`pde2path` without finite elements

Hannes Uecker

Institut für Mathematik, Universität Oldenburg, D26111 Oldenburg, hannes.uecker@uni-oldenburg.de

September 8, 2021

Abstract

We describe by means of some examples how to run `pde2path` on "general" right hand sides, i.e., right hand sides not obtained from a PDE discretization by the built–in FEM of `pde2path`. The examples are "PDEs" on graphs, and discretization of standard PDEs by Chebychev and FFT based methods. For the latter we also explain a "matrix–free" setup of `pde2path` (not forming Jacobians, but only using their action).

Contents

1 Introduction

The `Matlab` package `pde2path` [UWR14, Uec19, Uec21b, Uec21a] is originally set up for numerical continuation and bifurcation analysis of systems of PDEs of the form

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

over bounded domains $\Omega\subset\mathbb{R}^d$, $d=1$, $d=2$, or $d=3$, (1D, 2D, 3D case), with various boundary conditions (BCs). In (1), $u=u(x,t)\in\mathbb{R}^N$, $t\ge 0$, $x\in\Omega$, $\lambda\in\mathbb{R}^p$ is a parameter (vector), $M_d\in\mathbb{R}^{N\times N}$ is a (dynamical) mass matrix, which may be singular, and the coefficients c, a, b and f may depend on

x, λ and u. For (1), pde2path can generate, by a few convenience calls, a finite element method (FEM) discretization based on, e.g., OOPDE [Prü21], including the computation of the FEM (differentiation) matrices. With these, the user can setup the (discretized) right hand side (rhs) $G(u, \lambda)$, and then compute branches of steady and time periodic solutions of (1). For details, and many demos of numerical continuation of branches of solutions of PDEs, see the various tutorials which together with the software and demos can be downloaded at [Uec21b].

Here we explain how to use pde2path without the built in FEM, which often only requires a few modifications of the standard setup. As examples we choose variants of typical model problems, namely: Allen–Cahn type of equations

$$\partial_t u = c\Delta u + \lambda u - u^3, \quad u = u(x, t) \in \mathbb{R}; \tag{2}$$

the (modified) Schnakenberg problem

$$\partial_t U{=}D\Delta U{+}F(U), \quad U{=}\begin{pmatrix} u \\ v \end{pmatrix}, \quad F(U){=}\begin{pmatrix} -u{+}u^2v \\ \lambda{-}u^2v \end{pmatrix} + \sigma(u - 1/v)^2\begin{pmatrix} 1 \\ -1 \end{pmatrix}, \tag{3}$$

with diffusion matrix $D = \left(\begin{smallmatrix} 1 & 0 \\ 0 & d \end{smallmatrix}\right)$ and parameters λ, σ as an example of a pattern forming reaction–diffusion system; the Swift–Hohenberg equation

$$\partial_t u = -(1 + \Delta)^2 u + \lambda u + \nu u^2 - u^3, \quad u \in \mathbb{R}, \tag{4}$$

as a fourth order problem. However, in contrast to the (other) tutorials at [Uec21b] we do not treat these via FEM, but as follows:

- (2) and (3) on networks (graphs), where Δ is replaced by the network Laplacian.
- (2) and (3) on boxes (1D and 2D) with a Chebychev spectral discretization [WR00, Tre02], and Dirichlet BCs (DBCs) or Neumann BCs (NBCs).
- (2) and (3) on boxes (1D and 2D) with NBCs via Fourier spectral differentiation based on the discrete cosine transform dct.[1]
- (2) on disks with DBCs and NBCs by mixing a Chebychev discretization in radius with a Fourier discretization in angle.

As usual in pde2path, the rhs (and Jacobians), of (2)–(4), including the BCs where applicable, will be encoded in functions sG (and sGjac). Using the Matlab graph class, the equations on networks require only a few lines of coding. The main changes compared to the standard treatment of (2)–(4) on various domains and with various BCs via the FEM concern:

- At init we need to set up (the discretization of) the spatial domain, and then in particular (in our setting) a function oosetfemops that computes the pertinent discretized differential operators (on networks, or based on cheb or dct) used in sG and sGjac; see also Remark 2.1.
- The default (solution) plotting in pde2path is tuned to the FEM setting. To adapt, we set p.plot.pstyle=-1, which means that the default plotsol calls a function userplot, which in each of the above settings can be provided quite easily.

In §2 we review the default FEM data structures and functions in pde2path, which we subsequently modify (drop or overload) for the above problems. In §3 we start with problems on networks, which due to the Matlab graph class (and auxiliary functions such as WattsStrogatz) require only a few lines of code. In §4 we consider (2) and (3) via Chebychev spectral discretization. In §5 we turn to the spectral discretization of (2) and (4) via FFT, choosing dct since NBCs are most natural for

[1]By a "spectral" method we mean any method based on expansion into global functions, such as (Chebychev) polynomials or trigonometric functions, as opposed to local functions like the hat functions or splines in the FEM. Often, the spectral methods are called pseudospectral if the basis is associated with a grid (as it inevitably is) and hence the interpolants are band limited. See, e.g., [Boy01, Introduction].

pattern forming systems, and in §6 we give a short summary, noting pros and cons of the methods presented in §4 and §5 compared to the standard FEM setup. The demos coming with this tutorial are subdirectories of `pde2path/demos/modtut`, see Table 1.

Table 1: Demo directories in `pde2path/demos/modtut`.

directory	remarks
`acG`, `schnakG`	(2) and (3), respectively, on graphs
`ac1Dcheb`	(2) in 1D based on `cheb` with Neumann BCs
`ac2DDBC`, `ac2DNBC`	(2) via `cheb` in 2D with DBCs and NBCs
`schnak2D`	(3) in 2D with NBCs
`ac1Dfou`	(2) in 1D with NBCs based on `dct`
`sh1Dfou`, `sh2Dfou`	(4) in 1D and 2D with NBCs based on `dct`
`sh1Dmfree`, `sh2Dmfree`	(4) in 1D and 2D with NBCs based on `dct`, matrix–free linear algebra
`acdiscDBC`, `acdiskNBC`	(2) on disks
`altfou`	further demos based on `dct`, alternative implementations
`altcheb`	alternative implementations of `ac1Dcheb` and `ac2DNBC` which implement the BCs more directly, which might be more easy to generalize.

2 Default data and initialization of a `pde2path` struct `p`

In the following we assume that as usual all problem data is contained in the `pde2path` struct `p`. In the standard FEM setting (`OOPDE`, [Prü21]), this includes the object `p.pdeo` (with sub–objects `fem` and `grid`), which provides methods to generate FEM meshes, code BCs, and ultimately assemble FEM matrices `M` (mass matrix) and `K` (e.g., Laplacian, including the BCs). Typical initializations and first continuation steps in the FEM setup then run as follows:

1. Call `p=stanparam(p)` to initialize most fields in `p` with default values (see source of `stanparam.m` for default fields and values). Set `p.sw.fem=-1` to flag the use of `r=sG(p,u)` and `Gu=sGjac(p,u)` as rhs and Jacobian (as opposed to `G`, `Gjac` with different signatures aimed at a different type of FEM assembly).[2]
2. Call a `pdeo` constructor, for instance `p.pdeo=stanpdeo1D(p,vararg)`, where here and in the following `vararg` stands for variable arguments.
3. In a function `oosetfemops` (in the current directory), use `p.pdeo.assema` to generate `p.mat.M` (*always required*) and `p.mat.K` (and possibly further FEM matrices, e.g., for BCs).
4. Use `p.mat.M` and `p.mat.K` in a function `r=sG(p,u)` to encode the 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`, in particular `p.mat` and its subfields such as the preassembled matrices `M` and `K` is typically useful for simple coding.
5. Initialize `p.u` with a first solution (or a solution guess, to be corrected in a Newton loop).
6. Call `p=cont(p)` to (attempt to) continue the initial solution in some parameter, including bifurcation detection, localization, and saving to disk.
7. The current solution is plotted during `cont` via calling `plotsol(p)`, and similarly for a posteriori plotting solutions (e.g., from disk). The standard behavior (controlled by the switch `p.plot.pstyle` and other switches in `p.plot`) of `plotsol` is tuned to the FEM discretization. However, if `p.plot.pstyle=-1`, then `plotsol` immediately calls a function `userplot`, to be user–provided (in the `Matlab`–path, typically in the current directory).

[2]E.g. `sG` is the default setup for the function handle `p.fuha.sG`, i.e., `p.fuha.sG=@sG`; see Remark 2.1.

8. Call `p=swibra(dir,bpt,newdir)` to attempt branch switching at branch point `bpt` in directory `dir`; subsequently, call `p=cont(p)` again, with saving in `newdir`.
9. Perform special tasks such as fold or branch–point continuation; use `plotbra(dir,pt,vararg)` to plot bifurcation diagrams, and `plotsol(dir,pt,vararg)` to plot sample solutions.

Steps 1,2 and 5, and a call to `oosetfemops` are typically combined into an init-function, for instance `p=acinit(p,vararg)`. Importantly, to save disk space, during continuation, data in `p.mat` is by default *not saved to disk* in the standard save–function `stansavefu`. Thus, the function `p=loadp(dir,pt)` for loading a point from disk (as in `swibra` in step 8, or for plotting from disk), calls `oosetfemops` to restore the FEM matrices.[3]

When not using the built–in FEM, then obviously step 2 must first be modified (for instance simply omitted). Next, `oosetfemops` from step 4 should generate `p.mat.M` (always) and possibly other matrices, to be used in `sG` and/or `sGjac`. Again, this should happen in the function `oosetfemops` (which may for instance only contain the command `p.mat.M=speye(p.nu)`) as `p.mat` is not saved and instead `oosetfemops` is called when reloading points.

Remark 2.1 The rhs, Jacobian, 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 your rhs 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 encode these in the respective demo directory. For many handles in `p.fuha` there are standard choices, e.g., `p.fuha.savefu=@stansavefu`, which we very seldomly modify. Functions for which the "default choice" is more likely to be modified include, e.g.,[4]

- `p.fuha.outfu=@stanbra; % signature out=stanbra(p,u), branch output;`
- `p.fuha.lss=@lss; % signature [x,p]=lss(A,b,p), linear systems solver` $x = A^{-1}b$. Other options include, e.g., `lssbel` (bordered elimination) and `lssAMG` (preconditioned GMRES using `ilupack` [Bol11]).

For downward compatibility (since it was introduced rather recently), there is an *optional* handle

- `p.fuha.setops`. If `p.sw.sfem=`±1, and `p.fuha.setops` is set to, e.g., `mysetops` (with signature `p=mysetops(p)`), then `mysetops` is called by `setfemops`, otherwise `oosetfemops` (usually from the current directory) is called.

Thus, using `p.fuha.setops` different methods for generating operators (matrices) can be tested. ⌋

3 Allen–Cahn and Schnakenberg on networks

The `Matlab` class `graph` provides all we need to deal with "PDEs" on networks (aka undirected graphs), namely a graph Laplacian, and powerful plotting of graphs. An undirected graph Γ consists of *nodes* (points) p_j, $j = 1, \dots, n$ and *edges* encoded in the adjacency matrix A with $A_{ij} = 1$ if there is an edge between p_i and p_j. The *degree* k_i of a node p_i is the number of connecting edges, i.e., $k_i = \sum_{j=1}^n A_{ij}$. The *graph Laplacian* is expressed by the matrix $L_{ij} = A_{ij} - k_i\delta_{ij}$ such that for a function $u : \Gamma \to \mathbb{R}$, or simply $u = (u_1, \dots, u_n)$, we have $\Delta u = Lu$. The "diffusive mobility" on the graph is then given by $c\Delta u$ with diffusion constant $c \geq 0$.

We consider two types of graphs, namely:

- Watts–Strogatz (WS) "small world" networks [WS98], which in `Matlab` can be generated via the convenience function `WattsStrogatz`; depending on the parameters $K \in \mathbb{N}$ (average degree,

[3]Expert users can of course modify this as desired, e.g., modify `stansavefu` (and `loadp`) to save (part of) `p.mat`; alternatively FEM matrices such as the stiffness matrix `K` (Laplacian) could as well be stored as, e.g., `p.K`. In both cases, calls to `assema` upon reload of a point can be omitted.

[4]and we'll precisely modify the output function `stanbra` and the linear system solvers also in the demos below

with theoretically $n \gg K \gg \ln n \gg 1$) and $\beta \in [0,1]$ (rewiring coefficient), these graphs feature local clustering (independent of n) and short average path lengths.

- Barabasi–Albert (BA) scale–free networks [AB02], which we generate via BAgraphA (as a mod of BAgraph_dir by Tapan P Patel, obtained from Matlab–central). These feature a power–law degree distribution k^{-3}, and compared to small world networks longer average path lengths (approximately proportional to $\log n$), and less clustering.

To conveniently use these functions, libs/misc contains the function G=mygraph(np,sw) where np can be used to specify the number of nodes, and sw=0,...,4 to specify the graph type.[5]

We make no attempt to review details of the above graphs, and of their spectral theory or the theory of Turing type pattern formation for RD systems on them, which is less developed than for the classical Turing instability. Instead, we refer to the above references, and to [NM10, Wol12, HNM14, MW16] and the references therein, and simply aim to explain how to do continuation and bifurcation for systems like (3) on networks in pde2path, recovering results similar to those from the above references. However, for simplicity we start with the scalar problem (2).

3.1 Allen–Cahn

Listing 1 shows the five function files used to encode (2) on a graph.

```
function p=acinit(p,par,np,sw) % AC on graph, init, standard, except lines 4-6
p=stanparam(p); screenlayout(p); p.nc.neq=1; % standard params, scalar problem
p.sw.sfem=-1; % use the sG/sGjac and oosetfemops setting (without OOPDE!)
p.G=mygraph(np,sw); % generate graph (or load from disk)
p.np=p.G.numnodes; p.nu=p.np; % store #of points/unknowns
p.plot.pstyle=-1; % flag to call userplot in plotsol
p.nc.neig=min(20,p.np); % number of evals for bif-checking
p.u=zeros(p.np,1); p.u=[p.u; par']; % initial guess, parameters appended
p.plot.auxdict={'c','\lambda','c2','c3'}; % parameter names
p.nc.ilam=2;  p.sol.xi=1/(p.nu); % contine in par(2); and weight for arclength
p.sol.ds=0.1; p.nc.dsmax=0.5; % initial and max steplength
p.sw.bifcheck=2; p.sw.verb=2; p.nc.mu1=2; p.nc.mu2=0.5; % bif-detection settings
```

```
function p=oosetfemops(p) % for AC on graphs; M=Id, L=graph-laplacian
p.mat.M=speye(p.np); p.mat.L=p.G.laplacian;
```

```
function r=sG(p,u)  % AC on graph rhs
par=u(p.nu+1:end); u=u(1:p.nu); % split in par and PDE u
lam=par(2); c2=par(3); c3=par(4); f=lam*u+c2*u.^2+c3*u.^3; % "nonlinearity"
r=par(1)*p.mat.L*u-f; % residual
```

```
function Gu=sGjac(p,u) % AC on graph, Jacobian
n=p.nu; par=u(n+1:end);  u=u(1:n); lam=par(2); c2=par(3); c3=par(4);
fu=lam+2*c2*u+3*c3*u.^2; Fu=spdiags(fu,0,n,n); % local Jac, turned into matrix
Gu=par(1)*p.mat.L-Fu;  % Jac=c*Lap-f_u
```

```
function userplot(p,wnr) % plot graph data
figure(wnr); clf; h=plot(p.G); h.MarkerSize=5; % matlab graph-plot + makeup
h.NodeCData=p.u(1:p.np); h.NodeLabel={}; colormap cool; colorbar;
title([p.file.dir '/pt' mat2str(p.file.count-1)]); set(gca,'FontSize',14);
deg=degree(p.G); [~,order]=sort(deg,'descend'); % reorder by degree
figure(10); clf; xv=log(1:p.np); plot(xv,deg(order),'r'); % plot by degree
hold on; plot(xv,p.u(order)); xlabel('ln i'); % plot u by degree + some makeup
title([p.file.dir '/pt' mat2str(p.file.count-1)]);
```

[5] For sw=0, np is used as filename to load a graph from disk; otherwise see the source of mygraph.m for parameter settings, and as always feel free to modify in any way.

```
axis tight; ylabel(''); set(gca,'FontSize',14);
```

Listing 1: `acinit`, `oosetfemops`, `sG`, `sGjac` and `userplot` from `modtut/acG`.

Most of `acinit` is standard, and the only “graph–specific” settings are in line 4–6. In `oosetfemops` we use the `laplacian` method from the `graph` class, and `sG` and `sGjac` are implemented in a completely standard way. Similarly, no “graph–specific” settings at all occur in `cmds1.m`, see Listing 2. In the first 3 lines of `userplot` we use `plot` from the `graph` class, followed by some “makeup” (font sizes etc), and in the remainder we order the nodes by degree for “1D plots”.

```
%% AC on graph; Watts-Strogatz "small world" via sw=2, or already saved in G1
close all; keep pphome;
%% init and cont trivial branch
p=[]; par=[1 -0.2 0 -1]; % parameters c,lam,quad,cubic
sw=2; np=100; %sw=0; np='G1'; % uncomment 2nd part to load fixed graph
p=acinit(p,par,np,sw); p=setfn(p,'tr'); p=cont(p,15); % run continuation
% G=p.G; save('G1','G'); % if you like the graph, save it to disk
%% swibra to nontrivial branches
p=swibra('tr','bpt1','b1',0.1); p=cont(p,10); % spatially homogeneous
p=swibra('tr','bpt2','b2',0.1); pause; p=cont(p,10); % pause to inspect tangent
p=swibra('tr','bpt3','b3',0.1); p=cont(p,10);
%% bifurcation diagram plot
f=3; c=5; figure(f); clf; plotbra('tr',f,c,'cl','k','lsw',0);
plotbra('b1',f,c,'cl','b','lsw',0); plotbra('b2',f,c,'cl','r', 'lab',10);
plotbra('b3',f,c,'cl','m','lab',10); ylabel('max(u)');
%% solution plot, use pause to inspect (and export) plot
plotsol('b2','pt10'); pause; plotsol('b3','pt10');
```

Listing 2: `acG/cmds1.m`, generating Figure 1

Figure 1 shows some sample results from `cmds1`. We continue the trivial branch $u \equiv 0$, and find bifurcation points at the eigenvalues λ_j, $j = 1, 2, \ldots$ of the Laplacian, which are generically simple, with eigenfunctions ϕ_j. In summary:

- The first bifurcation is at $\lambda_1 = 0$ to the spatially homogeneous branch $u \equiv \pm\sqrt{\lambda}$.
- For small increasing j the eigenfunctions ϕ_j at λ_j consists of more and smaller “clusters”; for larger j the structure of the ϕ_j becomes difficult to understand graphically.
- The “patterns” on the bifurcating branches are essentially determined by the eigenfunction at bifurcation. (This will be very different for the Schnakenberg problem on BA graphs below).

These results appear to be rather independent of the network size n (but do depend on the average degree K and rewiring β), and we essentially chose $n = 100$ for graphical reasons. On a laptop, Fig. 1 is computed in a few seconds, and runtimes stay small (on the order of tenths of seconds for a continuation step) up to $n = 1000$, say.

In `cmds2` (same structure as `cmds1`) we consider (2) on a BA graph with $n = 250$, see Fig. 1(c,d). The main difference here is that for the BA graph many eigenvalues already cluster around $\lambda = 1$, with the eigenfunctions difficult to distinguish visually. Nevertheless, the “patterns” on the bifurcating branches still essentially seem to be determined by the eigenfunction at bifurcation. Again, this will be fundamentally different for the Schnakenberg problem on a (the same) BA graph.

3.2 Schnakenberg

In the demo `schnakG` we consider (3) with parameters $(\sigma, d_1, d_2) = (-0.3, 1, 500)$ on a WS graph (the same as in Fig. 1(a,b)), see `cmds1` and Fig. 2, and on a BA graph (same as in Fig. 1(c,d)), see `cmds2` and Fig. 3. The basic setup is as in the demo `acG`, and the main difference is that now we have a 2–component system of equations. Thus, in `sG` we compose the 2–component Laplacian from the scalar one, see Listing 3. Moreover, we find Hopf bifurcations (and period doublings), and hence

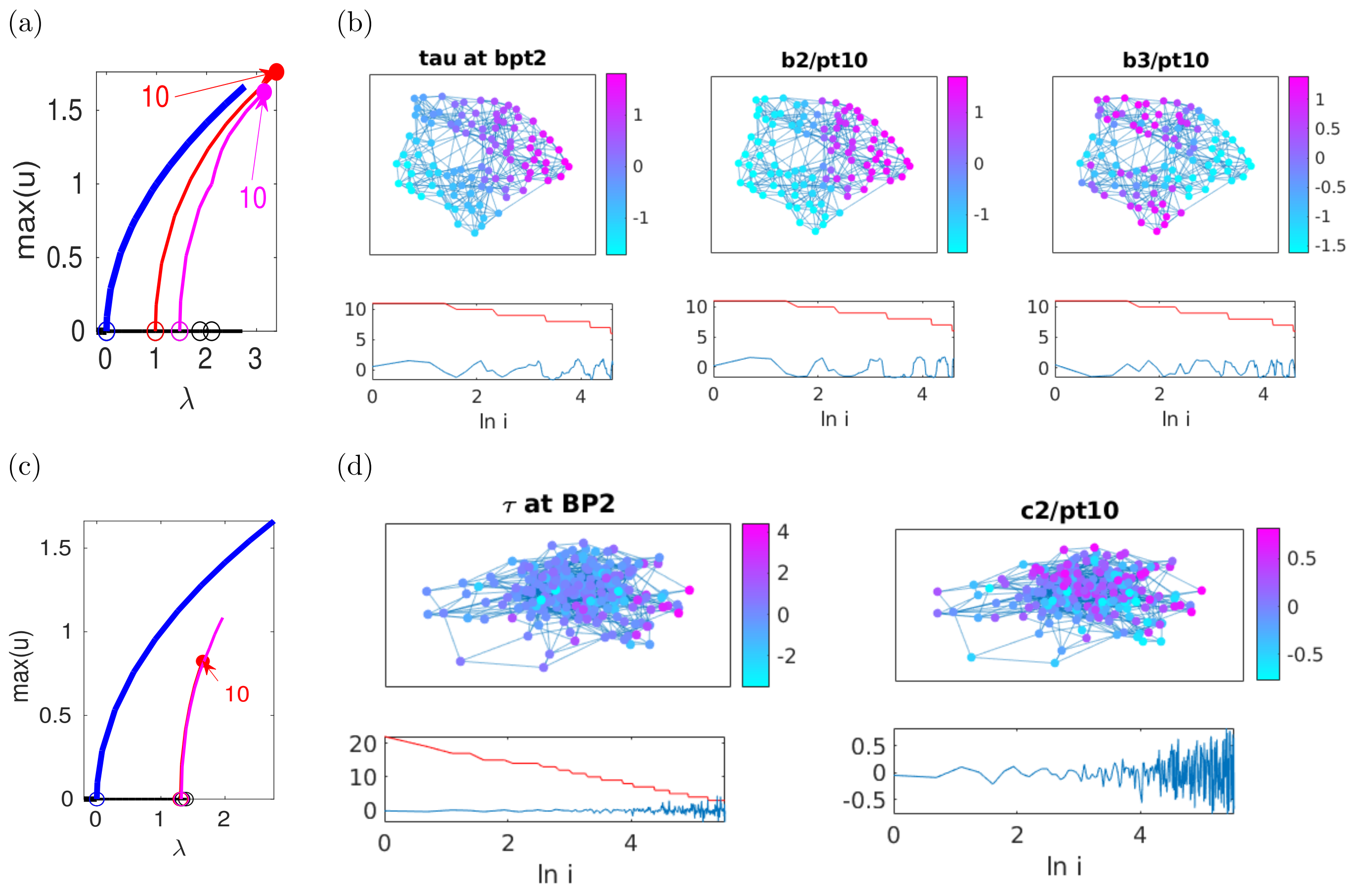

Figure 1: (a,b) (2) on WS graph, $n_p = 100$, $K = 4$, $\beta = 0.2$. (a) BD, with trivial $u = 0$ branch (black), and first three bifurcating branches b1 (blue, spatially homogeneous), b2 (red), and b3 (magenta). (b) tangent at BP2, and sample plots, via `plot` method of the `graph` class. Hubs (higher degree nodes) placed at center, nodes with smaller degree placed at the periphery. In the bottom plots, the nodes are ordered by decreasing degree (red line), and the blue line is u_i. (c,d) (2) on BA graph, $n_p = 250$, $m = 3$. (c) BD like in (a). (d) tangent at BP2, and sample plot b2/pt10. The red line in (b) illustrates the power law distribution of degrees.

also make a local modification of `hoplot` to plot Hopf orbits in the graph setting. In `cmds1` we then follow standard `pde2path` procedure, i.e., first continue the trivial branch, then a primary patterned branch, on which we find a Hopf bifurcation. Switching to the Hopf branch via `hoswibra`, we find a period doubling, and then switch to the bifurcating branch via `poswibra`, see the source of `cmds1.m` for details and further comments. The computation of the periodic orbits and Floquet multipliers runs very robustly.

```
function f=nodalf(p,u) % Schnakenberg
u1=u(1:p.np); u2=u(p.np+1:2*p.np); par=u(p.nu+1:end); % lam,sig,d
f1=-u1+u1.^2.*u2+par(2)*(u1-u2.^(-1)).^2;
f2=par(1)-u1.^2.*u2-par(2)*(u1-u2.^(-1)).^2;
f=[f1; f2];
```

```
function r=sG(p,u)  % (graph) PDE rhs
f=nodalf(p,u); % compute "nonlinearity" (everything but diffusion)
par=u(p.nu+1:end); d1=par(3); d2=par(4); u=u(1:p.nu); % split in par and PDE u
L=p.mat.L; K=[d1*L 0*L; 0*L d2*L];  % compose 2-compo Lapl. from scalar one
r=K*u-f; % the residual
```

Listing 3: `nodalf.m` (nonlinearity f, i.e., everything but diffusion), and `sG.m` from `schnakG`.

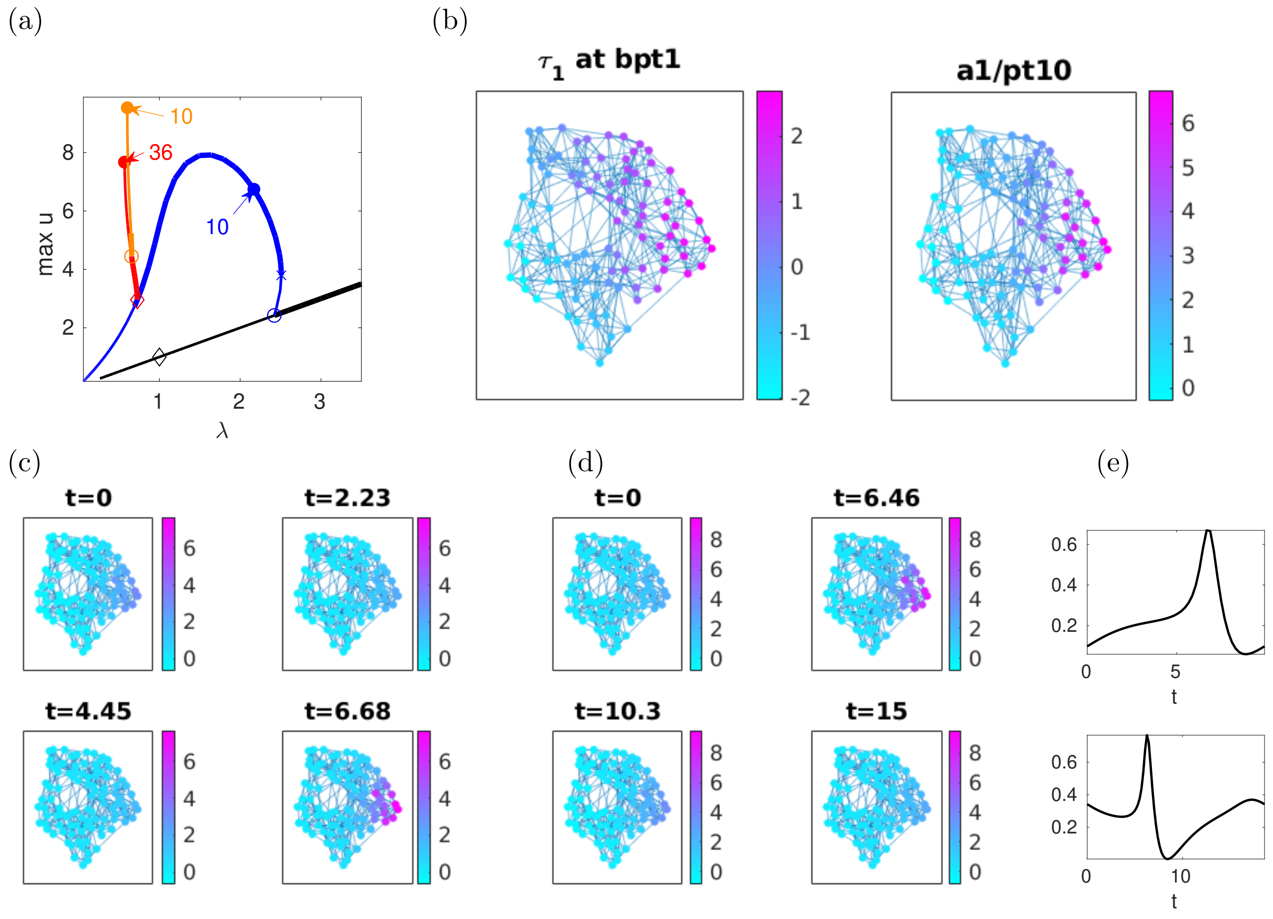

Figure 2: (3) with $(\sigma, d_1, d_2) = (-0.3, 1, 500)$ on a WS graph, n_p=100, K=4, β=0.2. (a) BD, with primary "Turing" branch (a1, blue), Hopf branch (h1, red), and PD (pd1, magenta). (b) tangent at at primary Turing bifurcation, and solution at pt10. (c) h1/pt36 and pd1/pt10, snapshots at selected time slices, showing "cluster oscillation". (e) Time–series for h1/pt36 (top) and pd1/10 (bottom) at a selected node.

In Fig. 2 we show sample results from `cmds1.m`. The primary Turing branch bifurcates subcritically and becomes stable after the fold, and the solutions keep their clearly clustered pattern determined by the eigenfunction at bifurcation all along the branch. The branch loses stability in a Hopf bifurcation at $\lambda \approx 0.8$. In the bifurcating Hopf branch the oscillations are localized in a cluster on "the right" in (c), and the same holds for the period–doubled branch bifurcating from the Hopf branch, see (d). In (e) we show time–series (of node 1, which is not in the strongly oscillating cluster). There are no further steady bifurcation points on the homogeneous (black) branch, but a Hopf point with bifurcation to a homogeneous oscillation at $\lambda = 1$. Altogether, this shows that the WS graph in the given parameter regime supports only clustered patterns (no single differentiated modes); this will be quite different on the BA graph considered next.

In Fig. 3 we consider (3) with $(\sigma, d_1, d_2) = (0, 0.4, 200)$ on the BA graph from Fig. 1(c,d). Here, the primary Turing bifurcation is at $\lambda \approx 9.2$, with many further small eigenvalues close by and subsequently bifurcating branches. Moreover, the continuation of the primary (blue) branch behaves quite differently than in Fig. 2: Away from the BP, the branch starts snaking [Kno08, ALB+10], and the already not clearly visible "pattern" at bifurcation decomposes into single differentiated nodes; on ways back an forth (in λ) single nodes (or several single modes) change their values from "neutral" (near the homogeneous solution $U_h = (\lambda, 1/\lambda)$) to "plus" $u \gg u_h$ or "minus" $u \ll u_h$, producing several stable "patterns". The same happens on the second (red) branch. These results are in agreement with [NM10, Wol12], and [MW16], to which we refer for analytical discussion.

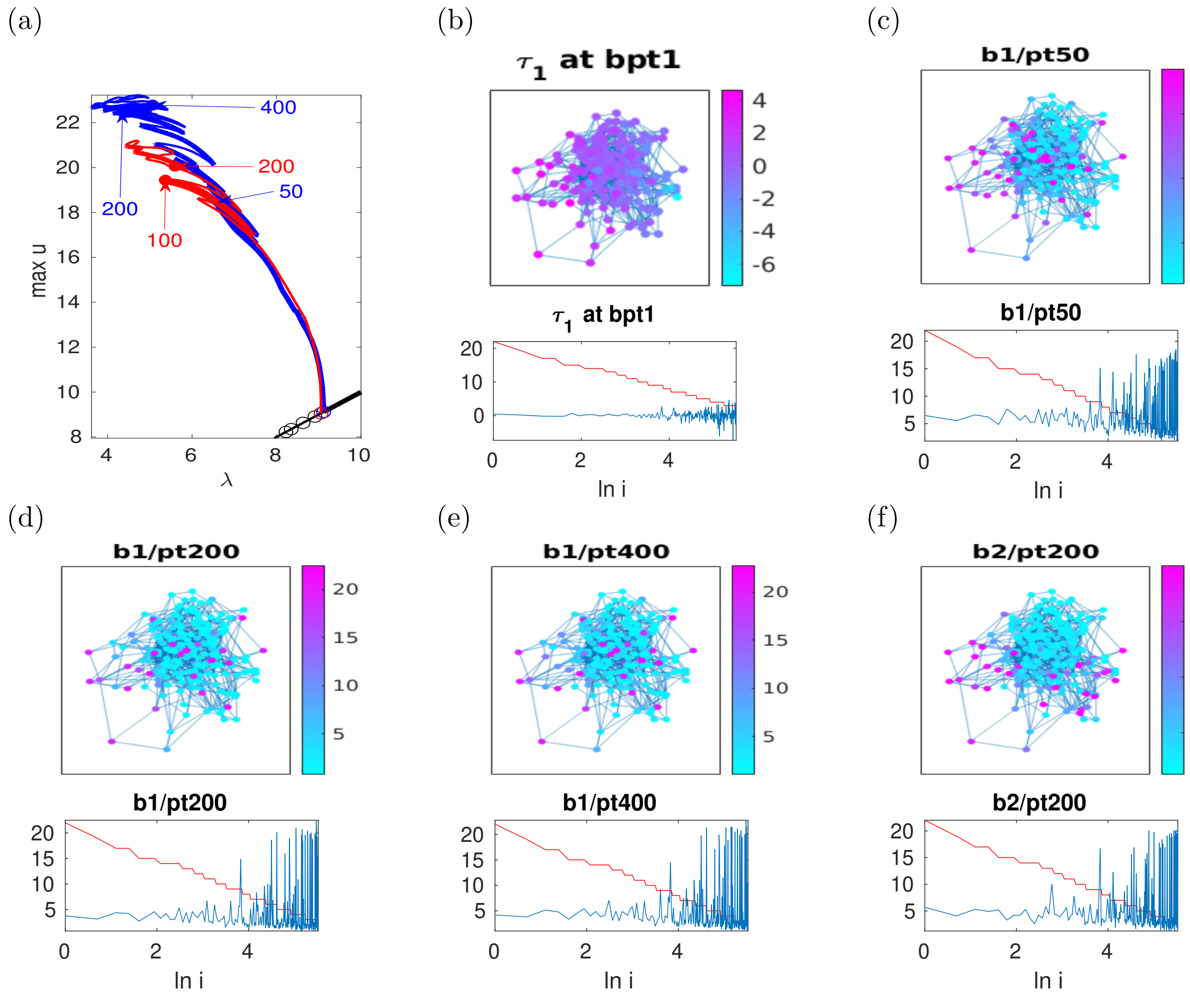

Figure 3: BA graph, $n_p = 250$, $m = 3$, $(\sigma, d_1, d_2) = (0, 0.4, 200)$. (a) BD of first two Turing branches b1 (blue) and b2 (red); snaking with many different stable solutions. (b) Tangent at first Turing (top), with nodes sorted by degree (see red line) at bottom. (c-e) samples from b1, (f) sample from b2.

4 Chebychev methods

We briefly recall some basics of Chebychev differentiation [WR00, Tre02], and then explain a `pde2path` Chebychev setup to compute branches for (2) and (3) over 1D and 2D boxes, with DBCs or NBCs. First restricting to the 1D case and the interval $I = [-1, 1]$, and functions $u : I \to \mathbb{R}$, the idea is to choose grid points

$$x_j = \cos(j\pi/n), \quad j = 0, \dots, n,$$

which cluster at the boundaries, and which in a certain sense minimize the interpolation error between a function $u : I \to \mathbb{R}$ and its interpolating polynomial p_n of degree at most n with $p_n(x_j) = u(x_j)$. Then we approximate derivatives of u (at the grid points) by the derivatives of p_n, i.e. set

$$\partial_x u(x_j) = \partial_x p_n(x_j). \tag{5}$$

Under some technical assumptions, see [Tre02, Theorem 5.5], for smooth (analytic) functions u it can then be shown that there exist constants $C, \alpha > 0$ such that

$$\sup_{x \in I} |u(x) - p_n(x)| \le C e^{-\alpha n}, \tag{6}$$

i.e., *exponential* convergence of the approximants p_n to u, and the same holds for derivatives $\partial_x^\nu u - \partial_x^\nu p_n$. Due to a connection to Fourier analysis, this is also called spectral differentiation, and the exponential error estimate is typical of spectral methods [Tre02].

On the other hand, since interpolation and differentiation are linear, (5) can be written as

$$(\partial_x u_j)_{j=0,\dots,n} = (Du)_j, \text{ with differentiation matrix } D \in \mathbb{R}^{(n+1)\times(n+1)}, \tag{7}$$

which for instance is computed by `cheb.m` which comes with [Tre02]. Moreover, higher order derivatives $\partial_x^m u$ are simply given by $(D_m u)_j$, where $D_m = D^m$. However, in contrast to the FEM, which yield sparse differentiation matrices D, D is a full matrix. Hence there is a trade–off between the spectral accuracy (6) (suggesting high accuracy with few grid points, i.e., small n) and the dense matrices $D, D_2, \dots$, in contrast to the sparse matrices (allowing large n) obtained from the FEM, but only yielding algebraic accuracy, e.g., $|\partial_x u'(x_j) - v_{f,j}| = \mathcal{O}(h^2) = \mathcal{O}(n^{-2})$ with $h = \mathcal{O}(1/n)$.

If $I = [-l_x, l_x]$ instead of $I = [-1, 1]$, then we obtain $\partial_x u_j$ by (7) and rescaling, i.e., $\partial_x u_j = \frac{1}{l_x}(Du)_j$. Higher dimensional derivatives can be obtained from the 1D matrices D_x and D_y, say, via tensor products. If for instance we want the Laplacian $\Delta u = (\partial_x^2 + \partial_y^2)u$ on a box $\Omega = [-l_x, l_x] \times [-l_y, l_y]$, then we generate 1D meshes $1 = x_0, \dots, x_{n_x} = -1$ and $1 = y_0, \dots, y_{n_y} = -1$ and associated D_x, D_y, namely (in `Matlab` notation) `[Dx,x]=cheb(nx); [Dy,y]=cheb(ny); Dx2=Dx^2; Dy2=Dy^2`. Then we set `L=kron(Dx2,eye(ny+1))./p.lx^2+kron(eye(nx+1),Dy2)./p.ly^2;` see below for further details, pertaining to BCs. Figure (4)(a) shows a resulting 2D rescaled 'tensor grid' with $l_x = 2, l_y = 1$ and, for graphical clarity, $n_x = 12, n_y = 6$, resulting in 112 mesh–points, and (b) shows the sparsity structure of the resulting `L`, with $n_z = 2352$ non–zeros out of $112^2 = 12544$ entries.

We consider two types of BCs in the demos: (i) DBCs and (ii) (homogeneous) NBCs. For (i), a useful strategy is to generate the Laplacian L for the full grid, but only treat the inner grid points as active. Thus, let u_a be the vector of values at the inner grid points. *Before* acting with L we extend u_a by u_b (for the boundary values), and *afterwards* we extract the residuals at the inner points again. A similar strategy can also be used for (ii), by introducing 'virtual' points beyond the boundary (see the subdirectories `ac1Dcheb, ac2DNBC` of `altcheb`), but for homogeneous NBCs ($\partial_n u = 0$), we can also put the contribution of the boundaries directly into L, for instance via the function `cheb2bc` from [WR00].

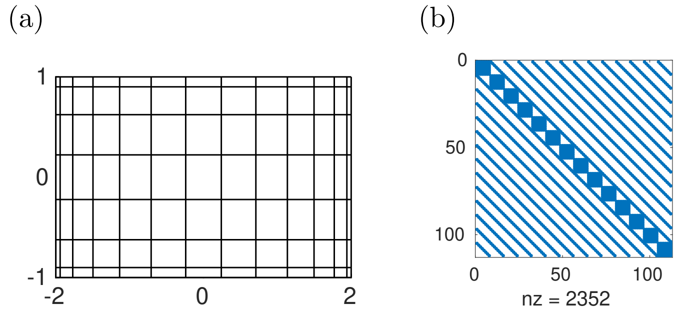

Figure 4: (a) tensor grid, (b) sparsity structure

Remark 4.1 Representing functions $u : [-1, 1] \to \mathbb{R}$ by Chebychev interpolants yields various further explicit formulas, e.g., for integration, root–finding, etc, and many of these have been put into the remarkable package(s) `chebfun` and `chebops`, see [PT10, TBD18], and `www.chebfun.org`. These also provide options to code and solve (adaptively, with guaranteed error bounds) 1D boundary value problems in 2 or 3 lines of `Matlab` code. However, here we restrict to just the function `[D,x]=cheb(n)` from [Tre02] and a few functions from [WR00], which for simplicity we put into `libs/misc`. ⌋

4.1 Allen–Cahn

ac1Dcheb. In the demo `ac1Dcheb` we consider (2) on $\Omega = (-2, 2)$ with homogeneous NBCs. The effect of these NBCs can be put into L directly, using `cheb2bc` from [WR00], see Listing 5, and we do not need to consider an extended mesh.[6] The function `acinit`, and the script `cmds1`, proceed almost exactly as for the graph case in §3.

```
function p=oosetfemops(p) % dx^2 via cheb
n=p.np; p.mat.M=speye(n); g=[0 1 0; 0 1 0]; % NBCs
[xt,D2]=cheb2bc(n,g); % following Reddy-Weideman
p.mat.L=D2/(p.lx^2); p.x=p.lx*xt; % rescaled Laplacian and mesh
```

```
function r=sG(p,u)  % PDE rhs
n=p.nu; par=u(n+1:end); u=u(1:n); lam=par(2); c2=par(3); c3=par(4); % split u
f=lam*u+c2*u.^2+c3*u.^3; f(1)=0; f(n)=0; % 'nonlinearity', zeroed on the bdry
r=-par(1)*p.mat.L*u-f; % compute L*u on extended domain
```

```
function Gu=sGjac(p,u) % Jacobian
n=p.nu; par=u(n+1:end); u=u(1:n); c=par(1); lam=par(2); c2=par(3); c3=par(4);
fu=lam+2*c2*u+3*c3*u.^2; fu(1)=0; fu(n)=0; % zero f_u on bdry
Fu=spdiags(fu,0,n,n); % local Jac, converted to matrix
Gu=-c*p.mat.L-Fu; % build Jac from bulk
```

```
function userplot(p,wnr) % mod of plotsol for Chebychev-diff setup
figure(wnr); clf; u=p.u(1:p.np); plot(p.x,u,'*-'); % plot
title([p.file.dir '/pt' mat2str(p.file.count-1)]); % and some makeup
axis tight; set(gca,'FontSize',14);
```

Listing 4: `oosetfemops`, `sG`, `sGjac` and `userplot` from `modtut/ac1Dcheb`.

In Fig. 4 we give the basic BD and sample solutions. The BPs from the trivial branch are at

$$\lambda_j = (j\pi/4)^2, \ \ \phi_j(x) = \sin(j\pi x/4), \ \ j = 0, 1, 2, \dots, \tag{8}$$

and even with $n = 30$ we obtain these with at least 5 digits accuracy.

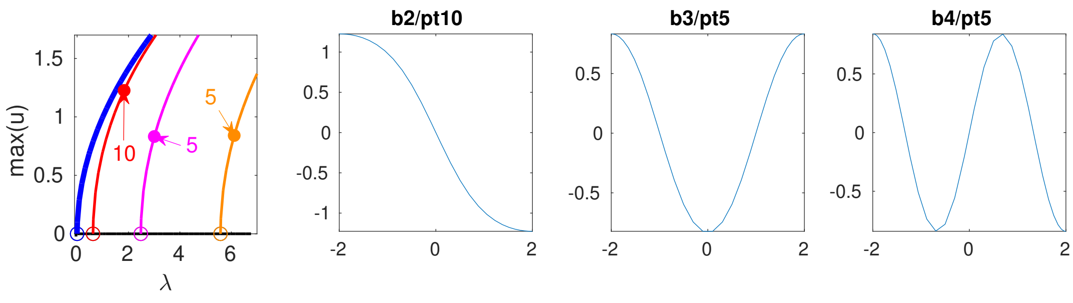

Figure 5: BD and sample solutions for (2) on $(-2, 2)$ with NBCs.

ac2DDBC. In the demo `ac2DDBC` we consider (2) on $\Omega = (-2, 2) \times (-1, 1)$ with the DBCs

$$u|_{\Gamma_1} = \gamma \cos(\pi y/2), \ \ \text{where } \Gamma_1 = \{x = 2\}, \quad u|_{\partial\Omega\setminus\Gamma_1} = 0. \tag{9}$$

[6]In `altcheb/ac1Dcheb/` we proceed differently, and this might be useful for generalizations: we choose a Chebychev mesh of $n + 2$ points, $x_1 = -1$, $x_{n+2} = 1$, and only treat $u(x_2), \dots, u(x_{n+1})$ as the $n = n_u$ unknowns or degrees of freedom (DoF). We can then extend **u** by the two boundary values $u(x_0) = u(x_1)$ and $u(x_{n+1}) = u(x_n)$ according to the NBCs, compute $\partial_x^2 u$ on the full (extended) grid, but then return the residual only at the genuine grid points. Similarly, we add the contribution from the boundaries to the Jacobian. Additionally, in `userplot`, the essential step is again to extend **u** by the boundary values. See also `altcheb/ac2DNBC` for a 2D version of this.

This is similar to [Uec21a, §6.3.1], where we use the FEM. The basic idea for the Chebychev discretization is explained above: We only use u on the inner grid points as DoF, and extend u on the boundary for evaluating the rhs, and for plotting. In `oosetfemops` we generate the Laplacian on the extended (tensor–product) grid, and, importantly, save the "bulk–indices" in `p.bui`.

```
function p=oosetfemops(p) % 2D, cheb, with DBCs
nx=p.nx; ny=p.ny; p.mat.M=speye(nx*ny); [Dx,x]=cheb(nx+1); D2x=Dx^2;
[Dy,y]=cheb(ny+1);  D2y=Dy^2; % Diff. matrices with two extra points for BCs
p.x=x; p.y=y; [xx,yy]=meshgrid(x,y); xx=xx(:); yy=yy(:);
p.lb=find(xx==-1); p.rb=find(xx==1); % left and right Bdry
p.bb=find(yy==-1); p.ub=find(yy==1); % bottom and top Bdry
p.bui=setdiff(1:(nx+2)*(ny+2),[p.lb;p.rb;p.bb;p.ub]); % bulk indizes
p.mat.L=kron(D2x,eye(ny+2))/p.lx^2+kron(eye(nx+2),D2y)/p.ly^2; % Laplacian
```

```
function r=sG(p,u)  % PDE rhs
n=p.nu; par=u(n+1:end); u=u(1:n); % split u into par and (active) field
lam=par(2); c2=par(3); c3=par(4); f=lam*u+c2*u.^2+c3*u.^3; % nonlin.
uf=zeros((p.nx+2)*(p.ny+2),1); % all u, to be filled by bulk and bdry values
uf(p.bui)=u(1:p.np); % filling in bulk
[xx,yy]=meshgrid(p.x,p.y); yy=yy(:); uf(p.rb)=par(5)*cos(pi/2*yy(p.rb)); % bd
r1=-par(1)*p.mat.L*uf;  % acting with L on full u,
r=r1(p.bui)-f;          % extract active (bulk) DoFs
```

```
function Gu=sGjac(p,u) % AC
n=p.nu; par=u(n+1:end); u=u(1:n); c=par(1); lam=par(2); c2=par(3); c3=par(4);
fu=lam+2*c2*u+3*c3*u.^2; Fu=spdiags(fu,0,n,n); % loc. Jac and convert to matrix
Gu=-c*p.mat.L(p.bui,p.bui)-Fu; % build Jac only for bulk
```

```
function userplot(p,wnr) % for AC2D with cheb and DBCs
figure(wnr); clf; n=p.nu; uf=zeros((p.nx+2)*(p.ny+2),1); % init full u with zero
par=p.u(p.nu+1:end); [xx,yy]=meshgrid(p.lx*p.x,p.ly*p.y);
uf(p.bui)=p.u(1:n); % put the bulk values into uf,
uf(p.rb)=par(5)*cos(pi/2*yy(p.rb)); % put bdry values and reshape to rectangle
uu=reshape(uf,p.ny+2,p.nx+2); surf(xx,yy,uu); axis tight; % plot and cosmetics
```

Listing 5: `oosetfemops`, `sG`, `sGjac` and `userplot` from `modtut/ac2DDBC`.

In Fig. 7(a) we give the basic BD with $\gamma = 0$ (homogeneous DBCs) $n_x = 40$ and $n_y = 20$, and $c = 1/2$; the BPs from the trivial branch are at $\lambda_{ij} = \frac{\pi^2}{2}(\frac{i^2}{16} + \frac{j^2}{4})$, $i, j = 1, 2, \ldots$, and are found with 5 digits accuracy; Fig. 7(b) shows a sample from continuation of `b1/b10` in γ.

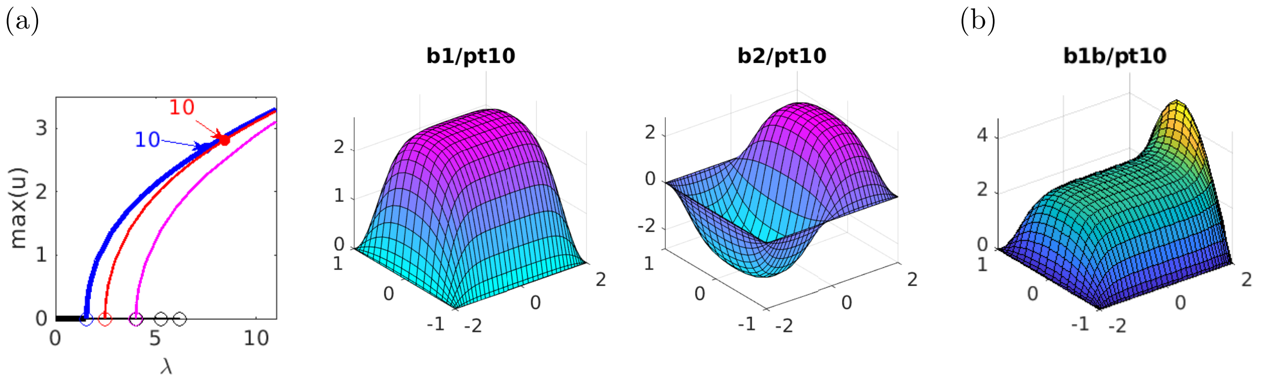

Figure 6: (a) BD for continuation in λ (b1 in blue, b2 in red) with homogeneous DBCs for (2) on $(-2, 2)\times(-1, 1)$ and sample solutions. (b) Sample solution after continuation in γ.

ac2DNBC. In the demo `ac2DNBC` we consider (2) on $\Omega = (-2,2) \times (-1,1)$ with (homogeneous) Neumann BCs $\partial_n u|_{\partial\Omega} = 0$, extending `ac1Dcheb` by tensor products, and storing the inner nodes in `p.bui`, see `oosetfemops` in Listing 6. The function `acinit.m` and the script `cmds1.m` are essentially as before. The BPs from the trivial branch now are at $\lambda_{ij} = \frac{\pi^2}{2}(\frac{i^2}{16} + \frac{j^2}{4})$, $i, j = 0, 1, \ldots$. In particular, the third BP at $\lambda = \pi^2/8$ is double, with kernel spanned by $\phi_{2,0} = \cos(2\pi x/4)$ and $\phi_{0,1} = \sin(\pi y/2)$, and the two bifurcating branches are on top of each other in the BD in Fig. 7.

```
function p=oosetfemops(p) % 2D, generate Chebychev-Lap with NBCs
nx=p.nx; ny=p.ny; p.mat.M=speye(nx*ny); g=[0 1 0; 0 1 0]; % BC code
[x,D2x]=cheb2bc(nx,g); [y,D2y]=cheb2bc(ny,g); % weideman
p.x=x; p.y=y; [xx,yy]=meshgrid(x,y); xx=xx(:); yy=yy(:);
lb=find(xx==-1); rb=find(xx==1); % left and right boundary
bb=find(yy==-1); ub=find(yy==1); % bottom and top boundary
p.bdi=[lb;rb;bb;ub]; % boundary indizes
L=kron(D2x,eye(ny))./p.lx^2+kron(eye(nx),D2y)./p.ly^2; % Lapl.
p.mat.L=sparse(L); p.x=x; p.y=y;
```

```
function r=sG(p,u)  % PDE rhs
n=p.nu; par=u(n+1:end); u=u(1:n); c=par(1); lam=par(2); c2=par(3); c3=par(4);
f=lam*u+c2*u.^2+c3*u.^3; f(p.bdi)=0; % zero nonlin on bdry
r=-c*p.mat.L*u-f;
```

```
function Gu=sGjac(p,u) % AC, with DBCs
par=u(p.nu+1:end); u=u(1:p.nu); c=par(1); lam=par(2); c2=par(3); c3=par(4);
fu=lam+2*c2*u+3*c3*u.^2; fu(p.bdi)=0; % local Jac, zeroed on boundary
n=p.nu; Fu=spdiags(fu,0,n,n); % local Jac, converted to matrix
Gu=-c*p.mat.L-Fu;      % build Jac
```

Listing 6: `oosetfemops`, `sG` and `sGjac` from `modtut/ac2DNBC`.

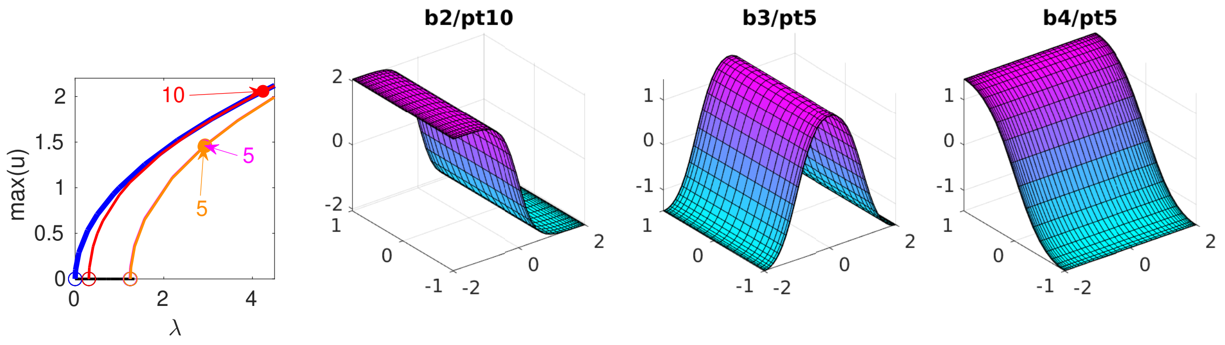

Figure 7: BD (with b1 (spatially homogeneous) in blue, b2 in red, b3 in orange, b4 in magenta), and sample solutions for (2) on $(-2,2) \times (-1,1)$ with NBCs.

4.2 Schnakenberg

In the demo `schnak2D` we essentially use the ideas from `ac2DNBC` to consider (3) with $(\sigma, d) = (0, 60)$ on $\Omega = (-l_x, l_x) \times (-l_y, l_y)$ with $l_y = 2\pi/k_c$, where $k_c = \sqrt{\sqrt{2} - 1}$ is the analytically known critical wave number, yielding the critical λ value $\lambda_c = \sqrt{60}\sqrt{3 - \sqrt{8}}$, cf. [Uec20, §4]. The specific domain Ω (its aspect ratio) pertains to a so called hexagonal dual lattice, and the first BP from the homogeneous branch $(u, v) = (\lambda, 1/\lambda)$ is double, with kernel spanned by $\cos(2\pi x/l_x)\phi$ and $\cos(\pi x/l_x)\cos(y/l_y)\phi$ with $\phi \in \mathbb{R}^2$. The primary bifurcating branches then are in the form of hexagons (transcritical) and stripes (supercritical). Again see [Uec20, §4] for background, and in particular [Uec20, Fig. 22] for a basic BD of (3) on the same Ω, computed by the standard FEM setup, with $n_u \approx 2700$ DoF. Here, in `cmds1.m` we obtain the same BD, see also Fig. 8, with $n_u = 1840$ DoF, which we can actually decrease to $n_u = 1020$ (choose $n_x = 30$ in `cmds1.m`).

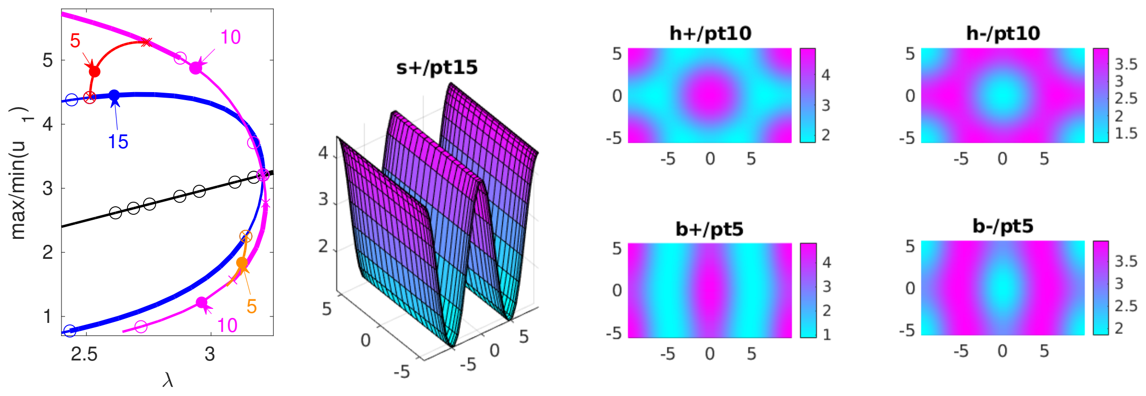

Figure 8: BD and sample solutions for (3) on $\Omega = (-l_x, l_x) \times (-l_x/\sqrt{3}, l_x/\sqrt{3})$ with NBCs, $l_x = 2\pi/k_c \approx 9.76$, $n_x = 40$, $n_y = 23$. Stripes (`s`$\pm$ blue), up hexagons (or spots `h+`) and down hexagons (or gaps `h-`, both red), and mixed modes (or beans `b`$\pm$, red and orange). In the BD we use $\max u_1$ for the $+$ branches, and $\min u_1$ for the $-$ branches.

As already said, for the implementation we reuse the ideas from above, and in particular the computation of the scalar Laplacian `p.mat.L` in `oosetfemops` as in `ac2DNBC`. From this we compose the 2–component diffusion matrix in `sG.m`, see Listing 7, where the nodal nonlinearity `nodalf` takes *exactly* the same form as on the graphs in §3.2, or in the standard FEM formulation.

```
function r=sG(p,u) % Schnakenberg
par=u(p.nu+1:end); f=nodalf(p,u); L=-p.mat.L; % nonlin and -Lapl
f(p.bdi)=0; f(p.np+p.bdi)=0; % zero nonlin on bdry for both comp
K=[L,0*L;0*L,par(3)*L]; r=K*u(1:p.nu)-f;  % diffusion matrix and residual
```

Listing 7: `sG` from `modtut/schnak2D`.

In `cmds2.m` we run the same problem on $\Omega = (-l_x, l_x) \times (-l_y, l_y)$, $l_x = 6\pi/k_c$, $l_y = l_x/(2\sqrt{3})$, $n_x = 100$, $n_y = 29$. This somewhat elongated domain roughly corresponds to [Uec20, Fig.23], and like there we obtain a snaking branch of a front between hexagons and stripes. However, already with the resulting relatively small `p.nu=5800`, the numerics become rather slow, i.e., significantly slower than in the original FEM setup with `p.nu` ≈ 20.000, but very sparse matrices. The non–sparse matrices suggest to use (preconditioned) iterative solvers (uncomment, e.g., line 11 in `cmds2`) but this does not yield a significant speedup.[7]

Thus, while these are rather superficial comparisons, so far the trade–off between spectral accuracy and relatively dense matrices (Chebychev) vs h^2–accuracy and sparse matrices (standard FEM) here lets the FEM win.

5 Fourier methods

As a second alternative to the built–in FEM, we explain how to discretize problems such as (2) and (4) via fast Fourier transform (FFT) $\mathcal{F}$ based methods. We focus on NBCs, and hence specifically

[7]This might change with matrix–free methods, which we did not yet check for the `cheb` discretizations, but which will be explained for FFT discretizations in the next section.

use the discrete cosine transform `dct`. FFT based methods are in particular simple and strong for constant coefficient parabolic problem of the form

$$\partial_t u = -G(u) := -L(\partial_x)u + f(u), \quad u|_{t=0} = u_0, \tag{10}$$

with periodic BCs, or DBC, or NBC, on a box $\Omega = (0, l_1) \times \ldots \times (0, l_d) \subset \mathbb{R}^d$. Taking the FFT of (10) yields

$$\partial_t \hat{u} = -\hat{G}(\hat{u}) = -\mu(k)\hat{u} + \hat{f}(\hat{u}), \quad \hat{u}|_{t=0} = \mathcal{F}u_0, \tag{11}$$

where $k \in \mathcal{L} = \pi\mathbb{Z}/l_1 \times \ldots \times \pi\mathbb{Z}/l_d$ are the wave vectors from the lattice $\mathcal{L}$, $\mu(k)$ are Fourier multipliers, e.g., $\mu(k) = |k|^2$ if $L = -\Delta$, and $\hat{f}(\hat{u}) = \mathcal{F}f(\mathcal{F}^{-1}\hat{u})$, which for the typical case of polynomial f can also be expressed via convolutions of $\hat{u}$.

Next we set $\hat{u}^j(k) = \hat{u}(t_j, k)$, $t_j = hj$, and using $\partial_t \hat{u}(t_j, \cdot) \approx \frac{1}{h}(\hat{u}^{j+1} - \hat{u}^j)$ approximate (11) by

$$\frac{1}{h}(\hat{u}^{j+1} - \hat{u}^j) = \mu(k)\hat{u}^{j+1} + \hat{f}(\hat{u}^j), \tag{12}$$

which yields the time stepping

$$\hat{u}^{j+1} = \hat{u}^j + h\eta(k,h)\hat{f}(\hat{u}^j), \quad \eta(k,h) = \frac{1}{1 + h\mu(k)}. \tag{13}$$

This is a semi–implicit Fourier method (as the linear terms on the rhs of (12) are evaluated at the next time–step $j+1$), and in a nutshell the great advantage is that usually (e.g., for $\mu(k) = |k|^2$ and similar), the multipliers η have $|\eta(k,h)| < 1$, and become smaller for larger h, and $|\eta(k,h)| \to 0$ as $|k| \to \infty$ (damping of higher Fourier modes). In particular, there are no stiffness–related (CFL–like) time stepsize conditions. Moreover, $\hat{f}$ can be evaluated in $\mathcal{O}(n \log n)$ time using FFT, including some possibly necessary de–aliasing, where n is the number of Fourier modes (=number of spatial DoF) used for the numerics. See [Uec09] for an ad hoc introduction to FFT time stepping, and the references therein for background.

For FFT methods for steady continuation problems, which require Jacobians of the rhs

$$G(u) = L(\partial_x)u - f(u) \text{ of (10), or of } \hat{G}(\hat{u}) = \mu(k) - \hat{f}(\hat{u}) \text{ in (11),}$$

the problem is that there is no way to avoid full Jacobians, because $\mathcal{F}$ and $\mathcal{F}^{-1}$ represent full matrices F and F^{-1} (with actually $F^{-1} = F^H$). Thus, if we work on G and want to evaluate $L(\partial_x)u$ as $F^{-1}\mu(k)Fu$ then $L(\partial_x)u = \partial_u(F^{-1}\mu(k)Fu) = F^{-1}\mu(k)F$ is full, even though $\mu(k)$ is nice and diagonal. Thus, also

$$J = \partial_u G = F^{-1}\mu(k)F - \partial_u f(u), \tag{14}$$

is full, although the second term is diagonal. Similarly, working entirely in Fourier–space as in (11), $\partial_{\hat{u}}\hat{f}(\hat{u}) = \partial_{\hat{u}}(Ff(F^{-1}\hat{u})) = F(\partial_u f(\hat{u}))F^{-1}$, and hence the second term in

$$\hat{J} = \partial_{\hat{u}}\hat{G} = \mu(k) - F(\partial_u f(u))F^{-1} \tag{15}$$

generates a full Jacobian (also if f is a polynomial and $\hat{f}(\hat{u})$ is evaluated via convolutions). Nevertheless, FFT–based continuation methods can be used efficiently on pattern forming problems with smooth solutions, because again smoothness in physical space is related to fast decay of the Fourier coefficients, namely exponential decay in case of analytic functions; see, e.g., [Tre02, Theorem 4.1]. Thus, we may hope to achieve high accuracy using the spectral differentiation $\partial_x^j u = \mathcal{F}^{-1}(\mathrm{i}k)^j\mathcal{F}u$ with relatively small n. See, e.g., [SAKR16, SAKR18] for continuation results based on FFT, also going to rather large scale, and in particular [Tuc20] and [SN16] and the references therein for (more general) methods for continuation in large scale problems with non–sparse (or only mildly sparse) Jacobians.

The full Jacobians suggest (preconditioned) iterative methods[8] for the linear systems that need to be solved in Newton steps, and similarly for the inverse vector iteration used in `eigs` to compute small eigenvalues (and associated eigenvectors) for bifurcation detection and branch switching. The preconditioner needs to take care of the stiffness generated by the differential operator (in x or in k). For this, (14) and (15) are quite different. In (14), $L(\partial_x) = F^{-1}\mu(k)F$ is full but only needs to be computed once, which suggests to just once compute an approximate inverse of L as a preconditioner, for instance via an incomplete LU decomposition (`ilu`) $L(\partial_x) \approx LU$, or, since $\mu(k)$ is real and diagonal and $L(\partial_x)$ as a matrix is symmetric, an incomplete Cholesky `ichol` decomposition $L(\partial_x) \approx L_C L_C^T$.[9] This *can* be used, see the demos `ac1Dfoux` and `sh*Dfoux` in `altfou`, but we find it more efficient to work directly in Fourier space and hence with $\hat{J}$ from (15).

The simple diagonal form of the multipliers $\mu(k)$ suggests a particularly simple and efficient preconditioner for the (full) matrix diag$\mu(k) + F(\partial_u f(u))F^{-1}$, which theoretically needs to be formed in every step (but should and easily can be avoided in a matrix free implementation). Assuming $\mu(k) \geq 0$ (as for $L(\partial_x) = -\Delta$ with $\mu(k) = |k|^2$ and $L(\partial_x) = (1+\Delta)^2$ with $\mu(k) = (1-|k|^2)^2$), we choose a $\delta > 0$ (typically $\delta = 0.1$, chosen heuristically) and then $L_C = \sqrt{\mu(k) + \delta}$ as left and right preconditioner in Fourier space.

Finally, the most expensive ($\mathcal{O}(n^3)$) term in (15) is $-F(\partial_u f(u))F^{-1}$, due to the full matrix products, and hence should not be formed. In a matrix free method, all that is needed is the action of $\hat{J}$ on a vector $\hat{v}$, namely, in `Matlab` notation

$$\hat{J}\hat{v} = \mu(k)\hat{v} - \texttt{dct}(\partial_u f(\texttt{idct}(\hat{u}))\texttt{idct}(\hat{v})), \tag{16}$$

where `idct` and `dct` run in $\mathcal{O}(n \log n)$.[10] This can be passed to an iterative solver such as `gmres` as a function handle, usually called `afun`, see, e.g., Listings 11 and 15.

In the following we first briefly consider the demo `ac1Dfou`, but then explain important tricks via the demos `sh1Dfou` (1D), `sh1Dmfree` (matrix free version of `sh1Dfou`) and the 2D versions `sh2Dfou` and `sh2Dmfree`. In Remark 5.1 at the end we comment on alternative versions `ac1Dfoux` and `sh*Dfoux` which work with (14) but are altogether slightly less efficient.

5.1 Allen–Cahn 1D

In the (simple) demo `ac1Dfou` we treat (2) on $\Omega = (0, 2\pi)$ with NBCs. The appropriate FFT then is `dct`. Given $u = (u_1, \dots, u_n)$, the standard form of `v=dct(u)` assumes an even (corresponding to NBCs) extension of u across u_1 at $x = 0$ to a 4π periodic function, and stores the discrete cosine coefficients as `v` $= (\hat{u}_0, \hat{u}_1, \dots, \hat{u}_{n-1})$. Thus, the multipliers corresponding to $-\partial_x^2$ are $\mu = (0, 1, 4, \dots, (n-1)^2)$. The transform can also be expressed as `v=F*u` with unitary (orthogonal) `F=dctmtx(n)`, such that $F^{-1} = F'$. Thus, $-\partial_x^2 u$ can be expressed as $F^T \text{diag}(\mu) F$, and this is implemented (with the rescaling to $(0, l_x)$ instead of $(0, 2\pi)$) in `oosetfemops`, see Listing 8. With this `p.mat.k2`, the remaining functions `sG, sGjac, userplot, acinit` and the script `cmds1` work *exactly* as in `ac1Dcheb`, yielding high accuracy for a discretization with $n = 30$ points ($n = 30$ Fourier modes, 6 digits accuracy for the first four

[8]which are also well suited for *matrix free methods* where the Jacobian J or $\hat{J}$ is never formed, see below

[9]The `Matlab` solvers such as `gmres, bicg, ...` generally allow left–right preconditioners, which are generally chosen as L, U (`ilu`) or L_C, L_C^T (`ichol`).

[10]Alternatively, instead of `idct` and `dct` we may use $u = F^T\hat{u}$ and $\hat{u} = Fu$ with the preassembled (orthogonal) `dct` matrix F. This is $\mathcal{O}(n^2)$, and thus still much faster than the $\mathcal{O}(n^3)$ for forming $-F(\partial_u f(u))F^T$, and we use this formulation in 2D, since preassembling the 2D F via tensor products of 1D F matrix gives a clearer code, and altogether for our n of interest up to $\mathcal{O}(10^4)$ seems at least as fast as the $\mathcal{O}(n \log n)$ for `dct2` and `idct2`.

BPs). In a local mod of stanbra we put the (normalized) L^2–norm $\left(\frac{1}{|\Omega|}\int_\Omega u^2\,\mathrm{d}x\right)^{1/2}$ on the branch via approximation as a Riemann sum.

```
function p=oosetfemops(p) % FFT based, u in Fourier-space
kf=pi/p.lx; n=p.np; p.mat.M=speye(n);
kv=kf*[0:n-1]'; % F-vectors (normalized to (0,lx))
p.mat.k2=kv.^2; p.mat.F=dctmtx(n); % store multipliers,  and dct matrix
```

```
function out=stanbra(p,u) % mod of stanbra to also put the L2-norm on the branch
uf=u(1:p.nu); np=p.nu/p.nc.neq; upde=p.mat.F'*uf; ul2=sqrt(sum(upde.^2)/p.nu);
out=[u(p.nu+1:end); max(abs(upde(1:np))); min(abs(upde(1:np))); ul2];
```

```
function r=sG(p,u)  % PDE rhs in Fourier
n=p.nu; par=u(n+1:end); uf=u(1:n); lam=par(2); c2=par(3); c3=par(4);
F=p.mat.F; u=F'*uf; f=lam*u+c2*u.^2+c3*u.^3; ff=F*f; % nonlin.
r=par(1)*p.mat.k2.*uf-ff;
```

Listing 8: oosetfemops, stanbra and sG from modtut/ac1Dfou.

5.2 Swift–Hohenberg

For the fourth order SH equation (4), i.e.,

$$\partial_t u = -(1+\Delta)^2 u + \lambda u + \nu u^2 - u^3, \quad u \in \mathbb{R}, \tag{17}$$

we set $L = (1+\Delta)^2$ with symbol $\mu(k) = (1-|k|^2)^2$ and consider $f(u) = \lambda u + \nu u^2 - u^3$ as the nonlinearity. Over, e.g., $\mathbb{R}^2$, the circle $|k| = 1$ of wave vectors becomes unstable at $\lambda = 0$, and over bounded boxes Ω it depends on the domain size which of the now discrete wave vectors first becomes unstable. Two standard cases are (i) a square with side–lengths $2\pi l$, $l \in \mathbb{N}$, where the two critical wave vectors at $\lambda = 0$ are $k = (1,0)$ and $k = (0,1)$, and (ii) a rectangle with side lengths $l_x = 2\pi l_1$ and $l_y = 2\pi l_2/\sqrt{3}$, $l_{1,2} \in \mathbb{N}$, where the three critical wave vectors are $k_1 = (1,0)$, $k_{2,3} = (-1/2, \pm\sqrt{3}/2)$. The latter, similarly used already in Fig. 8, corresponds to a so–called hexagonal dual lattice.

5.2.1 1D

sh1Dfou. First we consider the 1D case, over $\Omega = (0, 12\pi)$, such that the first three BPs are at $\lambda = 0$ ($k = 1$), $\lambda = 0.0255$ ($k = 11/12$), $\lambda = 0.0301$ ($K = 13/12$). Listing (9) shows the generation of the differentiation matrix in oosetfemops, and the implementation of sG and sGjac.

```
function p=oosetfemops(p) % SH via dct matrix F and multipl. matrix L
n=p.np; p.mat.M=speye(n); % mass matrix is Identity
kfx=pi/p.lx; kvx=kfx*[0:n-1]'; p.mat.F=dctmtx(n); % wave-nr and dct-matrix
p.mat.L=spdiags((1-kvx.^2).^2,0,n,n); % multipliers as a diag matrix
```

```
function r=sG(p,u)  % sh1D, rhs F-version
n=p.nu; par=u(n+1:end); uf=u(1:n); lam=par(1); c2=par(2); c3=par(3); % split
F=p.mat.F; u=F'*uf; ff=lam*u+c2*u.^2+c3*u.^3; f=F*ff; % "nonlinearity"
r=p.mat.L*uf-f;      % residual
```

```
function Gu=sGjac(p,u) % sh1D, (full) Jacobian via F
n=p.nu; par=u(n+1:end); uf=u(1:n); lam=par(1); c2=par(2); c3=par(3);
F=p.mat.F; u=F'*uf; fu=lam+2*c2*u+3*c3*u.^2;
Fu=F*(spdiags(fu,0,n,n)*F');   Gu=p.mat.L-Fu;
```

Listing 9: oosetfemops, sG and sGjac from modtut/sh1Dfou

Additionally setting up sGjac as usual, we can run (4) on $\Omega = (0, 12\pi)$ with high accuracy (again judged by the 6 correct digits of the first BPs) with just $n = 100$ Fourier modes in cmds1. Figure (9) shows some results, where we focus on the first Turing branch, and a secondary bifurcation to a snaking branch of a front between the primary pattern and $u \equiv 0$. Here we clearly achieve higher accuracy with fewer modes (and, despite the fullness of Jacobians comparable or higher speed) than with the built-in FEM for the same problem, cf. [Uec20, §3].

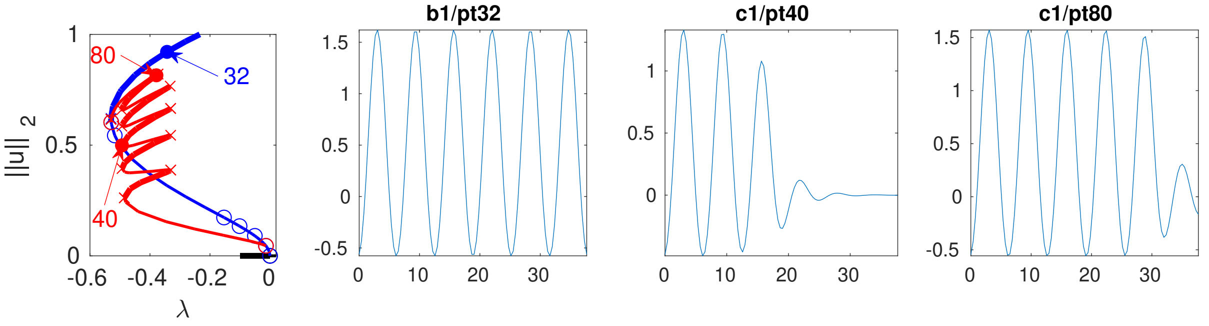

Figure 9: BD and sample solutions for (4) on $(0, 12\pi)$ with NBCs $\partial_x u|_{\partial\Omega} = \partial_x^3 u|_{\partial\Omega} = 0$; $n = 100$.

sh1Dmfree. In this demo we explain the matrix free setup based on (16), which is implemented in afun, see Listing 11, and is called by gmres. This naturally goes together with a few further modifications of the problem files for (17), and of the pde2path library routine belpi (bordered elimination (with) post iteration, see [UW17]). Since the signature of afun in gmres is (more or less) fixed, we find it most convenient to pass the needed multipliers μ and the local derivative $\partial_u f$ (computed in sGjac) via the (one and only, usually unused) pde2path global variable p2pglob. In Listing 11 we also collect the (somewhat) problem dependent interface lssgmres to gmres. This is also called in myeigsfu, which, by setting p.sw.eigssol=3, is called in the inverse vector iteration in eigs for computing p.nc.neig eigenvalues near p.nc.eigref, and the associated eigenvectors. We use the bordered elimination solver lssbel with lssgmres as inner solver for both, the $n \times n$ systems in natural parametrization, and the $(n+1) \times (n+1)$ systems in arclength parametrization. Here we have default border width p.bel.bw=0, in which case lssbel really just calls lssgmres, but in arclength we temporally set p.bel.bw=p.bel.bw+1, and lssgmres deals with the bulk part, while the $1 \times n$ borders are treated extra. This way we can use the $n \times n$ preconditioner on both, the standard and arclength extended systems. However, our (very simple) lssbel algorithm sometimes needs (1 or 2) post-iterations, and thus we also need to adapt the computation of the $n \times n$ systems' residuals in the library function belpi in the local directory.

```
function p=oosetfemops(p) % SH by dct, matrix free, multipl. stored in p2pglob
n=p.np; p.mat.M=speye(n); kfx=pi/p.lx; kvx=kfx*[0:n-1]'; % M and wave numbers
global p2pglob; p2pglob.mu=(1-kvx.^2).^2; % multiplier, used in sG and afun
p.mat.prec=spdiags(sqrt((1-kvx.^2).^2+1),0,n,n); % prec, used in lssgmres
```

```
function r=sG(p,u)  % matrix free, multipl.in p2pglob, needed in lssgmres anyway
n=p.nu; par=u(n+1:end); uf=u(1:n);lam=par(1);c2=par(2);c3=par(3);
u=idct(uf); ff=lam*u+c2*u.^2+c3*u.^3; f=dct(ff); % "nonlinearity" via dct (no F)
global p2pglob; r=p2pglob.mu.*uf-f;  % residual
```

```
function Gu=sGjac(p,u) % matrix free Jacobian,
global p2pglob; % p2pglob.fu further used in lssgmres
n=p.nu; par=u(n+1:end); uf=u(1:n); lam=par(1); c2=par(2); c3=par(3);
u=idct(uf); fu=lam+2*c2*u+3*c3*u.^2; p2pglob.fu=fu;
```

```
Gu=speye(n); % dummy, returned here cause size used as dimension at places
```

Listing 10: `oosetfemops`, `sG` and `sGjac` from `modtut/sh1Dmfree`. The multipliers μ and the local derivative $\partial_u f$ are stored and passed (for convenience) to `afun` in the global variable struct `p2pglob`.

```
function y=afun(uf) % lin.of SH used in lssgmres; p2pglob.fu filled in sGjac
global p2pglob; mu=p2pglob.mu; fu=p2pglob.fu;  y=mu.*uf-dct(fu.*idct(uf));
```

```
function [x,p]=lssgmres(A,b,p) % gmres interface for SH1D, see afun
n=size(b,1); try; ittol=p.ittol; limax=p.limax; catch; ittol=1e-8; limax=n; end
maxit=min(limax,n); L=p.mat.prec; % preconditioner (diagonal, shifted multipl)
tic;[x,flag,relres,iter]=gmres(@afun,b,[],ittol,maxit,L,L');iter=iter(2);t1=toc;
if p.sw.verb>2; fprintf('gmres-flag=%i, relres=%g, i=%i, time=%g\n',...
                        flag,relres,iter,t1); end
% following line is a typical 'fallback' (but not here, cause A is not set)
if flag>0; fprintf('gmres failed, using lss\n'); tic; x=A\b; toc, end
```

```
function y=myeigsfu(p,A,B,sig,b) % solver to be called in eigs; calls lssgmres
% (with global prec); here for SH: typical matrix free spectral differentiation
global p2pglob; fu=p2pglob.fu; p2pglob.fu=fu+sig; % put shift into fu
y=lssgmres(A,b,p); p2pglob.fu=fu;  % solve and restore fu
```

```
function [x,y,p,r]=belpi(A,b,c,d,f,g,p)
% BELPI: bordered elimination with post iterations;  here mod for matrix free
% (A is dummy) lss (called in bel), i.e., error tests must be adapted
global p2pglob; mu=p2pglob.mu; fu=p2pglob.fu;
[x,y,p]=bel(A,b,c,d,f,g,p);          % block-elim
fs=f-(mu.*x-dct(fu.*idct(x))+b*y); % modded, org: fs=f-(A*x+b*y);
gs=g-(c*x+d*y); r=norm([fs;gs],'inf'); iter=0;rs=r;
while r>p.bel.tol && iter<p.bel.imax
  [x1,y1,p]=bel(A,b,c,d,fs,gs,p); x=x+x1;y=y+y1; iter=iter+1; % corrector
  fs=f-(mu.*x-dct(fu.*idct(x))+b*y); % modded, org: fs=f-(A*x+b*y);
  gs=g-(c*x+d*y); r=norm([fs;gs],'inf'); rs=[rs,r];
end
if iter>0 && p.sw.verb>1
  str1=[num2str(iter),' post iterations done in belpi'];
  str2=['residuals were ',num2str(rs)]; disp(str1);disp(str2);
end
```

Listing 11: `afun`, `myeigsfu`, `lssgmres` and mod of the library function `belpi` from `modtut/sh1Dmfree`

Finally, Listing 12 shows a basic script for using the above setup. First (line 2) we need to declare the global variable `p2pglob`; then (line 6) we tell `pde2path` to use the linear system solver `lssbel` with border-width 0, tol=1e-4, at most 5 corrections in `belpi`, and `lssgmres` as inner solver, and to also run `eigs` via `myeigsfu`. In `lssgmres` we query for an (optional) tolerance `p.ittol`, which we set in l7, and finally we also set the number `p.nc.neig` of eigenvalues to compute to a rather small value. The reason is that the iterative linear system solvers are typically not well suited for the inverse vector iteration (IVI), and this is probably the main drawback of the setup: the IVI, in particular if there are several small eigenvalues very close together, needs rather accurate solutions, which is why we use the rather high accuracy `p.ittol=1e-8`, and hence `lssgmres` does need many (typically around 30-40) iterations for each linear system inside the IVI. See also the discussion in [Tuc20, §11.3] on other possibilities for preconditioning. Nevertheless, `cmds1` runs in about only 1s. In a second script `cmds2` with the same setup we go to somewhat larger scale $\Omega = (0, 30\pi)$ with $n = 600$ and compute a snake similar to Fig. 9, which is again robust and fast.

```
%% SH1D, via dct, matrix free (dummy Gu) via lssgmres and afun for Jac
close all; keep pphome; global p2pglob; % stores multiplier mu, and f_u
%% init, small dom, for testing, and cont trivial branch for a few steps
```

```
p=[]; par=[-0.1 2 -1];  % lam,quad,cubic
nx=100; lx=4*pi; dir='tr1'; p=shinit(p,lx,nx,par); p=setfn(p,dir);
p=setbel(p,0,1e-4,5,@lssgmres); p.sw.eigssol=3; % use gmres for Newton&Evals
p.ittol=1e-8; p.nc.neig=4; % tolerance in gmres, compute rather few Evals
p=cont(p,4); % go
%% switch to first bifurcating branch
p=swibra(dir,'bpt1','b1',0.01); p.nc.neig=1; p.nc.eigref=-0.2; p=cont(p,20);
```

Listing 12: Script `cmds1` from `modtut/sh1Dmfree`, small scale test case.

5.2.2 2D

sh2Dfou. Listing 12 shows the generation of the spectral differentiation matrix L in 2D, where similar to Listing 5 the 2D `dct` matrix `p.mat.F` is obtained from a tensor product of two 1D `dct` matrices `Fx` and `Fy`, and similar for the multiplier matrix `p.mat.L`.[11]

The remaining functions `sG`, `sGjac` and `stanbra` then stay exactly as in 1D. For testing we consider two cases. First, in `cmds1` we consider (4) on $\Omega = (0, 2\pi)^2$, such that the primary bifurcation is double, with kernel spanned by $\cos(x)$ and $\cos(y)$. By the D_4 symmetry of the square, we then have 3 bifurcating branches, namely horizontal and vertical stripes, and spots, which are found in `pde2path` by solving the pertinent cubic bifurcation equations via `cswibra`, see [Uec20, §3.3]. This runs very robustly already on a very coarse grid of `nx=ny=20`.

```
function p=oosetfemops(p) % SH 2D, full Jac via dct matrix F
nx=p.nx; ny=p.ny; n=nx*ny; p.mat.M=speye(n);
kfx=pi/p.lx; kvx=kfx*[0:nx-1]';  kfy=pi/p.ly; kvy=kfy*[0:ny-1]';
Fx=dctmtx(nx); Fy=dctmtx(ny); F=kron(Fx,Fy); % 1D dcts, and 2D
dd1=1-2*kvx.^2+kvx.^4; dd2=-2*kvy.^2+kvy.^4; % multipliers
D1=spdiags(dd1,0,nx,nx); D2=spdiags(dd2,0,ny,ny);
kx2=spdiags(kvx.^2,0,nx,nx); ky2=spdiags(kvy.^2,0,ny,ny);
L2x=kron(kx2,eye(ny)); L2y=kron(eye(nx),ky2);
L4x=kron(D1,eye(ny)); L4y=kron(eye(nx),D2);
L=L4x+L4y+2*L2x*L2y; p.mat.L=L; % multiplier matrix
p.mat.F=F; p.mat.prec=sqrt(L+speye(n)); % store F and prec
```

```
function r=sG(p,u)  % SH 2D, rhs via F
n=p.nu; par=u(n+1:end); uf=u(1:n); lam=par(1); c2=par(2); c3=par(3);
F=p.mat.F; u=F'*uf; f=lam*u+c2*u.^2+c3*u.^3; ff=F*f; r=p.mat.L*u(1:n)-ff;
```

Listing 13: `oosetfemops` and `sG` from `modtut/sh2Dfou`. Remainder like in `sh1Dfou`.

In `cmds2` and Fig. 10 we turn to a somewhat larger scale computation, namely (17) with $\nu = 2$ on $\Omega = (0, l_x) \times (0, l_y)$, $l_x = 12\pi$, $l_y = l_x x/\sqrt{3}$, i.e., a non–small domain with a hexagonal dual lattice. We run this on a coarse mesh with $n_x = 100$, $n_y = 56$, hence `p.nu=5600` DoF. This yields already rather slow computations, as the expensive term $F(\partial_u f(u))F^{-1}$ in `sGjac` takes several seconds. This motivates (a) to use a Newton chord method (flagged by `p.sw.newt=1`) instead of the default Newton method (flagged by `p.sw.newt=0`), and (b) the matrix free implementation described below, which yields the same results an order of magnitude faster, but first we comment on Fig. 10. The primary BP at $\lambda = 0$ is again double, with, due to the Neumann BCs, the kernel spanned by vertical stripes and hexagons. Using `qswibra` we switch to the hexagon branch (blue), and from this via `swibra` to a branch (red) of fronts between hexagons and $u \equiv 0$. This runs very robustly (using standard `cont`) on the coarse mesh, which is remarkable as similar computations based on the FEM setup require finer meshes (on the order of `p.nu=30000`, say), *and* heavy use of `pmcont` [UWR14] to avoid branch jumping.

[11]These simple tensor product structures are also the main reason why we use $\hat{u} = Fu$ and $u = F^T\hat{u}$, for the discrete 2D dct and inverse, instead of the built in `dct2` and `idct2`.

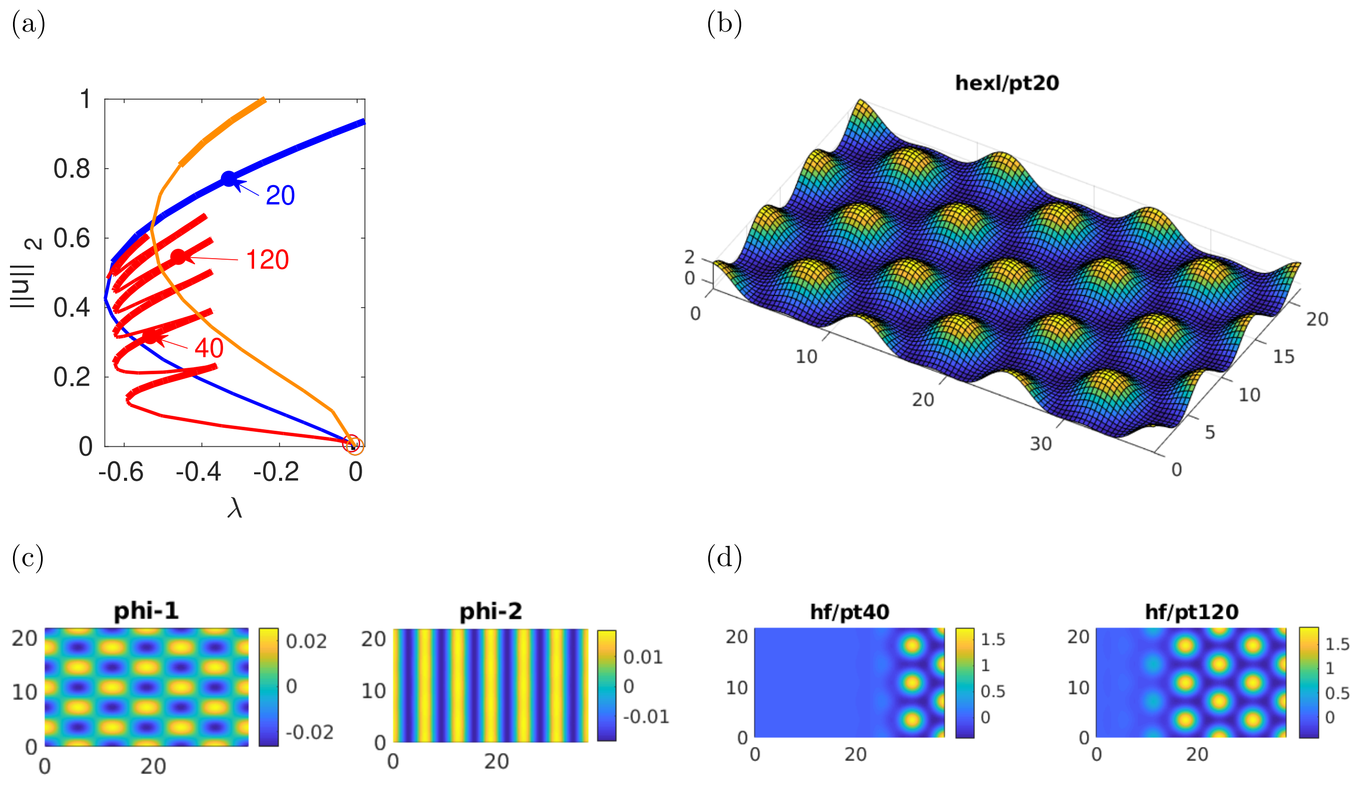

Figure 10: BD and sample solutions for (4) on $(0, l_x) \times (0, l_y)$, $l_x = 12\pi$, $l_y = l_x x/\sqrt{3}$, $n_x = 80$, $n_y = 46$ with NBCs $\partial_n u|_{\partial\Omega} = \partial_n \Delta u|_{\partial\Omega} = 0$. (a) hexagon (blue) and hex-front (red) branches, stripes in orange. (b) Hex sample solution, illustrating the quite coarse mesh. (c) Kernel vectors at the primary bifurcation. (d) Two samples from the snaking red branch.

This shows that the FFT based methods much better preserve symmetry. Also, the numerical kernel vectors ϕ_1 and ϕ_2 from Fig. 10(c) are exactly the "natural" kernel vectors $\phi_1 = \cos(x/2)\cos(\sqrt{3}y/2)$ and $\phi_2 = \cos(x)$, while in the FEM setup they are often somewhat distorted, as just some (orthogonal) base of the kernel is computed. As a consequence, in the situation of Fig. 10 we can in principle skip the computation of bifurcation directions τ by numerical solution of the algebraic bifurcation equations, and simply compose the bifurcation directions by hand, namely $\tau = \phi_1 + \phi_2$ for hexagons and $\tau = \phi_2$ for stripes, and this what we actually do for the stripes, thus skipping a call to `cswibra`.

sh2Dmfree. In this demo we consider a matrix free implementation of `sh2Dfou`, basically following `sh1Dmfree`, and obtaining for instance the results from Fig. 10 an order of magnitude faster, with one extra issue to consider. In `oosetfemops` in Listing 14 we start with essentially a mix of `oosetfemops` from `sh2Dfou` and from `sh1Dmfree`. Compared to the latter, here we also store F in `p2pglob.F`, for use in `sG, sGjac`, and `afun`, cf. Footnote 11. A special "trick" is the use of `p.needGu` in `sGjac`. If `p.needGu=1`, then the expensive *full Jacobian* is formed, which is otherwise omitted. The reason is that (presently) $\partial_u G$ is needed for branch switching at BPs of higher multiplicity via `qswibra` (quadratic bifurcation equations) or `cswibra`, which need to compute higher order directional derivatives.

```
function p=oosetfemops(p) % SH 2D, dct, matrix free, F passed via p2pglob
nx=p.nx; ny=p.ny; n=nx*ny; p.mat.M=speye(n);
% .. all as in sh2Dfou/oosetfemops ..., last line new:
global p2pglob; p2pglob.mu=diag(L); p2pglob.F=F;
```

```
function r=sG(p,u)  % SH 2D via dct with preassemble dct-matrix F and mult. mu
global p2pglob; F=p2pglob.F;
```

```
n=p.nu; par=u(n+1:end); uf=u(1:n); lam=par(1); c2=par(2); c3=par(3); % split
u=F'*uf; ff=lam*u+c2*u.^2+c3*u.^3;  f=F*ff; % "nonlinearity"
r=p2pglob.mu.*uf-f;     % residual
```

```
function Gu=sGjac(p,u) % matrix free Jac, except when called with p.needGu=1
global p2pglob; F=p2pglob.F;
n=p.nu; par=u(n+1:end); uf=u(1:n); lam=par(1); c2=par(2); c3=par(3);
u=F'*uf; fu=lam+2*c2*u+3*c3*u.^2; p2pglob.fu=fu;
try needGu=p.needGu; catch needGu=0; end
if needGu; t1=tic; Fu=F*(spdiags(fu,0,n,n)*F'); t1=toc(t1); fprintf('time for Gu
   : %g\n',t1);
   Gu=diag(p2pglob.mu)-Fu; % provide Gu for q(c)swibra
else Gu=speye(n);  end     % provide dummy
```

```
function y=afun(uf) % A*uf function for lssgmres for SH via fu (from sGjac)
global p2pglob; mu=p2pglob.mu; fu=p2pglob.fu; F=p2pglob.F;
y=mu.*uf-F*(fu.*(F'*uf));
```

Listing 14: `oosetfemops`, `afun`, `sG`, and `sGjac` from `modtut/sh2Dmfree`. For `p.needGu=1` we give up the matrix free approach in `sGjac` as `Gu` is needed for branch switching at BPs of higher multiplicity, cf. `cmds1` in Listing 15.

Listing 15 shows how to run this (mostly, i.e., except for using `Gu` in `qswibra`) matrix free approach on the same problem as in `sh2Dfou/cmds2`, to obtain the same results as in Fig. 10 much faster. In line 13, however, we switch on the computation of `Gu` in `sGjac` for `qswibra`. In fact, this could be avoided here, given the very clean numerical eigenvectors ϕ_1, ϕ_2 from Fig. 10(c), from which with some experience we see that the hexagon branch roughly corresponds to $\phi_1 + \phi_2$, such that (although it is transcritical) we can also switch to the hexagon branch via `p =gentau(p0,[1 1])` as we do for the stripes branch in line 22ff. However, in general it may not be clear what are the pertinent directions, and also $\lambda'(s)$ may need to be set to a good value in the transcritical case. These computations are all part of `qswibra`.

```
%% SH 2D mesh free, larger scale, hex-front-branch
close all; keep pphome; global p2pglob; % stores multipliers, f_u, and F
%% init
p=[]; par=[-0.01 2 -1]; lx=12*pi; ly=lx/sqrt(3); nx=100; ny=round(nx*ly/lx);
p=shinit(p,lx,ly,nx,ny,par); p=setfn(p,'tr2');
p.sol.ds=0.01; p.sw.verb=2; p.nc.neig=4; p.ps=2; % contour in userplot
p=setbel(p,0,1e-4,20,@lssgmres); p.sw.eigssol=3; % use gmres for Newton&Evals
p.limax=200; p.ittol=1e-8;  % max-it and tolerance in gmres
p.sw.bifcheck=2; p.nc.bisecmax=6; % bifdetec and loc. via Evals
tic; p=cont(p,2); toc % just 2 steps to find prim. bif (at lam=0)
%% 1BP double, use qswibra, Gu needed here and ONLY here, hence switch it on
aux=[]; aux.soltol=1e-10; aux.m=2;  aux.isotol=1e-12; p=loadp('tr2','bpt1');
p.needGu=1; p0=qswibra(p,aux); p0.needGu=0; % switch full Gu on/off for qswibra
p0.sw.spcalc=1; p0.nc.neig=1; p0.nc.eigref=-0.1; % just one Eval for stab.
p0.sw.verb=2; p0.sw.bifcheck=0; p0.nc.tol=1e-6;  % switch off bifcheck
p0.sol.ds=0.1; p0.nc.dsmax=0.1; p0.file.smod=10;
%% select tangent and cont, save first 2 steps for swibra to snake at 1st point
p=seltau(p0,2,'hexl',2);  p.sw.spcalc=0; p.file.smod=1; p.sol.ds=-0.01;
ta=tic; p=cont(p,2); toc(ta);
p.sw.spcalc=1; p.file.smod=10; p.nc.dsmax=0.1; p=cont(p,20);  % further steps
%% stripes via gentau
p=gentau(p0,[0 -1]);  p=setfn(p,'b1l'); p=cont(p,20);
%% hex-front via swibra from approximate BP,
p=swibra('hexl','pt1','hf',0.005); p.nc.dsmax=0.2; p.nc.dsmin=0.001;
p.nc.tol=1e-4; tic;p=cont(p,2);toc % 2 steps with large tol to get on the branch
p.nc.tol=1e-6; tic; p=cont(p,50); toc % back to small tolerance
```

Listing 15: First part of `cmds2` (last 6 lines for plotting) from `modtut/sh2Dmfree`.

Otherwise, we remark that for branch switching to the red snake we do not localize the pertinent BP, but simply call `swibra` on the first point on the hexagon branch (line 24), which yields the needed (approximate) kernel. Moreover, and relatedly, on the hex, stripe and hex2zero-front branches we only compute 1 eigenvalue near $\mu_0 = -0.1$ (see line 14) to decide on the stability of solutions. The reason, again, is that eigenvalue computations with `lssgmres` inside the inverse vector iteration are expensive due to the high accuracy required.

Thus, altogether we do need to apply a few "tricks" to make the continuation and bifurcation to branches of interest on this non–small domain robust and fast. Nevertheless, I believe this illustrates the strength of (matrix free) FFT methods for specific questions, and how to run them in `pde2path`.

Remark 5.1 a)The subdirectory `modtut/altfou` contains alternate versions for (2) in 1D, and 17 in 1D and 2D, based on FFT spectral differentiation but working in x–space, i.e., in the form $G(u) = \mathcal{F}^{-1}\mu(k)\mathcal{F}u - f(u)$. Consequently, in the Jacobian, the differential term $L(\partial_x) = \mathcal{F}^{-1}\mu(k)\mathcal{F}$ is full (but symmetric), but the term $\partial_u f(u)$ is diagonal, cf. the discussion after (14). Thus, for iterative methods here we can use an `ilu` or `ichol` preconditioner. This basically works, but slightly slower than the versions based on (15) discussed so far, and thus here we refrain from details and refer to the comments in the demos in `modtut/altfou`.

b) Naturally, also systems such as (3) can be treated via FFT by applying the spectral differentiation component–wise, and using, e.g., L_C=diag$(d_1\sqrt{|k|^2+1}, d_2\sqrt{|k|^2+1})$ as preconditioner. $\rfloor$

5.3 Problems on disks

For (2) on a disk $\Omega = \{(x,y) = r(\cos(\vartheta), \sin(\vartheta)) : r \in [0,R), \vartheta \in (0,2\pi]\}$ with either (homogeneous) NBCs

$$\partial_n u = 0 \text{ on } \partial\Omega = \{r = R\}, \tag{18}$$

or with DBCs, which we take in the form

$$u = \gamma\cos(\vartheta) \text{ on } \partial\Omega = \{r = R\}, \tag{19}$$

we can combine a Chebychev discretization in r with a Fourier discretization in ϑ. We follow [Tre02, Ch. 11] for the basic idea, and explain the setup for (18) in the demo `acdiskNBC`, using $R = 5$, and hence consider the same problem as (with the FEM) in [Uec21a, §6.8.3] and in the demo `demos/acsuite/acdisk`. For (19) we then have a very similar setup in the demo `acdiskDBC`. To deal with the rotational invariance of the problem (in case (19) only for $\gamma = 0$), for the continuation of branches with angular dependence we must add a rotational phase–condition, e.g.,

$$q(u) = \langle u, \partial_\vartheta u_{\text{old}}\rangle = 0, \tag{20}$$

where $\langle u, v\rangle = \int_\Omega uv \, d\Omega$ is the L^2–inner product and u_{old} is the solution from the last continuation step. Hence, for (18) we altogether consider

$$G(u) := -\Delta u - \lambda u + u^3 + s\partial_\vartheta u \overset{!}{=} 0, \quad q(u) := \langle u, \partial_\vartheta u_{\text{old}}\rangle \overset{!}{=} 0, \quad \partial_n u|_{\partial\Omega} = 0. \tag{21}$$

Let $R = 1$ and note that the general case follows by rescaling. To obtain a suitable r discretization (not too fine near the coordinate singularity $r = 0$), the idea is to generate the $2n_r$ Chebychev points $r_1{=}1, r_2, \dots, r_{n_r}|r_{n_r+1}, \dots, r_{2n_r}{=}-1$ in the interval $[-1,1]$, and use the tensor product of $(r_1, \dots, r_{n_r})$ with the equidistant angular mesh $0 < \vartheta_1 = 2\pi/n_a < \vartheta_2 < \ldots < \vartheta_{n_a} = 2\pi$ for the discretization. The polar coordinates Laplacian

$$\Delta u = \partial_r^2 u + r^{-1}\partial_r u + r^{-2}\partial_\vartheta^2 u, \tag{22}$$

then also is a tensor product of the differentiation matrix for ∂_ϑ^2 (which we implement as a Töplitz matrix [Tre02, Ch.3]), and the differentiation matrices for ∂_r, ∂_r^2 (Chebychev matrices). For the latter, we must take into account that although we discard all $r_j < 0$ we must add their contributions to ∂_r, ∂_r^2, see [Tre02, Ch.11].

Additionally, we need to set up a `userplot`, and it is useful to set up some function for evaluating integrals, for instance

$$\|u\|_2^2 = \frac{1}{|\Omega|}\int_\Omega u^2(x)\,\mathrm{d}x = \frac{1}{\pi R^2}\int_0^{2\pi}\int_0^R ru^2(r\cos\vartheta, r\sin\vartheta)\,\mathrm{d}r\ \ \mathrm{d}\vartheta \tag{23}$$

for plotting BDs. For the latter, we use the function `I=dchebint(p,u)`, where we evaluate the inner integral via `trapz`. For plotting, we mostly refer to `acdiskNBC/userplot`, and mainly remark that since $r = 0$ is *never* in the mesh, we give the option of "filled–in" plots by averaging u from $u|_{r=r_{\min}}$, controlled by the switch `p.ipz`, additional to the switch `p.ups` for plot styles. In Table 2 we summarize the functions used in the demo `acdiskNBC`, and in Fig. 11(a,b) we plot some results. The spectral accuracy allows quite coarse meshes (here `p.np=576`) for this simple problem, but for larger scale problems the only mildly sparse Jacobians may again become problematic.

Table 2: Overview of `acdiskNBC`, and of functions used from `pde2path/libs/misc/`.

function	remarks
`cmds1`	script for (21)
`p=acinit(p,R,nr,na,par)`	init; here mainly store the radius p.lx=R, and the # of discretization points nr (in r) and na (in ϑ) in p.nr, p.na, and call `oosetfemops`.
`p=oosetfemops(p)`	generate discretization and system matrices, based on `CFlapNBC`
`sG, sGjac, qf, qjac`	rhs, Jacobian, and constraint (20) and its derivative; with the disk Laplacian `p.mat.K` and the ∂_ϑ differentiation matrix `p.mat.Dphi`, these work as usual.
`userplot(p,wnr)`	plot solution; important switches are `p.ups` (user-plot-style), and `p.ipz` (interpolate to $r = 0$ if `ipz` > 0).
`[L,...,r]=CFLapNBC(p)`	Generate mesh and system matrices for Chebychev–Fourier discretization; in case of DBCs (demo `acdiskDBC`) we instead use `CFLapDBC`
`I=dchebint(p,u)`	integrate u (over Chebychev–Fourier mesh) over disk.

In Fig. 11(c,d) we consider (19) with $\gamma = 0$, yielding the expected results, and in Fig. 12 we then turn to the case of $\gamma \neq 0$ in (19). In (a), with samples in (b,c) we fix $\gamma = 0.1$ and see that the branches from Fig. 11(b) roughly remain, but now with imperfect bifurcations, cf. [Uec21a, §6.2.1]. In (d) we continue in γ, yielding an isola.

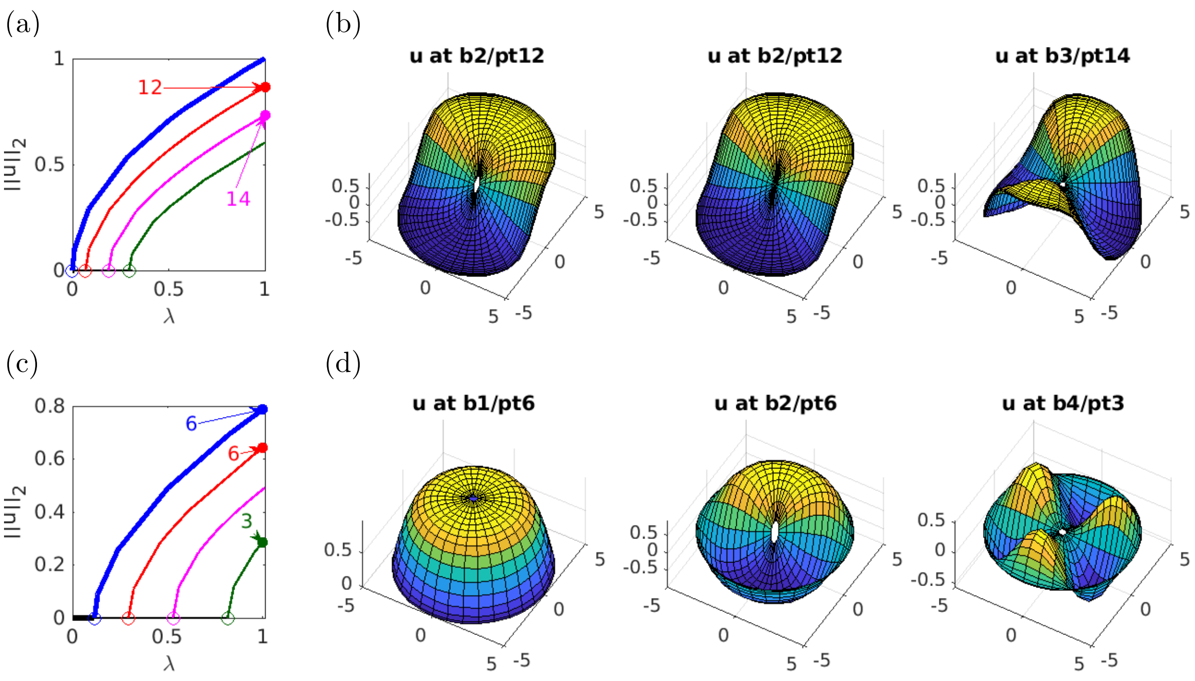

Figure 11: (a,b) BD and sample solutions for NBCs; the first two plots in (b) are the same solution, with the origin "filled in" in the second `p.ipz=1`. (c,d) BD and sample solutions for DBCs (19) with $\gamma = 0$.

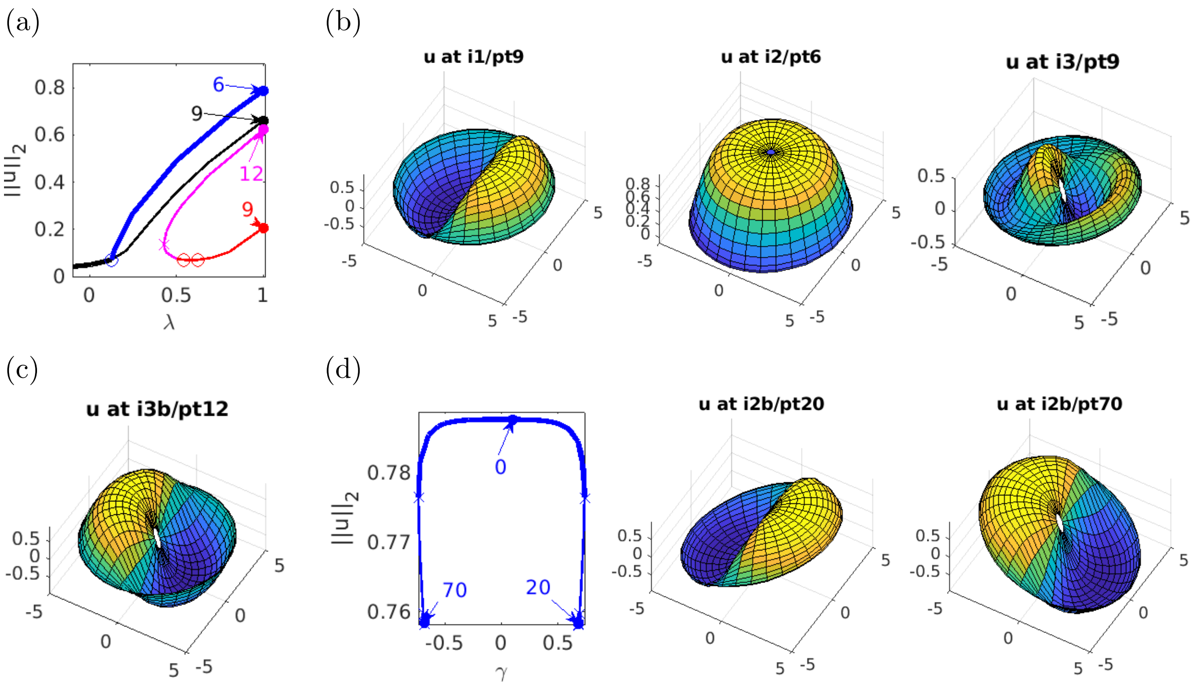

Figure 12: (a–c) BD and sample solutions for continuation in λ, $\gamma = 0.1$ fixed (`acdiskDBC/cmds1`). (d) BD and sample solutions for continuation of `i2/pt6` from (a) in γ (`acdiskDBC/cmds2`).

6 Summary

In order to explain how to use pde2path for problems $\frac{\mathrm{d}}{\mathrm{d}t}u = -G(u)$ with "arbitrary" rhs $-G(u)$, not using the built–in FEM discretizations, we explored (2) and (3) on graphs, and (2)–(4) as PDEs using Chebychev and Fourier spectral methods. Running pde2path on the graphs is easy and yields interesting results, which however we do not further discuss here as our primary goal was the explanation of data structures. For the alternative discretizations of the PDEs we obtain the analogous results to those based on the FEM discretization in [RU19, Uec20], with generally much coarser discretizations (smaller n), and which, moreover, seem generally more robust wrt to keeping symmetry while continuing branches of patterns (less "branch–jumping"). However, as the spectral methods yield full (or only mildly sparse) matrices, to also turn this into an speed advantage we need some additional tricks, and we explained some iterative linear system solver methods for the FFT based methods in §5. Moreover, spectral methods require specific domains (here boxes, or the disk as a special case) and BCs. In this sense, the FEM is more general, and in pde2path additionally comes with ready to use mesh–adaptation methods. Nevertheless, depending on the problem type it may be useful or necessary to implement a rhs independent of the built–in FEM, and with the above examples we hope to give some useful templates for this. For instance, given the (scalar) Laplacians `CFlapNBC` and `CFLapBDC` from §5.3 we can also set up vector valued problems such as (3) and (4) (as a 2nd order system) on disks. The latter is again amazingly rich in patterns, and, in a cubic–quintic version, has been studied via the FEM in [VKU21]. Using the Chebychev–Fourier discretization we recover the same patterns and branches with considerably less DoFs, but again with little speed advantages.

References

[AB02] R. Albert and A. Barabasi. Statistical mechanics of complex networks. *Rev. Mod. Physics*, 74:47–97, 2002.

[ALB$^+$10] D. Avitabile, D.J.B. Lloyd, J. Burke, E. Knobloch, and B. Sandstede. To snake or not to snake in the planar Swift-Hohenberg equation. *SIAM J. Appl. Dyn. Syst.*, 9(3):704–733, 2010.

[Bol11] M. Bollhöfer. ILUPACK V2.4, `www.icm.tu-bs.de/~bolle/ilupack/`, 2011.

[Boy01] J. P. Boyd. *Chebyshev and Fourier spectral methods.* Dover Publications, Inc., Mineola, NY, second edition, 2001.

[HNM14] Shigefumi Hata, Hiroya Nakao, and A.S. Mikhailov. Dispersal-induced destabilization of metapopulations and oscillatory Turing patterns in ecological networks. *Scientific Reports*, 4:3585, 2014.

[Kno08] E. Knobloch. Spatially localized structures in dissipative systems: open problems. *Nonlinearity*, 21:T45–T60, 2008.

[MW16] N. McCullen and T. Wagenknecht. Pattern formation on networks: from localised activity to Turing patterns. *Scientific Reports*, 6:27397, 2016.

[NM10] Hiroya Nakao and A.S. Mikhailov. Turing patterns in network-organized activator–inhibitor systems. *Nature Physics*, 6:544–550, 2010.

[Prü21] U. Prüfert. OOPDE, `https://tu-freiberg.de/fakult1/nmo/pruefert`, 2021.

[PT10] R. B. Platte and L. N. Trefethen. Chebfun: a new kind of numerical computing. In *Progress in industrial mathematics at ECMI 2008*, volume 15 of *Math. Ind.*, pages 69–87. Springer, Heidelberg, 2010.

[RU19] J.D.M. Rademacher and H. Uecker. The OOPDE setting of pde2path – a tutorial via some Allen-Cahn models, 2019.

[SAKR16] P. Subramanian, A. Archer, E. Knobloch, and A. Rucklidge. Three-dimensional icosahedral phase field quasicrystal. *Phys. Rev. Let.*, 117:075501, 2016.

[SAKR18] P. Subramanian, A. Archer, E. Knobloch, and A. Rucklidge. Spatially localized quasicrystalline structures. *New Journal of Physics*, 20:122002, 2018.

[SN16] J. Sánchez and M. Net. Numerical continuation methods for large-scale dissipative dynamical systems. *Eur. Phys. J. Special Topics*, 225:2465–2486, 2016.

[TBD18] L. N. Trefethen, Á. Birkisson, and T. A. Driscoll. *Exploring ODEs.* SIAM, Philadelphia, PA, 2018.

[Tre02] L.N. Trefethen. *Spectral methods in Matlab.* SIAM, 2002.

[Tuc20] L. S. Tuckerman. Computational challenges of nonlinear systems. In *Emerging Frontiers in Nonlinear Science*, pages 249–277. Springer, 2020.

[Uec09] H. Uecker. A short ad hoc introduction to spectral methods for parabolic PDE and the Navier–Stokes equations. In *Summer School Modern Computational Science, Oldenburg 2009*, pages 169–209. Universitätsverlag Oldenburg, 2009.

[Uec19] H. Uecker. Hopf bifurcation and time periodic orbits with pde2path – algorithms and applications. *Comm. in Comp. Phys*, 25(3):812–852, 2019.

[Uec20] H. Uecker. Pattern formation with pde2path – a tutorial, 2020.

[Uec21a] H. Uecker. *Numerical continuation and bifurcation in Nonlinear PDEs.* SIAM, Philadelphia, PA, 2021.

[Uec21b] H. Uecker. `www.staff.uni-oldenburg.de/hannes.uecker/pde2path`, 2021.

[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.

[VKU21] N. Verschueren, E. Knobloch, and H. Uecker. Localized and extended patterns in the cubic-quintic Swift-Hohenberg equation on a disk. *Phys. Rev. E*, 2021.

[Wol12] M. Wolfrum. The Turing bifurcation in network systems: Collective patterns and single differentiated nodes. *Physica D*, 241:1351–1357, 2012.

[WR00] J. A. C. Weideman and S. C. Reddy. A MATLAB differentiation matrix suite. *ACM Trans. Math. Software*, 26(4):465–519, 2000.

[WS98] D. J. Watts and S.H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 339:440–442, 1998.