tag{29}$$

For $d = 0$ we now have the bifurcation points

$$\lambda_{jlk} = (j/4)^2+(l/3)^2+(k/2)^2, \quad \phi_{jlk}(x,y,z) = \sin(j(x+2\pi)/4)\sin(l(y+3\pi/2)/3)\sin(l(z+\pi)/2), \tag{30}$$

from the trivial branch, $j, l, k = 1, 2, \ldots$. Again, we only need to slightly modify `acinit.m`, `oosetfemops.m`, while the remaining function files are exactly as in 1D and 2D.

```
function p=oosetfemops(p)  % ac3D
gr=p.pdeo.grid; p.nc.sf=1e3; % stiff spring constant for DBC via Robin-BC
% uncomment the ff line to identify the 6 boundary segments!
%for i=1:6; i, gr.identifyBoundarySegment(i); pause; end
[p.mat.K,p.mat.M,~]=p.pdeo.fem.assema(gr,1,1,1); % indep. of BC
bc1=gr.robinBC(1,0); bc2=gr.robinBC(1,'cos(y/3).*cos(z/2)');
gr.makeBoundaryMatrix(bc1,bc2,bc1,bc1,bc1,bc1);
[Q,G,~,~]=p.pdeo.fem.assemb(gr); p.mat.Q=Q; p.mat.G=G;  % the BC matrices
```

Listing 30: `ac3D/oosetfemops.m`; in order to assemble the boundary matrix we need to account for the 6 boundary segments (faces of the cuboid), which can be identified by uncommenting line 4. The inhomogeneous Dirichlet BC are on segment 2, and are generated as `bc2` in line 7. Line 8 then sets up the boundary matrix, which is assembled in line 9.

```
%pde=stanpdeo3D(lx,ly,lz,2*lx/nx);  % domain and mesh
pde=stanpdeo3D(lx,ly,lz,nx,round(ly/lx*nx),round(lz/lx*nx),sw); % alt.syntax
p.plot.pstyle=1;  p.plot.cm='cool'; % plotting stuff
p.plot.levc={'blue','red'}; p.plot.alpha=0.1; p.plot.ng=20;
```

Listing 31: `ac3D/acinit.m` (lines 4-7). In lines 4,5 (alternate syntaxes) we generate a 3D pde object, lines 6,7 contain 3D plotting specific settings, and the remainder of `acinit` is as usual.

Whereas script `cmds1.m` would work with very slight modification (mostly with respect to plotting) compared to `ac2D/cmds1`, in Cell 3 of `ac3D/cmds1.m` (Listing 32) we follow Remark 3.2. This provides an example of how to use `bploc` to localize branch points, which is easy here because we know the BPs analytically from (30), and the first three are all simple and well separated. In this situation the algorithm from Remark 3.2 is rather robust, and there is no need for an accurate guess for λ (or u, which of course we also know here, i.e. $u \equiv 0$). In case of multiple BPs, or of BPs that lie closer together (which naturally happens for the discretization due to anisotropy of the mesh), `bploc` typically still converges to a BP, but which one is selected is not simple to control. In the setting in Cell 1 with `nx=30` and hence 10350 grid points, the localization with `bploc` (Cell 3, with analytical `spjac`) is about 10 times faster than `findbif` (Cell 2), and this becomes more significant for larger `nx`. See also [UW17].

The remainder of `cmds1.m` deals with 3D solution plotting, see Fig. 10, and with fold continuation of fold on the first nontrivial branch. Here, switching off the assembled `spjac.m` in C8 shows that in this case a fold continuation becomes a factor 100 or more slower. In `cmds2.m` we give examples of 3D mesh adaptation based on `trullekrul`, see [Uec19c].

```
%% C1: init, nx=30->np=10350
p=[]; par=[1 0.3 1 0]; lx=2*pi; ly=3*pi/2; lz=pi; nx=30;
p=acinit(p,lx,ly,lz,nx,par); p.np, plotsol(p,1,1,1); p.fuha.spjac=@spjac;
p.nc.ilam=2; p.nc.lammax=2; p.sol.ds=0.1; p.nc.dsmax=0.2; p=setfn(p,'tr'); p0=p;
%% C2: use findbif to locate the BPs; here we comment out this cell and
% alternatively use bploc in the next cell (more efficient for large nx)
tic; p=findbif(p,3); toc
%% C3: localize BPs by setting lam to near a (here known) BP and using bploc
%p.branch=[bradat(p); p.fuha.outfu(p,p.u)];
p=p0; p=cont(p,1); % 1 step, only to generate tangent
tic; p=setlam(p,0.4); p=bploc(p); % localize 1st BP
p=setlam(p,0.6); p=bploc(p); p=setlam(p,0.8); p=bploc(p); toc % 2nd and 3rd BP
```

Listing 32: `ac3D/cmds1.m`, first 3 cells. C3 illustrates the use of `bploc` in order to localize branch points, which is significantly faster than `findbif` (or BP localization with regular `cont`) for large nx. The remainder of `ac3D/cmds.m` deals with plotting and a fold continuation.

5 Quasilinear equations

The `OOPDE` setup can also be used for quasilinear problems. As a tutorial example we consider the quasilinear Allen-Cahn equation from [UWR14, §3.4] given by

$$\partial_t u = \nabla \cdot [c(u)\nabla u] + f(u), \tag{31}$$

where $c(u) = c_0 + \delta u + \varepsilon u^2$ and $f(u) = \lambda u + u^3 - \gamma u^5$, with parameters $c_0, \delta, \varepsilon, \lambda, \gamma$. Using `OOPDE`, (31) can now be treated with unified interfaces in 1D, 2D and 3D, and with significant speedup. As before we focus on the steady version

$$-\nabla \cdot [c(u)\nabla u] - f(u) = 0, \tag{32}$$

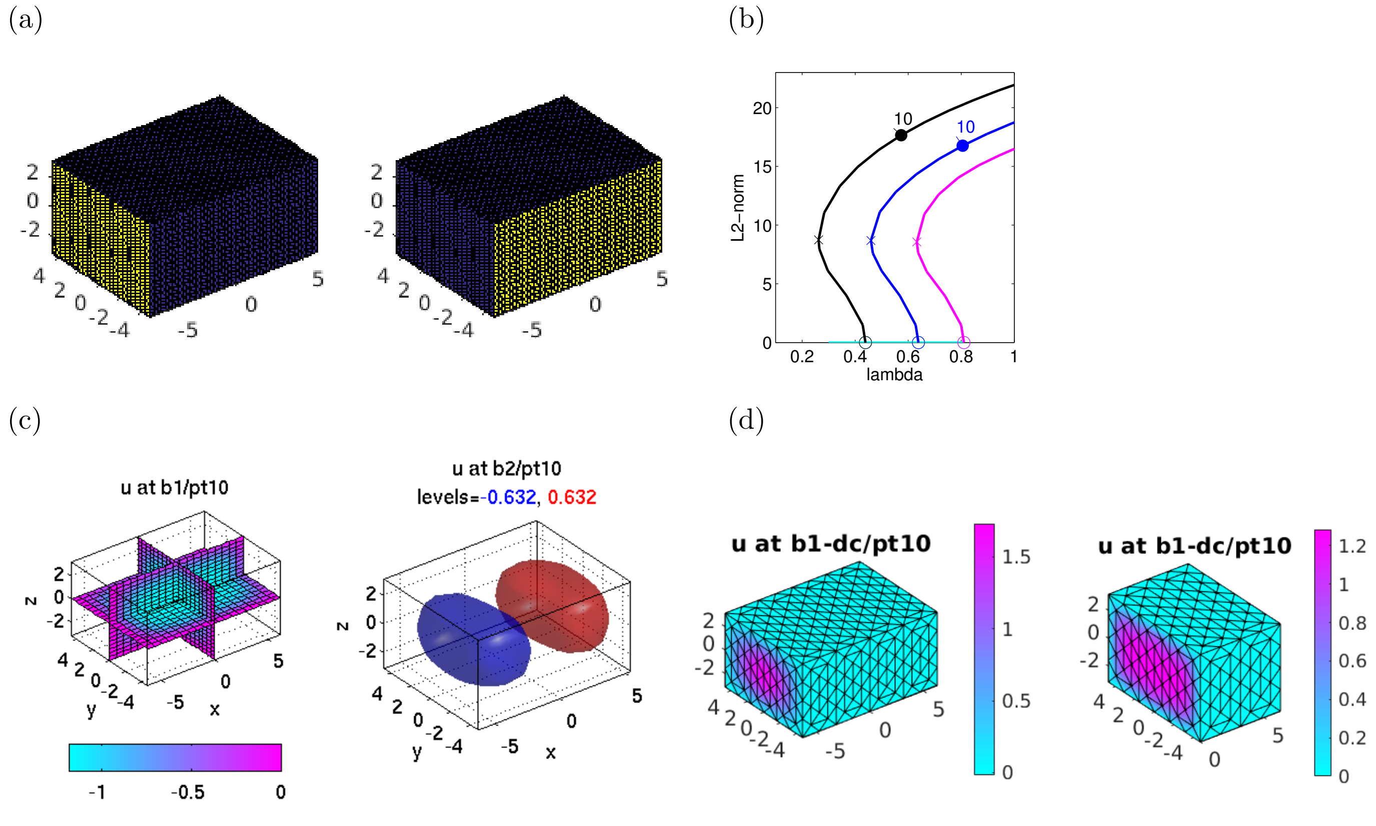


Figure 10: Selection of plots generated in `ac3D/cmds1.m`. (a) identifying boundary segments 2 (with the nontrivial BC) and 3. (b),(c) BD and example plots for $d = 0$, plotstyles 1 (slice-plot) and 2 (isosurface plot). (d) Example plots after continuation in d, plotstyle 3 (face plot) and 4 (cut–away plot).

over boxes, and for simplicity with homogeneous Dirichlet BC on all boundaries.

The FEM formulation of (32) reads

$$G(u) := [K(u) + sQ_{\mathrm{BC}}]u - F(u) = 0, \tag{33}$$

respectively $M\dot{u} = -G(u)$ for the evolutionary problem, where now

$$K_{ij}(u) = \int_\Omega c(u)\nabla\phi_i \cdot \nabla\phi_j \,\mathrm{d}x \tag{34}$$

depends on u. This needs $c(u)$ on the element centers, which is obtained from first interpolating u from the nodes to the element centers. Similarly, instead of using Mf, which is also possible and up to the $\mathcal{O}(h^2)$ interpolation error gives the same results, we now also evaluate F as in (10); this is combined in `[K,∼,F]=p.pdeo.fem.assema(grid,c,0,f)` in line 5 of `sG`, see Listing 35, where `c` is computed in the (very simple) extra function `cfu`, which is also used in `sGjac`.

In contrast to the previous sections, we now treat the 1D-3D cases jointly in one directory, as the differences mainly show up the init-file, see Listing 33, and in `oosetfemops` for the definition of "convection and gradient matrices", see Listing 34, needed in an implementation the Jacobian if as an operator acting on a function v we write it as

$$G_u(u)v = -\nabla \cdot (c(u)\nabla v) - \nabla \cdot (c_u(u)\nabla uv) - f_u(u)v. \tag{35}$$

The FEM formulation of the first term is $K(u)v$ as in (33), for the third we return to the Mf_u setting, and the second can be efficiently approximated as follows. We have (for definiteness assuming the 2D case) $\nabla\cdot(c_u(u)\nabla uv) = \partial_x(c_u(u)u_xv) + \partial_y(c_u(u)u_yv)$ which, assuming that we have (approximations of) u_x and u_y, translates into $K_x(c_u(u)u_xv) + K_y(c_u(u)u_yv)$, with the first order differentiation FEM matrices K_x, K_y, generated in `oosetfemops` via `p.mat.Kx=p.pdeo.fem.convection(grid,[1;0])` and `p.mat.Ky=p.pdeo.fem.convection(grid,[0;1])`, respectively. For these "convection matrices" we have $\partial_xu = M^{-1}K_xu$, $\partial_yu = M^{-1}K_yu$, or, in other words, $M\dot{u} = K_xu$ is the FEM for $\partial_tu = \partial_xu$. Since M^{-1} should never be used, to approximate u_x, u_y and, in 3D, u_z, we use differentiation matrices `Dx, Dy`, and `Dz`, which in `OOPDE` can be obtained via `fem.gradientMatrices(grid)`, see Listing 34. If, on a nontrivial solution, we compare this implementation of $G_{u,a}$ with numerical Jacobians $G_{u,n}$ (via `[Gu,Gn]=jaccheck(p)`), then we often get a relative error of about 10^{-3} or 10^{-4}, compared to 10^{-6} or smaller which we typically get for the semilinear problems discussed before. However, the approximation is 'good enough' for all Newton loops in the continuation to work.

On the other hand, similar approximations may not work for other quasilinear problems, see the demo `pftut/chemotax` [Uec20a]. Thus, there we explain an alternative method which is a 'hybrid' between analytical and numerical Jacobians, in a block form, with `numjac` only for difficult blocks such as $\partial_u\nabla\cdot(c(u)\nabla v)$. This is not needed here for the continuation, and slower than the above approximation, but nevertheless for illustration we include this method, which can be switched on via `p.jacsw=2` after initialization.

```
function p=acinit(p,lx,nx,par,dim) % acql
p=stanparam(p); p.nc.neq=1; p.nc.ilam=2; screenlayout(p);
p.sw.sfem=-1; p.dim=dim; p.fuha.sG=@sG; p.sw.jac=1;
p.jacsw=1; % additional switch for Jac: 1=approx.,
% 2=via numjac for \pa_u(div(c(u)\nab v), see sGjac*D for more comments
p.fuha.cfu=@cfu; % function handle for the nonlinear diffusion coeff
switch dim % domain setup
 case 1; pde=stanpdeo1D(lx,2/nx); p.fuha.sGjac=@sGjac1D;
 case 2; ly=0.9; ny=round(nx*ly/lx); p.plot.pstyle=3; p.fuha.sGjac=@sGjac2D;
     sw.sym=2; pde=stanpdeo2D(lx,ly,nx,ny,sw); % init with symmetric mesh
 case 3; ly=0.75*lx;lz=0.5*lx; pde=stanpdeo3D(lx,ly,lz,2*lx/nx);
     p.fuha.sGjac=@sGjac3D;
end
```

Listing 33: `acql/acinit.m` (lines 1-13), setting up different domains and function handles for the Jacobians depending on the switch `dim`, and also setting `p.jacsw` determining the behaviors of `sGjac*D`. Remainder of `acinit.m` as usual.

```
function p=oosetfemops(p) % for acql, hence no K (K assembled in sG)
gr=p.pdeo.grid; fem=p.pdeo.fem; [~,M,~]=fem.assema(gr,0,1,1); p.mat.M=M;
bc=gr.robinBC(1,0); gr.makeBoundaryMatrix(bc); % one for all
[Q,G,~,~]=fem.assemb(gr); p.mat.Q=Q; p.mat.G=G; % the BC matrices
E=center2PointMatrix(gr); % to map element different. matrices to nodal ones
p.mat.p2c=point2CenterMatrix(gr); % to interpolate from nodes to element centers
switch p.dim % set up differentiation and convection matrices for Jacobian
  case 1; p.mat.Dx=makeDx(p); p.mat.Kx=fem.convection(gr,1);
  case 2; [Dx,Dy]=fem.gradientMatrices(gr); p.mat.Dx=E*Dx; p.mat.Dy=E*Dy;
    p.mat.Kx=fem.convection(gr,[1;0]); p.mat.Ky=fem.convection(gr,[0;1]);
  case 3; [Dx,Dy,Dz]=fem.gradientMatrices(gr); p.mat.Dx=E*Dx; p.mat.Dy=E*Dy;
    p.mat.Dz=E*Dz;  p.mat.Kx=fem.convection(gr,[1;0;0]);
    p.mat.Ky=fem.convection(gr,[0;1;0]); p.mat.Kz=fem.convection(gr,[0;0;1]);
end
```

Listing 34: `acql/oosetfemops.m`, assembling M and the BC matrices as usual, but no fixed K. Additionally, we assemble and store the first order differentiation matrices `Dx` and `Kx`, and similarly, depending on the space dimension `Dy, Dz` and `Ky, Kz`.

```
function r=sG(p,u)  % rhs for ql-AC
```

```
par=u(p.nu+1:end); lam=par(2); ga=par(3);
u=u(1:p.nu); gr=p.pdeo.grid; ut=(p.mat.p2c*u)'; % interpolate to elem. centers
c=p.fuha.cfu(ut,par); f=lam*ut+ut.^3-ga*ut.^5; % coefficients for assembly
[K,~,F]=p.pdeo.fem.assema(gr,c,0,f); % assemble K and F (M not used)
r=K*u-F+p.nc.sf*(p.mat.Q*u-p.mat.G);
```

Listing 35: `acql/sG.m`, implementing the rhs by assembling K and F in each call.

```
function Gu=sGjac2D(p,u)  % Jac for ql-AC
par=u(p.nu+1:end); lam=par(2); ga=par(3); del=par(4); epsi=par(5);
n=p.nu; u=u(1:n); gr=p.pdeo.grid; fem=p.pdeo.fem;
ut=(p.mat.p2c*u)'; c=cfu(ut,par); % diff. coefficient defined on element centers
fu=lam+3*ut.^2-5*ga*ut.^4; % lin.of 'nonlinearity' f on triangles
[K,Fu,~]=fem.assema(gr,c,fu,0); % assembling nonlin.diff. and Fu
switch p.jacsw; % select how to compute Jacobian of divergence terms
  case 1; % approximate version, fast, and 'good enough'
    cu=del+2*epsi*u; % c_u (nodal) and f_u (triangles)
    ux=p.mat.Dx*u; uy=p.mat.Dy*u; % 1st derivatives as coefficients
    K1=p.mat.Kx*spdiags(cu.*ux,0,n,n)+p.mat.Ky*spdiags(cu.*uy,0,n,n);
    Gu=K-K1-Fu+p.nc.sf*p.mat.Q;  % putting it all together
  case 2;  % numjac for \pa_u div(c(u)\nab v)
    Kuvd=getKuvd(p,par,u,u);  Gu=K+Kuvd-Fu+p.nc.sf*p.mat.Q;
end
```

Listing 36: `acql/sGjac2D.m`, implementing (35) (in 2D). This mixes the classical FEM setup for K and the simplified (pre-assembled matrices) setup for the lower order terms. For 1D and 3D we only have to adapt `K1` in an obvious way. Alternatively, we can use a numerical Jacobian as described in the text.

Results of running (31) in 1D, 2D and 3D are shown in Figs. 11–13, based on the script files `cmds1D.m`, `cmds2D.m` and `cmds3D.m`, which follow standard procedure, with base parameters

$$(c_0, \gamma, \delta, \varepsilon) = (0.25, 1, -0.3, 0.2), \tag{36}$$

and λ as a continuation parameter. The trivial branch is $u \equiv 0$, and we follow the first two bifurcating branches.

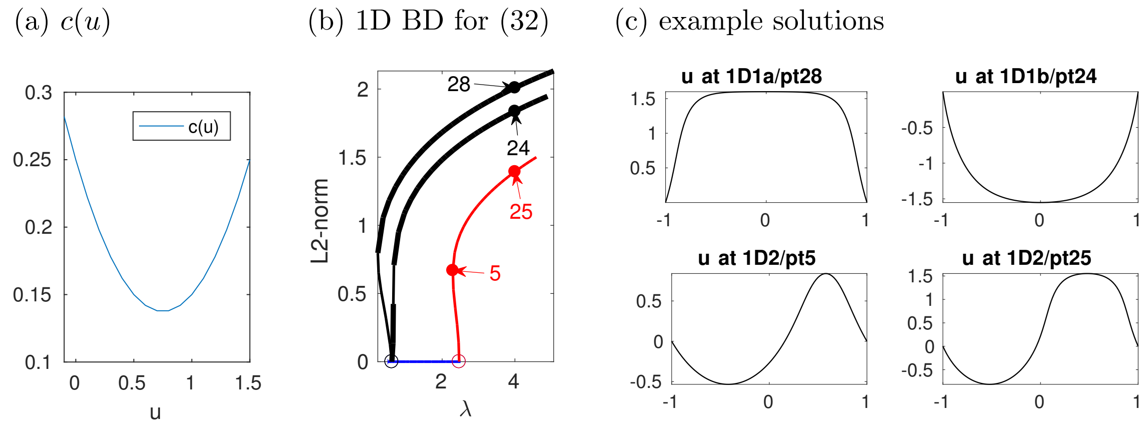


Figure 11: Results for the quasilinear Allen-Cahn equation (32), $\Omega = (-1, 1)$, $(c_0, \gamma, \delta, \varepsilon) = (0.25, 1, -0.3, 0.2)$, and λ as a continuation parameter. (a) $c(u)$, (b) Bifurcation diagram, and (c) five example solutions.

Figure 11(a) shows that $c(u)$ with parameters from (36) has a minimum near $u = 0.75$. Thus, for u near 0.75 we expect sharper gradients, and this is essentially illustrated in the solution plots. In Fig. 12(b) we additionally illustrate mesh–adaption based on the standard form of `e2rs`, and Fig. 13 for completeness illustrates some 3D results.

(a) 2D BD for (32) (b) example solution, with mesh adaption of 2D2a/pt15

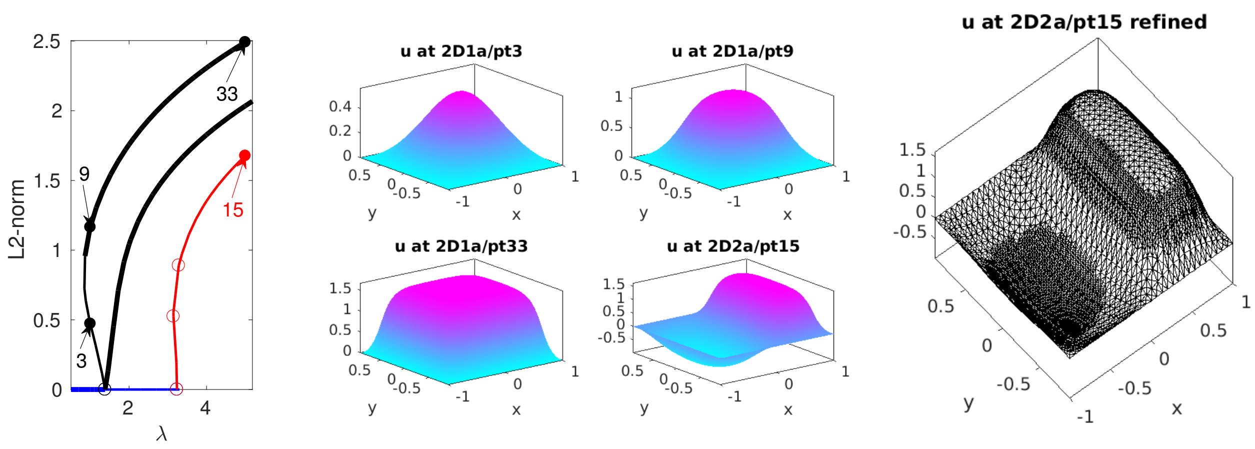


Figure 12: Results for (32), $\Omega = (-1,1)\times(-0.9,0.9)$, $(c_0,\gamma,\delta,\varepsilon) = (0.25,1,-0.3,0.2)$, and λ as a continuation parameter. (a) Bifurcation diagram. (b) Four example solutions and mesh adaption using `e2rs` as usual.

(a) 3D BD for (32) (b) 3 example solutions

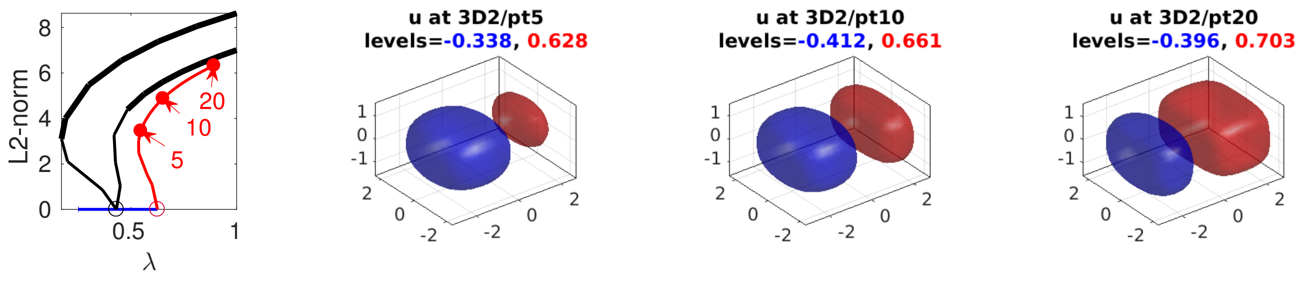


Figure 13: Results for (32), $\Omega = (l_x, l_x) \times (-l_y, l_y) \times (-l_z, l_z)$, $l_x = \pi, l_y = 3\pi/4, l_z = \pi/2$. Ω was discretized by a grid of $n_p = 7280$ points and $n_t = 37050$ tetrahedra, and the computation of (a) takes about 40s altogether.

Remark 5.1 A similar setup can be used for quasilinear equations in non–divergence form. For instance, $\Delta(c(u)) = \nabla\cdot\nabla c(u) = \nabla\cdot(c'(u)\nabla u)$, and the above setup replies with $\tilde{c} = c'(u)$. Similarly, $c(u)\Delta u = \nabla\cdot(c(u)\nabla u) - c'\langle\nabla u, \nabla u\rangle$, and we recommend to implement the second term using differentiation matrices `Dx` (and `Dy, Dz` in higher space dimension) as in the Jacobians above. ⌋

6 Global coupling, and further remarks

The aim of this tutorial was to give a soft introduction to the `OOPDE` setting of `pde2path`, using the Allen-Cahn equation (2) as an example. Starting with the simplest possible setting in `ac1D_simple`, we gradually introduced further features such as fold continuation, mesh adaptation and various boundary conditions with the exception of periodic BC. These are treated in detail [DU17]. Note that many files in `ac1D_simple` to `ac3D` are almost identical, with only minor changes in the initialization and plotting. However, a different setup has to be used for the right hand sides and Jacobians for quasilinear equations, as in `acql`. For mesh adaptation we restricted to 1D and 2D, based on standard error estimators. See [Uec19c] for an alternative mesh adaptation in 2D and 3D based on `trullekrul`.

Additionally, in the demo `acgc` we consider an AC equation with a global coupling, i.e., $G(u) = -c\Delta u - \lambda u - u^3 + \gamma u^5 + f_{\rm gc}(u)$, where $f_{\rm gc}(u) = \delta\langle u^4\rangle u$, and $\langle v\rangle = \frac{1}{|\Omega|}\int v(x)\,{\rm d}x$ denotes a (normalized) global average. The efficient implementation relies on Sherman-Morrison formulas for linear system solvers, described in [Uec20a].

The Allen-Cahn equation is scalar, and a natural extension of the setup is to systems of PDEs, for which we refer to the tutorials available at [Uec20c], with [dWDR$^+$20] as a directory and overview. Moreover, in the 2D and 3D cases we simplified things by setting up (non-square) domains, where the bifurcation points on the trivial branch are simple. Non-simple bifurcation points due to symmetries are discussed in [Uec19b] and [Uec20a], which includes various systems as example problems.

References

[Dör95] W. Dörfler. A robust adaptive strategy for the nonlinear Poisson equation. *Computing*, 55(4):289–304, 1995.

[DRUW14] T. Dohnal, J.D.M. Rademacher, H. Uecker, and D. Wetzel. pde2path - V2: faster FEM and periodic domains, 2014.

[DU17] T. Dohnal and H. Uecker. Periodic boundary conditions in pde2path, 2017.

[dW17] H. de Witt. Fold and branch point continuation in pde2path – a tutorial for systems, 2017.

[dWDR$^+$20] H. de Witt, T. Dohnal, J.D.M. Rademacher, H. Uecker, and D. Wetzel. pde2path - Quickstart guide and reference card, 2020.

[GU17] D. Grass and H. Uecker. Optimal management and spatial patterns in a distributed shallow lake model. *Electr. J. Differential Equations*, 2017(1):1–21, 2017.

[GUU19] D. Grass, H. Uecker, and T. Upmann. Optimal fishery with coastal catch. *Natural Resource Modelling*, (e12235), 2019.

[Mei89] Zhen Mei. A numerical approximation for the simple bifurcation problems. *Numer. Funct. Anal. Optim.*, 10(3-4):383–400, 1989.

[Mei00] Zhen Mei. *Numerical bifurcation analysis for reaction-diffusion equations.* Springer, 2000.

[Moo80] G. Moore. The numerical treatment of nontrivial bifurcation points. *Numer. Funct. Anal. Optim.*, 2(6):441–472 (1981), 1980.

[Prü16] U. Prüfert. OOPDE, `www.mathe.tu-freiberg.de/nmo/mitarbeiter/uwe-pruefert/software`, 2016.

[RU17] J.D.M. Rademacher and H. Uecker. Symmetries, freezing, and Hopf bifurcations of modulated traveling waves in pde2path, 2017.

[Uec16] H. Uecker. Optimal harvesting and spatial patterns in a semi arid vegetation system. *Natural Resource Modelling*, 29(2):229–258, 2016.

[Uec19a] H. Uecker. Hopf bifurcation and time periodic orbits with pde2path – algorithms and applications. *Comm. in Comp. Phys*, 25(3):812–852, 2019.

[Uec19b] H. Uecker. Multiple bifurcation points in pde2path, 2019.

[Uec19c] H. Uecker. Using `trullekrul` in `pde2path` – anisotropic mesh–adaptation for some Allen–Cahn models in 2D and 3D, Preprint, arXiv 1912.11130 , 2019.

[Uec20a] H. Uecker. Pattern formation with pde2path – a tutorial, 2020.

[Uec20b] H. Uecker. User guide on Hopf bifurcation and time periodic orbits with pde2path, 2020.

[Uec20c] H. Uecker. `www.staff.uni-oldenburg.de/hannes.uecker/pde2path`, 2020.

[UW17] H. Uecker and D. Wetzel. The pde2path linear system solvers – a tutorial, 2017.

[UWR14] H. Uecker, D. Wetzel, and J.D.M. Rademacher. pde2path – a Matlab package for continuation and bifurcation in 2D elliptic systems. *NMTMA*, 7:58–106, 2014.

[Wet17] D. Wetzel. A pde2path `plotsol` tutorial, 2017.