# Equation-Free function toolbox for Matlab/Octave

A. J. Roberts* et al.†

October 2, 2018

**Abstract**

This 'equation-free toolbox' facilitates the computer-assisted analysis of complex, multiscale systems. Its aim is to enable microscopic simulators to perform system level tasks and analysis. The methodology bypasses the derivation of macroscopic evolution equations by using only short bursts of microscale simulations which are often the best available description of a system (Kevrekidis & Samaey 2009, Kevrekidis et al. 2004, 2003, e.g.). This suite of functions should empower users to start implementing such methods—but so far we have only just started.

## Contents



*http://www.maths.adelaide.edu.au/anthony.roberts, http://orcid.org/0000-0001-8930-1552

†Be the first to appear here for your contribution.

# 1 Introduction

*Section contents*



**Users** Place this toolbox's folder in a path searched by MATLAB/Octave. Then read the subsection that documents the function of interest.

**Blackbox scenario** Assume that a researcher/practitioner has a detailed and *trustworthy* computational simulation of some problem of interest. The simulation may be written in terms of micro-positional coordinates $\vec{x}_i(t)$ in 'space' at which there are micro-field variable values $\vec{u}_i(t)$ for indices $i$ in some (large) set of integers and for time $t$. In lattice problems the positions $\vec{x}_i$ would be fixed in time (unless employing a moving mesh on the microscale); in particle problems the positions would evolve. The positional coordinates are $\vec{x}_i \in \mathbb{R}^d$ where for spatial problems integer $d = 1, 2, 3$, but it may be more when solving for a distribution of velocities, or pore sizes, or trader's beliefs, etc. The mirco-field variables could be in $\mathbb{R}^p$ for any $p = 1, 2, \ldots, \infty$.

Further, assume that the computational simulation is too expensive over all the desired spatial domain $\mathbb{X} \subset \mathbb{R}^d$. Thus we aim a toolbox to simulate only on macroscale distributed patches.

**Contributors** The aim of this project is to collectively develop a MATLAB/Octave toolbox of equation-free algorithms. Initially the algorithms will be simple, and the plan is to subsequently develop more and more capability.

MATLAB appears the obvious choice for a first version since it is widespread, reasonably efficient, supports various parallel modes, and development costs are reasonably low. Further it is built on BLAS and LAPACK so potentially the cache and superscalar CPU are well utilised. Let's develop functions that work for both MATLAB/Octave.

## 1.1 Create, document and test algorithms

For developers to create and document the various functions, we use an idea due to Neil D. Lawrence of the University of Sheffield:

- Each class of toolbox functions is located in separate directories in the repository, say `Dir`.
- Each toolbox function is documented as a separate subsection, with tests and examples as separate subsections.
- For each function, say `fun.m`, create a LaTeX file `Dir/fun.tex` of a section that `\input{Dir/*.m}`s the files of the function-subsection and the test-subsections, Table 1. Each such `Dir/fun.tex` file is to be `\include{}`ed from the main LaTeX file `equationFreeDoc.tex` so that people can most easily work on one section at a time.
- Each function-subsection and test-subsection is to be created as a MATLAB/Octave `Dir/*.m` file, say `Dir/fun.m`, so that users simply invoke the function in MATLAB/Octave as usual by `fun(...)`.

  Some editors may need to be told that `fun.m` is a LaTeX file. For example, TexShop on the Mac requires one to execute in a Terminal

  ```
  defaults write TeXShop OtherTeXExtensions -array-add "m"
  ```

- Table 2 gives the template for the `Dir/*.m` function-subsections. The format for a example/test-subsection is similar.
- Any figures from examples should be generated and then saved for later inclusion with the following (which finally works properly for MATLAB 2017+)

  ```
  set(gcf,'PaperPosition',[0 0 14 10])
  print('-depsc2',filename)
  ```

Table 1: example `Dir/*.tex` file to typeset in the master document a function-subsection, say `fun.m`, and the test/example-subsections.

```
% input *.m files for ... Author, date
%!TEX root = ../equationFreeDoc.tex
\section{...}
\label{sec:...}
\localtableofcontents
introduction...
\input{Dir/fun.m}
\input{Dir/funExample.m}
...
\subsection{To do}
...
```

Table 2: template for a function-subsection `Dir/*.m` file.

```
%Short explanation for users typing "help fun"
%Author, date
%!TEX root = ../equationFreeDoc.tex
%{
\subsection{\texttt{...}: ...}
\label{sec:...}
\localtableofcontents
Summary LaTeX explanation.
\begin{matlab}
%}
function ...
%{
\end{matlab}
Repeated as desired:
LaTeX between end-matlab and begin-matlab
\begin{matlab}
%}
Matlab code between %} and %{
%{
\end{matlab}
Concluding LaTeX before following final line.
%}
```

# 2 Projective integration of deterministic ODEs

*Section contents*



This section provides some good projective integration functions (Gear & Kevrekidis 2003*a*,*b*, Givon et al. 2006, e.g.). The goal is to enable computationally expensive dynamic simulations to be run over long time scales. Perhaps start by looking at Section 2.1 which codes the introductory example of a long time simulation of the Michaelis–Menton multiscale system of differential equations.

**Scenario** When you are interested in a complex system with many interacting parts or agents, you usually are primarily interested in the self-organised emergent macroscale characteristics. Projective integration empowers us to efficiently simulate such long-time emergent dynamics. We suppose you have coded some accurate, fine scale simulation of the interesting system, and call such code a microsolver.

The Projective Integration section of this toolbox consists of several functions. Each function implements over the long-time scale a variant of a standard

numerical method to simulate the emergent dynamics of the complex system. Each function has standardised inputs and outputs.

### Main functions

- Projective Integration by second or fourth order Runge–Kutta, `PIRK2()` and `PIRK4()` respectively. These schemes are suitable for precise simulation of the slow dynamics, provided the time period spanned by an application of the microsolver is not too large.
- Projective Integration with a General solver, `PIG()`. This function enables a Projective Integration implementation of any solver with coarse scale time steps. It does not matter whether the solver is a standard Matlab or Octave algorithm, or one supplied by the user. As explored in later examples, `PIG()` should only be used in very stiff systems.

The above functions share dependence on a user-specified 'microsolver', that accurately simulates some problem of interest.

### Minor functions

- "Constraint-defined manifold computing", `cdmc()`. This helper function, based on the method introduced in **?**, iteratively applies the microsolver and project the output backwards in time. The result is to constrain the fast variables close to the slow manifold, without advancing the current time by the duration of an application of the microsolver. This function can be used to reduce errors related to the simulation length of the microsolver in either the `PIRK` or `PIG` functions. In particular, it enables `PIG()` to be used on problems that are not particularly stiff.
- Black box microsolver generator, `bbgen()`. This simple function takes as input a standard solver with a recommended time step for microscale simulation, and returns a 'black box' microsolver for the Projective Integration functions.

The following sections describe the PIRK2() and PIG() functions in detail, providing an example for each. Descriptions for the minor functions follow, and an example of the use of cdmc(). PIRK4() (which is very similar to PIRK2()) concludes the section.

## 2.1 egPIMM: Example projective integration of Michaelis–Menton kinetics

*Subsection contents*



The Michaelis–Menten enzyme kinetics is expressed as a singularly perturbed system of differential equations for $x(t)$ and $y(t)$:

$$\frac{dx}{dt} = -x + (x + \tfrac{1}{2})y \quad \text{and} \quad \frac{dy}{dt} = \frac{1}{\epsilon}\big[x - (x+1)y\big].$$

As illustrated in Figure 2, the slow variable $x(t)$ evolves on a time scale of one, whereas the fast variable $y(t)$ evolves on a time scale of the small parameter $\epsilon$.

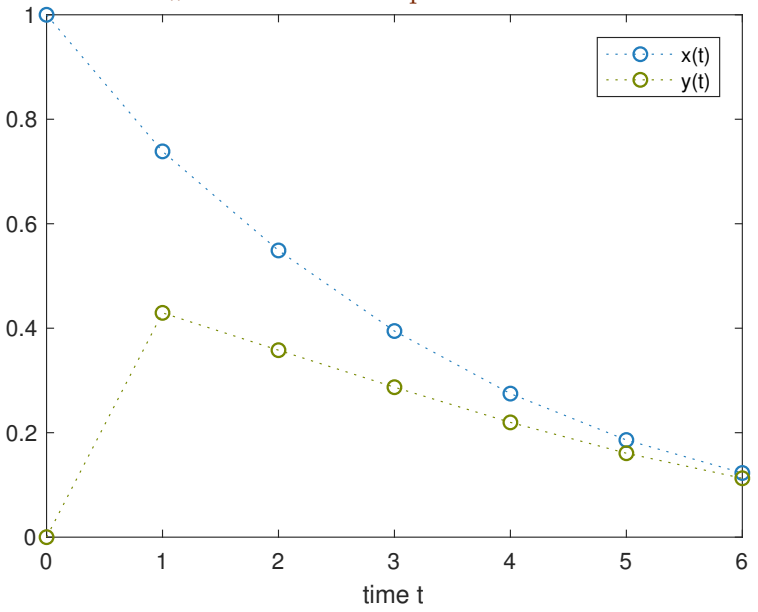
### 2.1.1 Invoke projective integration

Clear, and set the scale separation parameter $\epsilon$ to something small like 0.01. Here use $\epsilon = 0.1$ for clearer graphs.

```
clear all, close all
global epsilon
epsilon = 0.1
```

First, Section 2.1.2 encodes the computation of bursts of the Michaelis–Menten system in a function MMburst(). Second, here set macroscale times of computation and interest into vector ts. Then, invoke Projective Integration with PIRK2() applied to the burst function, say using bursts of simulations

Figure 1: Michaelis–Menten enzyme kinetics simulated with the projective integration of `PIRK2()`: macroscale samples.



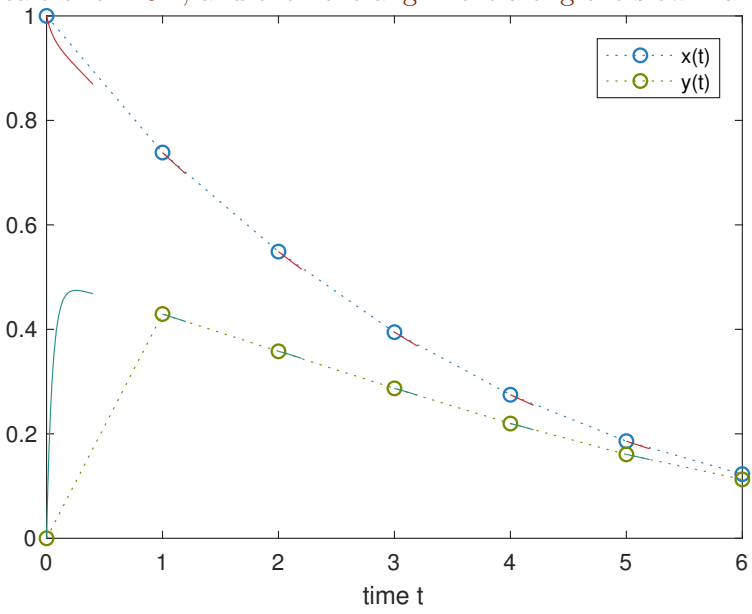

of length $2\epsilon$, and starting from the initial condition for the Michaelis–Menten system of $(x(0), y(0)) = (1, 0)$ (off the slow manifold).

```
ts = 0:6
xs = PIRK2(@MMburst, 2*epsilon, ts, [1;0])
plot(ts,xs,'o:')
xlabel('time t'), legend('x(t)','y(t)')
pause(1)
```

Figure 1 plots the macroscale results showing the long time decay of the Michaelis–Menten system on the slow manifold.

**Optional: request and plot the microscale bursts** Because the initial conditions of the simulation are off the slow manifold, the initial macroscale step appears to 'jump' (Figure 1). To see the initial transient attraction to

Figure 2: Michaelis–Menten enzyme kinetics simulated with the projective integration of `PIRK2()`: the microscale bursts show the initial transients on a time scale of $\epsilon = 0.1$, and then the alignment along the slow manifold.

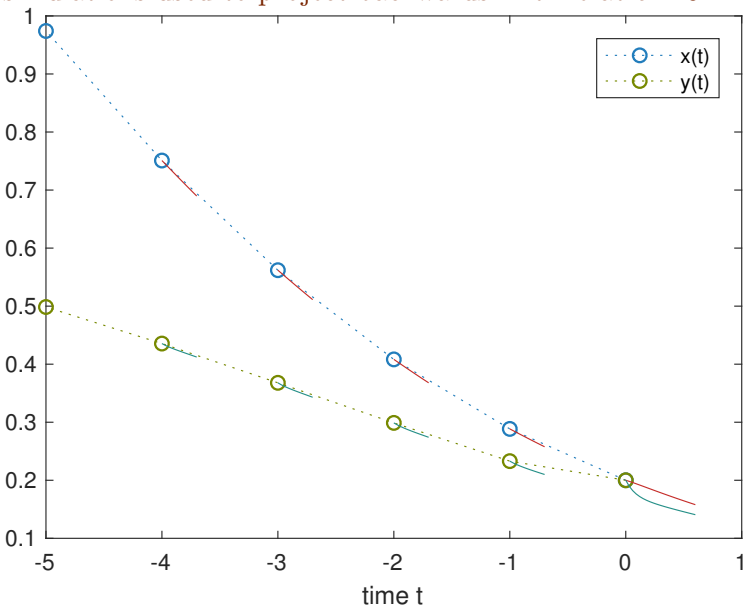



the slow manifold we plot some microscale data in Figure 2. Two further output variables provide this microscale burst information.

```
77 [xs,tMicro,xMicro] = PIRK2(@MMburst, 2*epsilon, ts, [1;0]);
78 figure, plot(ts,xs,'o:',tMicro,xMicro)
79 xlabel('time t'), legend('x(t)','y(t)')
80 pause(1)
```

Figure 2 plots the macroscale and microscale results—also showing that the initial burst is by default twice as long. Observe the slow variable $x(t)$ is also affected by the initial transient which indicates that other schemes which 'freeze' slow variables are less accurate.

Figure 3: Michaelis–Menten enzyme kinetics simulated backwards with the projective integration of `PIRK2()`: the microscale bursts show the short forward simulations used to project backwards in time at $\epsilon = 0.1$.




**Optional: simulate backwards in time** Figure 3 shows that projective integration even simulates backwards in time along the slow manifold using short forward bursts. Such backwards macroscale simulations succeed despite the fast variable $y(t)$, when backwards in time, being viciously unstable. However, backwards integration appears to need longer bursts, here $3\epsilon$.

```
ts = 0:-1:-5
[xs,tMicro,xMicro] = PIRK2(@MMburst, 3*epsilon, ts, 0.2*[1;1]);
figure, plot(ts,xs,'o:',tMicro,xMicro)
xlabel('time t'), legend('x(t)','y(t)')
```

### 2.1.2 Code a burst of Michaelis–Menten enzyme kinetics

Say use ode23() to integrate a burst of the differential equations for the Michaelis–Menten enzyme kinetics. Code differential equations in function dMMdt with variables $x$ = x(1) and $y$ = x(2). For the given start time ti, and start state xi, ode23() integrates the differential equations for a burst time of bT, and return the simulation data.

```
140 function [ts, xs] = MMburst(ti, xi, bT)
141     global epsilon
142     dMMdt = @(t,x) [ -x(1)+(x(1)+0.5)*x(2)
143           1/epsilon*( x(1)-(x(1)+1)*x(2) ) ];
144     [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
145 end
```

## 2.2 PIRK2(): projective integration of second order accuracy

This Projective Integration scheme implements a macroscale scheme that is analogous to second order Runge–Kutta integration.

```
14 function [x, tms, xms, rm, svf] = PIRK2(solver, bT, tSpan, x0)
```

### Input

- solver(), a function that produces output from the user-specified code for micro-scale simulation.

  ```
  [tOut, xOut] = solver(tStart, xStart, bT)
  ```

  - Inputs: tStart, the start time of a burst of simulation; xStart, the row $n$-vector of the starting state; bT, the total time to simulate in the burst.
  - Outputs: tOut, the column vector of solution times; and xOut, an array in which each *row* contains the system state at corresponding times.

- `bT`, a scalar, the minimum amount of time needed for simulation of the microsolver to relax the fast variables to the slow manifold.
- `tSpan` is an $\ell$-vector of times at which the user requests output, of which the first element is always the initial time. `PIRK2()` does not use adaptive time stepping; the macroscale time steps are (nearly) the steps between elements of `tSpan`.
- `x0` is an $n$-vector of initial values at the initial time `tSpan(1)`. Elements of `x0` may be `NaN`: they are included in the simulation and output, and often represent boundaries in space fields.

### Output

- `x`, an $\ell \times n$ array of the approximate solution vector. Each row is an estimated state at the corresponding time in `tSpan`. The simplest usage is then `x = PIRK2(solver,bT,tSpan,x0)`.

  However, microscale details of the underlying Projective Integration computations may be helpful. `PIRK2()` provides two to four additional outputs of the microscale bursts.

- `tms` is an $L$ dimensional column vector containing microscale times of burst simulations, each burst separated by `NaN`;
- `xms` is an $L \times n$ array of the corresponding microscale states—this data is an accurate simulation of the state and may help visualise more details of the solution.
- `rm`, a struct containing the 'remaining' applications of the microsolver required by the Projective Integration method during the calculation of the macrostep:
  - `rm.t` is a column vector of microscale times; and
  - `rm.x` is the array of corresponding burst states.

  The states `rm.x` do not have the same physical interpretation as those in `xms`; they are required in order to estimate the slow vector field

during the calculation of the Runge–Kutta increments, and do not in general resemble the true dynamics.

- svf, a struct containing the Projective Integration estimates of the slow vector field.
  - svf.t is an $2\ell$ dimensional column vector containing all times at which the Projective Integration scheme extrapolated along microsolver data to form a macrostep.
  - svf.dx is an $2\ell \times n$ array containing the estimated slow vector field.

**The projective integration code** Determine the number of time steps and preallocate storage for macroscale estimates.

```
76 nT=length(tSpan);
77 x=nan(nT,length(x0));
```

Get the number of expected outputs and set logical indices to flag what data should be saved.

```
84 nargs=nargout();
85 saveMicro = (nargs>1);
86 saveFullMicro = (nargs>3);
87 saveSvf = (nargs>4);
```

Run a preliminary application of the microsolver on the initial conditions to help relax to the slow manifold. This is done in addition to the microsolver in the main loop, because the initial conditions are often far from the attracting slow manifold. Require the user to input and output rows of the system state.

```
97 x0 = reshape(x0,1,[]);
98 [relax_t,relax_x0] = solver(tSpan(1),x0,bT);
```

Use the end point of the microsolver as the initial conditions.

```
105 tSpan(1) = tSpan(1)+bT;
106 x(1,:)=relax_x0(end,:);
```

If saving information, then record the first application of the microsolver. Allocate cell arrays for times and states for outputs requested by the user, as concatenating cells is much faster than iteratively extending arrays.

```
114 if saveMicro
115     tms = cell(nT,1);
116     xms = cell(nT,1);
117     tms{1} = reshape(relax_t,[],1);
118     xms{1} = relax_x0;
119     if saveFullMicro
120         rm.t = cell(nT,1);
121         rm.x = cell(nT,1);
122         if saveSvf
123             svf.t = nan(2*nT-2,1);
124             svf.dx = nan(2*nT-2,length(x0));
125         end
126     end
127 end
```

Loop over the macroscale time steps.

```
135 for jT = 2:nT
136     T = tSpan(jT-1);
```

Run the first application of the microsolver; since this application directly follows from the initial conditions, or from the latest macrostep, this microscale information is physically meaningful as a simulation of the system. Extract the size of the final time step.

```
144     [t1,xm1] = solver(T, x(jT-1,:), bT);
145     del = t1(end)-t1(end-1);
```

Find the needed time step to reach time `tSpan(n+1)` and form a first estimate `dx1` of the slow vector field.

```
152     Dt = tSpan(jT)-T-bT;
```

```
153     dx1 = (xm1(end,:)-xm1(end-1,:))/del;
```

Project along dx1 to form an intermediate approximation of x; run another application of the microsolver and form a second estimate of the slow vector field.

```
160     xint = xm1(end,:) + (Dt-bT)*dx1;
161     [t2,xm2] = solver(T+Dt, xint, bT);
162     del = t2(end)-t2(end-1);
163     dx2 = (xm2(end,:)-xm2(end-1,:))/del;
```

Use the weighted average of the estimates of the slow vector field to take a macrostep.

```
170     x(jT,:) = xm1(end,:) + Dt*(dx1+dx2)/2;
171
```

If saving trusted microscale data, then populate the cell arrays for the current loop iterate with the time steps and output of the first application of the microsolver. Separate bursts by NaNs.

```
179     if saveMicro
180         tms{jT} = [reshape(t1,[],1); nan];
181         xms{jT} = [xm1; nan(1,size(xm1,2))];
```

If saving all microscale data, then repeat for the remaining applications of the microsolver.

```
188         if saveFullMicro
189             rm.t{jT} = [reshape(t2,[],1); nan];
190             rm.x{jT} = [xm2; nan(1,size(xm2,2))];
```

If saving Projective Integration estimates of the slow vector field, then populate the corresponding cells with times and estimates.

```
197             if saveSvf
198                 svf.t(2*jT-3:2*jT-2) = [t1(end); t2(end)];
199                 svf.dx(2*jT-3:2*jT-2,:) = [dx1; dx2];
200             end
```

```
201         end
202     end
```

Terminate the main loop:

```
208 end
```

Overwrite `x(1,:)` with the specified initial condition `tSpan(1)`.

```
216 x(1,:) = reshape(x0,1,[]);
```

For additional requested output, concatenate all the cells of time and state data into two arrays.

```
223 if saveMicro
224     tms = cell2mat(tms);
225     xms = cell2mat(xms);
226     if saveFullMicro
227         rm.t = cell2mat(rm.t);
228         rm.x = cell2mat(rm.x);
229     end
230 end
```

This concludes `PIRK2()`.

```
237 end
```

## 2.3 Example: PI using Runge–Kutta macrosolvers

This script is a demonstration of the `PIRK()` schemes, that use a Runge–Kutta macrosolver, applied to a simple linear system with some slow and fast directions.

Clear workspace and set a seed.

```
11 clear
12 rng(1)
```

The majority of this example involves setting up details for the microsolver. We use a simple function `gen_linear_system` that outputs a func-

tion $f(t,x) = \mathbf{A}\vec{x} + \vec{b}$, where $\mathbf{A}$ has some eigenvalues with large negative real part, corresponding to fast variables and some eigenvalues with real part close to zero, corresponding to slow. The function requires that we specify bounds on the real part of the strongly stable eigenvalues,

```
22 fastband = [-5e2; -1e2];
```

and bounds on the real part of the weakly stable/unstable eigenvalues,

```
28 slowband = [-0.002; 0.002];
```

We now generate a random linear system with seven fast and three slow variables.

```
34 f = gen_linear_system(7,3,fastband,slowband);
```

Set the time step size and total integration time of the microsolver.

```
40 dt = 0.001;
41 micro.bT = 0.05;
```

As a rule of thumb, the time steps `dt` should satisfy `dt` $\leq 1/|$`fastband`$(1)|$ and the time to simulate with each application of the microsolver, `micro.bT`, should be larger than or equal to $1/|$`fastband`$(2)|$. We set the integration scheme to be used in the microsolver. Since the time steps are so small, we just use the forward Euler scheme

```
48 solver='fe';
```

(Other options: 'rk2' for second order Runge–Kutta, 'rk4' for fourth order, or any Matlab/Octave integrator such as 'ode45'.)

A crucial part of the PI philosophy is that it does not assume anything about the microsolver. For this reason, the microsolver must be a 'black box', which can be run by specifying an initial time and state, and a duration to simulate for. All the details of the microsolver are up to the user. We generate and save a black box microsolver.

```
61 bbm = bbgen(solver,f,dt);
62 micro.solver = bbm;
```

Set the macroscale times at which we request output from the PI scheme and the initial conditions.

```
tspan=0: 1 : 30;
IC = linspace(-10,10,10);
```

We implement the PI scheme, saving the coarse states in `x`, the 'trusted' applications of the microsolver in `xmicro`, and the additional applications of the microsolver in `xrmicro`. Note that the second and third outputs are optional and do not need to be set.

```
[x,xmicro,xrmicro] = PIRK4(micro,tspan,IC);
```

For verification, we also compute the trajectories using a standard solver.

```
[tt,ode45x] = ode45(f,tspan([1,end]),IC);
```

Figure 4 plots the output.

```
clf()
hold on
PI_sol=plot(tspan,x,'bo');
std_sol=plot(tt,ode45x,'r');
plot(xmicro{1},xmicro{2},'k.');
plot(xrmicro{1},xrmicro{2},'g.');
legend([PI_sol(1),std_sol(1)],'PI Solution',...
    'Standard Solution','Location','NorthWest')
xlabel('Time');
ylabel('State');
```

Save plot to a file.

```
set(gcf,'PaperPosition',[0 0 14 10])
print('-depsc2','PIRK')
```

## 2.4 PIG()

This is an approximate Projective Integration scheme in which the macrosolver is given by any user-specified scheme. Unlike the `PIRK` functions,

Figure 4: Demonstration of PIRK4(). From initial conditions, the system rapidly trannsitions to an attracting invariant manifold. The PI solution accurately tracks the evolution of the variables over time while requiring only a fraction of the computations of the standard solver.

PIG() can not estimate the slow vector field at the times expected by any user-specified scheme, but instead provides an estimate of the slow vector field at a slightly different time, after an application of the microsolver. Consequently PIG() will incur an additional global error term proportional to the burst length of the microscale simulator. For that reason, PIG() should be used with very stiff problems, in which the burst length of the microsolver can be short, or with the "constraint defined manifold" based microsolver provided by cdmc(), that attempts to project the variables onto the slow manifold without affecting the time.

```
14 function varargout = PIG(micro,macro,IC)
```

The inputs and outputs are necessarily a little different to the two PIRK2 functions.

### Input

- micro is a struct holding all information needed to run the microsolver:
  - micro.bT, a scalar, the minimum amount of time thought needed for integration of the microsolver to relax the fast variables to the slow manifold.
  - micro.solver(), a function that produces output from the user-specified code for micro-scale simulation. Usage:
    [tout,xout] = solver(t_in,x_in,bT) Inputs: t_in, the initialisation time; x_in $\in \mathbb{R}^n$, the initial state; bT, the time to simulate for.
    Outputs: tout, the vector of solution times, and xout, the corresponding states.

The remaining inputs to PIG() set the solver and parameters used for macroscale simulation.

- macro is a struct holding information about the macrosolver.
  - macro.solver(), the numerical method that the user wants to apply on a slow time scale. The solver should be formatted

as a standard numerical method in Matlab/Octave that can be called as `[t_out,x_out] = solver(f,tspan,IC)` for an ordinary differential equation $\frac{dx}{dt} = f(t,x)$, vector of input times `tspan` and initial condition `IC`. The function `f(t,x)` is not an input for `PIG()` but will instead be estimated by PI.

- `macro.tspan`, a vector of times at which the user requests output, of which the first element is always the initial time. If macro.solver can adaptively select time steps (e.g. `ode45`), then tspan can consist of an initial and final time only.

- `IC` is an $n$-vector of initial values at the time `tspan(1)`.

**Output** Standard usage is to output only macrosolver information, with the following usage:

```
x = PIG(micro,macro,IC)
```

- `x`, a cell array. `x{1}`, an $\ell$-vector of times at which PI produced output. `x{2}`, an $\ell \times n$ array of the approximate solution vector. Each row corresponds to an element of `x{1}`.

It is also possible to return the microsolver applications called by the PI method in executing the user-defined macrosolver. Much of this microscale data will not be an accurate solution of the system, and rather be a tool used to relax the fast variables close to the slow manifold in the process of executing a single macroscale time step.

```
[x,xm] = PIRK4(micro,tspan,IC)
```

- `xm`, a cell array containing the output of all applications of the microsolver. `xm{1}` is an $L$ dimensional column vector containing times; `xm{2}` is an $L \times n$ array of the corresponding microsolver output.

```
[x,xm,dx] = PIRK4(micro,tspan,IC)
```

- `dx`, a cell array containing the PI estimates of the slow vector field. `dx{1}` is an $\ell$ dimensional column vector containing all times at which

the PI scheme extrapolated along microsolver data to form a macrostep. `dx{2}` is an $\ell \times n$ array containing the estimated slow vector field.

The main body of the function now follows.

Get microscale and macroscale information from inputs, and compute the number of time steps at which output is expected.

```
71 solver = micro.solver;
72 bT = micro.bT;
73 tspan = macro.tspan;
74 csolve = macro.solver;
75 N=length(tspan)-1;
```

Get the number of expected outputs and set logical indices to flag what data should be saved.

```
82 nargs=nargout();
83 save_micro = (nargs>1);
84 save_dx = (nargs>2);
```

Run a first application of the microsolver on the initial conditions. This is done in addition to the microsolver in the main loop, because the initial conditions are often far from the attracting slow manifold.

```
93 IC = reshape(IC,[],1);
94 [relax_t,relax_IC] = solver(tspan(1),IC,bT);
```

Update the initial time.

```
101 tspan(1) = tspan(1)+bT;
```

Allocate cell arrays for times and states for any of the outputs requested by the user. If saving information, then record the first application of the microsolver. Note that it is unknown a priori how many applications of the microsolver will be required; this code may be run more efficiently if the correct number is used in place of `N+1` as the dimension of the cell arrays.

```
110 if save_micro
```

```
111     t_micro=cell(1,N+1); x_micro=cell(1,N+1);
112     n=1;
113     t_micro{n} = reshape(relax_t,[],1);
114     x_micro{n} = relax_IC;
115
116     if save_dx
117         dx_t = cell(1,N+1);
118         dx_hist = cell(1,N+1);
119     end
120 end
```

The idea of `PIG()` is to use the output from the microsolver to approximate an unknown function `ff(t,x)`, that describes the slow dynamics. This approximation is then used in the user-defined "coarse solver" `csolve()`. The approximation is described in the following function:

```
131     function [dx]=genProjection(tt,xx)
```

Run a microsolver from the given initial conditions.

```
137 [t_tmp,x_micro_tmp] = solver(tt,reshape(xx,[],1),bT);
```

Compute the standard PI approximation of the slow vector field.

```
143         del = t_tmp(end)-t_tmp(end-1);
144         dx = (x_micro_tmp(end,:)-x_micro_tmp(end-1,:))'/(del);
```

Save the microscale data, and the PI slow vector field, if requested.

```
150         if save_micro
151             n=n+1;
152             t_micro{n} = reshape(t_tmp,[],1);
153             x_micro{n} = x_micro_tmp;
154             if save_dx
155                 dx_t{n-1} = tt;
156                 dx_hist{n-1} = dx;
157             end
158         end
```

End genProjection().

```
    end
```

Define the approximate slow vector field according to PI.

```
ff=@(t,x) genProjection(t,x);
```

Do Projective Integration of ff() with the user-specified solver.

```
[t,x]=feval(csolve,ff,tspan,relax_IC(end,:)');
```

Write over x(1,:) and t(1), which the user expect to be IC and tspan(1) respectively, with the given initial conditions.

```
x(1,:) = IC';
t(1) = tspan(1);
```

Output the macroscale steps:

```
varargout{1} = {t x};

```

For each additional requested output, concatenate all the cells of time and state data into two arrays. Then, return the two arrays in a cell.

```
if save_micro
    t_micro = cat(1,t_micro{:});
    x_micro = cat(1,x_micro{:});
    varargout{2} = {t_micro x_micro};
     if save_dx
         dx_t = cat(1,dx_t{:});
         dx_hist = cat(1,dx_hist{:});
         varargout{3} = {dx_t dx_hist};
     end
end
```

This concludes PIG().

```
end
```

## 2.5 Example 2: PI using General macrosolvers

In this example the PI-General scheme is applied to a singularly perturbed ordinary differential equation. The aim is to allow a standard non-stiff numerical integrator, e.g. `ode45()`, to be applied to a stiff problem on a slow, long time scale.

```
clear
```

Set time scale separation and model.

```
epsilon = 1e-5;
f=@(t,x) [cos(x(1))*sin(x(2))*cos(t); (cos(x(1))-x(2))/epsilon];
```

Set the 'black box' microsolver to be an integration using a standard solver, and set the standard time of simulation for the microsolver.

```
micro.solver = @(tIC, xIC,T) feval('ode45',f,[tIC tIC+T],xIC);
micro.bT=20*epsilon;
```

Set initial conditions, and the time to be covered by the macrosolver. Set the macrosolver to be used as a standard, non-stiff integration scheme.

```
IC = [1 3];
tspan=[0 40];
macro.tspan = tspan;
macro.solver = 'ode45';
```

Now time and integrate the above system over `tspan` using `PIG()` and, for comparison, a brute force implementation of `ode45()`. Report the time taken by each method.

```
tic
[x,xmicro] = PIG(micro,macro,IC);
tPI=toc;
fprintf(['PI took %f seconds, using ode45 as the '...
    'macrosolver.\n'],tPI)
tic
[t45,xode45] = ode45(f,[tspan(1) tspan(end)],IC);
```

```
51 tODE45 = toc;
52
53 fprintf('Brute force ode45 took %f seconds.\n',tODE45)
```

Plot the output on two figures, showing the truth and macrosteps on both, and all applications of the microsolver on the first figure.

```
62 figure; set(gcf,'PaperPosition',[0 0 14 10])
63 hold on
64 PIm=plot(xmicro{1},xmicro{2},'b.');
65 PI=plot(x{1},x{2},'bo');
66 ODE45=plot(t45,xode45,'r-','LineWidth',2);
67 legend([PI(1),ODE45(1),PIm(1)],'PI Solution',...
68     'Standard Solution','PI microsolver')
69 xlabel('Time')
70 ylabel('State')
71 axis([0 40 0 3])
72
73 figure; set(gcf,'PaperPosition',[0 0 14 10])
74 hold on
75 PI2=plot(x{1},x{2},'bo');
76 ODE452=plot(t45,xode45,'r-','LineWidth',2);
77
78 legend([PI2(1),ODE452(1)],'PI Solution','Standard Solution')
79 xlabel('Time')
80 ylabel('State')
```

The output is plotted in Figure 5.

Notes:

- the problem may be made more, or less, stiff by changing the time scale parameter `epsilon`. `PIG()` will handle a stiffer problem with ease; but if the problem is insufficiently stiff, then the algorithm will produce nonsense. This problem is handled by `cdmc()`; see Section 2.7.
- The mildly stiff problem in Example 2.3 may be efficiently solved by a standard solver, e.g. `ode45()`. The stiff but low dimensional

Figure 5: Accurate simulation of a stiff nonautonomous system by PIG(). The microsolver is called on-the-fly by the macrosolver (here ode45).

problem in this example can be solved efficiently by a standard stiff solver, e.g. `ode15s()`. The real advantage of the PI schemes is in high dimensional stiff problems, that cannot be efficiently solved by most standard methods.

## 2.6 Minor functions

### 2.6.1 `cdmc()`

`cdmc()` iteratively applies the microsolver and then projects backwards in time to the initial conditions. The cumulative effect is to relax the variables to the attracting slow manifold, while keeping the final time for the output the same as the input time.

### Input

- `solver`, a black box microsolver suitable for PI. See any of `PIRK2(), PIRK4(), PIG()` for a description of `solver()`.
- `t`, an initial time
- `x`, an initial state
- `T`, a time period to apply `solver()` for

### Output

- `tout`, a vector of times. `tout(end)` will equal `t`.
- `xout`, an array of state estimates produced by `solver()`.

This function is intended to be used as a wrapper for the microsolver. That is, for instance if the problem of interest is a dynamical system that is not too stiff, and which can be simulated by the solver `sol(t,x,T)`, one would define `micro.solver = @(t,x,T) cdmc(sol,t,x,T)`. The original solver `sol()` would create large errors if used in a PI scheme; but the output of `cdmc()` will not.

```
28 function [tout, xout] = cdmc(solver,t,x,T)
```

Begin with a standard application of the microsolver.

```
34     [tt,xx]=feval(solver,t,x,T);
```

Project backwards to before the initial time, then simulate one burst forwards to obtain coordinates at t.

```
44 del = (xx(end,:) - xx(end-1,:))/(tt(end) - tt(end-1));
45 b_prop = 0.2;
46 x = xx(end,:) + (1+b_prop)*(tt(1)-tt(end))*del;
47 tr=2*tt(1)-tt(end);
48 [tout,xout]=feval(solver,tr,x',b_prop*T);
```

This concludes the function.

### 2.6.2 bbgen()

bbgen() is a simple function that takes a standard numerical method and produces a black box solver of the type required by the PI schemes.

```
12 function bb = bbgen(solver,f,dt)
```

#### Input

- solver, a standard numerical solver for ordinary differential equations
- f, a function f(t,x) taking time and state inputs
- dt, a time step suitable for simulation with f

**Output** bb = bb($t_{in}, x_{in}, T$) a 'black box' microsolver that takes initial conditions $(t_{in}, x_{in})$ and simulates forward a duration $T$.

```
28 bb = @(t_in,x_in,T) feval(solver,f,...
29 linspace(t_in,t_in+T,1+ceil(T/dt)),x_in);
30 end
```

## 2.7 Explore: PI using constraint-defined manifold computing

In this example the PI-General scheme is applied to a singularly perturbed ordinary differential equation in which the time scale separation is not too strong. The resulting simulation is not accurate. In parallel, we run the same scheme but with cdmc() used as a wrapper for the microsolver. This implementation successfully replicates the true dynamics.

```
9 clear
```

Set a weak time scale separation and model.

```
epsilon = 1e-2;
f=@(t,x) [cos(x(1))*sin(x(2))*cos(t); (cos(x(1))-x(2))/epsilon];
```

Set the 'naive' microsolver to be an integration using a standard solver, and set the standard time of simulation for the microsolver.

```
naivemicro.solver = @(tIC, xIC,T) feval('ode45',f,[tIC tIC+T],xIC);
naivemicro.bT=5*epsilon;
```

Create a second struct in which the solver is the output of `cdmc()`.

```
cdmicro.solver = @(t,x,T) cdmc(naivemicro.solver,t,x,T);
cdmicro.bT = naivemicro.bT;
```

Set initial conditions, and the time to be covered by the macrosolver. Set the macrosolver to be used as a standard, non-stiff integration scheme.

```
IC = [1 3];
tspan=0:0.5:40;
macro.tspan = tspan;
macro.solver = 'ode45';
```

Simulate using `PIG()` with each of the above microsolvers. Generate a trusted solution using standard numerical methods.

```
naivex = PIG(naivemicro,macro,IC);
x = PIG(cdmicro,macro,IC);
[t45,xode45] = ode45(f,[tspan(1) tspan(end)],IC);
```

Plot the output.

```
figure; set(gcf,'PaperPosition',[0 0 14 10])
hold on
nPI = plot(naivex{1},naivex{2},'bo');
PI=plot(x{1},x{2},'ko');
ODE45=plot(t45,xode45,'r-','LineWidth',2);

legend([nPI(1),PI(1),ODE45(1)],'Naive PIG Solution',...
    'PIG using cdmc','Accurate Solution')
xlabel('Time')
```

```
70 ylabel('State')
71 axis([0 40 0 3])
```

The output is plotted in Figure 6. The source of the error in the standard PIG() scheme is the burst length bT, that is significant on the slow time scale. Set bT to 20*epsilon or 50*epsilon[1] to worsen the error in both schemes. This example reflects a general principle, that most PI schemes will incur a global error term which is proportional to the simulation time of the microsolver and independent of the order of the microsolver. The PIRK() schemes have been written to minimise, if not eliminate entirely, this error, but by design PIG() works with any user-defined macrosolver and cannot reduce this error. The function cdmc() reduces this error term by attempting to mimic the microsolver without advancing time.

## 2.8 PIRK4()

This is a Projective Integration scheme in which the macrosolver is given by the fourth order Runge–Kutta method.

```
13 function varargout = PIRK4(micro,tspan,IC)
```

The inputs and outputs are standardised with PIRK2().

### Input

- micro is a struct holding all information needed to run the microsolver:
  - micro.bT, a scalar, the minimum amount of time thought needed for integration of the microsolver to relax the fast variables to the slow manifold.
  - micro.solver(), a function that produces output from the user-specified code for micro-scale simulation. Usage:

    ```
    [tout,xout] = solver(t_in,x_in,bT)
    ```

[1] this example is quite extreme: at bT=50*epsilon, it would be computationally much cheaper to simulate the entire length of tspan using the microsolver alone.

Figure 6: Accurate simulation of a weakly stiff nonautonomous system by PIG() using cdmc(), and an inaccurate solution using a naive application of PIG().

> Inputs: `t_in`, the initialisation time, `x_in` $\in \mathbb{R}^n$, the initial state, and `bT`, the total time of simulation.
> Outputs: `tout`, the vector of solution times, and `xout`, the corresponding states.

The remaining inputs to `PIRK4()` relate to the macroscale solution.

- `tspan` is an $\ell$-vector of times at which the user requests output, of which the first element is always the initial time.
- `IC` is an $n$-vector of initial values at the time `tspan(1)`.

**Output** Standard usage is to output only estimates of the state at the times given in `tspan`, with the following usage:

`x = PIRK4(micro,tspan,IC)`

- `x`, an $\ell \times n$ array of the approximate solution vector. Each row corresponds to an element of `tspan`.

`[x,xm] = PIRK4(micro,tspan,IC)`

- `xm`, a cell array containing the output of the applications of the microsolver that immediately follow a PI macrostep. `xm{1}` is an $L$ dimensional column vector containing times; `xm{2}` is an $L \times n$ array of the corresponding microsolver output. The data in `xm` is an accurate simulation and can be used alongside `x` when visualising the solution.

`[x,xm,xr] = PIRK4(micro,tspan,IC)`

- `xr`, a cell array containing the 'remaining' applications of the microsolver required by the PI method during the calculation of the macrostep. The applications of the microsolver in `xr` do not have the same physical interpretation as those in `xm`; they are required in order to estimate the slow vector field during the calculation of the Runge–Kutta increments, and will not in general have good error statistics.

`[x,xm,xr,dx] = PIRK4(micro,tspan,IC)`

- `dx`, a cell array containing the PI estimates of the slow vector field. `dx{1}` is an $\ell$ dimensional column vector containing all times at which the PI scheme extrapolated along microsolver data to form a macrostep. `dx{2}` is an $\ell \times n$ array containing the estimated slow vector field.

The main body of the m-file now follows.

Get microscale information from inputs.

```
68 solver = micro.solver;
69 bT = micro.bT;
```

Compute the number of time steps and create storage for output.

```
77 N=length(tspan)-1;
78 x=zeros(N+1,length(IC));
79
```

Get the number of expected outputs and set logical indices to flag what data should be saved.

```
86 nargs=nargout();
87 save_micro = (nargs>1);
88 save_full_micro = (nargs>2);
89 save_dx = (nargs>3);
```

Run a first application of the microsolver on the initial conditions. This is done in addition to the microsolver in the main loop, because the initial conditions are often far from the attracting slow manifold.

```
96 IC = reshape(IC,[],1);
97 [relax_t,relax_IC] = solver(tspan(1),IC,bT);
```

Set the initial macro-time and -state using the end point of the microsolver as the initial conditions.

```
104 tspan(1) = tspan(1)+bT;
105 x(1,:)=relax_IC(end,:);
```

Allocate cell arrays for times and states for any of the outputs requested by the user. If saving information, then record the first application of the microsolver.

```
112 if save_micro
113     t_micro=cell(1,N+1); x_micro=cell(1,N+1);
114     t_micro{1} = reshape(relax_t,[],1);
115     x_micro{1} = relax_IC;
116
117     if save_full_micro
118         t_r_micro = cell(1,N+1);
119         x_r_micro = cell(1,N+1);
120
121         if save_dx
```

```
122             dx_t = cell(1,N+1);
123             dx_hist = cell(1,N+1);
124         end
125     end
126 end
```

Main loop:

```
133 for n=1:N
134     t=tspan(n);
```

Run the first application of the microsolver; since this application directly follows from the initial conditions, or from the latest macrostep, this microscale information is physically meaningful as a simulation of the system. Extract the length of the burst in time and the size of the final time step.

```
143  [t1,xm1] = solver(t,x(n,:)',bT);
144     del=t1(end)-t1(end-1);
```

Find the needed time step to reach time `tspan(n+1)`, form a first estimate `dx1` of the slow vector field.

```
151     Dt=tspan(n+1)-t-bT;
152
153     dx1 = (xm1(end,:)-xm1(end-1,:))/del;
```

Project along `dx1` to form an intermediate approximation of `x`; repeat the above steps three more times in order to form four approximations of the slow vector field.

```
160       xint = xm1(end,:) + (Dt/2-bT)*dx1;
161     [t2,xm2] = solver(t+Dt/2,xint',bT);
162     dx2 = (xm2(end,:)-xm2(end-1,:))/del;
163
164     xint = xm1(end,:) + (Dt/2-bT)*dx2;
165     [t3,xm3] = solver(t+Dt/2,xint',bT);
166     dx3 = (xm3(end,:)-xm3(end-1,:))/del;
167
168     xint = xm1(end,:) + (Dt-bT)*dx3;
```

```
169     [t4,xm4] = solver(t+Dt,xint',bT);
170     dx4 = (xm4(end,:)-xm4(end-1,:))/del;
```

Use the weighted average of the estimates of the slow vector field to take a macrostep.

```
177     x(n+1,:) = xm1(end,:) + Dt*(dx1+2*dx2+2*dx3+dx4)/6;
```

If saving trusted microscale data, populate the cell arrays for the current loop iterate with the time steps and output of the first application of the microsolver.

```
184     if save_micro
185         t_micro{n+1} = reshape(t1,[],1);
186         x_micro{n+1} = xm1;
```

If saving all microscale data, populate the cell arrays for the current loop iterate with the time steps and output of the remaining applications of the microsolver.

```
193         if save_full_micro
194             t_r_micro{n+1} = [reshape(t2,[],1); reshape(t3,[],1); re
195             x_r_micro{n+1} = [xm2; xm3; xm4];
```

If saving PI estimates of the slow vector field, populate the corresponding cells with the times and estimates.

```
202             if save_dx
203                 dx_t{n} = [t1(end); t2(end); t3(end); t4(end)];
204                 dx_hist{n} = [dx1; dx2; dx3; dx4];
205             end
206         end
207     end
```

Terminate the main loop:

```
213 end
```

Write over `x(1,:)`, which the user expects to correspond to the input `tspan(1)`, with the given initial condition.

```
x(1,:) = IC';
```

Output the macroscale steps:

```
varargout{1} = x;

```

For each additional requested output, concatenate all the cells of time and state data into two arrays. Then, return the two arrays in a cell.

```
if save_micro
    t_micro = cat(1,t_micro{:});
    x_micro = cat(1,x_micro{:});
    varargout{2} = {t_micro x_micro};

    if save_full_micro
        t_r_micro = cat(1,t_r_micro{:});
        x_r_micro = cat(1,x_r_micro{:});
        varargout{3} = {t_r_micro x_r_micro};

        if save_dx
            dx_t = cat(1,dx_t{:});
            dx_hist = cat(1,dx_hist{:});
            varargout{4} = {dx_t dx_hist};
        end
    end
end
```

This concludes `PIRK4()`.

```
end
```

## 2.9 To do/discuss

- AJR: inconsistency between description $n$ and $\ell$, and the code's `N` and `n`?? Proposed a partial answer. Also need to be clear what need to be row/column vectors.

- AJR: is there any advantage to `reshape(t2,[],1)` over `t2(:)`? Answer appears to be nothing either way.
- AJR: can this code 'integrate' backwards in time using the forward time bursts? Answer: yes. But need to test its accuracy??
- AJR: replaced `varargout` as I believe many users would be put-off by the extra level of abstraction: OK? or not?
- AJR: remember, if used, cater for complex variable simulation by using real transpose `.'`.
- could implement Projective Integration by 'arbitrary' Runge–Kutta scheme; that is, by having the user input a particular Butcher table—surely only specialists would be interested
- can 'reverse' the order of projection and microsolver applications. The output at each user-requested coarse time would then be the end point of an application of the microsolver.
- some kind of minimally invasive checking is needed to ensure the burst length of the microsolver does not make the Projective Integration scheme redundant. For example, for systems that are not too stiff and for a fourth order Projective Integration scheme PIRK4, we should check that four applications of the microsolver do not bridge the gap between user-specified times.
- separate subsection for microsolver requirements? Then can point to it in each other function.
- Can maybe implement microsolvers that terminate a burst when the fast dynamics have settles using, for example, the 'Events' function handle in ode23.

# 3 Patch scheme for given microscale discrete space system

*Section contents*



The patch scheme applies to spatio-temporal systems where the spatial domain is larger than what can be computed in reasonable time. Then one may simulate only on small patches of the space-time domain, and produce correct macroscale predictions by craftily coupling the patches across unsimulated space (Hyman 2005, Samaey et al. 2005, 2006, Roberts & Kevrekidis 2007, Liu et al. 2015, e.g.).

The spatial discrete system is to be on a lattice such as obtained from finite difference approximation of a PDE. Usually continuous in time.

## 3.1 patchSmooth1()

Couples patches across space so a spatially discrete system can be integrated in time via the patch or gap-tooth scheme (Roberts & Kevrekidis 2007). Assumes that the sub-patch structure is *smooth* so that the patch centre-values are sensible macroscale variables, and patch edge values are determined

by macroscale interpolation of the patch-centre values. Need to pass patch-design variables to this function, so use the global struct `patches`.

```
16 function dudt=patchSmooth1(t,u)
17 global patches
```

### Input

- `u` is a vector of length `nSubP` · `nPatch` · `nVars` where there are `nVars` field values at each of the points in the `nSubP` × `nPatch` grid.
- `t` is the current time to be passed to the user's time derivative function.
- `patches` a struct set by `makePatches()` with the following information that is used here.
  - `.fun` is the name of the user's function `fun(t,u,x)` that computes the time derivatives on the patchy lattice. The array `u` has size `nSubP` × `nPatch` × `nVars`. Time derivatives must be computed into the same sized array, but the patch edge values will be overwritten by zeros.
  - `.x` is `nSubP` × `nPatch` array of the spatial locations $x_{ij}$ of the microscale grid points in every patch. Currently it *must* be a regular lattice on both macro- and micro-scales.

### Output

- `dudt` is `nSubP` · `nPatch` · `nVars` vector of time derivatives, but with zero on patch edges??

Reshape the fields `u` as a 2/3D-array, and sets the edge values from macroscale interpolation of centre-patch values. §3.3 describes function `patchEdgeInt1()`.

```
46 u=patchEdgeInt1(u);
```

Ask the user for the time derivatives computed in the array, overwrite its edge values with the dummy value of zero, then return to an integrator as column vector.

```
53 dudt=patches.fun(t,u,patches.x);
54 dudt([1 end],:,:)=0;
55 dudt=reshape(dudt,[],1);
```

Fin.

## 3.2 makePatches(): makes the spatial patches for the suite

Makes the struct `patches` for use by the patch/gap-tooth time derivative function `patchSmooth1()`.

```
14 function makePatches(fun,Xlim,BCs,nPatch,ordCC,ratio,nSubP)
15 global patches
```

### Input

- `fun` is the name of the user function, `fun(t,u,x)`, that will compute time derivatives of quantities on the patches.
- `Xlim` give the macro-space domain of the computation: patches are spread evenly over the interior of the interval [`Xlim(1)`, `Xlim(2)`].
- `BCs` somehow will define the macroscale boundary conditions. Currently `BCs` is ignored and the system is assumed macro-periodic in the domain.
- `nPatch` is the number of evenly spaced patches.
- `ordCC` is the order of interpolation across empty space of the macroscale mid-patch values to the edge of the patches for inter-patch coupling: currently must be in $\{-1, 0, \ldots, 8\}$.
- `ratio` (real) is the ratio of the half-width of a patch to the spacing of the patch mid-points: so `ratio` $= \frac{1}{2}$ means the patches abut; and `ratio` $= 1$ is overlapping patches as in holistic discretisation.
- `nSubP` is the number of microscale lattice points in each patch. Must be odd so that there is a central lattice point.

**Output** The *global* struct `patches` is created and set.

- `patches.fun` is the name of the user's function `fun(u,t,x)` that computes the time derivatives on the patchy lattice.
- `patches.ordCC` is the specified order of inter-patch coupling.
- `patches.alt` is true for interpolation using only odd neighbouring patches as for staggered grids, and false for the usual case of all neighbour coupling.
- `patches.Cwtsr` and `.Cwtsl` are the `ordCC`-vector of weights for the inter-patch interpolation onto the right and left edges (respectively) with patch:macroscale ratio as specified.
- `patches.x` is `nSubP` $\times$ `nPatch` array of the regular spatial locations $x_{ij}$ of the microscale grid points in every patch.

First, store the pointer to the time derivative function in the struct.

```
44 patches.fun=fun;
```

Second, store the order of interpolation that is to provide the values for the inter-patch coupling conditions. Maybe allow `ordCC` of 0 and $-1$ to request spectral coupling??

```
52 if ~ismember(ordCC,[-1:8])
53     error('makePatch: ordCC out of allowed range [-1:8]')
54 end
```

For odd `ordCC` do interpolation based upon odd neighbouring patches as is useful for staggered grids.

```
60 patches.alt=mod(ordCC,2);
61 ordCC=ordCC+patches.alt;
62 patches.ordCC=ordCC;
```

Check for staggered grid and periodic case.

```
68   if patches.alt & (mod(nPatch,2)==1)
69     error('Must have an even number of patches for a staggered grid'
```

```
  end
```

Might as well precompute the weightings for the interpolation of field values for coupling. (What about coupling via derivative values??)

```
if patches.alt  % eqn (7) in \cite{Cao2014a}
  patches.Cwtsr=[1
    ratio/2
    (-1+ratio^2)/8
    (-1+ratio^2)*ratio/48
    (9-10*ratio^2+ratio^4)/384
    (9-10*ratio^2+ratio^4)*ratio/3840
    (-225+259*ratio^2-35*ratio^4+ratio^6)/46080
    (-225+259*ratio^2-35*ratio^4+ratio^6)*ratio/645120 ];
else %
  patches.Cwtsr=[ratio
    ratio^2/2
    (-1+ratio^2)*ratio/6
    (-1+ratio^2)*ratio^2/24
    (4-5*ratio^2+ratio^4)*ratio/120
    (4-5*ratio^2+ratio^4)*ratio^2/720
    (-36+49*ratio^2-14*ratio^4+ratio^6)*ratio/5040
    (-36+49*ratio^2-14*ratio^4+ratio^6)*ratio^2/40320 ];
end
patches.Cwtsr=patches.Cwtsr(1:ordCC);
patches.Cwtsl=(-1).^((1:ordCC)'-patches.alt).*patches.Cwtsr;
```

Third, set the centre of the patches in a the macroscale grid of patches assuming periodic macroscale domain.

```
X=linspace(Xlim(1),Xlim(2),nPatch+1);
X=X(1:nPatch)+diff(X)/2;
DX=X(2)-X(1);
```

Construct the microscale in each patch, assuming Dirichlet patch edges, and a half-patch length of `ratio` · `DX`.

```
if mod(nSubP,2)==0, error('makePatches: nSubP must be odd'), end
```

```
113 i0=(nSubP+1)/2;
114 dx=ratio*DX/(i0-1);
115 patches.x=bsxfun(@plus,dx*(-i0+1:i0-1)',X); % micro-grid
```

Fin.

## 3.3 patchEdgeInt1(): sets edge values from macro-interpolation

*Subsection contents*



Couples patches across space by computing their edge values from macroscale interpolation. Consequently a spatially discrete system could be integrated in time via the patch or gap-tooth scheme (Roberts & Kevrekidis 2007). Assumes that the sub-patch structure is *smooth* so that the patch centre-values are sensible macroscale variables, and patch edge values are determined by macroscale interpolation of the patch-centre values. Pass patch-design variables via the global struct `patches`.

```
17 function u=patchEdgeInt1(u)
18 global patches
```

### Input

- `u` is a vector of length $\texttt{nSubP} \cdot \texttt{nPatch} \cdot \texttt{nVars}$ where there are `nVars` field values at each of the points in the $\texttt{nSubP} \times \texttt{nPatch}$ grid.
- `patches` a struct set by `makePatches()` with the following information.
  - `.fun` is the name of the user's function `fun(t,u,x)` that computes the time derivatives on the patchy lattice. The array `u` has size $\texttt{nSubP} \times \texttt{nPatch} \times \texttt{nVars}$. Time derivatives must be computed into the same sized array, but the patch edge values will be overwritten by zeros.

- .x is nSubP × nPatch array of the spatial locations $x_{ij}$ of the microscale grid points in every patch. Currently it *must* be a regular lattice on both macro- and micro-scales.
- .ordCC is order of interpolation, currently in $\{0, 2, 4, 6, 8\}$.
- .alt in $\{0, 1\}$ is one for staggered grid (alternating) interpolation.
- .Cwtsr and .Cwtsl

### Output

- u is nSubP × nPatch × nVars array of the fields with edge values set by interpolation.

Determine the sizes of things. Any error arising in the reshape indicates u has the wrong length.

```
49 [nM,nP]=size(patches.x);
50 nV=round(numel(u)/numel(patches.x));
51 u=reshape(u,nM,nP,nV);
```

With Dirichlet patches, the half-length of a patch is $h = dx(n_\mu - 1)/2$ (or $-2$ for specified flux), and the ratio needed for interpolation is then $r = h/\Delta X$. Compute lattice sizes from inside the patches as the edge values may be NaNs, etc.

```
58 dx=patches.x(3,1)-patches.x(2,1);
59 DX=patches.x(2,2)-patches.x(2,1);
60 r=dx*(nM-1)/2/DX;
```

For the moment?? assume the physical domain is macroscale periodic so that the coupling formulas are simplest. Should eventually cater for periodic, odd-mid-gap, even-mid-gap, even-mid-patch, dirichlet, neumann, ?? These index vectors point to patches and their two immediate neighbours.

```
69 j=1:nP; jp=mod(j,nP)+1; jm=mod(j-2,nP)+1;
```

The centre of each patch, assuming odd nM, is at

```
75 i0=round((nM+1)/2);
```

**Lagrange interpolation gives patch-edge values** So compute centred differences of the mid-patch values for the macro-interpolation, of all fields. Assumes the domain is macro-periodic.

```
84 if patches.ordCC>0 % then non-spectral interpolation
85   dmu=nan(patches.ordCC,nP,nV);
86   if patches.alt % use only odd numbered neighbours
87     dmu(1,:,:)=(u(i0,jp,:)+u(i0,jm,:))/2; % \mu
88     dmu(2,:,:)= u(i0,jp,:)-u(i0,jm,:); % \delta
89     jp=jp(jp); jm=jm(jm); % increase shifts to \pm2
90   else % standard
91     dmu(1,:,:)=(u(i0,jp,:)-u(i0,jm,:))/2; % \mu\delta
92     dmu(2,:,:)=(u(i0,jp,:)-2*u(i0,j,:)+u(i0,jm,:)); % \delta^2
93   end% if odd/even
```

Recursively take $\delta^2$ of these to form higher order centred differences (could unroll a little to cater for two in parallel).

```
99   for k=3:patches.ordCC
100    dmu(k,:,:)=dmu(k-2,jp,:)-2*dmu(k-2,j,:)+dmu(k-2,jm,:);
101  end
```

Interpolate macro-values to be Dirichlet edge values for each patch (Roberts & Kevrekidis 2007), using weights computed in `makePatches()` . Here interpolate to specified order.

```
108 u(nM,j,:)=u(i0,j,:)*(1-patches.alt) ...
109     +sum(bsxfun(@times,patches.Cwtsr,dmu));
110 u( 1,j,:)=u(i0,j,:)*(1-patches.alt) ...
111     +sum(bsxfun(@times,patches.Cwtsl,dmu));
```

**Case of spectral interpolation** Assumes the domain is macro-periodic. As the macroscale fields are $N$-periodic, the macroscale Fourier transform writes the centre-patch values as $U_j = \sum_k C_k e^{ik2\pi j/N}$. Then the edge-patch

values $U_{j\pm r} = \sum_k C_k e^{ik2\pi/N(j\pm r)} = \sum_k C'_k e^{ik2\pi j/N}$ where $C'_k = C_k e^{ikr2\pi/N}$. For nP patches we resolve 'wavenumbers' $|k| < \mathtt{nP}/2$, so set row vector ks $= k2\pi/N$ for 'wavenumbers' $k = (0, 1, \ldots, k_{\max}, -k_{\max}, \ldots, -1)$.

```
122 else% spectral interpolation
```

Deal with staggered grid by doubling the number of fields and halving the number of patches (makePatches tests there are an even number of patches). Then the patch-ratio is effectively halved. The patch edges are near the middle of the gaps and swapped.

```
130     if patches.alt % transform by doubling the number of fields
131         v=nan(size(u)); % currently to restore the shape of u
132         u=cat(3,u(:,1:2:nP,:),u(:,2:2:nP,:));
133         altShift=reshape(0.5*[ones(nV,1);-ones(nV,1)],1,1,[]);
134         iV=[nV+1:2*nV 1:nV]; % scatter interpolation to alternate fi
135         r=r/2; % ratio effectively halved
136         nP=nP/2; % halve the number of patches
137         nV=nV*2; % double the number of fields
138     else % the values for standard spectral
139         altShift=0;
140         iV=1:nV;
141     end
```

Now set wavenumbers.

```
147     kMax=floor((nP-1)/2);
148     ks=2*pi/nP*(mod((0:nP-1)+kMax,nP)-kMax);
```

Test for reality of the field values, and define a function accordingly.

```
154 if imag(u(i0,:,:))==0, uclean=@(u) real(u);
155     else               uclean=@(u) u;
156     end
```

Compute the Fourier transform of the patch centre-values for all the fields. If there are an even number of points, then zero the zig-zag mode in the FT and add it in later as cosine.

```
    Ck=fft(u(i0,:,:));
    if mod(nP,2)==0, Czz=Ck(1,nP/2+1,:)/nP; Ck(1,nP/2+1,:)=0; end
```

The inverse Fourier transform gives the edge values via a shift a fraction $r$ to the next macroscale grid point. Enforce reality when appropriate.

```
    u(nM,:,iV)=uclean(ifft(bsxfun(@times,Ck ...
        ,exp(1i*bsxfun(@times,ks,altShift+r)))));
    u( 1,:,iV)=uclean(ifft(bsxfun(@times,Ck ...
        ,exp(1i*bsxfun(@times,ks,altShift-r)))));
```

For an even number of patches, add in the cosine mode.

```
    if mod(nP,2)==0
        cosr=cos(pi*(altShift+r+(0:nP-1)));
        u(nM,:,iV)=u(nM,:,iV)+uclean(bsxfun(@times,Czz,cosr));
        cosr=cos(pi*(altShift-r+(0:nP-1)));
        u( 1,:,iV)=u( 1,:,iV)+uclean(bsxfun(@times,Czz,cosr));
    end
```

Restore staggered grid when appropriate. Is there a better way to do this??

```
if patches.alt
    nV=nV/2;  nP=2*nP;
    v(:,1:2:nP,:)=u(:,:,1:nV);
    v(:,2:2:nP,:)=u(:,:,nV+1:2*nV);
    u=v;
end
end% if spectral
```

Fin, returning the 2/3D array of field values.

### 3.3.1 `patchEdgeInt1test`: test spectral interpolation

A script to test the spectral interpolation of function `patchEdgeInt1()` Establish global data struct for the range of various cases.

```
clear all
global patches
nSubP=3
i0=(nSubP+1)/2; % centre-patch index
```

**Test standard spectral interpolation** Test over various numbers of patches, random domain lengths and random ratios.

```
for nPatch=5:10
nPatch=nPatch
Len=10*rand
ratio=0.5*rand
makePatches(@sin,[0,Len],nan,nPatch,0,ratio,nSubP);
kMax=floor((nPatch-1)/2);
```

**Test single field** Set a profile, and evaluate the interpolation.

```
for k=-kMax:kMax
  u0=exp(1i*k*patches.x*2*pi/Len);
  ui=patchEdgeInt1(u0(:));
  normError=norm(ui-u0);
  if abs(normError)>5e-14
    normError=normError
    error(['failed single var interpolation k=' num2str(k)])
  end
end
```

**Test multiple fields** Set a profile, and evaluate the interpolation. For the case of the highest wavenumber, squash the error when the centre-patch values are all zero.

```
for k=1:nPatch/2
  u0=sin(k*patches.x*2*pi/Len);
  v0=cos(k*patches.x*2*pi/Len);
  uvi=patchEdgeInt1([u0(:);v0(:)]);
```

```
58    normuError=norm(uvi(:,:,1)-u0)*norm(u0(i0,:));
59    normvError=norm(uvi(:,:,2)-v0)*norm(v0(i0,:));
60    if abs(normuError)+abs(normvError)>2e-13
61      normuError=normuError, normvError=normvError
62      error(['failed double field interpolation k=' num2str(k)])
63    end
64  end
```

End the for-loop over various geometries.

```
71  end
```

**Now test spectral interpolation on staggered grid** Must have even number of patches for a staggered grid.

```
79  for nPatch=6:2:20
80  nPatch=nPatch
81  ratio=0.5*rand
82  nSubP=3; % of form 4*N-1
83  Len=10*rand
84  makePatches(@simpleWavepde,[0,Len],nan,nPatch,-1,ratio,nSubP);
85  kMax=floor((nPatch/2-1)/2)
```

Identify which microscale grid points are $h$ or $u$ values.

```
91  uPts=mod( bsxfun(@plus,(1:nSubP)',(1:nPatch)) ,2);
92  hPts=find(1-uPts);
93  uPts=find(uPts);
```

Set a profile for various wavenumbers. The capital letter `U` denotes an array of values merged from both $u$ and $h$ fields on the staggered grids.

```
100 fprintf('Single field-pair test.\n')
101 for k=-kMax:kMax
102   U0=nan(nSubP,nPatch);
103   U0(hPts)=rand*exp(+1i*k*patches.x(hPts)*2*pi/Len);
104   U0(uPts)=rand*exp(-1i*k*patches.x(uPts)*2*pi/Len);
105   Ui=patchEdgeInt1(U0(:));
```

```
  normError=norm(Ui-U0);
  if abs(normError)>5e-14
    normError=normError
    error(['failed single sys interpolation k=' num2str(k)])
  end
end
```

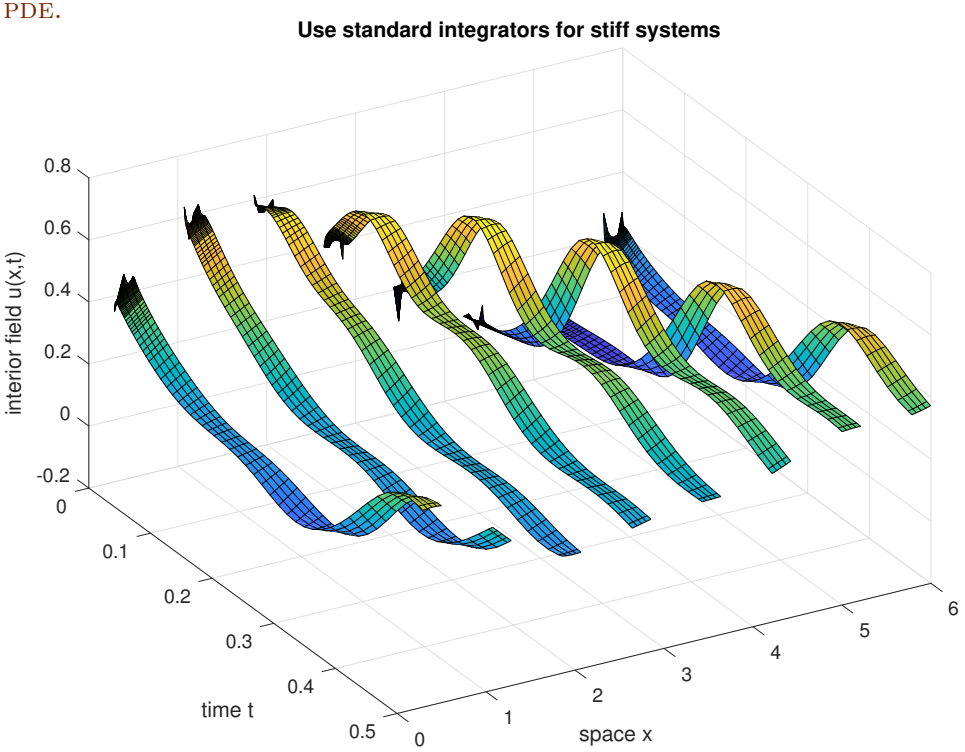
**Test multiple fields** Set a profile, and evaluate the interpolation. For the case of the highest wavenumber zig-zag, squash the error when the alternate centre-patch values are all zero. First shift the $x$-coordinates so that the zig-zag mode is centred on a patch.

```
fprintf('Two field-pairs test.\n')
x0=patches.x((nSubP+1)/2,1);
patches.x=patches.x-x0;
for k=1:nPatch/4
  U0=nan(nSubP,nPatch); V0=U0;
  U0(hPts)=rand*sin(k*patches.x(hPts)*2*pi/Len);
  U0(uPts)=rand*sin(k*patches.x(uPts)*2*pi/Len);
  V0(hPts)=rand*cos(k*patches.x(hPts)*2*pi/Len);
  V0(uPts)=rand*cos(k*patches.x(uPts)*2*pi/Len);
  UVi=patchEdgeInt1([U0(:);V0(:)]);
  normuError=norm(UVi(:,1:2:nPatch,1)-U0(:,1:2:nPatch))*norm(U0(i0,2
      +norm(UVi(:,2:2:nPatch,1)-U0(:,2:2:nPatch))*norm(U0(i0,1:2:nPa
  normuError=norm(UVi(:,1:2:nPatch,2)-V0(:,1:2:nPatch))*norm(V0(i0,2
      +norm(UVi(:,2:2:nPatch,2)-V0(:,2:2:nPatch))*norm(V0(i0,1:2:nPa
  if abs(normuError)+abs(normvError)>2e-13
    normuError=normuError, normvError=normvError
    error(['failed double field interpolation k=' num2str(k)])
  end
end
```

End for-loop over patches

```
end
```

Figure 7: field $u(x,t)$ tests the patch scheme function applied to Burgers' PDE.




**Finish** If no error messages, then all OK.

```
157 fprintf('\nIf you read this, then all tests were passed\n')
```

## 3.4 `BurgersExample`: simulate Burgers' PDE on patches

*Subsection contents*

Figure 7 shows an example simulation in time generated by the patch scheme function applied to Burgers' PDE. The inter-patch coupling is realised by fourth-order interpolation to the patch edges of the mid-patch values.

Establish global data struct for Burgers' PDE solved on $2\pi$-periodic domain, with eight patches, each patch of half-size ratio 0.2, with seven points within each patch, and say fourth order/spectral interpolation provides values for the inter-patch coupling conditions.

```
clear all, close all
global patches
nPatch=8
ratio=0.2
nSubP=7
Len=2*pi;
interpOrd=0
makePatches(@burgerspde,[0,Len],nan,nPatch,interpOrd,ratio,nSubP);
```

Set an initial condition, and check evaluation of the time derivative.

```
u0=0.3*(1+sin(patches.x))+0.05*randn(size(patches.x));
dudt=patchSmooth1(0,u0(:));
```

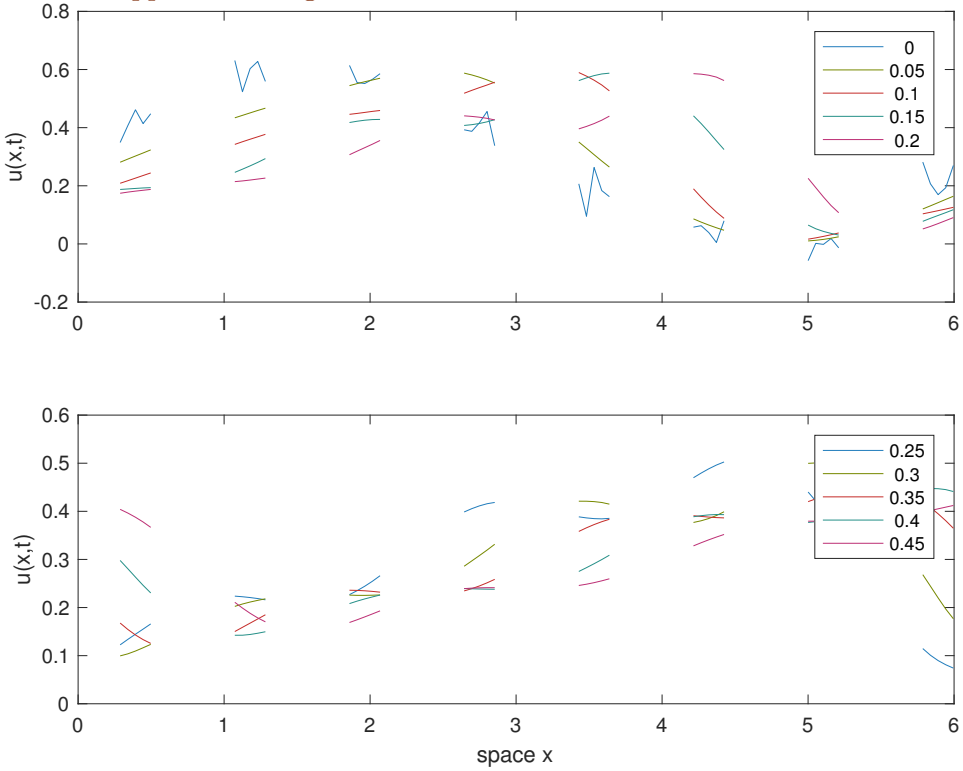
**Conventional integration in time** Integrate in time using standard MATLAB/Octave functions.

```
ts=linspace(0,0.5,60);
if exist('OCTAVE_VERSION', 'builtin') % Octave version
   ucts=lsode(@(u,t) patchSmooth1(t,u),u0(:),ts);
else % Matlab version
   [ts,ucts]=ode15s(@patchSmooth1,ts([1 end]),u0(:));
end
```

Plot the simulation, but here use only the microscale values interior to the patches. Use `nan` in the $x$-edges to leave gaps.

```
figure(1),clf
xs=patches.x; xs([1 end],:)=nan;
```

Figure 8: field $u(x,t)$ tests basic projective integration of the patch scheme function applied to Burgers' PDE.




```
surf(ts,xs(:),ucts')
title('Use standard integrators for stiff systems')
xlabel('time t'), ylabel('space x'), zlabel('interior field u(x,t)')
view(60,40)
print('-depsc2','ps1BurgersCtsU')
```

**Use projective integration** Now wrap around the patch time derivative function, `patchSmooth1`, the projective integration function for patch simulations as illustrated by Figure 8.

Mark that edge of patches are not to be used in the projective extrapolation by setting initial values to NaN.

```
u0([1 end],:)=nan;
```

Set the desired macro- and micro-scale time-steps over the time domain.

```
ts=linspace(0,0.45,10)
bT=2*(ratio*Len/nPatch/(nSubP/2-1))^2;
```

Projectively integrate in time with: DMD projection of rank `nPatch` + 1; guessed microscale time-step `dt`; and guessed numbers of transient and slow steps.

```
addpath('../ProjInt')
[us,tss,uss]=PIRK2(@patchBurst,bT,ts,u0(:));
```

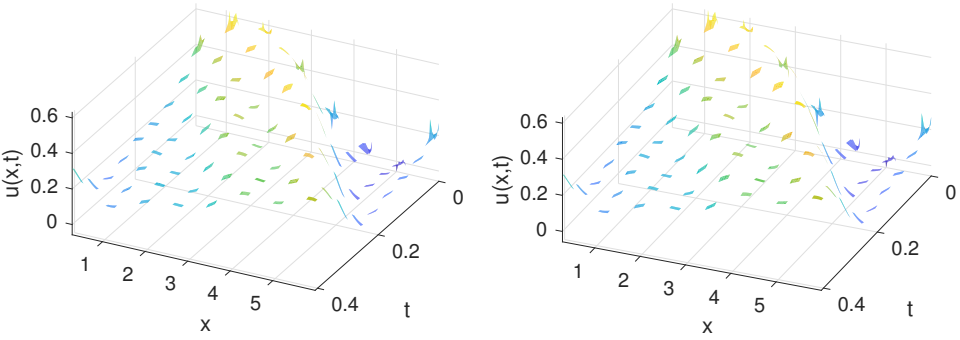
Plot the macroscale predictions to draw Figure 8, in groups of five in a plot.

```
figure(2),clf
k=length(ts); ls=nan(5,ceil(k/5)); ls(1:k)=1:k;
for k=1:size(ls,2)
  subplot(size(ls,2),1,k)
  plot(patches.x(:),us(ls(:,k),:)')
  ylabel('u(x,t)')
  legend(num2str(ts(ls(:,k))'))
end
xlabel('space x')
print('-depsc2','ps1BurgersU')
```

Also plot a surface of the microscale bursts as shown in Figure 9.

```
figure(3),clf
for k=1:2, subplot(2,2,k)
  surf(tss,patches.x(:),uss' ,'EdgeColor','none')
  ylabel('x'),xlabel('t'),zlabel('u(x,t)')
  axis tight, view(121-4*k,45)
end
print('-depsc2','ps1BurgersMicro')
```

Figure 9: stereo pair of the field $u(x,t)$ during each of the microscale bursts used in the projective integration.




### 3.4.1 `burgerspde()`: code the PDE inside patches

This function codes the lattice differential equations inside the patches.

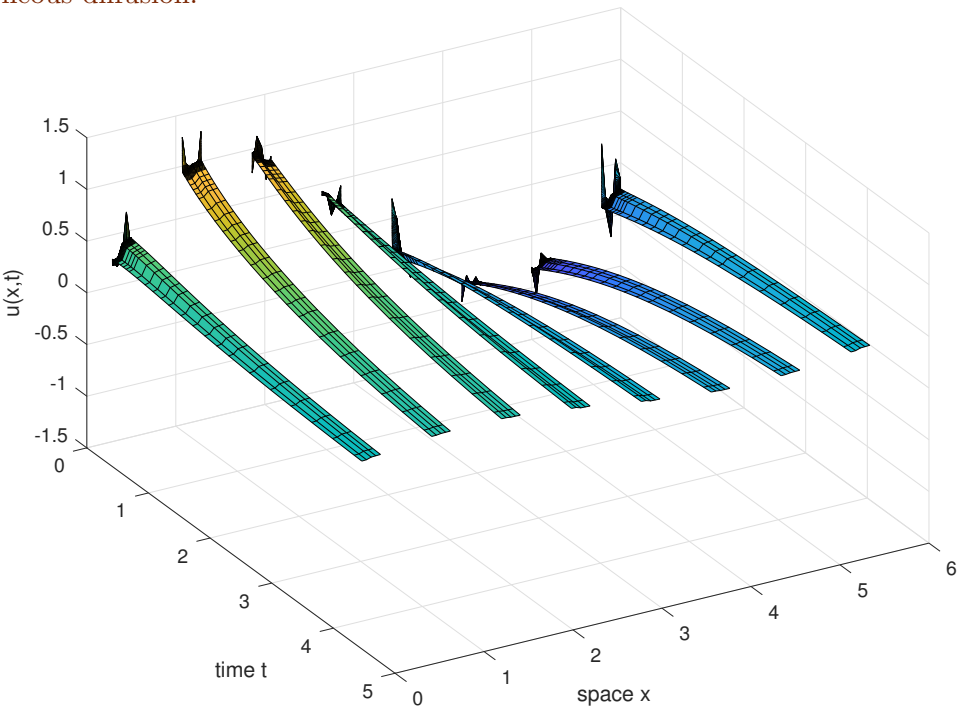
```
158 function ut=burgerspde(t,u,x)
159 dx=x(2)-x(1);
160 ut=nan(size(u));
161 i=2:size(u,1)-1;
162 ut(i,:)=diff(u,2)/dx^2 ...
163    -30*u(i,:).*(u(i+1,:)-u(i-1,:))/(2*dx);
164 end
```

### 3.4.2 `patchBurst()`: code a burst of the patches

```
170 function [ts, us] = patchBurst(ti, ui, bT)
171     [ts, us] = ode23(@patchSmooth1, [ti ti+bT], ui);
172 end
```

Fin.

Figure 10: field $u(x,t)$ tests the patch scheme function applied to heterogeneous diffusion.




## 3.5 HomogenisationExample: simulate heterogeneous diffusion in 1D on patches

Figure 10 shows an example simulation in time generated by the patch scheme function applied to heterogeneous diffusion. The inter-patch coupling is realised by fourth-order interpolation to the patch edges of the mid-patch values.

```
20 function HomogenisationExample
```

Consider a lattice of values $u_i(t)$, with lattice spacing $dx$, and governed by

the heterogeneous diffusion

$$\dot{u}_i = [c_{i-1/2}(u_{i-1} - u_i) + c_{i+1/2}(u_{i+1} - u_i)]/dx^2.$$

The macroscale, homogenised, effective diffusion should be the harmonic mean of these coefficients. Set the desired microscale periodicity, and microscale diffusion coefficients (with subscripts shifted by a half).

```
mPeriod=3
cDiff=2*rand(mPeriod,1)
cHomo=1/mean(1./cDiff)
```

Establish global data struct for heterogeneous diffusion solved on $2\pi$-periodic domain, with eight patches, each patch of half-size 0.1, and the number of points in a patch being one more than an even multiple of the microscale periodicity (which Bunder et al. (2017) showed is accurate). Fourth order interpolation provides values for the inter-patch coupling conditions.

```
global patches
nPatch=8
ratio=0.2
nSubP=2*mPeriod+1
Len=2*pi;
makePatches(@heteroDiff,[0,Len],nan,nPatch,4,ratio,nSubP);
```

Can add to the global data struct `patches` for use by the time derivative function (for example): here include the diffusivity coefficients, repeated to fill up a patch.

```
patches.c=repmat(cDiff,(nSubP-1)/mPeriod,1);
```

Set an initial condition, and test evaluation of the time derivative.

```
u0=sin(patches.x)+0.2*randn(size(patches.x));
dudt=patchSmooth1(0,u0(:));
```

**Conventional integration in time** Integrate in time using standard MATLAB/Octave functions.

```
ts=linspace(0,2/cHomo,60);
if exist('OCTAVE_VERSION', 'builtin') % Octave version
   ucts=lsode(@(u,t) patchSmooth1(t,u),u0(:),ts);
else % Matlab version
   [ts,ucts]=ode15s(@patchSmooth1,ts([1 end]),u0(:));
end
```

Plot the simulation.

```
figure(1),clf
xs=patches.x; xs([1 end],:)=nan;
surf(ts,xs(:),ucts')
xlabel('time t'),ylabel('space x'),zlabel('u(x,t)')
view(60,40)
%print('-depsc2','ps1HomogenisationCtsU')
```

**Use projective integration** Now wrap around the patch time derivative function, `patchSmooth1`, the projective integration function for patch simulations as illustrated by Figure 11.

Mark that edge of patches are not to be used in the projective extrapolation by setting initial values to NaN.

```
u0([1 end],:)=nan;
```

Set the desired macro- and micro-scale time-steps over the time domain.

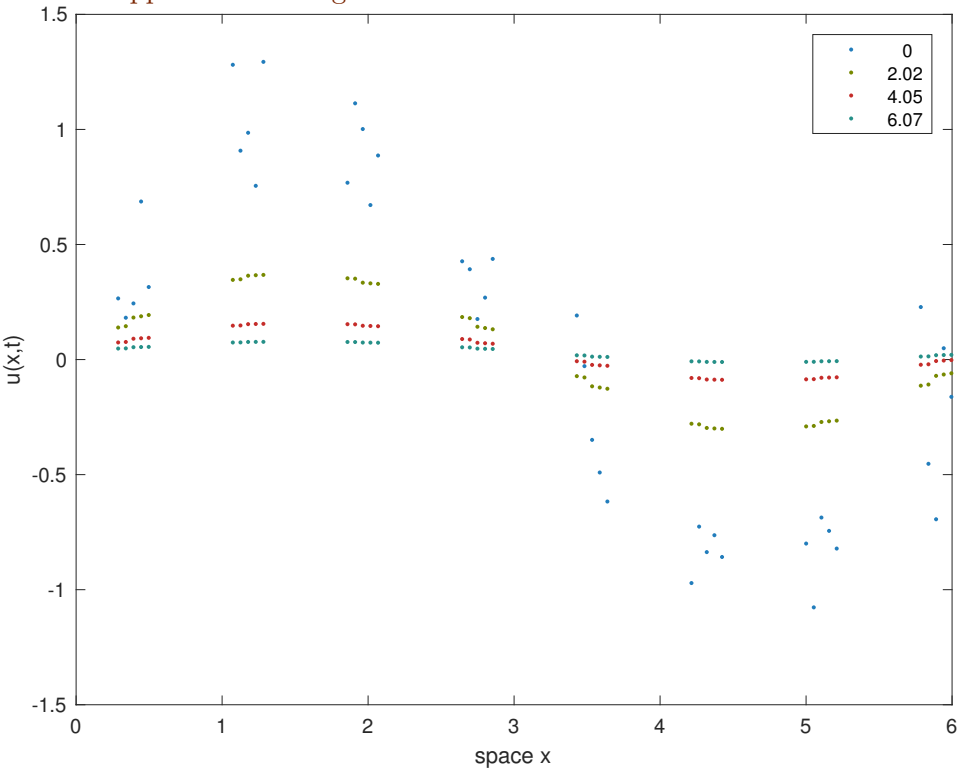
```
ts=linspace(0,3/cHomo,4)
dt=0.4*(ratio*Len/nPatch/(nSubP/2-1))^2/max(cDiff);
```

Projectively integrate in time with: DMD projection of rank `nPatch` + 1; guessed microscale time-step `dt`; and guessed numbers of transient and slow steps.

```
addpath('../ProjInt')
[us,uss,tss]=projInt1(@patchSmooth1,u0(:),ts ...
    ,nPatch+1,dt,[20 nPatch*2]);
```

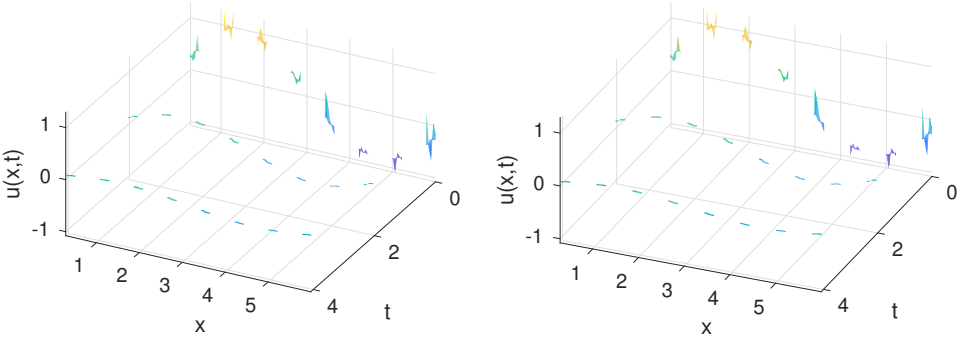
Plot the macroscale predictions to draw Figure 11, in groups of five in a plot.

Figure 11: field $u(x,t)$ tests basic projective integration of the patch scheme function applied to heterogeneous diffusion.




```
figure(2),clf
k=length(ts); ls=nan(5,ceil(k/5)); ls(1:k)=1:k;
for k=1:size(ls,2)
  subplot(size(ls,2),1,k)
  plot(xs(:),us(:,ls(~isnan(ls(:,k)),k)),'.')
  ylabel('u(x,t)')
  legend(num2str(ts(ls(~isnan(ls(:,k)),k))',3))
end
xlabel('space x')
%print('-depsc2','ps1HomogenisationU')
```

Figure 12: stereo pair of the field $u(x,t)$ during each of the microscale bursts used in the projective integration.




Also plot a surface of the microscale bursts as shown in Figure 12.

```
tss(end)=nan; %omit end time-point
figure(3),clf
for k=1:2, subplot(2,2,k)
  surf(tss,xs(:),uss,'EdgeColor','none')
  ylabel('x'),xlabel('t'),zlabel('u(x,t)')
  axis tight, view(121-4*k,45)
end
%print('-depsc2','ps1HomogenisationMicro')
```

End the main function

```
end
```

This function codes the lattice heterogeneous diffusion inside the patches.

```
function ut=heteroDiff(t,u,x)
global patches
dx=patches.x(2)-patches.x(1);
ut=nan(size(u));
i=2:size(u,1)-1;
ut(i,:)=diff(bsxfun(@times,patches.c,diff(u)))/dx^2 ;
end
```

Fin.

## 3.6 `waterWaveExample`: simulate a water wave PDE on patches

*Subsection contents*



Figure 13 shows an example simulation in time generated by the patch scheme function applied to a simple wave PDE. The inter-patch coupling is realised by third-order interpolation to the patch edges of the mid-patch values.

This section describes the nonlinear microscale simulator of the nonlinear shallow water wave PDE derived from the Smagorinski model of turbulent flow (Cao & Roberts 2012, 2016*a*). Often, wave-like systems are written in terms of two conjugate variables, for example, position and momentum density, electric and magnetic fields, and water depth $h(x,t)$ and mean lateral velocity $u(x,t)$ as herein. The approach applies to any wave-like system in the form

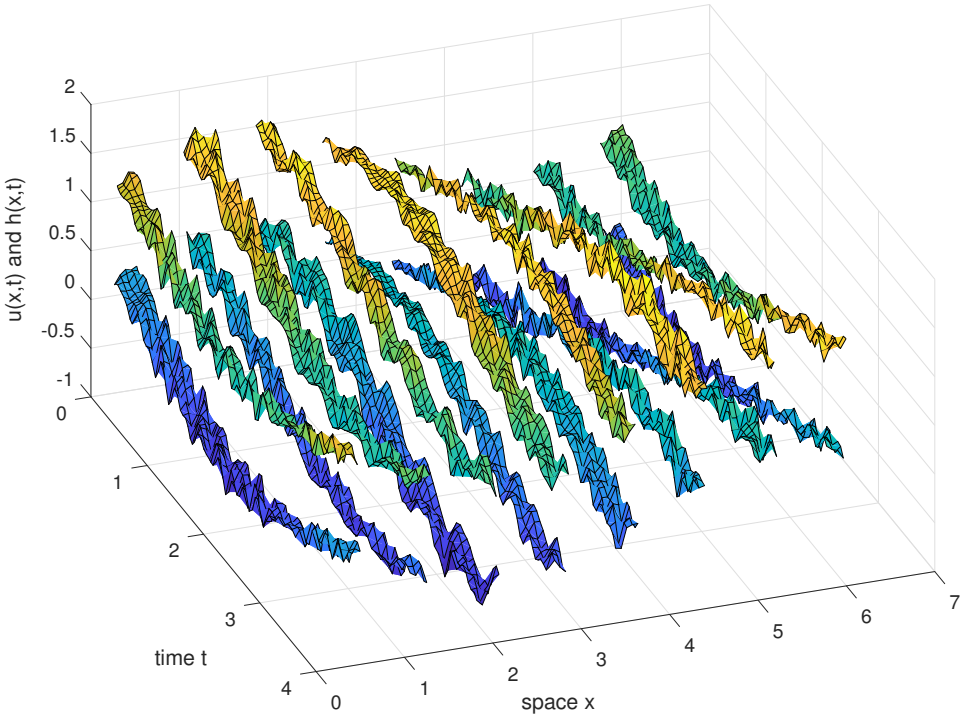
$$\frac{\partial h}{\partial t} = -c_1\frac{\partial u}{\partial x} + f_1[h,u] \quad \text{and} \quad \frac{\partial u}{\partial t} = -c_2\frac{\partial h}{\partial x} + f_2[h,u], \tag{1}$$

where the brackets indicate that the nonlinear functions $f_\ell$ may involve various spatial derivatives of the fields $h(x,t)$ and $u(x,t)$. Specifically, this section invokes a nonlinear Smagorinski model of turbulent shallow water (Cao & Roberts 2012, 2016*a*, e.g.) along an inclined flat bed: let $x$ measure position along the bed and in terms of fluid depth $h(x,t)$ and depth-averaged lateral velocity $u(x,t)$ the model PDEs are

$$\frac{\partial h}{\partial t} = -\frac{\partial (hu)}{\partial x}\,, \tag{2a}$$

$$\frac{\partial u}{\partial t} = 0.985\left(\tan\theta - \frac{\partial h}{\partial x}\right) - 0.003\frac{u|u|}{h} - 1.045u\frac{\partial u}{\partial x} + 0.26h|u|\frac{\partial^2 u}{\partial x^2}\,, \tag{2b}$$

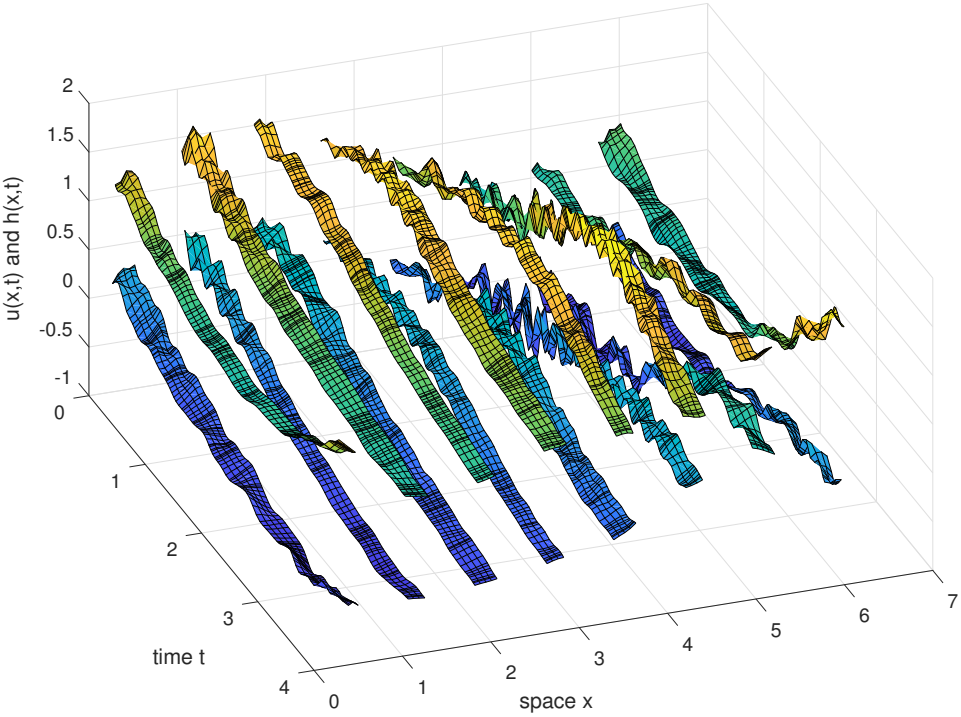
Figure 13: water depth $h(x,t)$ (above) and velocity field $u(x,t)$ (below) of the gap-tooth scheme function applied to simple wave PDE. A random component to the initial condition has long lasting effects on the simulation—but the macroscale wave still propagates.




where $\tan\theta$ is the slope of the bed. Equation (2a) represents conservation of the fluid. The momentum PDE (2b) represents the effects of turbulent bed drag $u|u|/h$, self-advection $u\partial u/\partial x$, nonlinear turbulent dispersion $h|u|\partial^2 u/\partial x^2$, and gravitational hydrostatic forcing $\tan\theta - \partial h/\partial x$. Figure 14 shows one simulation of this system—for the same initial condition as Figure 13.

For such wave systems, let's try a staggered microscale grid and staggered macroscale patches as introduced in Figures 3 and 4, respectively, by Cao & Roberts (2016*b*).

Figure 14: water depth $h(x,t)$ (above) and velocity field $u(x,t)$ (below) of the gap-tooth scheme the shallow water wave PDEs (2). A random component decays where the speed is non-zero.




```
56 function waterWaveExample
```

Establish global data struct for the PDEs (2) solved on $2\pi$-periodic domain, with eight patches, each patch of half-size 0.2, with eleven points within each patch, and third-order interpolation provides values for the inter-patch coupling conditions (higher order interpolation is smoother for these smooth initial conditions).

```
63 global patches
64 nPatch=8
65 ratio=0.2
```

```
66 nSubP=11 % of form 4*?-1
67 Len=2*pi;
68 makePatches(@simpleWavepde,[0,Len],nan,nPatch,3,ratio,nSubP);
```

Identify which microscale grid points are $h$ or $u$ values. Also store them in the struct `patches` for use by the time derivative function.

```
76 uPts=mod( bsxfun(@plus,(1:nSubP)',(1:nPatch)) ,2);
77 hPts=find(1-uPts);
78 uPts=find(uPts);
79 patches.hPts=hPts; patches.uPts=uPts;
```

Set an initial condition of a progressive wave, and check evaluation of the time derivative. The capital letter `U` denotes an array of values merged from both $u$ and $h$ fields on the staggered grids.

```
87 U0=nan(nSubP,nPatch);
88 U0(hPts)=1+0.5*sin(patches.x(hPts));
89 U0(uPts)=0+0.5*sin(patches.x(uPts));
90 U0=U0+0.05*randn(nSubP,nPatch);
91 dUdt0=patchSmooth1(0,U0(:));% check
92 %dUdt0=reshape(dUdt0,nSubP,nPatch)
```

**Conventional integration in time** Integrate in time using standard MATLAB/Octave functions the two cases of the simple wave equations and the water wave equations.

```
100 for k=1:2
```

When using `ode15s` we subsample the results because the sub-grid scale waves do not dissipate and so the integrator takes very small time steps for all time.

```
106 ts=linspace(0,4,41);
107 if exist('OCTAVE_VERSION', 'builtin') % Octave version
108    Ucts=lsode(@(u,t) patchSmooth1(t,u),U0(:),ts);
109 else % Matlab version
110    [ts,Ucts]=ode15s(@patchSmooth1,ts([1 end]),U0(:));
```

```
111     ts=ts(1:5:end);
112     Ucts=Ucts(1:5:end,:);
113 end
```

Plot the simulation.

```
119 figure(k),clf
120 xs=patches.x; xs([1 end],:)=nan;
121 surf(ts,xs(patches.hPts),Ucts(:,patches.hPts)'),hold on
122 surf(ts,xs(patches.uPts),Ucts(:,patches.uPts)'),hold off
123 xlabel('time t'),ylabel('space x'),zlabel('u(x,t) and h(x,t)')
124 view(70,45)
```

Print the graph.

```
130 if k==1, print('-depsc2','ps1WaveCtsUH')
131 else print('-depsc2','ps1WaterWaveCtsUH')
132 end
```

Now, change to the Smagorinski turbulence model (2) of shallow water flow, keeping other parameters and the initial condition the same. And end the loop to redo the simulation.

```
140 patches.fun=@waterWavepde;
141 dUdt0=patchSmooth1(0,U0(:));%check
142 end
```

**Use projective integration** As yet a simple implementation appears to fail, so it needs more exploration and thought.

End the main function

```
222 end
```

### 3.6.1 Simple wave PDE

This function codes the staggered lattice equation inside the patches for the simple wave PDE system $h_t = -u_x$ and $u_t = -h_x$. Here code for a staggered

microscale grid of staggered macroscale patches: the array

$$U_{ij} = \begin{cases} u_{ij} & i+j \text{ even}, \\ h_{ij} & i+j \text{ odd}. \end{cases}$$

The output `Ut` contains the merged time derivatives of the two staggered fields. So set the micro-grid spacing and reserve space for time derivatives.

```
241 function Ut=simpleWavepde(t,U,x)
242 global patches
243 dx=x(2)-x(1);
244 Ut=nan(size(U));
245 ht=Ut;
```

Compute the PDE derivatives at points internal to the patches.

```
251 i=2:size(U,1)-1;
```

Here 'wastefully' compute time derivatives for both PDEs at all grid points—for 'simplicity'—and then merges the staggered results. Since $\dot{h}_{ij} \approx -(u_{i+1,j} - u_{i-1,j})/(2\cdot dx) = -(U_{i+1,j} - U_{i-1,j})/(2\cdot dx)$ as adding/subtracting one from the index of a $h$-value is the location of the neighbouring $u$-value on the staggered micro-grid.

```
258 ht(i,:)=-(U(i+1,:)-U(i-1,:))/(2*dx);
```

Since $\dot{u}_{ij} \approx -(h_{i+1,j} - h_{i-1,j})/(2\cdot dx) = -(U_{i+1,j} - U_{i-1,j})/(2\cdot dx)$ as adding/subtracting one from the index of a $u$-value is the location of the neighbouring $h$-value on the staggered micro-grid.

```
264 Ut(i,:)=-(U(i+1,:)-U(i-1,:))/(2*dx);
```

Then overwrite the unwanted $\dot{u}_{ij}$ with the corresponding wanted $\dot{h}_{ij}$.

```
270 Ut(patches.hPts)=ht(patches.hPts);
271 end
```

### 3.6.2 Water wave PDE

This function codes the staggered lattice equation inside the patches for the nonlinear wave-like PDE system (2). As before, set the micro-grid spacing, reserve space for time derivatives, and index the internal points of the micro-grid.

```
function Ut=waterWavepde(t,U,x)
global patches
dx=x(2)-x(1);
Ut=nan(size(U));
ht=Ut;
i=2:size(U,1)-1;
```

Need to estimate $h$ at all the $u$-points, so into `V` use averages, and linear extrapolation to patch-edges.

```
ii=i(2:end-1);
V=Ut;
V(ii,:)=(U(ii+1,:)+U(ii-1,:))/2;
V(1:2,:)=2*U(2:3,:)-V(3:4,:);
V(end-1:end,:)=2*U(end-2:end-1,:)-V(end-3:end-2,:);
```

Then estimate $\partial hu/\partial x$ from $u$ and the interpolated $h$ at the neighbouring micro-grid points.

```
ht(i,:)=-(U(i+1,:).*V(i+1,:)-U(i-1,:).*V(i+1,:))/(2*dx);
```

Correspondingly estimate the terms in the momentum PDE: $u$-values in $\mathtt{U}_i$ and $\mathtt{V}_{i\pm1}$; and $h$-values in $\mathtt{V}_i$ and $\mathtt{U}_{i\pm1}$.

```
Ut(i,:)=-0.985*(U(i+1,:)-U(i-1,:))/(2*dx) ...
    -0.003*U(i,:).*abs(U(i,:)./V(i,:)) ...
    -1.045*U(i,:).*(V(i+1,:)-V(i-1,:))/(2*dx) ...
    +0.26*abs(V(i,:).*U(i,:)).*(V(i+1,:)-2*U(i,:)+V(i-1,:))/dx^2/2;
```

where the mysterious division by two in the 2nd derivative is due to using the averaged values of $u$ in the estimate:

$$\begin{aligned} u_{xx} &\approx \frac{1}{4\delta^2}(u_{i-2} - 2u_i + u_{i+2}) \\ &= \frac{1}{4\delta^2}(u_{i-2} + u_i - 4u_i + u_i + u_{i+2}) \\ &= \frac{1}{2\delta^2}\left(\frac{u_{i-2}+u_i}{2} - 2u_i + \frac{u_i+u_{i+2}}{2}\right) \\ &= \frac{1}{2\delta^2}\left(\bar{u}_{i-1} - 2u_i + \bar{u}_{i+1}\right). \end{aligned}$$

Then overwrite the unwanted $\dot{u}_{ij}$ with the corresponding wanted $\dot{h}_{ij}$.

```
325 Ut(patches.hPts)=ht(patches.hPts);
326 end
```

Fin.

## 3.7 To do

- Testing is so far only qualitative. Need to be quantitative.
- Multiple space dimensions.
- Heterogeneous microscale via averaging regions.
- Parallel processing versions.
- ??
- Adapt to maps in micro-time?

## References

Bunder, J. E., Roberts, A. J. & Kevrekidis, I. G. (2017), 'Good coupling for the multiscale patch scheme on systems with microscale heterogeneity', *J. Computational Physics* **accepted 2 Feb 2017**.

Bunder, J., Roberts, A. J. & Kevrekidis, I. G. (2016), 'Accuracy of patch dynamics with mesoscale temporal coupling for efficient massively parallel simulations', *SIAM Journal on Scientific Computing* **38**(4), C335–C371.

Calderon, C. P. (2007), 'Local diffusion models for stochastic reacting systems: estimation issues in equation-free numerics', *Molecular Simulation* **33**(9—10), 713—731.

Cao, M. & Roberts, A. J. (2012), Modelling 3d turbulent floods based upon the smagorinski large eddy closure, *in* P. A. Brandner & B. W. Pearce, eds, '18th Australasian Fluid Mechanics Conference'. http://people.eng.unimelb.edu.au/imarusic/proceedings/18/70%20-%20Cao.pdf

Cao, M. & Roberts, A. J. (2016*a*), 'Modelling suspended sediment in environmental turbulent fluids', *J. Engrg. Maths* **98**(1), 187–204.

Cao, M. & Roberts, A. J. (2016*b*), 'Multiscale modelling couples patches of nonlinear wave-like simulations', *IMA J. Applied Maths.* **81**(2), 228–254.

Gear, C. W. & Kevrekidis, I. G. (2003*a*), 'Projective methods for stiff differential equations: Problems with gaps in their eigenvalue spectrum', *SIAM Journal on Scientific Computing* **24**(4), 1091–1106. http://link.aip.org/link/?SCE/24/1091/1

Gear, C. W. & Kevrekidis, I. G. (2003*b*), 'Telescopic projective methods for parabolic differential equations', *Journal of Computational Physics* **187**, 95–109.

Givon, D., Kevrekidis, I. G. & Kupferman, R. (2006), 'Strong convergence of projective integration schemes for singularly perturbed stochastic differential systems', *Comm. Math. Sci.* **4**(4), 707–729.

Gustafsson, B. (1975), 'The convergence rate for difference approximations

to mixed initial boundary value problems', *Mathematics of Computation* **29**(10), 396–406.

Hyman, J. M. (2005), 'Patch dynamics for multiscale problems', *Computing in Science & Engineering* **7**(3), 47–53. http://scitation.aip.org/content/aip/journal/cise/7/3/10.1109/MCSE.2005.57

Kevrekidis, I. G., Gear, C. W. & Hummer, G. (2004), 'Equation-free: the computer-assisted analysis of complex, multiscale systems', *A. I. Ch. E. Journal* **50**, 1346–1354.

Kevrekidis, I. G., Gear, C. W., Hyman, J. M., Kevrekidis, P. G., Runborg, O. & Theodoropoulos, K. (2003), 'Equation-free, coarse-grained multiscale computation: enabling microscopic simulators to perform system level tasks', *Comm. Math. Sciences* **1**, 715–762.

Kevrekidis, I. G. & Samaey, G. (2009), 'Equation-free multiscale computation: Algorithms and applications', *Annu. Rev. Phys. Chem.* **60**, 321—44.

Kutz, J. N., Brunton, S. L., Brunton, B. W. & Proctor, J. L. (2016), *Dynamic Mode Decomposition: Data-driven Modeling of Complex Systems*, number 149 *in* 'Other titles in applied mathematics', SIAM, Philadelphia.

Liu, P., Samaey, G., Gear, C. W. & Kevrekidis, I. G. (2015), 'On the acceleration of spatially distributed agent-based computations: A patch dynamics scheme', *Applied Numerical Mathematics* **92**, 54–69. http://www.sciencedirect.com/science/article/pii/S0168927414002086

Plimpton, S., Thompson, A., Shan, R., Moore, S., Kohlmeyer, A., Crozier, P. & Stevens, M. (2016), Large-scale atomic/molecular massively parallel simulator, Technical report, http://lammps.sandia.gov.

Roberts, A. J. & Kevrekidis, I. G. (2007), 'General tooth boundary conditions for equation free modelling', *SIAM J. Scientific Computing* **29**(4), 1495–1510.

Roberts, A. J. & Li, Z. (2006), 'An accurate and comprehensive model of thin fluid flows with inertia on curved substrates', *J. Fluid Mech.* **553**, 33–73.

Samaey, G., Kevrekidis, I. G. & Roose, D. (2005), 'The gap-tooth scheme for homogenization problems', *Multiscale Modeling and Simulation* **4**, 278–306.

Samaey, G., Roberts, A. J. & Kevrekidis, I. G. (2010), Equation-free computation: an overview of patch dynamics, *in* J. Fish, ed., 'Multiscale methods: bridging the scales in science and engineering', Oxford University Press, chapter 8, pp. 216–246.

Samaey, G., Roose, D. & Kevrekidis, I. G. (2006), 'Patch dynamics with buffers for homogenization problems', *J. Comput Phys.* **213**, 264–287.

Svard, M. & Nordstrom, J. (2006), 'On the order of accuracy for difference approximations of initial-boundary value problems', *Journal of Computational Physics* **218**, 333–352.