

*Lecture given at the International Summer School Modern Computational Science
(August 15-26, 2011, Oldenburg, Germany)*
In *MCS11 – Simulation of Extreme Events*, R. Leidl, A. Hartmann (Eds.),
BIS-Verlag der Carl von Ossietzky Universität Oldenburg, 197–226, 2011

Interfacing Fortran routines from Matlab in a simple and efficient way – with applications to ordinary and partial differential equations

*Hannes Uecker
Faculty of Mathematics and Science
Carl von Ossietzky Universität Oldenburg
D-26111 Oldenburg
Germany*

Abstract. We explain the usage of MEX files to call Fortran routines from Matlab in a “quick and dirty” but simple and efficient way. Our main examples are interfaces to the ODE solver SODEX and the PDE solver PDE TWO. We apply the SODEX interface to the van der Pol oscillator and Chua’s circuit, illustrating a significant speedup compared to Matlab integrators, and use the PDE TWO interface to integrate a reaction diffusion system, the porous medium equation, a wave equation, some shallow water equations, and some more examples.

Contents

1	Introduction	2
1.1	The baby MEX : <code>timesa</code>	3
2	An ODE solver: SODEX	5
2.1	The solver	5
2.2	Two examples	6
3	PDE TWO	12
3.1	The class of problems	12
3.2	The gateway	13
3.3	Four examples	14
4	Summary	26

1 Introduction

Matlab is a powerful mathematical software that originated as an “easy input – easy output” interface to numerical subroutines/libraries such as BLAS and LAPACK, written in Fortran. Today it is widely used in academia and industry and has a huge number of built in functions and features. However, even though via so called MEX files **Matlab** offers a convenient way to interface additional Fortran routines or libraries, at least in academic teaching this option does not seem to be much used. Therefore, here we review some basics of this. There are two possible advantages to using external library routines in **Matlab**:

- a) A **Matlab** routine is not available for the specific task, but a library routine is.
- b) Even if there is a **Matlab** routine, some library routine may still be faster (sometimes by orders of magnitude, see the ODE examples below).

We assume that the reader is familiar with the basic usage of **Matlab**, and otherwise refer to the extensive documentation and tutorials everywhere on the internet (and within **Matlab**). We restrict to Fortran source code and thus only assume that the reader can read and write some elementary Fortran. However, even if the reader has never used Fortran before, this should be no problem: with the templates provided, anyone who has a very basic understanding of **Matlab** (or, e.g., C, Java, ...) can write the elementary fortran codes needed ¹.

After some introductory toy problem we focus on the ODE (ordinary differential equation) solver SODEX and the PDE (partial differential equation) solver PDETWO. For this we assume some familiarity with PDEs and ODEs and their numerical solutions, on a very basic level. The number of references we give is rather short, since for most of the following much excellent information can be found on the internet; in particular we again recommend the **Matlab** documentation and, for introductory purposes, the Wikipedia articles on, e.g., ordinary and partial differential equations, numerical solutions, stiff ODEs, etc ². Of the large variety of books on ODE and PDE we recommend [Str94] for ODEs, [Str92, Eva98, Rob01] for PDEs and modeling, and [HNW93, HW96, Co098] for numerical aspects. See also [Uec09] and the references therein.

All **Matlab**, MEX and library routines (in source code) used here can be downloaded from [Uec11], as well as possible updates and extensions. The folder contains a file README, which describes the folder contents and includes setup remarks. The folder only contains “free software”, but concerning copyright and license details we refer to the source codes of the libraries, and, e.g., concerning PDETWO to the ACM license agreement³. The routines should run on any computer with **Matlab** and a Fortran compiler (supported by **Matlab**). We tested them on a linux system with **Matlab** R2006b and **gfortran**.

¹additionally, there are many Fortran tutorials on the web

²as is natural for software issues, we also give a number of urls for documentation and references

³<http://www.acm.org/publications/policies/softwarecrnotice/>

Remark 1.1 A free and almost compatible alternative to `Matlab` is `octave`⁴. Like `Matlab`, `octave` has the option of interfacing libraries and producing executable so called “octfiles” from C and Fortran source files. Here we focus on `Matlab` MEX files, but in fact our setup of a MEX gateway to PDETWO was motivated by a similar (but more elaborate) `octave` gateway [Wea06].]

1.1 The baby MEX : timesa

The way to call Fortran subroutines from `matlab` is via MEX (MatlabEXecutable) files. As already said, on this there is extensive help, documentation and tutorials⁵, and thus here we only briefly explain the basic steps, with a small twist, which does not seem to be widely used. More sophisticated library routines `sub` like e.g. ODE or PDE solvers typically call user supplied subroutines `subsub`, where the interface to `subsub` is fixed. Then, if either the interface to `sub` or from `sub` to `subsub` does not allow the user to pass all parameters needed, the way to do this is via `COMMON` blocks. This is (naturally) the situation we encounter with virtually all library ODE or PDE solvers, and thus here we want to explain a solution in conjunction with `Matlab` using an otherwise standard baby MEX file, namely the multiplication of a vector `x` with a scalar `a`.

MEX (source) files⁶ are Fortran files, and thus (contrary to `Matlab`) require some formatting⁷ and moreover in principle need declaration of variables. However, we often use the declaration `IMPLICIT DOUBLE PRECISION (A-H,O-Z)` which gives variables with names starting with `A-H` or `O-Z` automatically the type double, and all other variables the type integer. This is handy, but also dangerous!

Here is our first MEX file, condensed to what is necessary, called `timesa.F`:

```

1 #include "fintrf.h"
2     subroutine mexFunction(nlhs, plhs, nrhs, prhs)
3 ! Gateway routine to multiply input vector x by scalar a
4 ! compile as mex timesa.F, call as [y]=timesa(x,nx,a)
5     IMPLICIT DOUBLE PRECISION (A-H,O-Z)
6     parameter (nmax=10) ! maximal length for x
7     mwpointer mxCreateDoubleMatrix,mxGetPr,mxCreateNumericMatrix
8     double precision mxGetScalar
9     mwpointer prhs(*),plhs(*),x_pr, a_pr, y_pr
10    dimension x(nmax)
11    common /para/ a

```

⁴www.gnu.org/software/octave/

⁵a starting point is www.mathworks.com/help/techdoc/matlabexternal/f7667dfi1.html

⁶in the following, by MEX file we mainly mean the respective (Fortran) source codes (*.F files), while strictly speaking the MEX file is the compiled file with ending *.mexglx (linux) or *.mexw32 (windows 32-bit) or similar, which can be called from `Matlab`

⁷we just mention that commands must start in column 7 or later, that any character in column 6 indicates a continuation of the previous line, that everything after column 72 will be ignored, and that comments are behind a C or !. Historically, Fortran programs are UPPER CASE. However, since no compiler seems to actually require this (they are case-insensitive), and since programming is always a matter of copy-paste, here for convenience we freely mix UPPER and lower case letters

```
12     x_pr=mxGetPr(prhs(1));nx=mxGetScalar(prhs(2));
13     call mxCopyPtrToReal8(x_pr,x,nx);          ! input x
14     a=mxGetScalar(prhs(3));                    ! input a
15     call timesa(x,nx)                          ! call the calculation routine
16     plhs(1)=mxCreateDoubleMatrix(1,nx,0)      ! prepare return
17     y_pr=mxGetPr(plhs(1)); call mxCopyReal8ToPtr(x,y_pr,nx);
18     return; end
19
20     subroutine timesa(x,nx) ! ‘‘computational’’ subroutine
21     IMPLICIT DOUBLE PRECISION (A-H,O-Z)
22     dimension x(nx)
23     character(200) mstring          ! for printing info
24     common /para/ a
25     !write(mstring,*) 'a=',a; k=mexPrintf(mstring);
26     do k=1, nx; x(k)=a*x(k); enddo
27     return; end
```

Any (Fortran) MEX file is named *.F, with * the desired name of the routine, and starts with the preprocessor macro on line 1, followed by the Gateway Routine, always declared as `subroutine mexFunction(nlhs, plhs, nrhs, prhs)`. The arguments are abstract in the sense that the data contained must be extracted in a rather explicit way. First we remark that the main use of `nlhs` and `nrhs` is to check the correct number of input and output arguments. This we clip in our condensed file, see again footnote 5.

In Line 6 we define the maximal length of vectors to be processed; instead of this we could use dynamic memory allocation, but for simplicity we also clip this. For this, see also Remarks 2.1 and Remarks 3.1 below.

Lines 7 to 9 contain the necessary MEX constructions: everything passed to or from a MEX routine should be dealt with as a pointer. Line 10 provides the actual vector to pass to the computational subroutine, and Line 11 defines the common block `/para/`, which will be available to all subroutines in the MEX file.

Lines 12 to 14 process the input, which is then passed to `timesa`. In `subroutine timesa` we commented out line 25 which gives an example of output to the `Matlab` command window; this may be useful for debugging. On return from `timesa` (to line 16), the result is copied to the output pointer `y_pr`, and then control returns to `matlab`.

Now typing `mex timesa.F` in `Matlab` and, e.g., `x=[1 2 3]; nx=3; a=2; y=timesa(x,nx,a)` produces `y = 2 4 6`, as expected.

As already said, this was a rather stripped (for instance, no error checking) version of a MEX file for a completely trivial task; again see footnote 5 for many more explanations concerning structs like `x_pr=mxGetPr(prhs(1))` and many more examples. The only thing we want to point out is the use of the common block `/para/`. Again, this construction is rather useless here, but it will be handy when we need to pass parameters to subroutines, for which we have no control of the shape of the interface, as will be the case in the next sections.

2 An ODE solver: SODEX

Many processes are described by ordinary differential equations (ODE), which are equations of the form

$$u'(t) = f(t, u(t)) \quad (1)$$

for an unknown function $u \in C^1(I, \mathbb{R}^p)$, where $I \subset \mathbb{R}$ is an interval, $f : I \times \mathbb{R}^p \rightarrow \mathbb{R}^p$ is called the vector field, and $C^1(I, \mathbb{R}^p)$ denotes the space of continuously differentiable functions from the interval I to \mathbb{R}^d . An initial value problem consists of (1) together with an initial condition (IC) $u_0 \in \mathbb{R}^p$ at some time t_0 , i.e., $u|_{t=t_0} = u_0$. For $p \geq 2$, (1) is sometimes also called a system of ODEs.

Often, solutions to (1) cannot be found analytically and thus we need to retreat to numerical approximations. The simplest method would be the explicit Euler method: since $f(t, u(t)) = u'(t) \approx \frac{1}{h}(u(t+h) - u(t))$ for small h , given $u(t)$ and a stepsize h we approximate $u(t+h) \approx u(t) + hf(t, u(t))$. However, in particular for so called stiff ODEs this has severe problems and often does not work at all, see, e.g. [HW96] or the `Matlab` documentation, or [Uec09] and the references therein for further discussion. Therefore, we want something better.

`Matlab` has a number of built in ODE solvers. The most popular are `ode45` for non stiff problems and `ode15s` for stiff problems. Additionally, a multitude of library routines are available in C and Fortran. In this section we use model problems to illustrate a MEX gateway to the ODE solver SODEX, available at [Hai11], where a large number of other ODE integrators and other software is available⁸. Comparing SODEX to builtin `Matlab` routines we shall see that for instance for the standard problem of the (stiff) Van der Pol oscillator, SODEX yields a

“speedup by a factor of about 320”

compared to the best `Matlab` solvers for this problem⁹.

2.1 The solver

SODEX is designed to solve stiff ODEs $My' = f(x, y)$ where the independent variable t is now called x and $M \in \mathbb{R}^{p \times p}$ is a so called mass matrix. This includes so called differential algebraic equations (DAEs) where M may be singular. See [HW96, §IV.9].

The Fortran interface is

```

1      SUBROUTINE SODEX(N,FCN,IFCN,X,Y,XEND,H, RTOL,ATOL,ITOL,
2      &    JAC ,IJAC,MLJAC,MUJAC,DFX,IDFX,MAS,IMAS,MLMAS,MUMAS,
3      &    SOLOUT,IOUT,WORK,LWORK,IWORK,LIWORK,IDID)
```

We only discuss a few of the parameters of SODEX; all details can be found in `sodex.f`. On input, N ($= p$ in our notation) is the dimension of the ODE,

⁸we chose SODEX for no particular reason, and other solvers may well outperform SODEX, which in fact is listed as an “oldie” under [Hai11]

⁹this statement will be put into perspective below

X is the current time (value of the independent variable) ,
 Y is the current solution, and
 FCN describes the right hand side f of the ODE. This FCN *must* read

```

1      SUBROUTINE FCN(N,X,Y,F)
2      REAL*8 X,Y(N),F(N)
3      F(1)=...   ETC.
4      RETURN
5      END
    
```

Thus there is no option to pass parameters to FCN. Here the **COMMON** construction as in §1 becomes usefull. Additionally, the user must supply subroutines **JAC** and **SOLOUT**, which however in the simplest setting can be empty and thus shall not be discussed here. On successful return from **SODEX**, amongst other things, **X** and **Y** are overwritten by **X=XEND** and **Y= y(x)**, respectively.

We now explain by two examples a MEX interface to SODEX which is called in the form

$$[\mathbf{ynew} \ t_{\mathbf{new}}] = \mathbf{fconsodex}(\mathbf{y}, \mathbf{t}, \mathbf{dt}, \mathbf{para}, \mathbf{npara}, \mathbf{n})$$

where **fcn** stands for some particular ODE, $[\mathbf{y} \ \mathbf{t}]$ are the current values on input, **dt, para, npara, n** are the length of the time interval, the parameters, the number of parameters, and the number of ODEs, respectively, and **ynew** is the numerically calculated value of $\mathbf{y}(t_{\mathbf{new}} = \mathbf{t} + \mathbf{dt})$.

Remark 2.1 Thus, what we really aim at and provide here is a “quick and dirty” gateway to SODEX in the following sense: we do not pass a function handle to a user provided **Matlab** function FCN. Instead, FCN is coded (in Fortran) within the MEX file. Therefore, *for each ODE we need a separate MEX file which must be compiled*. Moreover, there will be no dynamic memory management. This is not elegant, but the MEX files are easy to understand. Moreover, coding FCN in Fortran and compiling may be expected to be faster than a **Matlab** FCN. See <http://www-m3.ma.tum.de/Software/ODEHome> for more elegant (but also much more elaborate) MEX interfaces (in C) to ODE solvers, which pass all relevant parameters and also function pointers. See also Remark 3.1.]

2.2 Two examples

2.2.1 Van der Pol oscillator

The van der Pol oscillator $y'' - \mu(1 - y^2)y' + y = 0$ with $\mu > 0$ a parameter models some electric circuit. Rewritten as a first order system for $\mathbf{u} = (y, y')$ we have

$$\begin{aligned} u_1' &= u_2, \\ u_2' &= \mu(1 - u_1^2)u_2 - u_1, \end{aligned} \tag{2}$$

which is of the form (1). We are mainly interested in μ large. Near $u_1 = 0$ the Jacobian of the right hand side has eigenvalues $\approx \pm\sqrt{\mu}$, i.e., eigenvalues of strongly different real parts. This characterizes a so called stiff system.

Here is a MEX gateway to SODEX. The stripped¹⁰ version of the MEX file `vdpsodex.F` reads

```

1  #include "fintrf.h"
2  c    vdpsodex.F, MEX interface to sodex for van der Pol
3  c    compile: mex vdpsodex.F sodex.o decsol.o
4  c    call: [y t]=vdpsodex(x,t,h,p,npara,n)
5      subroutine vdp(n,t,x,y) ! the actual rhs to integrate, here vdp
6      implicit real*8 (a-h,o-z)
7      dimension x(n),y(n), dp(100)
8      common /para/ dp
9      dmu=dp(1); y(1)=x(2); y(2)=dmu*(1-x(1)**2)*x(2)-x(1);
10     return; end
11
12     subroutine mexFunction(nlhs, plhs, nrhs, prhs) ! Interface
13     IMPLICIT REAL*8 (A-H,O-Z)
14     parameter(nd=100,nparm=100) ! maximal number of ODES/parameters
15     dimension x(nd),y(nd),p(nparm)
16     common /para/ p
17     ... usual data extraction ...
18     call sostep(x,t,h,n) ! call the computation
19     ... usual procession of result to pass back to matlab
20     return; end
21
22     subroutine sostep(x,t,h,n) ! "helper interface"
23     ... define variables/switches needed by SODEX, see vdpsodex.F and sodex.f
24     CALL SODEX(N,vdp,IFCN,t,x,t+h,hi,RTOL,ATOL,ITOL,
25     & jaco,IJAC,MLJAC,MUJAC,jaco,IDFX,jaco,IMAS,MLMAS,MUMAS,
26     & SOLOUT,IOUT,WORK,LWORK,IWORK,LIWORK,IDID)
27     return; end
28
29 c auxiliary routines (may be dummy)
30     SUBROUTINE SOLOUT(NR,XOLD,X,Y,N,IRTRN) ! useful for intermed. output
31     return; end
32     SUBROUTINE jaco(N,X,Y,DFY,LDFY)
33     return; end

```

Naturally, subroutine `vdp(n,t,x,y)` is the right hand side f from (2). We put it first since in most cases *this is the only subroutine which the user must edit when changing the ODE*. Moreover, typically the user only needs to *edit one or two lines*, here line 9.

The gateway routine starts in line 12. In line 14 we hardwire the maximal number of ODEs and parameters¹¹. Then we have a few lines of extracting the data from `prhs`, in particular fill the parameter common `/para/`. We then call the “helper interface” `sostep`, which deals with setting values to the (large number of) parameters used to

¹⁰here and in the following we usually shorten the original files a bit; we use “... *** ...” to indicate omissions which can be looked up in `vdpsodex.F` if desired

¹¹again, this is not elegant, but practical

control (and needed to call) SODEX. Upon return from `sostep` we process the result to pass it back to `Matlab`. In lines 29 resp. 33 there are two additional subroutines required by SODEX. With our parameter settings in `sostep` they are however never called. Thus they are only required by the compiler and may be dummy.

Altogether, `vdpsodex.F` is 50 lines of code. To compile it, in `matlab` we type `mex vdpsodex.F sodex.o decsol.o`. This assumes that `sodex.o` and `decsol.o` are available for linking, i.e. that we already compiled `sodex.f` and `decsol.f` (the latter contains linear algebra routines required by SODEX). This can be done for instance by calling `gfortran -c sodex.f decsol.f` from a shell, or typing `mex -c sodex.f decsol.f` at the `Matlab` prompt¹².

A basic `Matlab` script (see `vdptest.m`) which calls `vdpsodex` reads ¹³

```

1 % comparison of sodex-mex with built in solvers
2 global mu; mu=500; tend=10000; x=[2;0];
3 tic; [tv1,yv1]=ode15s(@vdprhs,[0 tend],x);s1=toc
4 tic; [tv2,yv2]=ode23s(@vdprhs,[0 tend],x);s2=toc
5 npara=1;para=mu;nsteps=100;t=0;tend=10000;n=2;
6 x=[2; 0]; dt=tend/nsteps; sol=zeros(2,nsteps); tv=zeros(1,nsteps);
7 tic;
8 for k=1:nsteps
9     [x t]=vdpsodex(x,t,dt,para,npara,n); sol(:,k)=x;tv(k)=t;
10 end
11 s3=toc, plot(tv1,yv1(:,1),'-',tv,sol(1,:),'+');
```

We set $\mu = 500$ and in lines 2-4 for later comparison first solve (2) with the built in `Matlab` solvers `ode15s` and `ode23s`.¹⁴ For these we obviously need the right hand side of (2) as a `Matlab` function as well, which needs the parameter μ , which is therefore declared as `global` in line 1. See `vdprhs.m`.

Lines 5-7 prepare the call to `vdpsodex` which is done in the loop in lines 8-10. The remainder of the file plots the results, and compares run times. Figure 1 illustrates that the solution shows so called relaxation oscillations, with intervals where u changes rapidly, as is typical for stiff problems.

For the execution times, on a notebook with an Intel Core 2 Duo SU7300 ULV processor, we have, approximately, in seconds,

s1=ode15s	s2=ode23s	s3=vdpsodex
5.5	5.1	0.0016

Thus, `vdpsodex` beats the two `Matlab` solvers by a factor of about 320. However, a few comments are in order:

¹²where additional compiler flags may be helpful; for instance, use `gfortran -c -O3 sodex.f decsol.f` for code optimization

¹³the original `Matlab` scripts in most cases contain some more comments; moreover, they usually use `cell mode` and hence should be conveniently stepped through using the `Matlab` editor.

¹⁴all other `Matlab` solvers fail for this problem or at least give executions larger by orders of magnitude

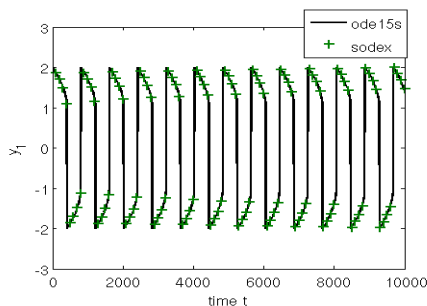


Figure 1: Output of vdpctest.m

Remark 2.2 a) s_1, s_2, s_3 are “typical times” in the sense that repeated calls may slightly differ due to other system loads.

b) The behaviour of the `Matlab` solvers can be tweaked in many ways by setting a number of parameters. Here we just used the defaults, for instance `RelTol = .001` and `AbsTol = .000001`. Of course, we run SODEX with the same values.

c) The `Matlab` solvers produce output $t = t_0, t_1, \dots, t_N$, $y = y(t_0), \dots, y(t_N)$ in a controlled form “suitable for plotting”, i.e. intermediate output is created whenever the solution y varies on a certain scale. This is convenient but (of course) consumes time. However, calling `ode15s` in the form

```
nsteps=100;t=0;tend=10000; x=[2; 0];tend=10000;dt=tend/100;tv=0:dt:tend;
tic; [tv3,yv3]=ode15s(@vdp rhs, tv, x);s1=toc
```

which produces no intermediate output, again gives $s_1 \approx 5$, thus no gain in speed. Similarly, calling `ode15s` in a `for` loop like `vdp sodex` gives $s_1 \approx 6$.

d) On the other hand, `vdp sodex` produces no intermediate output, it just returns the solution at the end of the desired time interval. That is why we need the loop in lines 8-10 to produce intermediate output by hand, controlled by `nsteps`; increasing `nsteps` to have denser output increases s_3 , but only slightly. On the other hand, calling, e.g., `[x t]=vdp sodex(x,0,tend,para,npara,n)` we get $s_3 \approx 0.001$.

e) Given these caveats, it is clear that even for the specific problem (including machine, `Matlab` release, compiler) we cannot strictly claim that “SODEX is 320 times faster than `ode15s`”. However, we can say that for the above problem SODEX in the given setting is *significantly, i.e., by orders of magnitude* faster than `ode15s`.]

Alltogether, the comparison of different algorithms for some (class of) task(s) is a difficult problem¹⁵, and the above comparison is in no way meant to be rigorous. Rather we invite the reader to make his or her own comparisons for his or her specific problem at hand.

At least equally important as speed is accuracy. Here we only take Fig. 1 as proof that both `ode15s` and SODEX produce the same (and hence “accurate”) results. In general we may say that the `Matlab` solvers as well as library solvers have sophisticated

¹⁵“There is no algorithm you can’t fool, unless you don’t understand it” (folklore)

error control such that for “generic” systems accuracy is not a problem and the solutions can be trusted. Nevertheless one should be aware that this is not quite true for “chaotic” problems. An example will be given now.

2.2.2 Chua’s circuit

Chua’s circuit is another model from electrical circuit theory, namely

$$u_1' = a(u_2 - u_1 - g(u_1)), \quad u_2' = u_1 - u_2 + u_3, \quad u_3' = -bu_2, \quad (3)$$

with parameters $a, b, c, d \in \mathbb{R}$ and $g(u) = cu + \frac{1}{2}(d - c)(|u + 1| - |u - 1|)$. For certain parameters it has so called chaotic orbits.

To integrate (3) with SODEX, copy `vdpsodex.F` to `chuasodex.F` and then change lines 5-11 to

```

1      subroutine chua(n,t,x,y)    ! Chua's circuit
2      implicit real*8 (a-h,o-z)
3      dimension x(n),y(n), dp(100)
4      common /para/ dp
5      a=dp(1); b=dp(2); c=dp(3); d=dp(4);
6      g=c*x(1)+0.5*(d-c)*(dabs(x(1)+1)-dabs(x(1)-1))
7      y(1)=a*(x(2)-x(1)-g); y(2)=x(1)-x(2)+x(3); y(3)=-b*x(2)
8      return; end

```

That’s all. In Matlab, type “`mex chuasodex.F sodex.o decsol.o`” and then run (see `chuatest.m`)

```

1  global a b c d; a=15; b=25.58; c=-5/7; d=-8/7; x=[2; 0; 0]; tend=100;
2  tic; [tv1,yv1]=ode15s(@chua,[0 tend],x);s1=toc
3  tic; [tv2,yv2]=ode45(@chua,[0 tend],x);s2=toc
4  npara=4;para=[a b c d]; sol=zeros(3,nsteps); tv=zeros(1,nsteps);
5  nsteps=1000; t=0; tend=100; n=3; x=[2; 0; 0]; dt=tend/nsteps; tic;
6  for k=1:nsteps
7      [x t]=chuasodex(x,t,dt,para,npara,n); sol(:,k)=x;tv(k)=t;
8  end
9  s3=toc, plot(tv1,yv1(:,1),'+',tv2,yv2(:,1),tv,sol(1,:),'-x');

```

This integrates (3) using `ode15s`, `ode45` and SODEX, with parameters (taken from [Lyn04, §13.4]) $(a, b, c, d) = (15, 25.58, -5/7, -8/7)$, and produces the output in Fig.2. The runtimes are

$$(s_1(\text{ode15s}), s_2(\text{ode45}), s_3(\text{SODEX})) \approx (1.45, 0.33, 0.03).$$

Thus, here SODEX can be considered about 12 times faster than `ode45`, and about 44 times faster than `ode15s`.

We start by remarking that (3) with the above parameters is *not* a stiff ODE. Therefore, `ode45` is faster than `ode15s`, and similarly, (3) is not particularly in the

class for which SODEX was designed¹⁶. Nevertheless, SODEX is still by far faster than the `Matlab` solvers, which must be attributed to the speed advantage of Fortran.

Looking at Fig. 2, this example also illustrates the issue of accuracy; (3) is a *chaotic* ODE, which roughly means that arbitrary small changes in initial data may lead to completely different (bounded) behavior after some time. Numerical error leads to such changes in initial data (for the next step) and thus *no* numerical solution can be accurate after some time. Here `ode15s` and SODEX agree somewhat longer with each other than with `ode45`, probably because `ode15s` and SODEX are in the same class of (implicit) integrators, but we must accept that we don't know the solution with any accuracy after some time t_1 , with here, say, $t_1 = 10$.

In conjunction with the so called Lorenz system this is often compared to weather forecast. For weather forecast the time t_1 is generally accepted to be about 15 days.

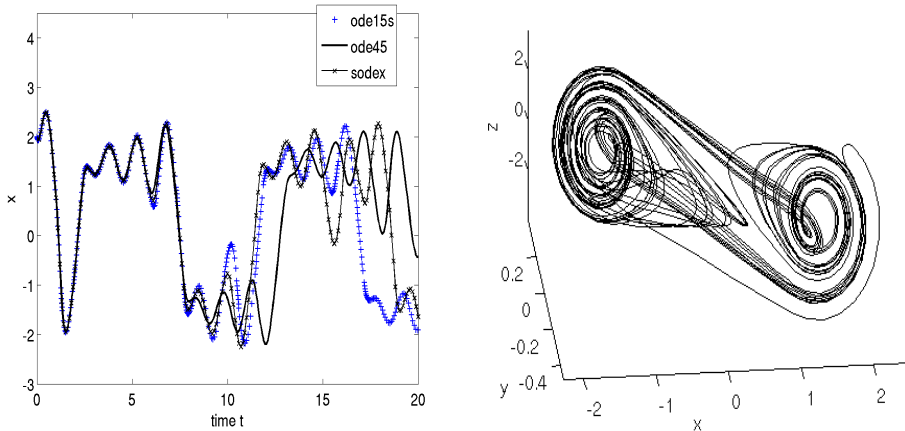


Figure 2: Output of `chuatest.m`. Left: time-series $u_1(t)$. Right: 3D illustration of the chaotic attractor.

For didactic reasons we suggest the following exercise, –and provide “solutions” in [Uec11]. However, at this point the reader may as well set up his or her own ODEs.

Exercise 2.3 Set up the following ODEs for SODEX.

- Consider the Fitz-Hugh–Nagumo system $u' = f(u) - v$, $v' = \varepsilon(u - \gamma v)$, with $f(u) = u(1 - u)(u - a)$ and $\varepsilon, \gamma > 0$ and $a \in \mathbb{R}$ some parameters, see, e.g., [Kee88, chapter 12]. Set, e.g., $(a, \gamma) = (0.25, 1)$ and let $\varepsilon \searrow 0$.
- Consider the Lorenz system $x' = \sigma(y - x)$, $y' = \rho x - y - xz$, $z' = -\beta z + xy$, with parameters σ, β, ρ . Fix $\sigma = 10$, $\beta = 8/3$ and set, e.g., $\rho = 0.5$, $\rho = 2$, $\rho = 10$ and $\rho = 27$.]

¹⁶see also <http://www1.uni-hamburg.de/W.Wiedl/Skripte/Matlab/ODE/RUBY/> (in german) for an enlightening discussion how to dynamically detect stiffness and choosing the best integrator

3 PDETWO

We now consider some time dependent problems described by partial differential equations (PDE), which are equations of the form

$$G(t, x, u, \partial_t u, \partial_{x_1} u, \partial_{x_2} u, \dots) = 0. \quad (4)$$

Here $u : [t_0, t_1] \times \Omega \rightarrow \mathbb{R}^p$ is an unknown function of time $t \in [t_0, t_1]$ and space $x \in \Omega \subset \mathbb{R}^d$, and (4) expresses a relation between u and its partial derivatives $\partial_t u, \partial_{x_1} u$ and so on. If $p = 1$ then (4) is also called a scalar PDE, while for $p \geq 2$ it is called a system of PDE.

PDE are everywhere; probably the easiest and best known examples are

- the transport equation $\partial_t u(t, x) - c \partial_x u(t, x) = 0$ ($d = 1, p = 1$), where $c \in \mathbb{R}$ is a speed;
- the heat equation or diffusion equation $\partial_t u(t, x) - D \partial_x^2 u(t, x) = 0$ ($d = 1, p = 1$), where $D > 0$ is called diffusion coefficient;
- the wave equation $\partial_t^2 u(t, x) - c^2 \partial_x^2 u(t, x) = 0$ ($d = 1, p = 1$); where $c > 0$ is the wave speed;
- the Poisson problem $\Delta u = f$ ($d = 2, p = 1$), where $\Delta u = (\partial_{x_1}^2 + \partial_{x_2}^2)u$ is the Laplace operator and $f : \Omega \rightarrow \mathbb{R}$ is some given function.

Other important PDE include Reaction–Diffusion systems, Navier–Stokes equations, Euler equations, the Schrödinger equation, Maxwell’s equations, Einstein’s equations, and many more. Since PDE are everywhere and analytical solution techniques are rare, we are interested in the numerical solution of PDE.

It is sometimes said that every (class of) PDE is different (*), in the sense that additionally to its specific analytical difficulties every PDE requires its own special numerics. Nevertheless, we now present the general solver PDETWO [MS81b, MS81a], written in Fortran, and applicable to a rather large class of systems of PDE in two space dimensions, and give a `Matlab` interface together with a few examples. The idea is that even while the above statement (*) is basically true, for quick inspection of a given PDE it is useful to have a general solver, even if it cannot resolve some fine properties of solutions. Moreover, results of the general solver can be used for comparison in the developing and testing of more specific solvers: it is always good to have a second tool for some task.

3.1 The class of problems

The general form (4) is of course rather useless, and we need to be more specific. PDETWO deals with initial boundary value problems for systems of p PDEs defined over rectangular domains

$$R = \{(x, y) : a_1 \leq x \leq b_1, a_1 \leq y \leq b_1\}.$$

The l^{th} PDE, $l = 1, \dots, p$, has the form

$$\partial_t u_l = f_l \left(t, x, y, u_1, \dots, u_p, \partial_x u_1, \dots, \partial_x u_p, \partial_y u_1, \dots, \partial_y u_p, \partial_x(D_{l,1}^h \partial_x u_1), \dots, \partial_x(D_{l,p}^h \partial_x u_p), \partial_y(D_{l,1}^v \partial_y u_1), \dots, \partial_y(D_{l,p}^v \partial_y u_p) \right) \quad (5)$$

The diffusion coefficients $D_{l,j}^h$ (horizontal) and $D_{l,j}^v$ (vertical) can be functions of t, x, y, u .

On the boundaries ∂R of R we need boundary conditions; on the horizontal boundaries $a_1 \leq x \leq a_2$, $y = b_1$ or $y = b_2$, these are assumed to be of the form

$$a_l^h u_l + b_l^h \partial_y u_l = c_l^h, \quad l = 1, \dots, p \quad (6)$$

and on the vertical boundaries $b_1 \leq y \leq b_2$, $x = a_1$ or $x = a_2$ we have

$$a_l^v u_l + b_l^v \partial_x u_l = c_l^v, \quad l = 1, \dots, p. \quad (7)$$

Here $a_l^h, b_l^h, c_l^h, a_l^v, b_l^v$ and c_l^v can be functions of t, x, y , and also of u if $b_l^* \neq 0$.

Thus, PDETWO can deal with the first two examples above, respectively with their 2D ($d = 2$) generalizations, and also with the wave equation by rewriting it as a first order system. Moreover, it can deal with such systems of arbitrary dimensions $p \geq 2$, all sorts of time- and space dependent coefficients and boundary conditions, and almost all sorts of nonlinearities¹⁷.

All the user of PDETWO has to do is provide a (not necessarily uniform) mesh $(x_i, y_j)_{i=1, \dots, nx, j=1, \dots, ny}$ for R , some initial conditions, the functions f_l , the diffusion coefficients $D_{l,k}^h$ and $D_{l,k}^v$, and the boundary functions $a_l^h, b_l^h, c_l^h, a_l^v, b_l^v, c_l^v$. PDETWO takes care of converting (5) into a (huge) system of ODEs by approximating the spatial derivatives by finite differences taking into account the boundary conditions and subsequently calls a GEAR (implicit) method or Adams (explicit) method for time integration. Such an approach is usually called method of lines. Probably the biggest restrictions are the rectangular domain, the absence of mixed derivatives like $\partial_x \partial_y u$, and the lack of support for stationary (i.e. time independent) problems. We now set up a MEX interface to PDETWO and illustrate some of the power and versatility of PDETWO by some examples.

3.2 The gateway

With NPDE= p , finite differences based on a spatial mesh $(x_i, y_j)_{i=1, \dots, nx, j=1, \dots, ny}$ of R convert the NPDE PDEs (5) into NODE=NPDE*NX*NY coupled ODEs. The user of PDETWO calls the driver routine

DRIVEP(NODE, TO, H, U, TOUT, EPS, MF, INDEX, WORK, IWORK, X, Y), where

¹⁷thus, PDETWO can be considered as a 2D version of the built in Matlab solver `pdepe`

X,Y are the vectors $x_i, i = 1, \dots, nx, y_j, j = 1, \dots, ny$,
U is the array $U(NPDE, NX, NY)$ of u -values at the meshpoints
TO, TOUT are initial and (desired) final time,
H, EPS are the initial stepsize and the local error,
MF is a switch for the integration method (see below),
INDEX is used as startup and error indicator,
WORK is real workspace, and
IWORK is integer workspace also used to pass arguments like $NPDE, NX, NY$.

Additionally, the user must provide subroutines **BNDRYV**, **BNDRYH**, **DIFFH**, **DIFFV**, **F** which as explained above specify the boundary conditions, the diffusion-fluxes, and f from (5). On exit, if integration was successful **U** contains the solution at **TOUT**.

Our goal is to call **DRIVEP** from **Matlab**, where we want to pass **t0**, **tout**, **h**, **epsi**, **mf**, **index**, **x**, **y**, **u**, and parameters for **F**, **BNDRYV**, etc. On return we are naturally interested in $U=U(TOUT)$, but for reasons explained below we also return **INDEX** and **H** (the last stepsize used). Thus the call of the **MEX** routine driving **DRIVEP** shall read

`[u,index,h]=*step(t0,tout,h,epsi,mf,index,x,y,u,para,npara)`

where ***** will be replaced by a suitable name for the PDE we want to integrate.

Remark 3.1 Remark 2.1 also applies here. Since we do not pass function handles to user provided **Matlab** functions **F**, **BNDRH**, **BNDRV**, **DIFFH**, **DIFFV**, for each PDE we need an individual **MEX** file which must be compiled. Therefore we also refrain from passing workspace (or at least dimensions $NPDE, NX, NY$) to ***step**. See [Wea06] for a nice **octave** implementation of a gateway to **PDETWO** (and also to the related routine **PDEONE**) which does it better.]

3.3 Four examples

3.3.1 A Reaction–Diffusion system

Reaction–Diffusion (RD) systems are PDE of the form

$$\partial_t u = D(u)\Delta u + f(u), \quad u = u(t, x) \in \mathbb{R}^p, x \in \Omega \subset \mathbb{R}^d, t \geq 0, \quad (8)$$

where $D(u) \in \mathbb{R}^{p \times p}$ is a positive definite matrix called diffusion matrix. Naturally, (8) must be completed with an initial condition (IC) $u|_{t=0} = u_0$ and with boundary conditions (BC). Reaction–Diffusion systems appear throughout Chemistry [MK98, EE95], Biology [Mur89] and many other branches of science, and (thus) also play a major role in PDE theory [Smo94]. For $\Omega = R$ a rectangle and typical forms of BCs

we may use PDETWO to numerically approximate solutions, and in fact we regard solving RD systems as one of the main applications of PDETWO.

A typical example is the Schnakenberg model [Sch79]

$$\begin{aligned}\partial_t u_1 &= \Delta u_1 - u_1 + u_1^2 u_2, \\ \partial_t u_2 &= d\Delta u_2 + b - u_1^2 u_2,\end{aligned}\tag{9}$$

where u_1, u_2 denote the concentrations of two chemical species, and $b, d > 0$ are parameters. This is a prototype for a reaction diffusion system with a Turing instability, which means that for some parameter regions (9) develops some (roughly) spatially periodic patterns. Often, animal coat patterns are related to Turing instabilities.

For simplicity, as boundary conditions for (9) we choose Neumann in both components, i.e.

$$\partial_n u_1 = 0 \text{ and } \partial_n u_2 = 0 \text{ on all four boundaries of } R,\tag{10}$$

where n denotes the outer normal to R . Note that much more general BC would be allowed by PDETWO.

The stripped MEX file for (9),(10) reads (see `schnakstep.F`)

```

1 #include "fintrf.h"
2     subroutine mexFunction(nlhs, plhs, nrhs, prhs) ! Gateway to DRIVEP
3     IMPLICIT DOUBLE PRECISION (A-H,O-Z)
4     parameter (nx=100, ny=100, npde=2, nwork=20000000)
5     dimension X(nx),Y(ny),U(npde,nx,ny), WORK(nwork), para(100)
6     integer IWORK(100000)
7     ... more declarations and preprocessing of data
8     CALL DRIVEP(NODE,TO,H,U,TOUT,EPS,MF,INDEX,WORK,IWORK,X,Y)
9     ... postprocessing
10    return; end
11
12    SUBROUTINE BNDRYH (T,X,Y,U,AH,BH,CH,NPDE) ! HORIZONTAL BC
13    IMPLICIT DOUBLE PRECISION (A-H,O-Z)
14    DIMENSION U(NPDE),AH(NPDE),BH(NPDE),CH(NPDE)
15    AH(1)=0.0; BH(1)=1.0; CH(1)=0; AH(2)=0.0; BH(2)=1.0; CH(2)=0
16    RETURN; END
17
18    SUBROUTINE BNDRYV (T,X,Y,U,AV,BV,CV,NPDE) ! VERTICAL BC
19    IMPLICIT DOUBLE PRECISION (A-H,O-Z)
20    DIMENSION U(NPDE),AV(NPDE),BV(NPDE),CV(NPDE)
21    AV(1)=0.0; BV(1)=1.0; CV(1)=0; AV(2)=0.0; BV(2)=1.0; CV(2)=0
22    RETURN; END
23
24    SUBROUTINE DIFFH (T,X,Y,U,DH,NPDE) ! HORIZONTAL DIFFUSION COEFFICIENTS
25    IMPLICIT DOUBLE PRECISION (A-H,O-Z)
26    DIMENSION U(NPDE),DH(NPDE,NPDE)
27    DH(1,1)=1; DH(1,2)=0; DH(2,1)=0.0; DH(2,2)=1.0;
28    RETURN; END

```

```

29
30      SUBROUTINE DIFFV(T,X,Y,U,DV,NPDE) ! VERTICAL DIFFUSION COEFFICIENTS
31      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
32      DIMENSION U(NPDE),DV(NPDE,NPDE)
33      DV(1,1)=1; DV(1,2)=0; DV(2,1)=0.0; DV(2,2)=1.0;
34      RETURN; END
35
36      SUBROUTINE F(T,X,Y,Uv,UX,UY,DUXX,DUYY,DUdT,NPDE) !PDE
37      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
38      DIMENSION Uv(NPDE),UX(NPDE),UY(NPDE),DUXX(NPDE,NPDE),
39      * DUYY(NPDE,NPDE),DUdT(NPDE), para(100)
40      common /hupara/ para
41      d=para(1); b=para(2); u=uv(1);v=uv(2);
42      dlapu=DUXX(1,1)+DUYY(1,1); dlapv=DUXX(2,2)+DUYY(2,2);
43      DUDT(1)=dlapu-u*u**2*v; DUDT(2)=d*dlapv+b-u**2*v;
44      RETURN; END

```

In line 4 we first define parameters `nx=100`, `ny=100`, `npde=2`, with obvious meanings. *These must have the same values as the fields `x,y,u` used to call `schnakstep(...)`.* Next we also define the parameter `nwork` describing the size of real workspace for PDETWO. At least for `MF=22` (see below for the meaning of this parameter), PDETWO needs quite a lot of workspace (on the order of `NODE*NX`, see `pde2d.f` for details), thus we are rather generous here. Line 5 then dimensionalizes the inputs and workspace, and similarly in line 6 we prepare necessary integer workspace for PDETWO. The remainder of the gateway routine consist of extracting and preprocessing (including an error check this time) the data from `prhs`, calling `DRIVEP` and postprocessing, see `schnakstep.F` for details. Lines 12 to 44 contain the routines `BNDRYH`, `BNDRYV`, `DIFFH`, `DIFFV` and `F` describing (9),(10) in PDETWO syntax.

A very basic `Matlab` file (see `schnakdr.m` for a more elaborate version) calling `schnakstep` is

```

1  npde=2; nx=100; ny=100; u=zeros(npde,nx,ny); lx=50; ly=lx; t=0;
2  x=linspace(-lx,lx,nx);y=linspace(-ly,ly,ny); [X,Y]=meshgrid(x,y);
3  X=X'; Y=Y'; % transpose, since u=u(nx,ny)
4  bp=3; dp=60;
5  u(2, :, :)=1/bp+0*X; yp1=-0.4*ly; yp2=-0.6*ly; p1=0.01; p2=0.02;
6  u(1, :, :)=bp+0.1*((((X+0.5*lx).^2+(Y-0.4*ly).^2)<30)...
7      +(((X-0.5*lx).^2+(Y-0.4*ly).^2)<30)+(Y>yp2+p2*X.^2 & Y<yp1+p1*X.^2));
8  para=zeros(1,100); para(1:2)=[dp, bp]; npara=2; % parameters
9  t0=0; t1=50; epsi=0.0000001; h=0.00001; mf=22; index=1;
10 tic;[u, index, h]=schnakstep(t0,t1,h,epsi,mf,index,x,y,u,para,npara); toc,h
11 u1=squeeze(u(1, :, :)); u2=squeeze(u(2, :, :));
12 figure(1);pcolor(X,Y,u1);figure(2);pcolor(X,Y,u2);
13 dt=t1-t0;dt=asknu('dt (0 for end)',dt); isteps=10; % start integration loop
14 while(dt>0);
15     ic=0; isteps=asknu('isteps (0 for end)',isteps); % user control
16     while(ic<isteps)
17         t0=t1; t1=t0+dt; ic=ic+1;

```

```

18     tic; [u, index, h]=schnakstep(t0,t1,h,epsi,mf,index,x,y,u,para,npara); toc
19     u1=squeeze(u(1,:,:)); figure(1);clf;pcolor(X,Y,u1);
20     u2=squeeze(u(2,:,:)); figure(2);clf;pcolor(X,Y,u2);
21     end;
22     dt=asknu('dt (0 for end)',dt);
23 end

```

Lines 1 and 2 are obvious (grid) preparations. The reason for line 3 is that X, Y produced by $[X, Y]=\text{meshgrid}(x, y)$ have dimensions $n_y \times n_x$, which corresponds to the mathematical notion of a $n_y \times n_x$ matrix, while PDETWO expects U to be dimensionalized as $U(\text{npde}, n_x, n_y)$ (and therefore $u=\text{zeros}(\text{npde}, n_x, n_y)$ in line 1). Thus before filling u with initial data using the (convenient) $\text{meshgrid } X, Y$ we need to first transpose X and Y . Note that we use a rather coarse uniform grid, in fact $x_{i+1}-x_i = y_{j+1}-y_j = 1.01$; this is justified since from some theoretical considerations it is easy to know in advance that the patterns which emerge must have a wavelength of about 4π .

Line 4 sets the PDE parameters, and lines 5 to 7 fill the initial conditions. We choose $v \equiv 1/b$ while for u we do something more interesting – can you guess from the code? Starting in line 8 we prepare the call to `schnakstep`, and in lines 10 to 13 we call `schnakstep` and plot the result.

Lines 14 to 23 contain the (in our view) typical application of PDETWO. We repeatedly call `schnakstep`, as long as we please, with control returning to the user after each `isteps` many steps¹⁸. There the user can inspect the solution and for instance produce figures, or even interrupt the execution to e.g., change parameters etc. In the repeated calls of `schnakstep` the returning of `index` and `h` becomes useful: On input, PDETWO (i.e. `DRIVEP`) uses `index` to indicate if this is a first call (`index=1`) or a successive call (`index=0`). Successive calls need less preprocessing and are therefore faster. Similarly, on input, `h` is the initial stepsize; choosing the final stepsize `h` from the last step usually reduces the costs of startup. Additionally, the size of `h` is interesting for the user since it gives an indication about how fast (`h` small) or slow (`h` large, at least for implicit integration `MF=22`) the dynamics are¹⁹. Here we also finally want to comment on parameter `MF` in line 12.

Remark 3.2 Essentially, there are two options for `MF` in PDETWO:

a) `MF=10` means an explicit time integration. For parabolic PDE (e.g. Reaction Diffusion systems) this has the disadvantage that due to stiffness it requires very small time steps (h on the order of δ^2 , where δ is the meshsize of the spatial discretization). On the other hand, explicit time stepping for each step only requires few function evaluations, only simple algebraic operations, and little workspace.

b) `MF=22` means an implicit time integration. For this, in principle there is no restriction on the size of h , but in each step a linear system must be solved. Therefore, PDETWO requires workspace of order `NODE*NX=NX*NX*NY`. This

¹⁸`asknu` is a helper routine asking the user for input, see `asknu.m`

¹⁹of course similar remarks apply to `SODEX`

may be prohibitive for large N_x, N_y . In particular, if possible, the PDE should be set up in such a way that $N_x \leq N_y$.

For parabolic problems, if possible (concerning workspace) $MF=22$ is expected to be faster; if the dimensions ($npde, nx, ny$) are too large, try $MF=10$. In any case one should make a quick check of speeds with both $MF=10$ and $MF=22$.]

Remark 3.3 All implicit solvers (also `ode15s`, SODEX, etc) can be considerably sped up by providing an analytical Jacobian of f . In PDETWO, use $MF=21$. At least for PDE, however, this usually requires a major programming effort, and is therefore contrary to the spirit of PDETWO (and easy use of software in general).]

Remark 3.4 One can of course also pass other information back to `Matlab`, for instance number of function calls etc; see line 1527 in `pde2d.f`. For simplicity we restrict to `index` and `h`.]

Figure 3 shows the initial condition for u , and the solution at times as noted, indicating the emergence of stripes pattern and convergence to a stationary solution. The execution times/stepsize for integration intervals of size $dt=100$ with $MF=22$ are

time interval	0-100	100-200	...	400-500	...	900-1000	...	1900-2000
runtime (s)	122	34	...	10	...	5	...	4
last stepsize	1	2	...	3	...	9	...	25

With time the stepsize increases as the solution becomes (quasi)stationary, and thus runtime decreases. On the other hand, for the explicit scheme ($MF=10$) we obtain

time interval	0-100	100-200	200-300	300-400	400-500	...
runtime (s)	380	370	340	340	335	...
last stepsize	0.0024	0.0016	0.0117	0.0081	0.0102	...

and this continues until $t = 2000$, i.e., the convergence to a stationary solution does not help speed up the calculations.

Remark 3.5 Problem (9),(10) can also efficiently solved using a semi-implicit pseudo-spectral method, see [Uec09]. For this, here we have to replace the basis functions $e_{kl} = \exp\left(i\left(k\frac{2\pi}{L_x}x + l\frac{2\pi}{L_y}y\right)\right)$ from [Uec09] suitable for periodic boundary conditions by $e_{kl} = \cos\left(k\frac{2\pi}{L_x}(x - L_x) + l\frac{2\pi}{L_y}(y - L_y)\right)$, suitable for Neumann BC, i.e., instead of FFT we have to use a discrete cosine transform `dct2` in `Matlab`. This can be combined with error and stepsize control, and then we may expect competitive execution times with PDETWO.

However, this is only possible since we know the eigenfunctions e_{kl} of the Laplacian under Neumann (or Dirichlet, or periodic) boundary conditions, i.e., in particular due to the constant coefficients. For non-constant coefficient principal part, or for more complicated boundary conditions, spectral methods are not available or become much more complicated, while these variations are no problem for PDETWO.]

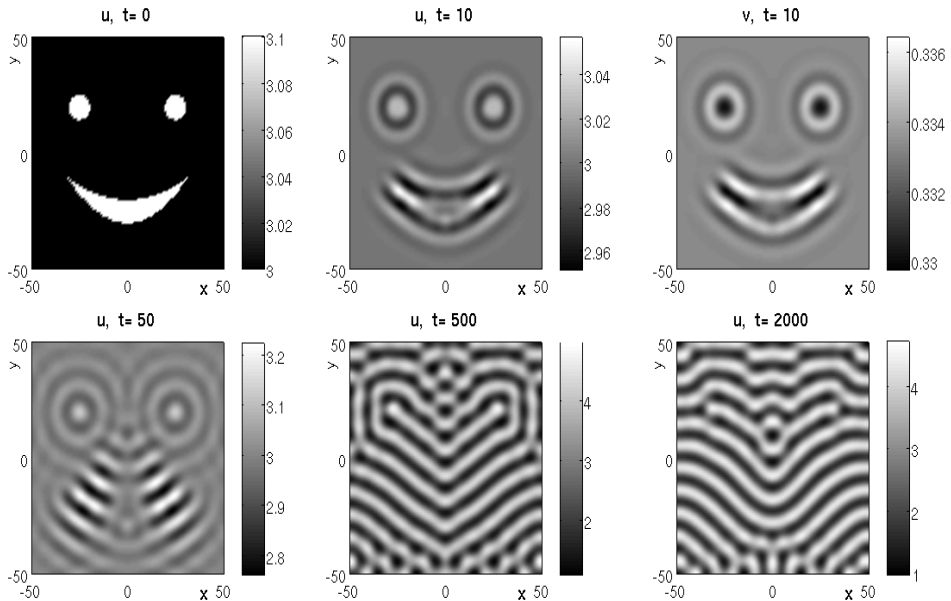


Figure 3: Output of `schnakdr.m`

3.3.2 Nonlinear diffusion: a porous medium equation

So called porous media are described by equations of the form

$$\partial_t u = \frac{1}{m} \Delta(u^m), \quad u = u(t, x, y) \geq 0, \quad (11)$$

where $m \geq 2$ is a parameter. Additional to the obvious differences between (9) and (11) we want to point out the (mathematically) important fact that (11) is quasilinear which means that the highest derivatives (here Δu) appear in a nonlinear way, cf. also (15). Here we use the porous medium equation (11) to illustrate that on the one hand, PDETWO can efficiently integrate such quasilinear equations. On the other hand, using a known exact solution, and by monitoring mass conservation, we illustrate some difficulties of the finite difference method. This shows that while PDETWO can be used to quickly obtain the basic behaviour, some fine properties of solutions can only be recovered using specialized numerics.

The d -dimensional generalization of (11),

$$\partial_t u = \frac{1}{m} \Delta(u^m), \quad x \in \Omega \subset \mathbb{R}^d, \quad (12)$$

has a family of self-similar fundamental solutions, called Barenblatt solutions. Let

$m_c = (d - 2)/d$, $m > m_c$, $m \neq 1$. Then, for arbitrary $D > 0$, $T \geq 0$,

$$U_{D,T}(t, x) = R(t)^{-d} \left(D - \frac{m-1}{2d(m-m_c)} \left| \frac{x}{R(t)} \right|^2 \right)_+^{\frac{1}{m-1}}, \quad R(t) = (T+t)^{\frac{1}{d(m-m_c)}} \quad (13)$$

is an exact solution of (12), where $u_+(x) := \max(0, u(x))$ denotes the positive part. Since U is not differentiable at the interface $\Gamma(t)$ from $U(t, \cdot) \equiv 0$ to $U(t, \cdot) > 0$, these are in fact only so called weak solutions.

For instance, for $d = 2$ and $D = T = 1$ we have the exact solution

$$U(t, x) = (1+t)^{-1/m} \left(1 - \frac{m-1}{4m} |x|^2 / (1+t)^{1/m} \right)_+^{1/(m-1)}, \quad (14)$$

which we shall use to validate the results of PDETWO. Therefore, we use $U(0, x)$ as IC for (11) on $R = [-L_x, L_x] \times [-L_y, L_y]$, with Dirichlet BC $u|_{\partial R} = 0$. Note that under these BC, the mass $M(t) = \int_R u \, dx$ is always conserved, i.e., by Gauss theorem,

$$\partial_t M = \frac{1}{m} \Delta(u^m) \, dx = \int_{\partial R} \nabla(u^m) \cdot n \, d\Gamma = 0.$$

Whenever such a conserved quantity is known, it should be monitored to evaluate the performance of a numerical scheme.

To put (11) into PDETWO we note that it is equivalent to

$$\partial_t u = u^{m-1} \Delta u + (m-1) u^{m-2} |\nabla u|^2 =: f(u, u_x, u_y, u_{xx}, u_{yy}; m). \quad (15)$$

Now consider `pmstep.F`. In the gateway routine we only replace `parameter (nx=100, ny=100, npde=2)` by `parameter (nx=50, ny=50, npde=1)` since these are the (n_x, n_y) values we want to use for this problem. Next, put `AV(1)=1.0; BV(1)=0.0; CV(1)=0` resp. `AH(1)=1.0; BH(1)=0.0; CH(1)=0` into `BNDRYV`, `BNDRYH` to encode Dirichlet boundary conditions, and `DH(1,1)=1.0`; resp `DH(1,1)=1.0`; into `DIFFV`, `DIFFH`. To encode f from (15) we finally let

```

1      SUBROUTINE F(T,X,Y,U,UX,UY,DUXX,DUYY,DU DT, NPDE) !PDE
2      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
3      DIMENSION U(NPDE), UX(NPDE), UY(NPDE), DUXX(NPDE, NPDE),
4      * DUYY(NPDE, NPDE), DU DT(NPDE), para(100)
5      common /hupara/ para
6      dm=para(1); ulap=DUXX(1,1)+DUYY(1,1);
7      DU DT(1)=u(1)**(dm-1)*ulap
8      1 +(dm-1)*u(1)**(dm-2)*(ux(1)**2+uy(1)**2)
9      RETURN; END
    
```

That's all. A simple Matlab driver is `pmldr.m`, which is an obvious modification of `schnakdr.m`. Execution speed is non-critical here, everything is very fast. Some simulation results are shown in Fig.4. The top row shows the initial condition and

the solution at time $t = 10$. The error plot ($u_{\text{num}}(10, \cdot) - U(10, \cdot)$ with U from (14)) on the bottom left shows that the numerical solution u decays somewhat too quickly, in particular at the interface from $u \equiv 0$ to $u > 0$. Of course, this interface, where u is not smooth, must be expected to cause problems for any method which does not deal with it in a special way. In particular, the error at the interface does not decrease if we increase, e.g., \mathbf{nx}, \mathbf{ny} . Finally, related to this is that the mass $\langle u(t) \rangle = \int_R u(t, x) \, dx$ is not well conserved (bottom right), in particular not initially.

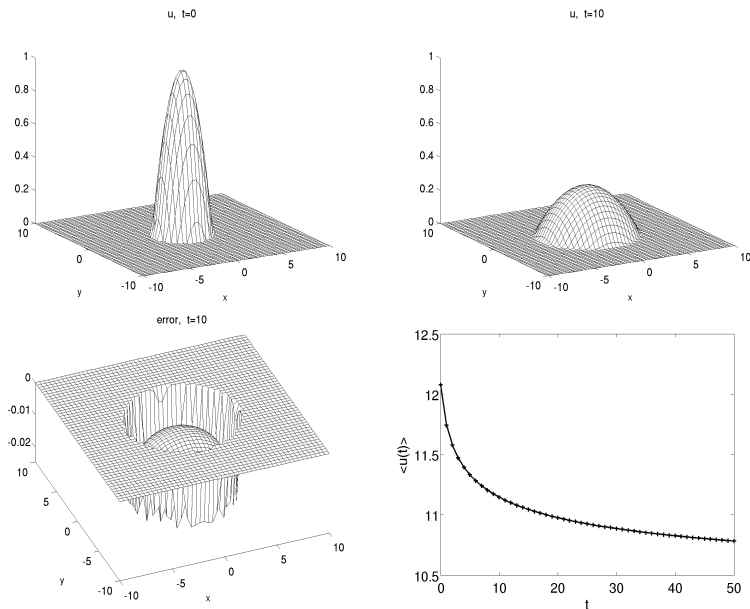


Figure 4: Output of pmdr.m

3.3.3 A wave equation

The wave equation

$$\partial_t^2 w + a \Delta w = 0, \quad w = w(t, x) \in \mathbb{R}, \quad t \in \mathbb{R}, \quad x \in R = [-L_x, L_x] \times [-L_y, L_y], \quad (16)$$

$a > 0$ a parameter, and its variants are examples of a so called hyperbolic PDE. Here we briefly show how to solve (16) with PDETWO, taking the example of an (x, y) dependent a , namely $R = [-10, 10] \times [-10, 10]$ and

$$a(x) = \begin{cases} 1 & x \in R \setminus \Omega \\ a_0 & x \in \Omega = [0, 1] \times [-3, 3]. \end{cases} \quad (17)$$

Moreover, we again take Neumann boundary conditions, i.e., $\partial_n w|_{\partial R} = 0$ where n is the outer normal to R . This may be used as a model to study the scattering of waves at the *obstacle* Ω .

To have the form required by PDETWO we rewrite (16) as a first order system for $u = (u_1, u_2) = (w, \partial_t w)$, namely

$$\partial_t \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} u_2 \\ a(x)\Delta u_1 \end{pmatrix}, \quad \partial_n u_1 = 0 \text{ and } \partial_n u_2 = 0 \text{ on } \partial R. \quad (18)$$

As initial conditions we want to use $(u, u_t)|_{t=0} \equiv (0, 0)$ except for a small pulse in u located left of the obstacle.

Now copy `schnakstep.F` to `wavestep.F`. In the gateway routine we only replace `parameter (nx=100, ny=100, ...)` by `parameter (nx=200, ny=200, ...)` since these are the values we want to use for this problem. Next, the routines `BNDRYV`, `BNDRYH` (since again we use Neumann BC) and `DIFFV`, `DIFFH` may stay as before, we only need to replace lines 41 to 50 by

```

1      SUBROUTINE F(T,X,Y,U,UX,UY,DUXX,DUYY,DU DT,NPDE) !PDE
2      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
3      DIMENSION U(NPDE),UX(NPDE),UY(NPDE),DUXX(NPDE,NPDE),
4      * DUYY(NPDE,NPDE),DU DT(NPDE), para(100)
5      common /hupara/ para
6      a=1;if((x.gt.0).and.(x.lt.1).and.(y.gt.-3).and.(y.lt.3)) a=para(1);
7      DU DT(1)=u(2); DU DT(2)=a*(DUXX(1,1)+DUYY(1,1))
8      RETURN; END

```

Line 6 encodes the obstacle Ω , where a_0 is passed to `F` via the common `hupara`. After calling `mex wavestep.F pde2d.o`, a simple matlab driver looks very much like `schnakdr.m`, see `wavedr.m`. Figure 5 shows the output. Here we use `MF=10`. In fact, with `nwork=2000000` in `wavestep.F` we cannot use `MF=22` since then we need `nwork ≥ 97122838`. However, for hyperbolic problems explicit schemes are typically superior to implicit schemes. Execution speed, anyway, is not an issue: we need about 0.5s for a `dt=1` step.

At this point the reader should start setting up his or her favorite PDE; however, for the following exercises we again provide solutions at [Uec11]. Below, the interested reader can also find one more example.

Exercise 3.6 Set up the following PDE for PDETWO.

a) Gray–Scott, namely

$$\begin{aligned} \partial_t u &= d_1 \partial_x^2 u - uv^2 + f(1 - u), \\ \partial_t v &= d_2 \partial_x^2 v + uv^2 - (f + k)v, \end{aligned} \quad (19)$$

$(x, y) \in R = (-L_x, L_x) \times (L_y, L_y)$, $\partial_n u = \partial_n v = 0$ on ∂R , with typically $0 < d_2 < d_1$ in applications, parameters f, k , and $(x, y) \in (-L_x, L_x) \times (L_y, L_y)$. Study self replicating

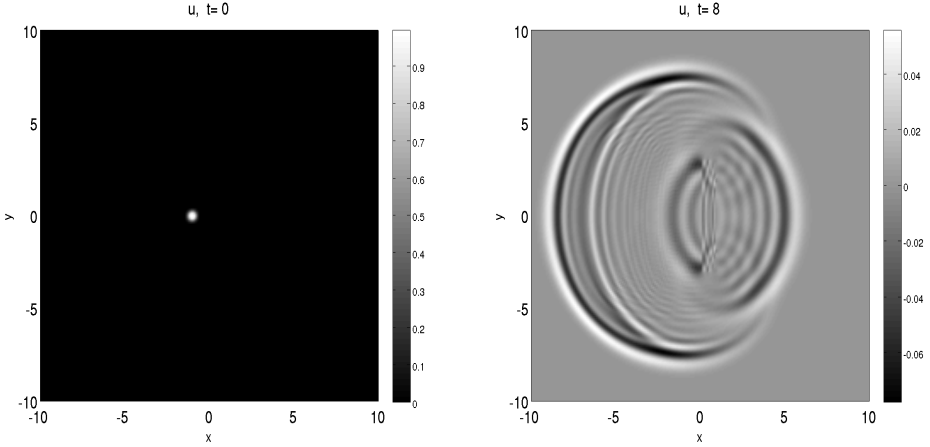


Figure 5: Output of `wavedr.m`

patterns for $(d_1, d_2)=(1, 0.5)$, $(f, k)=(0.038, 0.06)$. Compare to [Uec09, Exercise 3.18].
b) The 2D Nonlinear Schrödinger equation (NLS) [SS99] for a *complex* amplitude u reads

$$\partial_t u = i(\Delta u + \alpha|u|^2 u), \quad (20)$$

where $\alpha = +1$ is called the focussing and $\alpha = -1$ is called the defocussing case. Set $u = u_1 + iu_2$, rewrite (20) as a real system for (u_1, u_2) , and simulate with “pulse-like initial conditions” and Dirichlet or Neumann boundary conditions.

c) A three component RD system describing traveling interface modulations is [RLK⁺11]

$$\partial_t u = u - u^3 - v - \delta(u - u_s)q^2 + d_u \Delta u + d_{uq} \Delta q + \eta \partial_x u, \quad (21a)$$

$$\partial_t v = \varepsilon(u + \beta - v) + d_v \Delta v + \eta \partial_x v, \quad (21b)$$

$$\partial_t q = (1 - q)(q - a)(q + 1) + \gamma(1 - q^2)(u - u_s) + d_{uq} \Delta u + \Delta q + \eta \partial_x q, \quad (21c)$$

with diffusion constants, $d_u, d_{uq}, d_v > 0$, parameters $\beta, \gamma, \delta, \eta \in \mathbb{R}$, $\varepsilon > 0$ and $-1 < a < 1$. For the IC consider wedges in q ,

$$q_0(x, y) = \begin{cases} -1 & x < x_0 - m|y| \\ 1 & x \geq x_0 - m|y| \end{cases},$$

where $\pm m \in \mathbb{R}$ are the slopes of the sides and $x_0 \in \mathbb{R}$ is the position of the tip. For (u, v) choose $(u_0, v_0) = (u_s, v_s)$ where $u_s = -\beta^{1/3}$, $v_s = u_s + \beta$. For the BCs choose

$$(u, v)|_{\partial\Omega} = (u_s, v_s) \quad \text{and} \quad q = \pm 1 \quad \text{on} \quad x = \pm L, \quad q(x, \pm L) = q_f(x - x_1),$$

where $q_f(x) = \tanh(x/\sqrt{2})$. As a starting point, choose domain size $L = 15\pi$ and parameters

$$\begin{aligned} d_u &= 0.09, d_v = 0.01, d_q = 1, d_{uq} = 0.1, \beta = 0.2, \delta = 0.5, \varepsilon = 0.03, \\ \gamma &= -0.05, a = -0.1, \eta = -0.15, x_0 = L/4, m = 1, x_1 = -3L/4. \end{aligned}$$

Fig. 6 shows some plots you should get. Examine, e.g., the role of γ .]

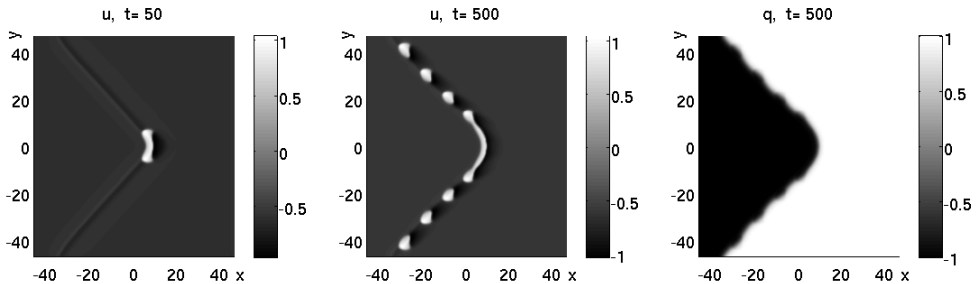


Figure 6: Traveling interface excitations in (21)

3.3.4 Some shallow water equations

Our final example is somewhat more elaborate. We study a shallow water system (with a viscous regularization), which describes flows with a free surface in situations where the horizontal dimensions of the flow are much larger than the vertical dimension. Roughly, we consider water in a channel $(x, y) \in R = [-L_x, L_x] \times [-L_y, L_y]$ with bottom topography described by a function $b(x, y)$. The unknowns are the fluid depth h and some averaged flows u, v , and our shallow water equations are

$$\partial_t h = -\partial_x(hu) - \partial_y(hv) + \nu_1 \Delta h, \quad (22a)$$

$$\partial_t u = -u\partial_x u - v\partial_y u - (\partial_x h + \partial_x b) + \nu_2 \Delta u, \quad (22b)$$

$$\partial_t v = -u\partial_x v - v\partial_y v - (\partial_y h + \partial_y b) + \nu_2 \Delta v. \quad (22c)$$

For $\nu_1, \nu_2 = 0$ these are the standard shallow water equations, but we added some small dissipation to the right hand side to avoid so called shocks, which need special numerical treatment. In particular the term $\nu_1 \Delta h$ in (22a) is rather unphysical; see, e.g., [BDM07], for more physical so called viscous shallow water equation.

However, our goal here is to illustrate once more that it is easy and fun to simulate (22) with PDETWO, and thus to obtain some interesting first insight into (22) also with $\nu_1, \nu_2 = 0$.

We consider water flowing mainly in x -direction. If the bottom is flat, then (22) under suitable boundary conditions has stationary solutions of the form

$$(h, u, v) = (h_s, u_s, 0),$$

where h_s, u_s are a reference water depth and “streamwise” flow. We now want to consider the situation where such a flow is influenced by some obstacles at the bottom (“stones in the river bed”). Therefore, as boundary conditions we choose

$$\begin{aligned} h &= h_s \text{ and } u = u_s \text{ at in- and outflow, i.e. at } x = \pm L_x, \\ \partial_n h &= 0 \text{ and } \partial_n u = 0 \text{ at the lateral walls, i.e. at } y = \pm L_y, \\ \partial_n v &= 0 \text{ at in- and outflow, and } v = 0 \text{ at the lateral walls.} \end{aligned} \quad (23)$$

A simple obstacle in an otherwise flat bottom might for instance be of the form

$$b(x, y) = \max(0, p(x, y)), \text{ where } p(x, y) := p_h \left(1 - \frac{(x - x_p)^2 + (y - y_p)^2}{p_w^2} \right). \quad (24)$$

This describes a “spike” in the river bed, with parameters p_h, x_p, y_p, p_w , corresponding to spike height, x -position, y -position, and width. Of course we might also consider “holes” of the form $b(x, y) = \min(0, -p(x, y))$, or several of such spikes, or combinations with holes, or whatever the problem requires or our imagination comes up with.

Finally, as initial conditions we choose

$$h|_{t=0} = h_s - b(x, y), \quad u|_{t=0} \equiv u_s, \quad v|_{t=0} \equiv 0, \quad (25)$$

which as pointed out above is a stationary solution if $b(x, y) = 0$. Physically, for $b(x, y) \neq \text{const}$, this might be thought of as the obstacles from (24) appearing at $t = 0$ instantaneously (without displacing the surface) in a laminar stationary flow.

We now simulate (22)–(24) using PDETWO, where for the code, although it is again not long, we refer to `swdr.m` and `swstep.F`. We use

$$L_x = 5, L_y = 3, h_s = 1, u_s = 1, \nu_1 = 0.005, \nu_2 = 0.05,$$

grid resolutions `nx=250`, `ny=100`, and first put one spike in the middle of the domain with height and width equal to 0.5 ($p_x = 0, p_y = 0, p_h = 0.5, p_w = 0.5$ in (24)). Execution times are a few seconds for time steps `dt=1` (`MF=10`), and the results are shown in Fig. 7. The top row shows the bottom contour and the free surface positions $b(x, y) + h(t, x, y)$, while the bottom row shows the streamwise flow u . Initially, a so called hydraulic jump develops locally over the spike: in particular, h strongly decreases on the lee side while u strongly increases. With time, this hydraulic jump spreads first over the full channel width $y \in [-L_y, L_y]$, and then also in x , creating a so called undular bore also upstream.

Just for fun, in Fig. 8 we put a spike near a wall ($(p_x, p_y, p_h, p_w) = (-1, -3, 0.5, 0.5)$) and a “hole” at $(x, y) = (1, 1)$ (with $(p_h, p_w) = (0.4, 0.4)$). We finally remark, that the flow fields (u, v) can be conveniently plotted using `quiver` in `Matlab`. For details we again refer to `swdr.m` and `swstep.F`. Finally we remark that (22)–(23) again has a conserved quantity, namely the mass $\int_R h(t, x, y) d(x, y)$, which is rather well conserved numerically by PDETWO.

Exercise 3.7 See what happens if you set ν_1 or ν_2 or both to zero. Also toy around with u_s, h_s , and of course examine different bottom topographies.]

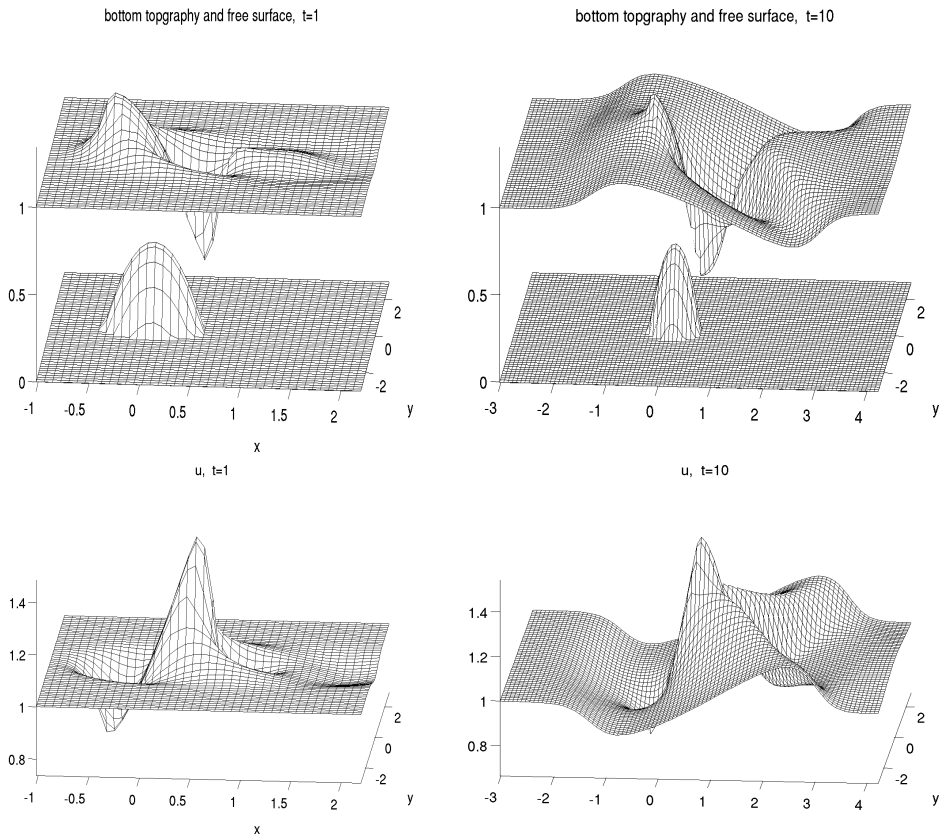


Figure 7: Flow over one spike; only part of the domain is shown, and only every fourth grid point is used for plotting.

4 Summary

There are plenty of numerical libraries around. Additional to the above examples and the collection at [Hai11]²⁰, we like to mention the free packages ODEPACK²¹, the GSL (GNU Scientific Library), and the packages at Netlib²². Numerical Recipes (F, C, C++) is available at most universities, and often also the commercial libraries IMSL, NAG as well. This list is by no means meant to be exhaustive.

As *Matlab* offers a convenient way to interface such libraries, using them may save a lot of both programming and execution time. Finally, we believe that it is not necessary to write too elaborate MEX files, unless one wants to contribute a “standalone *Matlab* routine”, which hides all MEXing from the user. Here we rather

²⁰with some (C) MEX files at www-m3.ma.tum.de/Software/ODEHome

²¹<https://computation.llnl.gov/casc/software.html>

²²<http://netlib.sandia.gov/master/index.html>

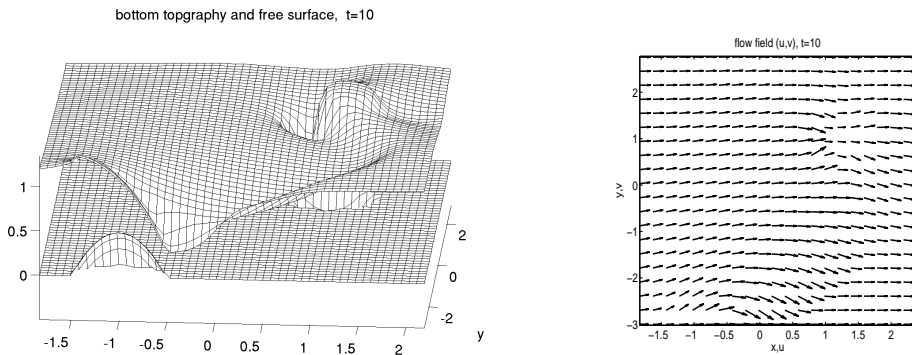


Figure 8: Flow over a spike and a hole; only part of the domain is shown, and only every fourth grid point is used for plotting. For the vectorfield plot, the v component is 5 times magnified.

opted for easy gateways from the programming point of view (and which should be faster concerning runtime), where the user does some editing of the MEX source code, cf. Remarks 2.1 and 3.1.

References

- [BDM07] D. Bresch, B. Desjardins, and G. Metivier. Recent mathematical results and open problems about shallow water equations. In *Analysis and simulation of fluid dynamics*. Birkhäuser, 2007.
- [Coo98] J. Cooper. *Introduction to partial differential equations with MATLAB*. Birkhäuser, 1998.
- [EE95] M. Eiswirth and G. Ertl. Pattern formation on catalytic surfaces. In R. Kapral and K. Showalter, editors, *Chemical Waves and Patterns*, pages 447–483. Kluwer, Dordrecht, 1995.
- [Eva98] L. C. Evans. *Partial differential equations*, volume 19 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI, 1998.
- [Hai11] E. Hairer. Software for ODEs: www.unige.ch/~hairer/software.html, 2011.
- [HNW93] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving ordinary differential equations. I*. Springer, 1993.
- [HW96] E. Hairer and G. Wanner. *Solving ordinary differential equations. II: Stiff and differential-algebraic problems*. Springer, 1996.
- [Kee88] J. P. Keener. *Principles of applied mathematics*. Addison-Wesley, 1988.

- [Lyn04] S. Lynch. *Dynamical systems with applications using MATLAB*. Birkhäuser, 2004.
- [MK98] H. Mori and Y. Kuramoto. *Dissipative structures and chaos*. Springer, Berlin, 1998.
- [MS81a] D. K. Melgaard and R. F. Sincovec. Algorithm 565: PDETWO / PSETM / GEARB: Solution of systems of two-dimensional nonlinear partial differential equations. *ACM Transactions on Mathematical Software (TOMS)*, 7(1):126–135, 1981.
- [MS81b] D. K. Melgaard and R. F. Sincovec. General software for two-dimensional nonlinear partial differential equations. *ACM Transactions on Mathematical Software (TOMS)*, 7(1):106–125, 1981.
- [Mur89] J. D. Murray. *Mathematical Biology*. Springer, Berlin, 1989.
- [RLK⁺11] M. Rafti, F. Lovis, V. Krupennikova, R. Imbihl, and H. Uecker. A new type of traveling interface modulations in a catalytic surface reaction, Preprint, 2011.
- [Rob01] J. C. Robinson. *Infinite-dimensional dynamical systems*. Cambridge University Press, 2001.
- [Sch79] J. Schnakenberg. Simple chemical reaction systems with limit cycle behaviour. *J. Theoret. Biol.*, 81(3):389–400, 1979.
- [Smo94] J. Smoller. *Shock waves and reaction-diffusion equations*. Springer, New York, 1994.
- [SS99] C. Sulem and P.-L. Sulem. *The nonlinear Schrödinger equation*. Springer-Verlag, New York, 1999.
- [Str92] W. A. Strauss. *Partial differential equations – An Introduction*. John Wiley & Sons Inc., New York, 1992.
- [Str94] S. H. Strogatz. *Nonlinear Dynamics and Chaos*. Perseus, New York, 1994.
- [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.
- [Uec11] H. Uecker. Software for MCS 2011, Oldenburg, `mcs11-husoft.tar` at www.staff.uni-oldenburg.de/hannes.uecker/soft.html; solutions in `mcs11-husol.tar`, 2011.
- [Wea06] J. L. Weatherwax. Software for solving PDEs with Octave, PDETWOG: An octave gateway routine to `pdetwo.f`, web.mit.edu/wax/www/software/code/pdetwo/doc/pdetwo.html, 2006.