Linear system solvers in `pde2path` – a tutorial

Hannes Uecker[1,*], Daniel Wetzel[1,•]

[1] Institut für Mathematik, Universität Oldenburg, D26111 Oldenburg
[*] hannes.uecker@uni-oldenburg.de, [•] danieldwetzel@gmail.com

March 29, 2017

Abstract

We explain two implementations of linear system solvers in `pde2path`, and discuss their usage and performance via two tutorial examples.

1 Introduction

The `Matlab` bifurcation and continuation package `pde2path` [UWR14] discretizes PDEs to obtain equations of the form $\partial_t u = G(u,\lambda)$, where $\lambda \in \mathbb{R}$ is a parameter, and $u = u(t) \in \mathbb{R}^{n_u}$ with n_u large, in particular if the underlying PDE lives over 3D spatial domains. We assume that there is a branch b of stationary solutions and that we already know a solution (u_0, λ_0) on b. Let s be a parametrization of b and $(u_0,\lambda_0) = (u(s_0), \lambda(s_0))$. To go from one point on the branch to the next one, `pde2path` sets up a predictor by using the tangent and starts a Newton iteration. Implemented are a 'natural' continuation method for which `pde2path` fixes λ to $\lambda_0 + \delta$ and solves $G(u, \lambda_0 + \delta) = 0$, and a pseudo arclength continuation, for which `pde2path` solves $H(u,\lambda) = \begin{pmatrix} G(u,\lambda) \\ p(u,\lambda,s)\end{pmatrix} = \begin{pmatrix}\mathbf{0}\\0\end{pmatrix}$, with $p(u,\lambda,s) = \left\langle \tau_0, \begin{pmatrix} u-u_0 \\ \lambda-\lambda_0\end{pmatrix}\right\rangle_\xi - \mathrm{d}s$, $\quad \mathrm{d}s = s - s_0$, where $\tau_0 = (\dot u_0, \dot\lambda)$, $\dot{} = \frac{\mathrm{d}}{\mathrm{d}s}$, is the tangent at b in s_0 with $\|\tau\|_\xi = 1$. Here, $\|\tau\|_\xi = \sqrt{\langle \tau,\tau\rangle_\xi}$, $\left\langle \begin{pmatrix} u\\ \alpha\end{pmatrix}, \begin{pmatrix} v \\ \beta\end{pmatrix}\right\rangle = \xi\,\langle u, v,\rangle_2 + (1-\xi)\alpha\beta$, with $0 < \xi < 1$ a factor to give an appropriate weight to the single parameter λ compared to the large vector u. Geometrically, if $|\dot\lambda|$ is large (near 1), then the branch b is almost 'horizontal', while if $\dot\lambda$ is near 0, then the branch is 'steep', and in the default setting we use $\dot\lambda$ to switch between natural and arclength continuation, according to

$$|\dot\lambda| > \eta \text{ (natural), or } |\dot\lambda| \le \eta \text{ (arclength),} \tag{1}$$

where η=`p.sw.lamdtol`=0.5 by default in `pde2path`.

For solving at fixed λ, the Newton steps are given by

$$u^m = u^{m-1} - G_u^{-1}(u^{m-1})G(u^{m-1}), \tag{2}$$

and for arclength by

$$U^m = U^{m-1} - H_U^{-1}(U^{m-1})H(U^{m-1}) \text{ with } H_U(U) = \begin{pmatrix} G_u(U) & G_\lambda(U) \\ \xi \dot u_0 & (1-\xi)\dot\lambda_0 \end{pmatrix}, \quad U = (u,\lambda). \tag{3}$$

After a new point is found on the branch, `pde2path` calculates the new tangent τ via

$$H_U \tau = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{4}$$

Solving (2), (3) and (4) are significant costs in the continuation process (the most significant costs if spectral computations are switched off), and therefore it is desirable to use efficient solvers. An important feature of H_U in (3) and (4) is that the Jacobian G_u is a large but sparse matrix, which often has some favorable structure for linear system solvers. For instance, in the case of discretizations of 1D PDE systems it consists of tridiagonal blocks corresponding to the Laplacian, one for each component of the underlying PDE system. On the other hand, the full matrix H_U in (3) and (4) may not have a favorable structure due to the borders $G_\lambda \in \mathbb{R}^{n_u \times 1}$ and $(\xi \dot{u}_0, (1-\xi)\dot{\lambda}_0) \in \mathbb{R}^{1\times(n_u+1)}$. Thus, a well known idea is to use bordered elimination methods [Gov00] for solving (3) and (4).

In `pde2path`, if `p` is the current problem struct, then the linear system solver (LSS) used for solving $Ax = b$ are given by `p.fuha.lss` (for (2)) and `p.fuha.blss` (for (3) and (4)). Simple and in many cases sufficiently efficient choices are `p.fuha.lss=@lss` and `p.fuha.blss=@lss`, where `lss` is simply an interface to `Matlab`'s backslash operator \, which is a highly optimized LSS. However, for a given problem it may be advantageous to use different solvers, and it is the purpose of this tutorial to give a brief introduction to two such LSS which come with `pde2path`. These should also be seen as templates for the user to implement further methods or use third party packages.

2 An iterative LSS using `ilupack`

We found the AMG-solver of the package `ilupack` (GMRES with incomplete *LU* preconditioner) to be an efficient LSS for many of our problems. The package can be downloaded at [Bol11]. To use it, mex its `Matlab` interface and put the corresponding folder into `Matlab`'s path, see also the `pde2path` quickstart guide and reference card [dWDR+17]. For solving a linear system with `ilupack` one has to set up some control parameters. The easiest way to do so is via `AMGinit.m`, where also the meaning and default settings of these parameters are explained.

A `pde2path` interface to solve $Ax = b$ via the AMG-solver is provided as `[x,p]=lssAMG(A,b,p)`, where `p` is the pertinent `pde2path` struct. This also calls `AMGinit` to set parameters, but the `ilupack` parameters can be (re)set in `pde2path` via the field `p.ilup`, i.e., `p.ilup` may contain fields that overwrite the values returned from `AMGinit`. So far we mainly use the drop tolerance `droptol` in the incomplete *LU* factorization, and the maximum number of GMRES iterations `maxit`. Additionally, `p.ilup` contains a field `maxitmax` (default 1000), which sets an upper bound on `maxit`, and `droptolmin` (default 10^{-8}), which sets a lower bound on `droptol`, see below.[1] A convenience function is `p=setilup(p,droptol,maxit)`. The `lssAMG` algorithm then works as follows:

- Set the `ilupack` control parameters.
- Check if a preconditioner `prec` of correct size is stored in `p.mat.prec`. If not, then compute `prec` and store it in `p.mat.prec`, for re-use if possible.
- Run the AMG-solver and check convergence.
- If convergence, then return.
- If no convergence and `prec` was not yet updated, then update and try to solve again.
- If no convergence, and `prec` was already updated, then depending on the general `pde2path` interaction switch `p.sw.inter` proceed as follows
  - If `p.sw.inter`≤ 0 then, depending on on `nch:=p.ilup.noconvhandling`:
    `nch`=0: stop;
    `nch`=2: if `droptol`$\geq$`10*droptolmin`, then let `droptol=droptol/10`, update `prec`, and try to solve again, else stop
    `nch`=1: if `maxit`$\leq$ `p.ilup.maxitmax/2`, then let `maxit=2*maxit`, and try to solve again, else stop.
  - If `p.sw.inter`>0, then the user can stop, or interactively choose new settings for `p.ilup.droptop` and `p.ilup.maxit`, and afterwards update `prec` and try to solve again.

[1]For our problems we typically work with `droptol`= 10^{-3} or 10^{-4}, and then GMRES typically converges with less than 50 iterations. If it doesn't, then empirically it is better to update the preconditioner (to the current Jacobian G_u) more frequently than to change `droptol` or `maxit`. Moreover, we remark that on some machines we found that setting `maxit`=30 or smaller may give segmentation faults, which never occured for `maxit`=50 or larger.

3 A bordered elimination method

The Jacobian H_U consists of G_u and borders. Moreover, G_u itself may have borders, for instance in case that G does not only consist of the PDE but also involves constraints such as mass conservation. To generally deal with problems $Ax = F$ where A has borders, which for instance also play a big role in the Hopf case [Uec17b], we use a simple bordered elimination method `lssbel`, controlled by the field `p.bel` containing the parameters listed in Table 1.

Table 1: Parameters for bordered elimination methods

parameter	meaning
`p.bel.tol`	residual tolerance
`p.bel.imax`	max. number of post iterations
`p.bel.bw`	border width for G_u

To explain the algorithm, consider $Ax = F$ with $A \in \mathbb{R}^{n\times n}$ with border width $m \ge 0$, $F \in \mathbb{R}^n$, and split A and F into

$$A = \begin{pmatrix} B & b \\ c & d \end{pmatrix} \text{ and } F = \begin{pmatrix} f \\ g \end{pmatrix}$$

with $B \in \mathbb{R}^{(n-m)\times(n-m)}$, $b \in \mathbb{R}^{(n-m)\times m}$, $c \in \mathbb{R}^{m\times(n-m)}$, $d \in \mathbb{R}^{m\times m}$. Compute

$$v = B^{-1}b, \quad w = B^{-1}f, \quad \delta = d - cv, \quad y_0 = \delta^{-1}(g - cv), \quad x_0 = w - vy \tag{5}$$

$$f_s = f - (Bx_0 + by_0), \quad g_s = g - (cx_0 + dy_0), \quad \text{and the residual } r = \left\| \begin{pmatrix} f_s \\ g_s \end{pmatrix} \right\|_\infty . \tag{6}$$

If $r <$`p.bel.tol`, then $x = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$. If not, then iterate

$$\begin{aligned} &v = B^{-1}b, \quad w = B^{-1}f_s, \quad \delta = d - cv, \quad y_1 = \delta^{-1}(g_s - cv), \quad x_1 = w - vy, \qquad (7)\\ &x_0 \leftarrow x_0 + x_1, \quad y_0 \leftarrow y_0 + y_1, \end{aligned}$$

as long as the residual $r \ge$`p.bel.tol` and the number of iterations is smaller than `p.bel.imax`. If this does not converge, then, depending on `p.sw.inter`, we use `p.fuha.innerlss` (see below) on the full system, or ask for user interaction. `lssbel` is a simple method, which however works well for our examples; see [Gov00] for much more sophisticated methods.

For the $(n-m) \times (n-m)$ linear systems to be solved in (5) and (7), the user must set an 'inner' linear system solver via `p.fuha.innerlss`. The standard setting (via calling `stanparam`) is `p.fuha.innerlss=@lss`. A convenience function to set parameters is `p=setbel(p,bw,beltol, belimax,innerlss)`, which sets

(α) `p.bel.bw=bw`; `p.bel.tol=tol`; `p.bel.imax=imax`;

(β) `p.fuha.lss=@lssbel; p.fuha.blss=@blssbel; p.fuha.innerlss=@innerlss;`

In `pde2path`, we may want to use `lssAMG` for both, (2) and (3), respectively `lssAMG` as an inner LSS for `lssbel` for both. The extended system (3) has at least one border and exactly one border more than (2), and it is advantageous to use the same `prec` for `lssAMG` as inner solver for both. For this set `p.fuha.lss=@lssbel` and `p.fuha.blss=@blssbel` where in `blss` we temporarily increase the border width by 1. If G_u has for instance border width 3, then set `p.bel.bw`=3, and `lssbel` and `blssbel` work with border widths 3 and 4, respectively. A convenience call is `p=setbelilup(p,bw,tol,imax,dtol)`, which additionally to (α) and (β) above sets

(γ) `p.fuha.innerlss=@lssAMG;` `p.ilup.droptol=dtol`; `p.ilup.droptolS=dtol/10;`

4 Two Examples

Whether a significant advantage of `lssbel` or `lssAMG`, or the combination of both, over the default \ solver occurs depends on many factors. In general, `lssAMG` beats \ already for intermediate $n_u = n_u^*$, where our experience is something like $n_u^* = 10^4$, and certainly for $n_u > 10^5$. These are only empirical values, and may still vary strongly, for instance depending on how often (after how many continuation steps) a new `prec` is needed. Similarly, whether `lssbel` gives an advantage over \ strongly depends on how much more efficiently the unbordered systems $Bv = b$ and $Bw = f$ in (5) can be solved than the full system $Ax = F$.

This tutorial comes with two demo directories `schnak1D` and `ac3D`, which are just two but somewhat typical examples illustrating the usefulness of `lssbel` and `lssAMG`. Another class of examples for huge advantages of `lssbel` over solvers without bordered elimination are Hopf problems [Uec17b], where block tri-diagonal matrices occur, and the full matrix has additional borders of width 2.

4.1 1D Schnakenberg

Huge advantages of `lssbel` (rather independent of `p.fuha.innerlss`) over `lss` can be found for 1D problems. As an example, in `lss/schnak1D` we consider the modified Schnakenberg model

$$0 = D\Delta U + N(u,\lambda) + \sigma\left(u - \frac{1}{v}\right)^2 \begin{pmatrix}1\\-1\end{pmatrix}, \quad N(U,\lambda) = \begin{pmatrix}-u+u^2v\\ \lambda-u^2v\end{pmatrix}, \tag{8}$$

with $U = (u,v)(x) \in \mathbb{R}^2$, $D = \left(\begin{smallmatrix}1&0\\0&d\end{smallmatrix}\right)$, where we take λ as the bifurcation parameter and restrict to d=60, $\sigma = -0.6$, and $x \in \Omega \subset \mathbb{R}$. A primary bifurcation of a periodic solution then occurs for decreasing λ to $\lambda_c = \sqrt{60}\sqrt{3-\sqrt{8}} \approx 3.21$ with critical wave number $k_c = \sqrt{\sqrt{2}-1}$, see [UWR14, UW14, dW17]. In particular, the `pde2path` files `sG.m, sGjac, oosetfemops.m` and `schnakinit.m` in `schnak1D` defining the rhs, the Jacobian, the FEM matrices and the general model implementation follow closely those described in `demos/schnakfold`, and here we restrict to discussing the script `cmds.m` in Listing 1, and the results.

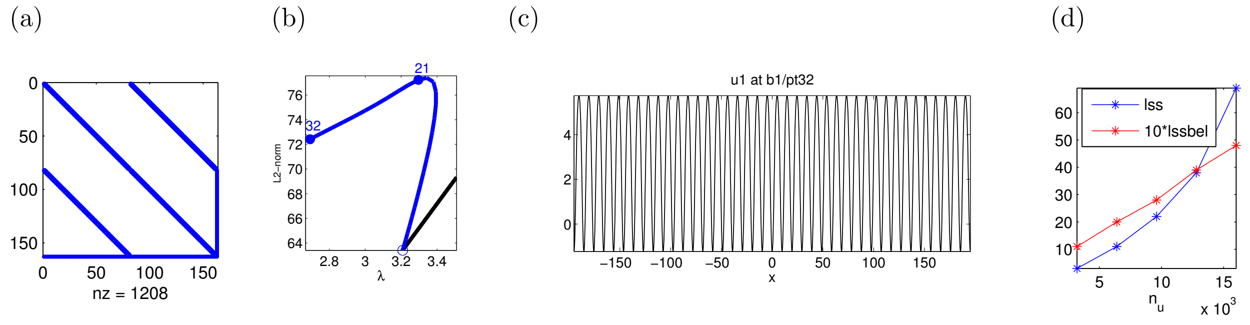

Figure 1: Illustration of borders (a), branch (b) and solution plots (c) for the test problem, and performance (d), time in seconds. In (a) we use the settings from Listing 1 but with `nper=1`. Since G_1 (the first component of (8)) does not depend on λ, we only have a vertical border (last column of the matrix) in the second half. In (b) and (c) we show results for `nper=40`.

```
%% 1 - initialising the problem
close all; keep pphome; p=[]; par=[sqrt(60)*sqrt(3-sqrt(8))+0.2, -0.6, 60];
nper=80; npp=60; p=schnakinit(p,nper,npp,par); % also use nper=80, 120, 160, 200
%% 2 - continue trivial branch to find BP
tic; p=findbif(p,1); toc
%% 3 - switch to periodic branch and continue. For comparison of \ and
% lssbel, switch off stuff not related to lss
p=swibra('p','bpt1','b1',0.02); p.sw.spcalc=0; p.sw.foldcheck=0; p.sw.bifcheck=0;
p.sw.verb=2; p0=p; t1=tic; p=cont(p,50); t1=toc(t1); % cont with default settings
p=p0; bw=0; beltol=1e-4; belmaxit=5; p=setbel(p,bw,beltol,belmaxit,@lss); % lssbel
t2=tic; p=cont(p,50); t2=toc(t2);
```

```
fprintf('t1=%g, t2=%g\n', t1,t2); plotsol(p,1,1,1);
%% 4 - plots
figure(3); clf; plotbra('p',3,0); plotbra('b1','pt32',3,0,'lab',[21,32],'cl','b');
xlabel('\lambda'); plotsol('b1','pt32');
x=[32 64 96 128 160]; y1=[3 11 22 38 69]; y2=[1.1 2 2.8 3.9 4.8];
plot(1000*x,y1,'*-',1000*x,10*y2,'r*-'); legend('lss','10*lssbel'); xlabel('n_u');
```

Listing 1: Script `schnak1D/cmds.m`. Cells 1,2 are preparatory, in Cell 3 we compare the linear system solvers, and Cell 4 gives plots for illustration. We consider (8) on domains of size nper$\times 2\pi/k_c$, first for nper= 40 with npp= 40 grid-points per period $2\pi/k_c$, hence nu= 3200 unknowns. In Cell 2 we find the primary bifurcation point, and in Cell 3 we follow the bifurcating branch, first with the default setting of using `Matlab`'s \ for both, `p.fuha.lss` and `p.fuha.blss`. Here we switch off all features (plotting, saving, spectral computations) not relevant for evaluating the LSS performance. Then setting `p=setbel(p,0,1e-4,5,@lss)`, i.e., border width 0, means that we still use `Matlab`'s \ for the non-bordered system $G_u v = b$, and `lssbel` effectively only occurs in `p.fuha.blss` with border width 1. Subsequently we run the same script with nper=80,. . . ,200 thus nu=6400, . . . , 16000. See Fig.1 and Table 2 for example plots and performance.

The rather large domains `nper=40,...,200` in Listing 1 are somewhat artificial, but were chosen because (a) such longs domain are needed for studying localized patterns (see, e.g., [UW14]), and (b) (here more to the point) to go to at least intermediate scale with n_u. The results[2] in Table 2 show the advantage of `lssbel` over `lss` in 1D. In particular, `lssbel` scales linearly with n_u. In Fig.1(b) we also mark Point 21 on the branch (for nper=40), because this is where the continuation switches from arclength to natural according to (1). Afterwards, the bordered matrices only occur in the computation of the new tangent but not in Newton loops, and thus the setting with `p.fuha.blss=@lss` is almost as fast as with `p.fuha.blssbel=@blssbel`. The speed advantage of `blssbel` occurs on the arclength segment from the bifurcation point to Point 21 on the branch. No post-iterations were needed for `blssbel` in this example.

Table 2: Speed test. All settings are as in Listing 1.

`p.fuha.lss`	`p.fuha.blss`	$n_u =$	3200	4800	6400	12800	16000
lss	blss	time(s):	3	11	22	38	59
lssbel	blssbel	time(s):	1.13	2	2.8	3.9	4.8

We refrain from going to 2D or 3D with (8) because the correct continuation of branches in pattern forming systems in higher space dimensions is a rather delicate problem, due to undesired branch switching. See, e.g., the discussion of `pmcont` in [UWR14, §4.3]. Moreover, when we do follow 2D or 3D branches for (8), then `p.fuha.blssbel=@blssbel` with \ as inner solver no longer has a speed advantage over `p.fuha.blss=@lss`. We believe that the reason for this is that $\partial_u G$ in higher dimension has a less favorable sparsity structure for \ since it no longer consists of tridiagonal blocks. Rather, in higher space dimension, using `lssAMG` as inner solver turns out to be a good choice, as discussed next.

4.2 3D Allen-Cahn

The second demo is called `ac3D`, and essentially uses the same setup as in [RU17, §4.2], namely the steady cubic-quintic Allen-Cahn equation

$$0 = -d\Delta u - \lambda u - u^3 + u^5 \tag{9}$$

on the domain $\Omega = (-l_x, l_x) \times (-l_y, l_y) \times (-l_z, l_z)$ with homogeneous Dirichlet boundary conditions. We choose $l_x = l_y = l_z = \pi/2$, $d = 1$ and take λ as a bifurcation parameter. The trivial branch of (9)

[2] on a standard laptop with an Intel i7-4510U processor with 3 GHz maximal frequency, two cores, and four threads

is given by $u = 0$, and bifurcations to solutions of the form $u = A\cos(k_x x)\cos(k_y y)\cos(k_z z) + \text{h.o.t.}$ with $A \in \mathbb{R}$ occur at $\lambda = k_x^2 + k_y^2 + k_z^2$, where the boundary conditions enforce $k = (k_x, k_y, k_z) \in \mathbb{N}^3$. Thus, the first bifurcation from the trivial branch is at $\lambda = 3$ and is simple, $k = (1, 1, 1)$, while at $\lambda = 6$ we have a triple bifurcation point with $k = (2, 1, 1), k = (1, 2, 1)$ and $k = (1, 1, 2)$. Here we focus on the performance of the LSS during continuation of the first nontrivial branch.

The folder `ac3D` contains the functions `acinit.m`, `oosetfemops.m`, `sG.m`, `sGjac.m`, `spjac.m`, and the script `cmds.m`, and we refer to [RU17, §2] for a discussion of the functions. In particular, `spjac.m` encodes $\partial_u(\partial_u G\phi)$ needed in fold continuation, see [RU17, §3.1.2]. Because (9) has the odd symmetry $u \mapsto -u$, the same extended system as for fold localization and continuation can also be used for the localization of branch points [WS84]. This is useful here, because spectral computations also become slow for large n_u, at least in the standard `pde2path` setting, and here we are only interested in the performance of the LSS for large n_u.

Thus, in `cmds.m` we switch off spectral computations. After init in C1, with parameter nx, in C2 we find the first BP (near $\lambda = 3$) using `bploc`, see [RU17, §3.1.2], which for large n_u becomes an order of magnitude faster than the default setting, based on eigenvalues and bisection. The core comparison of the LSS then takes place in C3; see the caption of Listing 2 for further discussion.

```
%% demo to compare \ and lssbel with lssAMG as inner solver
close all; keep pphome;
%% C1: init (generic), then specific settings (could also be set in init)
p=[]; par=[1 3 1 0]; lx=pi/2; ly=pi/2; lz=pi/2; nx=40; % choose different nx here
p=acinit(p,lx,ly,lz,nx,par); p.np, plotsol(p,1,1,1); p.fuha.spjac=@spjac;
p.nc.ilam=2; p.nc.lammax=5; p.sol.ds=0.1; p.nc.dsmax=1;
dir=mat2str(nx); p=setfn(p,dir); p.sw.verb=3; % outdir, and high verbosity
p.sw.spcalc=0; p.sw.bifcheck=0; p.sw.foldcheck=0; % swith off spectral stuff
bw=0; beltol=1e-3; belmaxit=5; droptol=1e-3; amgmaxit=50;  % param. for blssbel,
p=setbelilup(p,bw,beltol,belmaxit,droptol,amgmaxit); % lssAMG as inner LSS
p=cont(p,1); % 1 step, only to generate tangent
%% C2: localize BPs by setting lam to near a (here known) BP and using bploc
tic; p=bploc(p); toc % localize 1st BP
p=swibra(dir,'bpt1','b',0.01);  p=cont(p,1); % compute 1 point on bif. branch
%% C3: comparison of lss by cont till lam=1;
p0=loadp('b','pt1'); p0.file.smod=0; p0.plot.pmod=0;
p=p0; t1=tic; p=cont(p); t1=toc(t1); % using lssbel with lssAMG
p=p0; p.fuha.lss=@lss; p.fuha.blss=@lss; % switching back to \
t2=tic; p=cont(p); t2=toc(t2); fprintf('t1=%g, t2=%g\n',t1,t2);
```

Listing 2: Script `ac3D/cmds.m`. Cells 1 and 2 are preparatory. In Cell 1 we initialize with discretization parameter nx, set some standard parameters, and in line 8 switch off all spectral computations, as here we only want to evaluate the LSS performance. In lines 9,10 we set `lssbel` and `blssbel`, with inner solver `p.fuha.innerlss=@lssAMG`, and do one step on the trivial branch to obtain the tangent (though here it is trivial). In Cell 2 we then compute the first bifurcation point via `bploc`, and compute just one point on the nontrivial branch as a starting point for the LSS comparison in Cell 3. Again we have border width 0, and thus bordered elimination only occurs for the extended systems. Cell 4 deals with plotting, see Fig. 2. Cells 1-3 can then be repeated with different nx.

Table 3: Speed test. All settings are as in Listing 2.

`p.fuha.lss`	`p.fuha.blss`	`p.fuha.innerlss`	$n_u =$	8×10^3	27×10^3	64×10^3	125×10^3
lss	blss	NA	time(s):	7	73	330	1430
lssbel	blssbel	lssAMG	time(s):	2.6	11	31	72

Running the script `cmds.m` with $n_x = 20, 30, 40, 50$, i.e., with $n_u = 8.000, 27.000, 64.000, 125.000$ unknowns, gives the results in Table 3. Additionally we remark that for $n_x = 20, 30, 40, 50$ a typical preconditioner computation (with droptol=1e-3) takes about $0.2, 1.4, 4, 9$ seconds. `blssbel` with

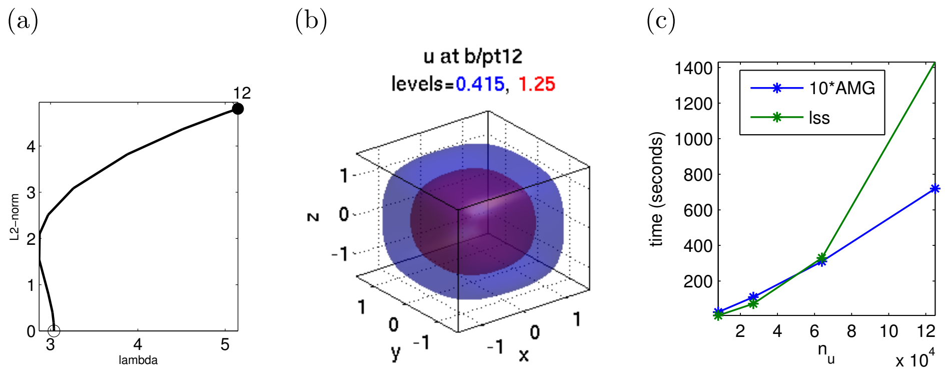

Figure 2: Branch and solution plots (a),(b) and computation times (c) for the test problem.

`lssAMG` as inner solver seems to scale linearly for this problem, and becomes highly advantageous at large n_u. In fact, depending on memory, `Matlab`'s \ here tends to become useless for n_u not much larger, $n_u = 200.000$, say. Of course, this again is only an example, and quite friendly to `blssbel` because as in §4.1 no post-iterations were needed in `blssbel`, and because `lssAMG` needed at most 2 preconditioner computations in each run.[3]

5 Summary and Remarks

We gave two examples of how the performance of continuation with `pde2path` can be significantly improved by choosing suitable LSS. It would be somewhat premature to draw general conclusions from these examples, and to be on the safe side for now we keep `p.fuha.lss=@lss` and `p.fuha.blss=@blss` (`Matlab`'s \ as a highly optimized LSS) as default settings in `stanparam`. However, the examples are not especially tweaked, and we close with the following recommendations and remarks, also supported by further experiments.

1. In 1D, try `p.fuha.lss=@lssbel`, `p.fuha.blss=@blssbel` with `p.fuha.innerlss=@lss`, since `p.fuha.innerlss=@lssAMG` typically does not seem to increase performance in 1D. A convenience call for this is `p=setbel(p,bw,beltol,belmaxit,@lss)`, see Listing 1.
2. For 2D or 3D larger scale problems, try `p.fuha.lss=@lssbel` and `p.fuha.blss=@blssbel` with `p.fuha.innerlss=@lssAMG`. A typical convenience call to switch on these settings is `p=setbelilup(p,bw,beltol,belmaxit,droptol,amgmaxit)`, see Listing 2.
3. Similar settings as in 1. and 2. are also highly recommended for Hopf problems, see [Uec17b].
4. In our tests we switched off all spectral computations, which in the standard setting are needed to detect and localize bifurcations, and for stability information.[4] For large n_u (on the order of 10^5, say), eigenvalue computations become another severe and often dominating bottleneck. However, if a few eigenvalues near some shifts σ_j are sufficient (`p.sw.bifcheck=2` setting, see [Uec17a]), then again a significant speed-up can be obtained by using `lssAMG` as a linear system solver inside `Matlab`'s `eigs`. Details of this will be given elsewhere, and so will details of how to use `bploc` for branch point localization during continuation, and how to deal with multiple bifurcation points as for instance at $\lambda = 6$ in the `ac3D` example.

References

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

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

[3]During these continuations, the algorithm switches back and forth between natural and arclength three times according to (1), which shows the usefulness of using the same `prec` for both.

[4]In §4.2 we also computed the primary bifurcation point based on `bploc`, which reduces the spectral computations to computing one eigenvalue and the associated eigenvector during `spcontini` within `bpcont`.

[dWDR$^+$17] H. de Witt, T. Dohnal, J. Rademacher, H. Uecker, and D. Wetzel. pde2path - Quickstart guide and reference card, 2017. Available at [Uec17c].

[Gov00] W. Govaerts. *Numerical methods for bifurcations of dynamical equilibria.* SIAM, 2000.

[RU17] J. Rademacher and H. Uecker. The OOPDE setting of pde2path – a tutorial via some Allen-Cahn models, 2017.

[Uec17a] H. Uecker. Hopf bifurcation and time periodic orbits with pde2path – algorithms and applications, Preprint, 2017.

[Uec17b] H. Uecker. User guide on Hopf bifurcation and time periodic orbits with pde2path, 2017.

[Uec17c] H. Uecker. `www.staff.uni-oldenburg.de/hannes.uecker/pde2path`, 2017.

[UW14] H. Uecker and D. Wetzel. Numerical results for snaking of patterns over patterns in some 2D Selkov-Schnakenberg Reaction-Diffusion systems. *SIADS*, 13-1:94–128, 2014.

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

[WS84] B. Werner and A. Spence. The computation of symmetry-breaking bifurcation points. *SIAM J. Numer. Anal.*, 21(2):388–399, 1984.