Supplementary information (comments on demo implementations) for *Continuation and bifurcation in Nonlinear PDEs – Algorithms, applications, and experiments*

Hannes Uecker

Institut für Mathematik, Carl von Ossietzky Universität Oldenburg, hannes.uecker@uol.de

October 11, 2021

General comments. In [6] we consider three example problems for numerical continuation and bifurcation analysis in nonlinear PDEs. Here we comment on the associated `pde2path` implementations, in the demos `seco`, `sh35disk`, and `schnakdc`, and additionally on the Allen–Cahn dead core demo `acdc`, which prepares `schnakdc`, all included in the download `DMVdemos` at [9]. We start with general comments on `pde2path`, see also [7, Chapter 5], and then give overviews and specifics for the individual demos. In Table 1 we list acronyms used in [6] and here.

Table 1: Acronyms.

BC	Boundary Condition	BD	Bifurcation Diagram	BP	Branch Point
DC	Dead Core	FP	Fold Point	HP	Hopf Point
FEM	Finite Element Method	PC	Phase Condition	PO	Periodic Orbit

`pde2path` aims at PDE problems of the form

$$M_d\partial_t u = -G(u,\lambda) = \nabla\cdot(c\otimes\nabla u) - au + b\otimes\nabla u + f(u), \tag{1}$$

with $u = u(x,t) \in \mathbb{R}^N$ (N components), $x \in \Omega \subset \mathbb{R}^d$ some bounded domain, $d \in \{1,2,3\}$ (the 1D, 2D and 3D case, respectively), and time $t \geq 0$, and where (1) can be complemented with various BCs. We refer to [6] or [7] for the meaning of terms in $G(u,\lambda)$. The basic idea is to first discretize (1) in space using some FEM built into `pde2path`, and then treat the resulting high–dimensional ODEs, respectively algebraic systems in case of steady states of (1).

As usual in `pde2path` we assume that all problem data is contained in the `MATLAB` struct `p`. This includes the object `p.pdeo` (with sub–objects `fem` and `grid`), which provides methods to generate FEM meshes and assemble FEM matrices `M` (mass matrix) and `K` (e.g., Laplacian, including the BCs), and several more, for instance for mesh adaptation. Typical initializations and first continuation steps run as follows (Listings 1–3 contain concrete examples):

1. Call `p=stanparam()` to initialize the fields in `p` with defaults values.
2. Call a `pdeo` constructor, for instance `p.pdeo=stanpdeo1D(p,aux)` (which discretizes an interval), where here and in the following `aux` or ... stands for variable arguments.
3. Initialize `p.u` with a first solution (or a solution guess, to be corrected in a Newton loop), and append the parameters at the end of `p.u`.
4. In a function `oosetfemops` (in the working directory[1]), use `p.pdeo.assema` to generate `p.mat.M`

[1] `MATLAB` always searches the working directory first, which is an easy way to overload `pde2path` library functions

(the dynamical mass matrix, *always required*) and `p.mat.K` (a Laplacian), and possibly further FEM matrices, e.g., for BCs.

5. Use `p.mat.M` and `p.mat.K` in a function `r=sG(p,u)` to encode the (discretized) PDE (and the Jacobian in `Gu=sGjac(p,u)`). Here, the input argument `u` contains the "PDE unknowns" u and the parameters (appended at the end), and `p` is typically useful for simple coding, in particular the subfields of `p.mat` such as the preassembled matrices `M` and `K`.
6. Call `p=cont(p)` to (attempt to) continue the initial solution in some parameter, including bifurcation detection, localization, saving to disk, and plotting.
7. Call `p=swibra(dir,bpt,newdir)` (or `p0=qswibra(dir,bpt,aux); p=seltau(...)`, see [6, Algorithms 2.2 and 2.3]) to attempt branch switching at branch point `bpt` in directory `dir`. For Hopf bifurcations, call `p=hoswibra(...)`, for PO branch switching call `p=poswibra(...)`.[2] In all cases, subsequently call `p=cont(p)` again, with saving in `newdir`.
8. Perform special tasks such as fold or branch–point continuation; use `plotbra(dir,pt,aux)` to plot bifurcation diagrams, and `plotsol(dir,pt,aux)` to plot sample solutions.

Steps 1–3, and a call to `oosetfemops` are typically combined into an init-function, for instance `p=secoinit(...)` in Listing 2.

Remark 1. a) The right hand side G, Jacobian $\partial_u G$, and a number of further functions needed/used to run `pde2path` on a problem `p`, are interfaced via function handles in `p.fuha`. For instance, you can give the function encoding G in (1) any name, e.g., `myrhs`, with signature `r=myrhs(p,u)`, and then set `p.fuha.sG=@myrhs`. In most demos, we simply keep the "standard names" `sG` and `sGjac` and code these functions in the respective demo directory. For other handles in `p.fuha` there are standard choices which we seldomly modify, e.g. `p.fuha.savefu=@stansavefu`. Functions for which the "default choice" is more likely to be modified include, e.g.,

- `p.fuha.outfu=@stanbra; % signature out=stanbra(p,u)`. Here the user defines what data (typically parameter values and some norm(s) of u) is used for branch output, e.g., for later plotting of BDs.
- `p.fuha.lss=@lss; % signature [x,p]=lss(A,b,p)`. Linear systems solver $x = A^{-1}b$. The default `lss` is just an interface to `MATLAB`'s \; other options include, e.g., `lssbel` (bordered elimination) and `lssAMG` (preconditioned GMRES using `ilupack` [2]).

b) An important feature of the FEM used to spatially discretize the PDEs is its flexibility with respect to the domain shape and BCs, and its flexibility and well established procedures for adaptive mesh refinement. `pde2path` comes with a number of convenience functions to discretize standard domains (intervals, rectangles, cuboids, disks, sectors, balls, ...), and with methods for mesh adaptation in 1D, 2D, and 3D, based on standard error–estimators. Here, we only use some mesh adaptation in the demo `acdc`, and refer to [7, Chapters 4 and 6] for details. ⌋

The demo `seco`. In [6, §3.1] we consider the 2–component reaction system

$$\begin{aligned} \partial_t u_1 &= \partial_x^2 u_1 + \frac{u_2 - u_1}{(u_2 - u_1)^2 + 1} - \tau u_1, \\ \partial_t u_2 &= d\partial_x^2 u_2 + \alpha(j_0 - (u_2 - u_1)), \end{aligned} \tag{2}$$

set up in [4] as a model for **semi**con**ductors. Throughout we fix $(\tau, d) = (8, 0.05)$, and initially also $\alpha = 0.02$, and take j_0 as the primary continuation/bifurcation parameter. For all (j_0, α), (2) has the

[2]There are further specialized methods for branch switching, for instance `twswibra` for switching to traveling wave branches.

unique spatially homogeneous steady state

$$u^* = (u_1^*, u_2^*), \qquad u_1^* = \frac{j_0}{\tau(j_0^2+1)}, \quad u_2^* = j_0 + u_1^*. \tag{3}$$

For suitable parameter choices, there are codimension–2 points, near which u^* may undergo either a (steady) Turing or a Hopf bifurcation, see [6, Fig. 4].

Table 2: Scripts and functions in `seco`.

file	purpose, remarks
`cmds1,cmds2`	scripts, generating [6, Fig. 5 and Fig. 6].
`secoinit`	initialization of problem struct `p` with standard parameter values, call of `stanpdeo1D` to generate a 1D PDE object (interval, with mesh), initialization of u with the homogeneous steady state, call of `oosetfemops` to generate the FEM matrices.
`oosetfemops`	assemble and store the mass matrix M, and the (1-component) Neumann-Laplacian K.
`sG,sGjac`	rhs of (4), and Jacobian.
`nodaljac`	”local” derivatives (terms in Jac without spatial derivatives), called in `sGjac` and `spufu`.
`bpjac`	implements $\partial_u(G_u\phi)$ for BP continuation, see also `hpjac` for HP continuation.
`secobra`	mod of library function `hobra`; subtract steady state from solution for branch output
`getss`	convenience function to compute steady state u^* from parameters.
`spufu`	”spectral” user function, used to plot dispersion relations.

Table 2 list the files used to implement and run (2). in the demo `seco`, and Listing 1 shows the three basic functions essentially needed in a typical `pde2path` demo. In `secoinit` we set up the domain, the initial solution (3), based on the parameters passed to `secoinit` in the script file `cmds1`, and set a few `pde2path` control parameters as appropriate for this problem. This, to some extent, is trial and error; moreover, switches and numerical controls such as `p.nc.dsmax` can be (and often are) changed later any time. Additionally, in line 5 we switch on the bordered elimination linear system solver `lssbel`, which in 1D, due to the band structure of M and K, usually yields a significant speedup. Here the border of the Jacobian $\partial_u H$ of the extended system $H = (G, p)$, with p the arclength equation [6, (6)], only has width 1 from p, and the 0 in `setbel` means that $\partial_u G$ itself has no border.

In the function `oosetfemops` we assemble (the 1–component) mass matrix M and stiffness matrix K; we store the (block–diagonal) *system* mass matrix in `p.mat.M` (always needed), and the (1–component Neumann–)Laplacian in `p.mat.K`. We do not need BC matrices, and compose the system diffusion matrix from `p.mat.K` in `sG`; this allows, e.g., to do also continuation in diffusion constants. Altogether, the residual `r` in `sG` consists of the diffusion terms $(-\Delta u_1, -d\Delta u_2)$ and the “nonlinearity” f, which contains all terms without derivatives.[3]

```
function p=secoinit(lx,nx,par) % init for MWBS'97 semiconductor model
p=stanparam(); p.nc.neq=2;      % init with stanparam, 2-compo-system
p.sol.ds=-0.01; p.sol.dsmax=0.05; p.sw.bifcheck=2; % reset some pars
p.fuha.outfu=@secobra; p.plot.bpcmp=8; % output function handle, and compo
p=setbel(p,0,1e-6,10,@lss); % use bordered elim. lss (always good in 1D),
% with border width 0, tolerance 1e-6, and at most 10 iterations
pde=stanpdeo1D(lx,2*lx/nx); p.vol=2*lx; % standard 1D PDE object
p.np=pde.grid.nPoints; p.pdeo=pde; % store number of grid-points, and pdeo
p.nu=p.np*p.nc.neq; p.sol.xi=1/p.nu; % DoFs, and weight for arclength
```

[3]Here f is computed in a simplified FEM setup as $Mf_{\rm nodal}$ from the mass matrix and the nodal values of f. The genuine (P^1–) FEM would require interpolating u to the element centers and then evaluating f; the error between the two is bounded by h^2, where h is the (local) mesh width. See [7, §4.1] for further comments.

```
p.nc.lammin=0; p.nc.lammax=5; p.nc.dsmax=0.1; % lam-range, and max stepsize
j0=par(1); tau=par(3); als=j0/(tau*(j0^2+1)); us=als+j0; % initial sol
u=als*ones(p.np,1);v=us*ones(p.np,1);p.u=[u;v;par]; % append pars and store
p.nc.ilam=1; % select the (initial) primary active par, here j0
p.sw.sfem=-1; p=oosetfemops(p); % set FEM matrices
```

```
function p=oosetfemops(p)
% generate FEM matrices, here just mass M and  Neumann-Laplacian K
[K,M,~]=p.pdeo.fem.assema(p.pdeo.grid,1,1,1);
p.mat.M=[M 0*M;0*M M]; p.mat.K=K; % store matrices
```

```
function r=sG(p,u) % rhs for MWBS'97 semiconductor model
% split u into pars and PDE fields u1 and u2:
par=u(p.nu+1:end); j0=par(1); al=par(2); tau=par(3); D=par(4);
u1=u(1:p.np); u2=u(p.np+1:2*p.np);
% compute nodal 'nonlinearity' (terms without spat.derivatives):
f1=(u2-u1)./((u2-u1).^2+1)-tau*u1; f2=al*(j0-(u2-u1)); f=[f1;f2];
Ks=p.mat.K; K=[Ks 0*Ks; 0*Ks D*Ks]; % the diffusion matrix
r=K*u(1:p.nu)-p.mat.M*f;             % the residual
```

Listing 1: The three "basic" functions in seco, following standard pde2path principles

In 1D (more precisely: for a small number n_u of DoF) speed is usually not an issue, and hence sGjac.m implementing $\partial_u G$ could be omitted, setting p.sw.jac=0 to use numerical Jacobians. However, $\partial_u G$ can in fact be implemented in a few lines, see Listing 2, and this should be considered good practice. Here we also "outsource" the Jacobian nodaljac of the nonlinearity as it is also called in spufu.

```
function Gu=sGjac(p,u) % Jac for MWBS'97 semiconductor model
n=p.np; [f1u,f1v,f2u,f2v]=nodaljac(p,u); % Jac of 'nonlinearity'
Fu=[[spdiags(f1u,0,n,n),spdiags(f1v,0,n,n)];% put f_u into block matrix
    [spdiags(f2u,0,n,n),spdiags(f2v,0,n,n)]];
D=u(p.nu+4); Ks=p.mat.K; K=[Ks 0*Ks; 0*Ks D*Ks];% diffusion matrix
Gu=K-p.mat.M*Fu;                                  % Jacobian
```

```
function [f1u,f1v,f2u,f2v]=nodaljac(p,u) % for seco
n=p.np; u1=u(1:n); u2=u(n+1:2*n); den=(u2-u1).^2+1; % u, and a denominator
par=u(p.nu+1:end); al=par(2); tau=par(3); % parameters in nonlinearity
f1u=-1./den+2*(u2-u1).^2./(den.^2)-tau; f1v=-f1u-tau;
f2u=al*ones(n,1);  f2v=-f2u;
```

Listing 2: The Jacobian sG, and nodaljac which computes the local derivatives, and is also called in spufu to compute dispersion relations.

Listing 3 gives the first 9 lines of the script file cmds1, of altogether 70 lines, which contain many plotting commands and somewhat detailed comments. The file is organized in *cells*, started by %%, which can be executed individually, which we strongly recommend. In the second cell, we use initeig to compute a guess for shifts for Hopf eigenvalues, cf. [7, §3.3].

```
%% MWBS97 model, trivial and Turing branches, and bif from Turing
al=0.02; j0=3.3; tau=0.05; D=8; kc=(al*tau/D)^0.25; nw=8; % parameters,
lx=nw*pi/kc; nx=nw*40; par=[j0;al;tau;D]; dir='0'; % dom.size and nx
p=secoinit(lx,nx,par); p=setfn(p,dir); % init, set output dir for tr.branch
%% compute guess for Hopf-spectral shift, then cont trivial branch
p=initeig(p,0.5); p.nc.neig=[10,10]; p.nc.eigref(1)=-0.05;
p.nc.dsmax=0.05; p.sw.verb=1; p.file.smod=5; p=cont(p,30);
%% primary Turing mode
```

```
p=swibra('0','bpt1','T1'); p.nc.dsmax=0.05; p.sol.ds=-0.02; p=cont(p,60);
```

Listing 3: Start of script cmds1.m, generating [6, Fig.5]. We choose (initial) parameters, the domain size $2l_x = 2n_w\pi/k_c$ with $n_w = 8$ and critical wave number k_c such that 8 waves fit into Ω at criticality, and choose 40 points per wave for the spatial discretization. Then we compute (via `initeig`) a guess for the temporal wave number ω_1 near which we aim to detect Hopf eigenvalues, and continue the trivial branch in j_0. Subsequently, we switch to a number of Turing modes, to the localized Turing mode, and from the generated snaking branch to some localized Hopf modes. Then we deal with plotting.

The further files from Table 2 are optional and/or for special tasks. The function `bpjac.m` implements $\partial_u(G_u\phi)$ which appears in BP continuation, cf. [6, §2.4], and to `hpjac.m` implementing a similar (large) matrix of derivatives of $G_u\phi_r$ and $G_u\phi_i$ needed for HP continuation, cf.[7, §3.6.1 and §4.4]. In 1D, these functions can be omitted using numerical Jacobians, but in 2D (or even 3D) this becomes rather slow. Finally, `secobra` is a modification of the library function `hobra`, which generates `[pars; m1; m2; m3; m4]` for branch–output, where $m_1 = T$ (period, 0 for steady states), $m_2 = \max(u_1)$, $m_3 = \min(u_1)$, and $m_4 = \|u\|_2$ as defined in [6, (51)].

Listing 4 shows the first step to compute the Turing–resp. Hopf bifurcation lines in j_0–α parameter space ([6, Fig. 4]) via BP and HP continuation. Here we choose α as the new primary active parameter (at position 2 in the parameter vector), and hence j_0 becomes a secondary active parameter. Our main comment is that to improve robustness we here need to increase the parameter `p.nc.del` (steplength for the finite differences for parameter derivatives, which are always done numerically) from its default 10^{-4} to 10^{-2}. The HP continuation works similarly via `p=hpcontini('0','hpt1',2,'hpc1')`, and then we plot the results. Listing 5 first shows a typical call of `hoswibra` and subsequent continuation of the periodic orbit (first 4 lines). Then we illustrate the rather special task for the system (2), of how to splice together a Turing mode and a Hopf mode to obtain a "mixed solution", see [6, Fig.6].

```
%% BP and HP cont, needs a larger delta for FDs for the param-derivatives
p=bpcontini('0','bpt1',2,'bpc1'); p.sol.ds=0.005; p.nc.dsmax=0.2;
p.nc.del=1e-3; p.plot.bpcmp=1; p.nc.tol=1e-6; p=cont(p,50);
```

Listing 4: BP continuation from cmds1b.m. Next we do the same for HP1, then plot, while at the end of cmds1b.m we plot the dispersion relations from [6, Fig. 4]

```
%% spatially homogeneous primary Hopf; use tl=40 points in t;
ds=0.1; aux.tl=40; p=hoswibra('0','hpt1',ds,4,'H1',aux); p.nc.dsmax=2;
p.sw.bifcheck=0; p.file.smod=1;% switch off bif.detection, save every point
p.hopf.flcheck=1; p.hopf.fltol=1e-6; % switch on multiplier comp
p.sw.verb=0; p=setbel(p,2,1e-4,5,@lss); p=cont(p,40); % use bel, and go!
%% splice together Hopf and Turing: choose xcut and j0 by trial and error
p1=loadp('T1','pt25'); p=loadp('H1','pt25'); xcut=-160; j0guess=3.35;
x=getpte(p);idx=find(x>xcut);%find indizes where to replace Hopf by Turing
p.u(p.nu+1)=j0guess; p.hopf.lam=j0guess; % set j0 to the guess j0guess
for j=1:p.hopf.tl % put Turing-soln into Hopf-data
 p.hopf.y(idx,j)=p1.u(idx); p.hopf.y(idx+p.np,j)=p1.u(idx+p.np);
end
```

Listing 5: Start of script cmds2.m, illustrating a basic call of `hoswibra`, and "how to mix solutions". The remainder of the script deals with continuation of the obtained branch, and plotting.

The demo `sh35disk`. In [6, §3.1], following [11], we consider the cubic–quintic Swift–Hohenberg (SH) equation $\partial_t u = -(1+\Delta)^2 u + \varepsilon u + \nu u^3 - u^5$ on a disk with Neumann BCs for u and Δu. Setting

$u_1 = u$ and $u_2 = \Delta u$ we rewrite the 4th order SH equation as the 2nd order system

$$M_d \partial_t \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} -\Delta u_2 - 2u_2 - (1-\varepsilon)u_1 + f(u_1) \\ -\Delta u_1 + u_2 \end{pmatrix} \tag{4}$$

with a singular dynamical mass matrix $M_d = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$, $f(u_1) = \nu u_1^3 - u_1^5$, and Neumann BCs for u_1 and u_2. The implementation again follows standard principles of pde2path demos, see Table 3 for the used files and comments. Compared to the demo seco, the functions q*.m are new. They implement-phase conditions [6, §2.4]: After standard preparations in shinit and oosetfemops, qy and qyder are the y–phase conditions for localized solutions on the half disk, see Listing 6, while on the full disk we also need qx and qrot and combinations thereof.

Table 3: Scripts and functions in sh35disk.

file	purpose, remarks
cmds1	Basic script, generating [6, Figures 7–9].
shinit	initialization; setup of half–disk domain and mesh, initialization with $u \equiv 0$.
oosetfemops	assembly of mass matrix M, Laplacian K, and of matrices for phase conditions.
sG, sGjac	rhs of (4), and Jacobian.
qy, qyder	phase condition $q^y(u) = \langle \partial_y u_{\text{old}}, u \rangle = (\texttt{Ky} * \texttt{p.u})' * \texttt{u}$, with $\partial_u q^y(u) = (\texttt{Ky} * \texttt{p.u})'$. Similar for ∂_x – or ∂_ϕ –phase conditions (qx or qrot), where $q^{rot}(u) = \langle (-y\partial_x + x\partial_y)u_{\text{old}}, u \rangle = (\texttt{Krot} * \texttt{p.u})' * \texttt{u}$, and combinations such as qxy for x–and y–phase condition, qxyr (all three) and their derivatives.
qyon	convenience function to switch on the y–phase condition; similar for qxon and qroton etc.
spjac	implements $\partial_u(G_u \phi)$ for fold continuation.
shJ, shbra	shJ computes the energy $\mathscr{F}$ [6, (53)], and is called and put onto output branch in shbra (modification of library function stanbra).
h2fdisk	(mirror u from) half disk to full disk. See also h2fdisk0 where instead of mirroring we extend u by 0.

```
function q=qy(p,u) % phase condition for transl.invariance in y
n=p.np; u0y=p.mat.Dy(1:n,1:n)*p.u(1:n); q=u0y'*u(1:n);
```

```
function qu=qyder(p,u) % derivative of transl.phase condition in y
qy=(p.mat.Dy(1:p.np,1:p.np)*p.u(1:p.np))';  qu=[qy, 0*qy];
```

Listing 6: Vertical shift phase condition qy and derivative qyder.

Remark 2. A special feature, also compared to the examples from [7], is that for (4) we use a piecewise quadratic FEM following [5]. The main advantage is that this is more robust than the default P^1–FEM wrt “branch jumping”. See also [8], and [10] for an online version of this demo. ⌋

The demo acdc. To show how dead core (DC) problems of type [6, (56) and (63)] can be treated with pde2path, we first consider modifications of the Allen–Cahn type equation from [1], given by

$$\partial_t u = c_2 \partial_x^2 u + f(u,x) \text{ in } \Omega = (0,1), \quad \partial_x u(0,t) = \partial_x u(1,t) = 0, \quad u|_{t=0} = u_0, \tag{5}$$

where $f(u,x) = u(1-u)(u-a(x))$ with $a \in C^1([0,1],(0,1))$. For fixed x, the ODE $\frac{\mathrm{d}}{\mathrm{d}t}u = f(x,u)$ has the two stable fixed points $u = 0$ and $u = 1$, and the unstable fixed point $u = a(x)$. The idea to allow an x–dependent $a(x)$ is that for non–constant choices of a, e.g., periodic a, and small $c_2 > 0$ there exist stable states u for (5) for which $u(x) \approx 0$ on some intervals $I_1, \dots, I_n$, and $u(x) \approx 1$ roughly on the complements, with narrow interfaces in between. This is partly motivated by applications, see [1], but mostly by the fact [3] that for any constant $a \equiv a_0 \in (0,1)$ states with interfaces are unstable and the only stable steady states are the constant states $u \equiv 0$ or $u \equiv 1$. Similar results also hold for the case of Dirichlet BCs: For instance, requiring $u(0,t) = 1, u(1,t) = 0$, the only stable steady state is monotonously decreasing and thus has exactly one interface.

Motivated by (5), to obtain a DC problem with possibly many stable DC solutions we thus consider the toy problem

$$\partial_t u = c_2 \Delta u + f(u,\lambda,x) \text{ in } \Omega, \quad u = 1 \text{ on } \partial\Omega, \quad u_+ = \max(u,0), \tag{6}$$

respectively its steady version, with parameters $c_2 > 0$ and $\lambda > 0$, first in 1D, $\Omega = (0,1)$, and with

$$f(u,\lambda,x) = \tilde{f}(u,x) - \lambda(u_+^\gamma - u), \quad \tilde{f}(u,x) = u(u-1)(a(x)-u). \tag{7}$$

We choose $a(x) = 0.6 + 0.2\cos(8\pi x)$, which later we also use in 2D as $a = a(r)$, $r = \sqrt{x^2+y^2}$. For any $x \in \Omega$ and $\gamma \in (0,1)$, $u = 1$ is always a fixed point of the ODE $\frac{\mathrm{d}}{\mathrm{d}t}u = f(u,\lambda)$, but as λ increases, $u = 1$ loses stability in a transcritical bifurcation. Consequently the trivial solution $u \equiv 1$ of (6) is stable for small $\lambda > 0$ but then loses stability at some $\lambda_0 > 1$ to an unimodal branch, which develops a dead core in the middle of the domain.

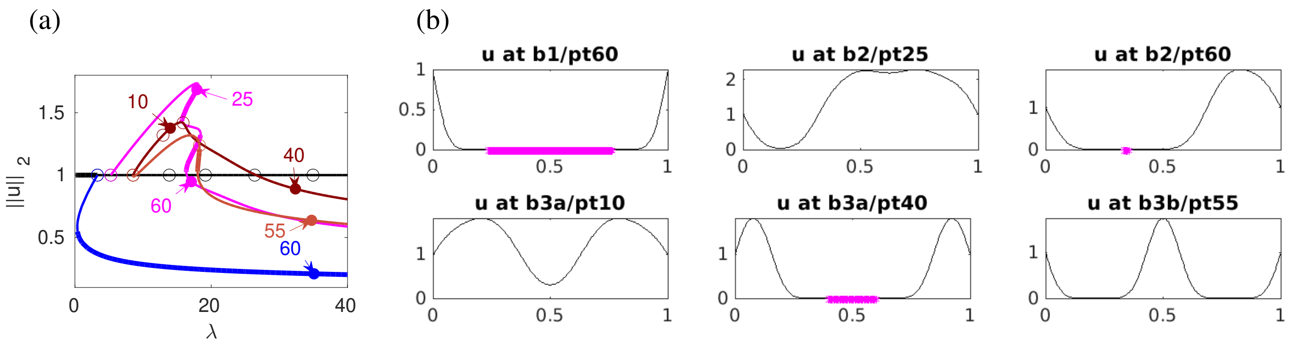

Figure 1: Results from `acdc/cmds1D` for(6) over $\Omega = (0,1)$ with $(\gamma, c) = (0.85, 0.01)$. (a) BD; trivial branch $u \equiv 1$ (black), and first three nontrivial branches, b1 (blue, transcritical, only 1 direction), b2 (magenta, pitchfork), b3a and b3b (dark/light brown, transcritical). (b) Samples, with dead cores ($u < 10^{-8}$) marked by magenta crosses in b1/60, b2/60, and b3a/40. For smaller γ we also find (stable) solutions with several (disjoint) dead cores.

In Fig. 1(a) we show a BD and some sample solutions for the case $\Omega = (0,1)$, $(\gamma, c_2) = (0.85, 0.01)$. Depending on the parameters, in particular γ and c_2, also some higher bifurcations (with kernels $\phi_\ell = \sin(\ell\pi x)$, $\ell \geq 2$) from the trivial branch lead to branches which develop stable interfaces and subsequently stable dead cores consisting of several disjoint intervals.

Table 4: Scripts and functions in `acdc`.

file	purpose, remarks
`cmds1D, cmds2D`	Basic scripts, generating Figures 1 and 2.
`acinit`	initialization; setup of domains and (initial) meshes, initialization with $u \equiv 1$.
`oosetfemops`	assembly of mass matrix M and Laplacian K.
`sG, sGjac, afu`	rhs of (6), Jacobian, and the x–(resp. r–) dependent function a.
`nloop, nloopext`	modifications of the Newton loop algorithms, incorporating (9).
`mypsol`	customized plotting to visualize DCs, see also `mpsol2D`.

Table 4 list the files for the implementation of (6), in 1D and 2D. The two main issues we need to deal with are the non–differentiability of f from (7) at $u = 0$, and with keeping $u \geq 0$ throughout, in particular during Newton iterations such as

$$u_{n+1} = u_n - [\partial_u G(u_n)]^{-1} G(u_n), \tag{8}$$

and similar for the extended systems [6, (22b)]. To keep $u \geq 0$ we make local copies of the `pde2path` library functions `nloop` and `nloopext` in the working directory, and, e.g., in `nloop.m` add the command

$$\mathtt{u1(1:p.nu)} = \mathtt{max(u1(1:p.nu),0)} \tag{9}$$

after computing the update $\mathtt{u1}{=}u_{n+1}$, and similar in `nloopext.m`. For $u \searrow 0$ we have $\partial_u f(u,\lambda,x) \nearrow \infty$ and f is not differentiable at $u = 0$. Thus, to approximate the (non-existing) "Jacobian" for $G(u) = -c_2\Delta u - f(u,\lambda,x)$ we use

$$G_u(u)v = -c_2\Delta v - h(u,\lambda,x)v \text{ with } h(u,\lambda,x) = \partial_u f(\max(u,\delta),\lambda,x), \tag{10}$$

with a small $\delta > 0$, namely $\delta = 10^{-6}$. Together, (9) and (10) can be seen as an ad hoc Newton method for (7), which exploits that DCs are valid solutions, and since we do *not* modify (7), we solve the original problem.[4]

In Fig. 2 we essentially study the same problem as in Fig. 1, now in 2D. We use an almost disk shaped domain, which however we slightly disturb to an ellipse with major axis $e = 1.1$ to break the rotational invariance. We also use a larger diffusion $c_2 = 0.05$ to avoid too steep interfaces, and start with a rather coarse mesh of $\mathtt{nt} \approx 3300$ elements. Naturally, steep interfaces suggest adaptive mesh refinement. In `pde2path`, mesh adaptation is based on user defined elements–to–refine–selector functions, linked as `p.fuha.e2rs`. The default choice is the library function `e2rs`, using a standard element wise error estimate η_τ, see [7, §4.2.1]. Here, instead of η_τ we just use the (discrete) curvature $(|\Delta u|)_\tau$ as an ad hoc selector, yielding the refined meshes in Fig. 2(b), with around 10000 elements. This mesh adaptation is needed to continue the nontrivial branches to larger λ, i.e., $\lambda > 20$, say. The primary bifurcating branch (c1) shows an elliptic DC for $\lambda > \lambda_c \approx 9.4$ (with free boundary roughly parallel to the domain boundary $\partial\Omega$), and the second branch shows a DC for $\lambda > 21$, which becomes non–convex for $\lambda > 22.5$.

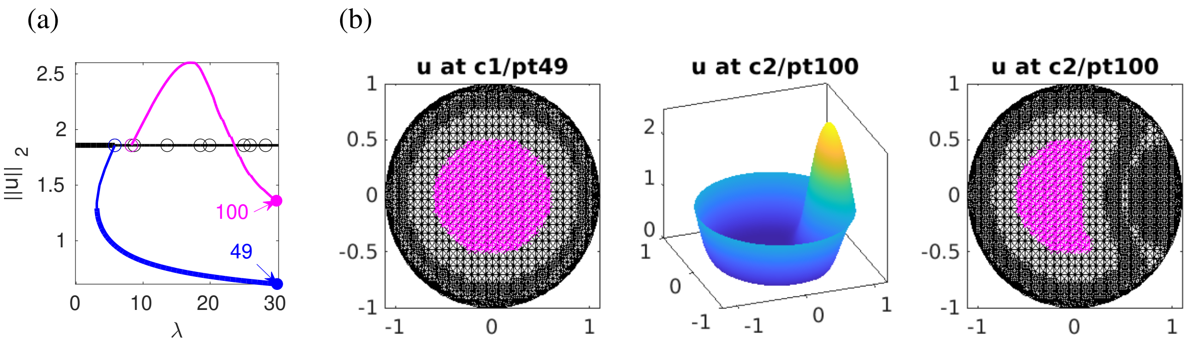

Figure 2: Results from `acdc/cmds2D` for (6) over an ellipse with $e = 1.1$, $(\gamma, c) = (0.85, 0.05)$. (a) BD of trivial and first two nontrivial branches c1 (blue) and c2 (magenta), with mesh adaptation each 10th continuation step. (b) Sample solutions, also illustrating the adapted meshes, with dead cores marked by magenta dots.

[4]Naturally, the false Jacobians for solutions involving DCs also generate wrong spectra, and the consequences of this for bifurcation detection and handling still need to be studied; see [6] for further comments.

The demo `schnakdc`**.** In [6, §3.3] we consider the Schnakenberg reaction diffusion system

$$\partial_t u = \Delta u - u_+^{1/m} + u^2 v, \quad \partial_t v = d\Delta v + \lambda - u^2 v. \tag{11}$$

The term $-u_+^{1/m}$ replaces the standard term $-u$, and for $m > 1$ is singular as $u \searrow 0$. This leads to DC pattern formation, including Hopf bifurcations from solution branches with DCs, where the (time–)oscillations at least near bifurcation are strongly localized at the "live" part $u > 0$. For the implementation we closely follow the demo `acdc`, where additionally we also use modified Newton loops for periodic orbits, namely `honloopexp`, similar as in (9). Table 5 lists and comments on the files used in the demo `schnakdc`.

Table 5: Scripts and functions in `schnakdc`; quite similar to `acdc`.

file	purpose, remarks
`cmds1D`, `cmds2D`	Basic scripts, generating [6, Figs 10 and 11].
`schnakinit`	initialization; setup of domains (1D and 2D).
oosetfemops	assembly of mass matrix M and Laplacian K.
`sG, sGjac`	rhs of (11), and Jacobian.
`nloop, nloopext`	modifications of the Newton loops, incorporating `upos`, and similar in `honloopext`.
`mypsol`	customized plotting to visualize DCs, see also `mpsol2D`, and `myhopl` (for Hopf orbits).

References

[1] S. B. Angenent, J. Mallet-Paret, and L. A. Peletier. Stable transition layers in a semilinear boundary value problem. *J. Differential Equations*, 67(2):212–242, 1987.

[2] M. Bollhöfer. ILUPACK V2.4, `www.icm.tu-bs.de/~bolle/ilupack/`, 2011.

[3] N. Chafee. Asymptotic behavior for solutions of a one-dimensional parabolic equation with homogeneous Neumann boundary conditions. *J. Differential Equations*, 18:111–134, 1975.

[4] M. Meixner, A. De Wit, S. Bose, and E. Schöll. Generic spatiotemporal dynamics near codimension-two Turing-Hopf bifurcations. *Phys. Rev. E*, 55(6, part A):6690–6697, 1997.

[5] C. Pozrikidis. *Introduction to finite and spectral element methods using MATLAB®*. CRC Press, Boca Raton, FL, second edition, 2014.

[6] H. Uecker. Continuation and bifurcation for Nonlinear PDEs – algorithms, applications, and experiments. *Jahresbericht DMV*, 2021.

[7] H. Uecker. *Numerical continuation and bifurcation in Nonlinear PDEs*. SIAM, Philadelphia, PA, 2021.

[8] H. Uecker. pde2path with higher order finite elements, 2021. Available at [9].

[9] H. Uecker. `www.staff.uni-oldenburg.de/hannes.uecker/pde2path`, 2021.

[10] N. Verschueren. Pattern formation on a finite disk using the SH35 equation. `https://nverschueren.bitbucket.io/sh35p2p.html`, 2021. Online tutorial.

[11] N. Verschueren, E. Knobloch, and H. Uecker. Localized and extended patterns in the cubic-quintic Swift-Hohenberg equation on a disk. *Phys. Rev. E*, (104):014208, 2021.