# Equation-Free function toolbox for Matlab/Octave

A. J. Roberts* John Maclean† et al.‡

October 29, 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

**1 Introduction** **4**

**2 Projective integration of deterministic ODEs** **5**

2.1 egPIMM: Example projective integration of Michaelis–Menton kinetics . . . . . 7

2.1.1 Invoke projective integration . . . . . 7

2.1.2 Code a burst of Michaelis–Menten enzyme kinetics . . 9

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

†http://www.adelaide.edu.au/directory/john.maclean

‡Appear here for your contribution.

2.2 PIRK2(): projective integration of second order accuracy . . . 10
2.2.1 If no arguments, then execute an example . . . . . 13
2.2.2 The projective integration code . . . . . 14
2.2.3 If no output specified, then plot simulation . . . . . 18
2.3 Example: PI using Runge–Kutta macrosolvers . . . . . 19
2.4 PIG(): Projective Integration with a General macrosolver . . 22
2.5 Example 2: PI using General macrosolvers . . . . . 27
2.6 Minor functions . . . . . 30
2.6.1 cdmc() . . . . . 30
2.6.2 bbgen() . . . . . 31
2.7 Explore: PI using constraint-defined manifold computing . . 32
2.8 PIRK4(): projective integration of fourth order accuracy . . . 33
2.8.1 The projective integration code . . . . . 36
2.8.2 If no output specified, then plot simulation . . . . . 40
2.9 To do/discuss . . . . . 41

**3 Patch scheme for given microscale discrete space system 43**
3.1 configPatches1(): configures spatial patches in 1D . . . . . 44
3.1.1 If no arguments, then execute an example . . . . . 45
3.1.2 The code to make patches . . . . . 47
3.2 patchSmooth1(): interface to time integrators . . . . . 49
3.3 patchEdgeInt1(): sets edge values from interpolation over the macroscale . . . . . 50
3.4 BurgersExample: simulate Burgers' PDE on patches . . . . . 55
3.4.1 Script code to simulate a micro-scale space-time map . 56
3.4.2 burgersMap(): discretise the PDE microscale . . . . . 59
3.4.3 burgerBurst(): code a burst of the patch map . . . . 60
3.5 HomogenisationExample: simulate heterogeneous diffusion in 1D ... . . . . 61
3.5.1 Script to simulate via stiff or projective integration . . 61
3.5.2 heteroDiff(): heterogeneous diffusion . . . . . 66
3.5.3 heteroBurst(): a burst of heterogeneous diffusion . . 66
3.6 waterWaveExample: simulate a water wave PDE on patches . 67
3.6.1 Script code to simulate wave systems . . . . . 69
3.6.2 simpleWavePDE(): simple wave PDE . . . . . 72

3.6.3 `waterWavePDE()`: water wave PDE . . . . . 73
3.7 To do . . . . . 74
3.8 Miscellaneous tests . . . . . 74
3.8.1 `patchEdgeInt1test`: test the spectral interpolation . 74

**A Create, document and test algorithms** **79**

**B Aspects of developing a 'toolbox' for patch dynamics** **81**
B.1 Macroscale grid . . . . . 81
B.2 Macroscale field variables . . . . . 81
B.3 Boundary and coupling conditions . . . . . 82
B.4 Mesotime communication . . . . . 83
B.5 Projective integration . . . . . 83
B.6 Lift to many internal modes . . . . . 84
B.7 Macroscale closure . . . . . 84
B.8 Exascale fault tolerance . . . . . 85
B.9 Link to established packages . . . . . 85

# 1 Introduction

**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. Appendix A outlines some details for contributors.

# 2 Projective integration of deterministic ODEs

*Section contents*


2.1 egPIMM: Example projective integration of Michaelis–Menton kinetics . . . 7
2.1.1 Invoke projective integration . . . 7
2.1.2 Code a burst of Michaelis–Menten enzyme kinetics . . . 9
2.2 PIRK2(): projective integration of second order accuracy . . . 10
2.2.1 If no arguments, then execute an example . . . 13
2.2.2 The projective integration code . . . 14
2.2.3 If no output specified, then plot simulation . . . 18
2.3 Example: PI using Runge–Kutta macrosolvers . . . 19
2.4 PIG(): Projective Integration with a General macrosolver . . . 22
2.5 Example 2: PI using General macrosolvers . . . 27
2.6 Minor functions . . . 30
2.6.1 cdmc() . . . 30
2.6.2 bbgen() . . . 31
2.7 Explore: PI using constraint-defined manifold computing . . . 32
2.8 PIRK4(): projective integration of fourth order accuracy . . . 33
2.8.1 The projective integration code . . . 36
2.8.2 If no output specified, then plot simulation . . . 40
2.9 To do/discuss . . . 41


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 complex 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 macroscale 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 projects 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*

2.1.1 Invoke projective integration . . . . . 7
2.1.2 Code a burst of Michaelis–Menten enzyme kinetics . . 9

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

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

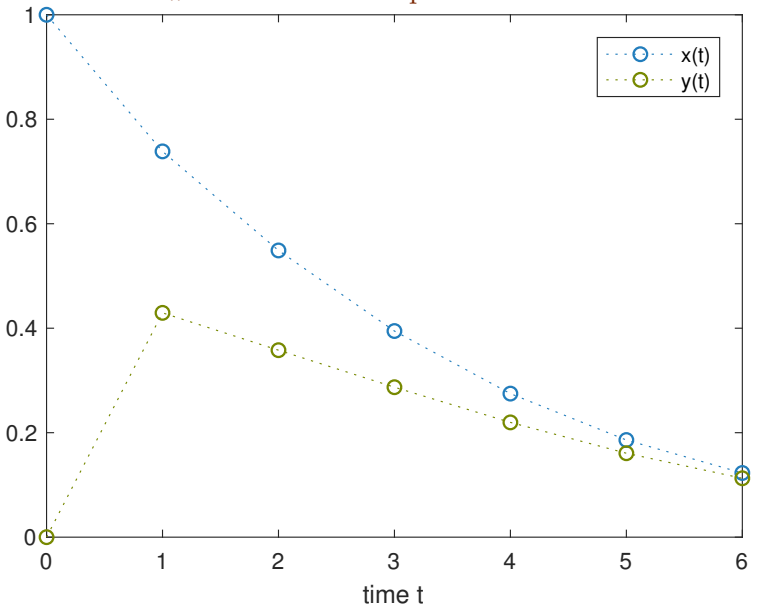
As illustrated in Figure 2, the slow variable $x(t)$ evolves on a time scale of one, whereas the fast variable $y(t)$ evolves on a time scale of the small parameter $\epsilon$.

### 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.

```
31 clear all, close all
32 global epsilon
33 epsilon = 0.1
```

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




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 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).

```
48 ts = 0:6
49 xs = PIRK2(@MMburst, 2*epsilon, ts, [1;0])
50 plot(ts,xs,'o:')
51 xlabel('time t'), legend('x(t)','y(t)')
52 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 the slow manifold we plot some microscale data in Figure 2. Two further output variables provide this microscale burst information.

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

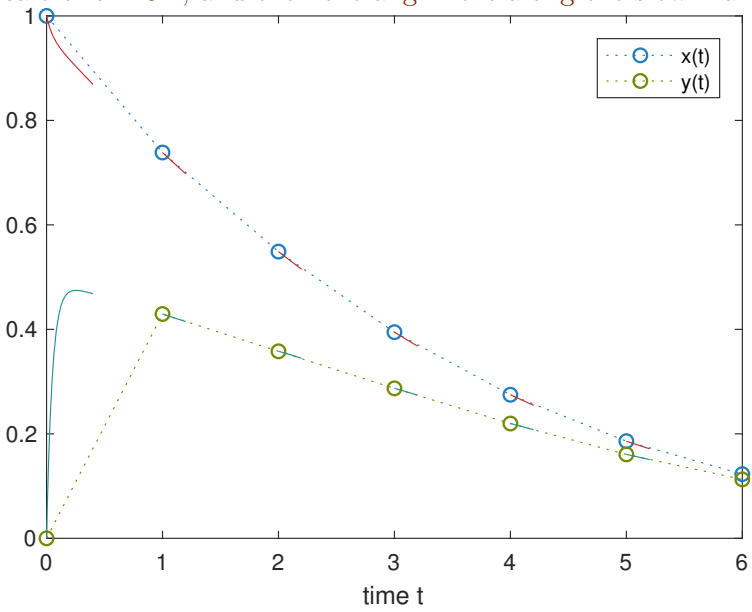
**Optional: simulate backwards in time** Figure 3 shows that projective integration even simulates backwards in time along the slow manifold using short forward bursts. Such backwards macroscale simulations succeed despite the fast variable $y(t)$, when backwards in time, being viciously unstable. However,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

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.



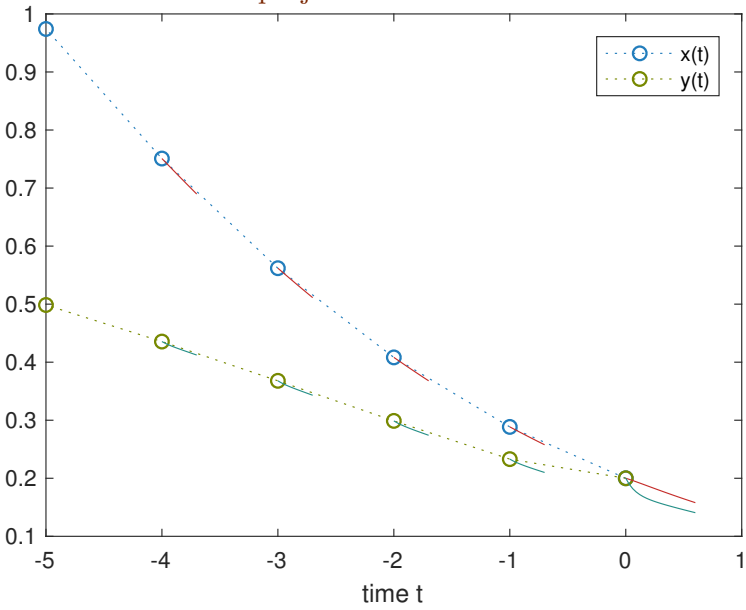

```
function [ts, xs] = MMburst(ti, xi, bT)
    global epsilon
    dMMdt = @(t,x) [ -x(1)+(x(1)+0.5)*x(2)
          1/epsilon*( x(1)-(x(1)+1)*x(2) ) ];
    [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
end
```

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

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

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

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 $\epsilon = 0.1$.




**Input** If there are no input arguments, then this function applies itself to the Michaelis–Menton example: see the code in Section 2.2.1 as a basic template of how to use.

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

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

  – Inputs: `tStart`, the start time of a burst of simulation; `xStart`, the row $n$-vector of the starting state; `tSim`, 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** If there are no output arguments specified, then a plot is drawn of the computed solution `x` versus `tSpan`.

- `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 optional outputs of the microscale bursts.

- `tms`, optional, is an $L$ dimensional column vector containing microscale times of burst simulations, each burst separated by `NaN`;
- `xms`, optional, 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`, optional, 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; the rm.x 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, optional, a struct containing the Projective Integration estimates of the slow vector field.
  - svf.t is a $2\ell$ dimensional column vector containing all times at which the Projective Integration scheme is extrapolated along microsolver data to form a macrostep.
  - svf.dx is a $2\ell \times n$ array containing the estimated slow vector field.

### 2.2.1 If no arguments, then execute an example

```
113 if nargin==0
```

**Example code for Michaelis–Menton dynamics** The Michaelis–Menten enzyme kinetics is expressed as a singularly perturbed system of differential equations for $x(t)$ and $y(t)$ (encoded in function MMburst):

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

With initial conditions $x(0) = 1$ and $y(0) = 0$, the following code computes and plots a solution over time $0 \le t \le 6$ for parameter $\epsilon = 0.05$ using bursts of length $3\epsilon$.

```
130 epsilon = 0.05
131 ts = 0:6
132 [x,tms,xms,rm,svf] = PIRK2(@MMburst, 3*epsilon, ts, [1;0]);
133 figure, plot(ts,x,'o:',tms,xms)
134 title('Projective integration of Michaelis--Menten enzyme kinetics')
135 xlabel('time t'), legend('x(t)','y(t)')
```

Upon finishing execution of the example, exit this function.

```
141 return
142 end%if no arguments
```

**Example function code for a burst of ODEs** Integrate a burst of length `bT` of the ODEs for the Michaelis–Menten enzyme kinetics at parameter $\epsilon$ inherited from above. Code ODEs in function `dMMdt` with variables $x =$ `x(1)` and $y =$ `x(2)`. Starting at time `ti`, and state `xi` (row), here code the midpoint integration scheme.

```
156 function [ts, xs] = MMburst(ti, xi, bT)
157     dMMdt = @(t,x) [ -x(1)+(x(1)+0.5)*x(2)
158           1/epsilon*( x(1)-(x(1)+1)*x(2) ) ];
159     ts = linspace(ti,ti+bT,ceil(bT*5/epsilon))';
160     dt = ts(2)-ts(1);
161     xs = nan(length(ts),length(xi));
162     xs(1,:) = xi;
163     for j = 1:length(ts)-1
164       xMidpoint = xs(j,:)+dt/2*dMMdt(ts(j),xs(j,:)).';
165       xs(j+1,:) = xs(j,:)+dt*dMMdt(ts(j)+dt/2,xMidpoint).';
166     end
167 end
```

### 2.2.2 The projective integration code

Determine the number of time steps and preallocate storage for macroscale estimates.

```
181 nT=length(tSpan);
182 x=nan(nT,length(x0));
```

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

```
190 nArgs=nargout();
191 saveMicro = (nArgs>1);
192 saveFullMicro = (nArgs>3);
193 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.

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

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

```
215 tSpan(1) = tSpan(1)+bT;
216 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.

```
226 if saveMicro
227     tms = cell(nT,1);
228     xms = cell(nT,1);
229     tms{1} = reshape(relax_t,[],1);
230     xms{1} = relax_x0;
231     if saveFullMicro
232         rm.t = cell(nT,1);
233         rm.x = cell(nT,1);
234         if saveSvf
235             svf.t = nan(2*nT-2,1);
236             svf.dx = nan(2*nT-2,length(x0));
237         end
238     end
239 end
```

### Loop over the macroscale time steps

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

If two applications of the microsolver would cover the entire macroscale time-step, then do so (setting some internal states to `NaN`); else proceed to projective step.

```
    if 2*abs(bT)>=abs(tSpan(jT)-T) & bT*(tSpan(jT)-T)>0
        [t1,xm1] = solver(T, x(jT-1,:), tSpan(jT)-T);
        x(jT,:) = xm1(end,:);
        t2=nan; xm2=nan(1,size(xm1,2));
        dx1=xm2; dx2=xm2;
    else
```

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.

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

Check for round-off error.

```
    xt=[reshape(t1(end-1:end),[],1) xm1(end-1:end,:)];
    roundingTol=1e-8;
    if norm(diff(xt))/norm(xt,'fro') < roundingTol
    warning(['significant round-off error in 1st projection at T=' n
    end
```

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

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

```
303     xint = xm1(end,:) + (Dt-bT)*dx1;
304     [t2,xm2] = solver(T+Dt, xint, bT);
305     del = t2(end)-t2(end-1);
306     dx2 = (xm2(end,:)-xm2(end-1,:))/del;
```

Check for round-off error.

```
312     xt=[reshape(t2(end-1:end),[],1) xm2(end-1:end,:)];
313     if norm(diff(xt))/norm(xt,'fro') < roundingTol
314     warning(['significant round-off error in 2nd projection at T=' n
315     end
```

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

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

Now end the if-statement that tests whether a projective step saves simulation time.

```
331     end
```

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.

```
341     if saveMicro
342         tms{jT} = [reshape(t1,[],1); nan];
343         xms{jT} = [xm1; nan(1,size(xm1,2))];
```

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

```
351         if saveFullMicro
352             rm.t{jT} = [reshape(t2,[],1); nan];
353             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.

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

Terminate the main loop:

```
end
```

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

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

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

```
if saveMicro
    tms = cell2mat(tms);
    xms = cell2mat(xms);
    if saveFullMicro
        rm.t = cell2mat(rm.t);
        rm.x = cell2mat(rm.x);
    end
end
```

### 2.2.3 If no output specified, then plot simulation

```
if nArgs==0
    figure, plot(tSpan,x,'o:')
    title('Projective Simulation with PIRK2')
end
```

This concludes `PIRK2()`.

```
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 function $f(t,x) = \mathbf{A}\vec{x} + \vec{b}$, 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 variables. The function `gen_linear_system()` 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 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 is run by specifying an initial time and state, and a duration to simulate for. All the details of the microsolver must be set by the user. We generate and save a black box microsolver.

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

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

```
69 tSpan=0: 1 : 30;
70 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.

```
79 [x, tms, xms, rm] = PIRK4(solver, bT, tSpan, IC);
```

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

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

Figure 4 plots the output.

```
98 tmsr = rm.t; xmsr = rm.x;
99 clf()
100 hold on
101 PI_sol=plot(tSpan,x,'bo');
102 std_sol=plot(tt,ode45x,'r');
103 plot(tms,xms,'k.');
104 plot(tmsr,xmsr,'g.');
```

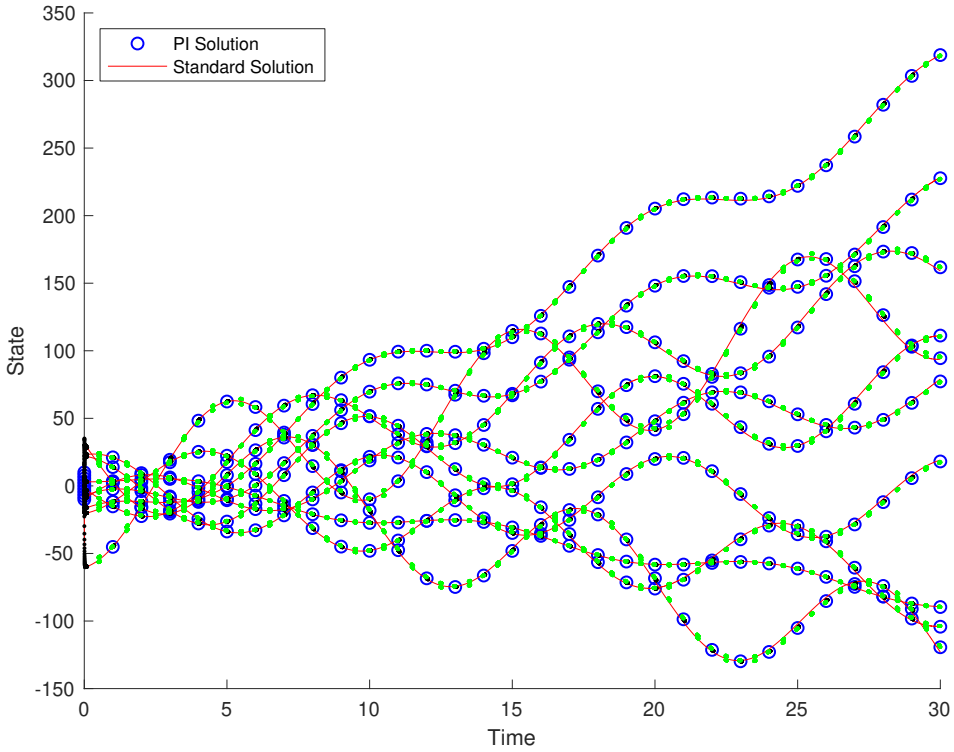



Figure 4: Demonstration of PIRK4(). From initial conditions, the system rapidly trannsitions to an attracting invariant manifold. The PI solution accurately tracks the evolution of the variables over time while requiring only a fraction of the computations of the standard solver.

```
105 legend([PI_sol(1),std_sol(1)],'PI Solution',...
106     'Standard Solution','Location','NorthWest')
107 xlabel('Time');
108 ylabel('State');
```

Save plot to a file.

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

## 2.4 PIG(): Projective Integration with a General macrosolver

This is an approximate Projective Integration scheme in which the macrosolver is given by any user-specified scheme. Unlike the PIRK functions, PIG() cannot 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 [t, x, tms, xms, svf] = PIG(solver,bT,macro,IC)
```

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

### Input

- solver(), a function that produces output from the user-specified code for micro-scale simulation. Usage:
  [tout,xout] = solver(t_in,x_in,tSim) Inputs: t_in, the initialisation time; x_in $\in \mathbb{R}^n$, the initial state; tSim, the time to simulate

for.
Outputs: tout, the vector of solution times, and xout, the corresponding states.

- bT, a scalar, the minimum amount of time thought needed for integration of the microsolver to relax the fast variables to the slow manifold.

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

- macro, 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 is 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, 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, but rather will consist of simulations 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.

```
68 tspan = macro.tspan;
69 csolve = macro.solver;
70 nT=length(tspan)-1;
71 sIC = length(IC);
```

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

```
78 nArgs=nargout();
79 saveMicro = (nArgs>1);
80 saveSvf = (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.

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

Update the initial time.

```
97  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.

```
106 if saveMicro
107     tms=cell(nT+1,1); xms=cell(nT+1,1);
108     n=1;
109     tms{n} = reshape(relax_t,[],1);
110     xms{n} = relax_IC;
111
112     if saveSvf
113         svf.t = cell(nT+1,1);
114         svf.dx = cell(nT+1,1);
115     end
116 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

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

Run a microsolver from the given initial conditions.

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

Compute the standard PI approximation of the slow vector field.

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

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

```
146         if saveMicro
147         n=n+1;
148         tms{n} = [reshape(t_tmp,[],1); nan];
149         xms{n} = [x_micro_tmp; nan(1,sIC)];
150         if saveSvf
151             svf.t{n-1} = tt;
152             svf.dx{n-1} = dx;
153         end
154         end
```

End `genProjection()`.

```
160 end
```

Define the approximate slow vector field according to PI.

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

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

```
176 [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.

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

Output the macroscale steps:

```
192
```

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.

```
200 if saveMicro
201     tms = cell2mat(tms);
202     xms = cell2mat(xms);
203     if saveSvf
```

```
204         svf.t = cell2mat(svf.t);
205         svf.dx = cell2mat(svf.dx);
206     end
207 end
```

This concludes `PIG()`.

```
215 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.

```
10 clear
```

Set time scale separation and model.

```
17 epsilon = 1e-4;
18 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.

```
25 solver = @(tIC, xIC,T) feval('ode45',f,[tIC tIC+T],xIC);
26 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.

```
33 IC = [1 3];
34 tspan=[0 40];
35 macro.tspan = tspan;
36 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
[t,x,tms,xms] = PIG(solver,bT,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);
tODE45 = toc;

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.

```
figure; set(gcf,'PaperPosition',[0 0 14 10])
hold on
PIm=plot(tms,xms,'b.');
PI=plot(t,x,'bo');
ODE45=plot(t45,xode45,'r-','LineWidth',2);
legend([PI(1),ODE45(1),PIm(1)],'PI Solution',...
    'Standard Solution','PI microsolver')
xlabel('Time')
ylabel('State')
axis([0 40 0 3])

figure; set(gcf,'PaperPosition',[0 0 14 10])
hold on
PI2=plot(t,x,'bo');
ODE452=plot(t45,xode45,'r-','LineWidth',2);

legend([PI2(1),ODE452(1)],'PI Solution','Standard Solution')
xlabel('Time')
ylabel('State')
```

The output is plotted in Figure 5.

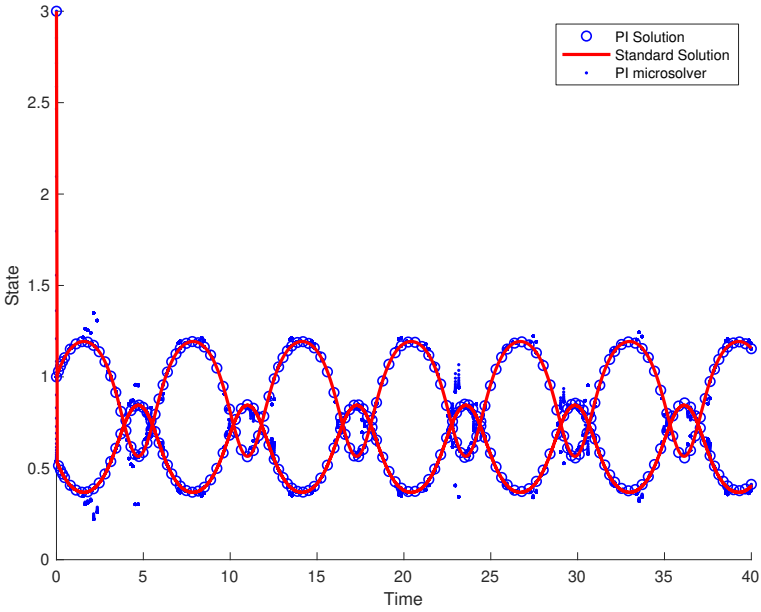



Figure 5: Accurate simulation of a stiff nonautonomous system by PIG(). The microsolver is called on-the-fly by the macrosolver (here ode45).

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 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 a wrapper for the microsolver. 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
cSol = @(t,x,T) cdmc(sol,t,x,T),
and thereafter use csol() in place of sol() as the solver for any PI scheme. The original solver sol() would create large errors if used in a PI scheme, but the output of cdmc() will not.

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

Begin with a standard application of the microsolver.

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

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

```
45 del = (xx(end,:) - xx(end-1,:))/(tt(end) - tt(end-1));
46 b_prop = 0.2;
47 x = xx(end,:) + (1+b_prop)*(tt(1)-tt(end))*del;
48 tr=2*tt(1)-tt(end);
49 [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 initialises at $(t_{in}, x_{in})$ 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 second implementation successfully replicates the true dynamics.

```
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.

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

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

```
cSol = @(t,x,T) cdmc(naiveSol,t,x,T);
```

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.

```
[nt,nx] = PIG(naiveSol,bT,macro,IC);
[ct,cx] = PIG(cSol,bT,macro,IC);
[t45,xode45] = ode45(f,[tspan(1) tspan(end)],IC);
```

Plot the output.

```
60 figure; set(gcf,'PaperPosition',[0 0 14 10])
61 hold on
62 nPI = plot(nt,nx,'bo');
63 PI=plot(ct,cx,'ko');
64 ODE45=plot(t45,xode45,'r-','LineWidth',2);
65
66 legend([nPI(1),PI(1),ODE45(1)],'Naive PIG Solution',...
67     'PIG using cdmc','Accurate Solution')
68 xlabel('Time')
69 ylabel('State')
70 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(): projective integration of fourth order accuracy

This Projective Integration scheme implements a macrosolver analogous to the fourth order Runge–Kutta method.

```
15 function [x, tms, xms, rm, svf] = PIRK4(solver, bT, tSpan, x0)
```

The inputs and outputs are standardised with PIRK2().

[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.

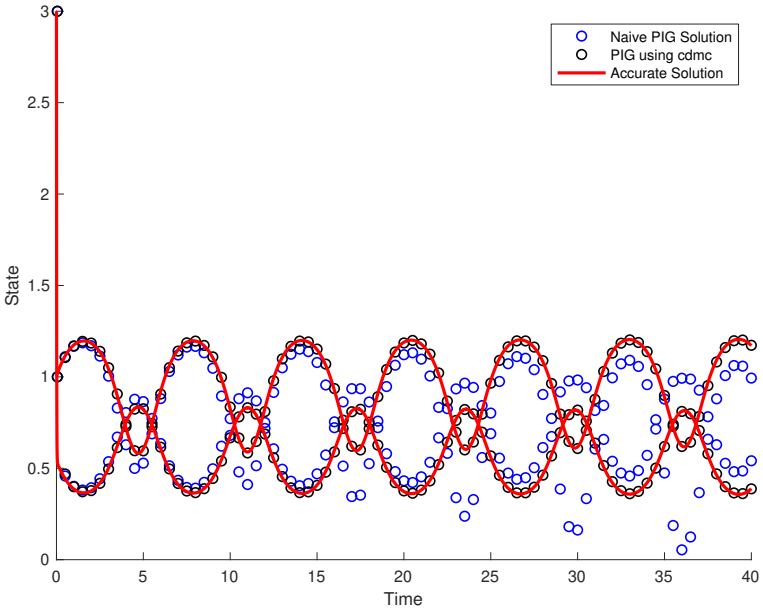



Figure 6: Accurate simulation of a weakly stiff nonautonomous system by PIG() using cdmc(), and an inaccurate solution using a naive application of PIG().

## Input

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

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

  – Inputs: `tStart`, the start time of a burst of simulation; `xStart`, the row $n$-vector of the starting state; `tSim`, 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. PIRK4() 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** If there are no output arguments specified, then a plot is drawn of the computed solution x versus tSpan.

- 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 = PIRK4(solver,bT,tSpan,x0).

  However, microscale details of the underlying Projective Integration computations may be helpful. PIRK4() provides two to four optional outputs of the microscale bursts.

- tms, optional, is an $L$ dimensional column vector containing microscale times of burst simulations, each burst separated by NaN;
- xms, optional, 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, optional, 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; the rm.x 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, optional, a struct containing the Projective Integration estimates of the slow vector field.
  - svf.t is a $4\ell$ dimensional column vector containing all times at which the Projective Integration scheme is extrapolated along microsolver data to form a macrostep.
  - svf.dx is a $4\ell \times n$ array containing the estimated slow vector field.

### 2.8.1 The projective integration code

Determine the number of time steps and preallocate storage for macroscale estimates.

```
109 nT=length(tSpan);
110 x=nan(nT,length(x0));
```

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

```
118 nArgs=nargout();
119 saveMicro = (nArgs>1);
120 saveFullMicro = (nArgs>3);
121 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.

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

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

```
143 tSpan(1) = tSpan(1)+bT;
144 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.

```
154 if saveMicro
155     tms = cell(nT,1);
156     xms = cell(nT,1);
157     tms{1} = reshape(relax_t,[],1);
158     xms{1} = relax_x0;
159     if saveFullMicro
160         rm.t = cell(nT,1);
161         rm.x = cell(nT,1);
162         if saveSvf
163             svf.t = nan(4*nT-4,1);
164             svf.dx = nan(4*nT-4,length(x0));
165         end
166     end
167 end
```

### Loop over the macroscale time steps

```
175 for jT = 2:nT
176     T = tSpan(jT-1);
```

If four applications of the microsolver would cover the entire macroscale time-step, then do so (setting some internal states to NaN); else proceed to projective step.

```
184     if 4*abs(bT)>=abs(tSpan(jT)-T) & bT*(tSpan(jT)-T)>0
185         [t1,xm1] = solver(T, x(jT-1,:), tSpan(jT)-T);
186         x(jT,:) = xm1(end,:);
187         t2=nan; xm2=nan(1,size(xm1,2));
```

```
        t3=nan; t4=nan; xm3=xm2; xm4 = xm2; dx1=xm2; dx2=xm2;
    else
```

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.

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

Check for round-off error.

```
    xt=[reshape(t1(end-1:end),[],1) xm1(end-1:end,:)];
    roundingTol=1e-8;
    if norm(diff(xt))/norm(xt,'fro') < roundingTol
    warning(['significant round-off error in 1st projection at T=' n
    end
```

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

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

```
    xint = xm1(end,:) + (Dt/2-bT)*dx1;
    [t2,xm2] = solver(T+Dt/2, xint, bT);
    del = t2(end)-t2(end-1);
    dx2 = (xm2(end,:)-xm2(end-1,:))/del;

    xint = xm1(end,:) + (Dt/2-bT)*dx2;
    [t3,xm3] = solver(T+Dt/2, xint, bT);
    del = t3(end)-t3(end-1);
    dx3 = (xm3(end,:)-xm3(end-1,:))/del;

```

```
241     xint = xm1(end,:) + (Dt-bT)*dx3;
242     [t4,xm4] = solver(T+Dt, xint, bT);
243     del = t4(end)-t4(end-1);
244     dx4 = (xm4(end,:)-xm4(end-1,:))/del;
```

Check for round-off error.

```
250     xt=[reshape(t2(end-1:end),[],1) xm2(end-1:end,:)];
251     if norm(diff(xt))/norm(xt,'fro') < roundingTol
252     warning(['significant round-off error in 2nd projection at T=' n
253     end
```

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

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

Now end the if-statement that tests whether a projective step saves simulation time.

```
269     end
```

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.

```
279     if saveMicro
280         tms{jT} = [reshape(t1,[],1); nan];
281         xms{jT} = [xm1; nan(1,size(xm1,2))];
```

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

```
289         if saveFullMicro
290             rm.t{jT} = [reshape(t2,[],1); nan;...
291                         reshape(t3,[],1); nan;...
292                         reshape(t4,[],1); nan];
293             rm.x{jT} = [xm2; nan(1,size(xm2,2));...
294                         xm3; nan(1,size(xm2,2));...
295                         xm4; nan(1,size(xm2,2))];
```

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

```
            if saveSvf
                svf.t(4*jT-7:4*jT-4) = [t1(end); t2(end); t3(end); t
                svf.dx(4*jT-7:4*jT-4,:) = [dx1; dx2; dx3; dx4];
            end
        end
    end
```

Terminate the main loop:

```
end
```

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

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

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

```
if saveMicro
    tms = cell2mat(tms);
    xms = cell2mat(xms);
    if saveFullMicro
        rm.t = cell2mat(rm.t);
        rm.x = cell2mat(rm.x);
    end
end
```

### 2.8.2 If no output specified, then plot simulation

```
if nArgs==0
    figure, plot(tSpan,x,'o:')
    title('Projective Simulation with PIRK4')
end
```

This concludes `PIRK4()`.

```
357 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*


3.1 `configPatches1()`: configures spatial patches in 1D . . . . . 44
3.1.1 If no arguments, then execute an example . . . . . 45
3.1.2 The code to make patches . . . . . 47
3.2 `patchSmooth1()`: interface to time integrators . . . . . 49
3.3 `patchEdgeInt1()`: sets edge values from interpolation over the macroscale . . . . . 50
3.4 `BurgersExample`: simulate Burgers' PDE on patches . . . . . 55
3.4.1 Script code to simulate a micro-scale space-time map . 56
3.4.2 `burgersMap()`: discretise the PDE microscale . . . . . 59
3.4.3 `burgerBurst()`: code a burst of the patch map . . . . 60
3.5 `HomogenisationExample`: simulate heterogeneous diffusion in 1D ... . . . . 61
3.5.1 Script to simulate via stiff or projective integration . . 61
3.5.2 `heteroDiff()`: heterogeneous diffusion . . . . . 66
3.5.3 `heteroBurst()`: a burst of heterogeneous diffusion . . 66
3.6 `waterWaveExample`: simulate a water wave PDE on patches . 67
3.6.1 Script code to simulate wave systems . . . . . 69
3.6.2 `simpleWavePDE()`: simple wave PDE . . . . . 72
3.6.3 `waterWavePDE()`: water wave PDE . . . . . 73
3.7 To do . . . . . 74
3.8 Miscellaneous tests . . . . . 74
3.8.1 `patchEdgeInt1test`: test the spectral interpolation . 74


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.

**Quick start** For an example, see Section 3.1.1 for basic code that uses the provided functions to simulate Burgers' PDE.

## 3.1 `configPatches1()`: configures spatial patches in 1D

*Subsection contents*

3.1.1 If no arguments, then execute an example . . . . . 45
3.1.2 The code to make patches . . . . . 47

Makes the struct `patches` for use by the patch/gap-tooth time derivative function `patchSmooth1()`. Section 3.1.1 lists an example of its use.

```
17 function configPatches1(fun,Xlim,BCs,nPatch,ordCC,ratio,nSubP)
18 global patches
```

**Input** If invoked with no input arguments, then executes an example of simulating Burgers' PDE—see Section 3.1.1 for the example code.

- `fun` is the name of the user function, `fun(t,u,x)`, that computes time derivatives (or time-steps) of quantities on the patches.
- `Xlim` give the macro-space domain of the computation: patches are equi-spaced 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 equi-spaced 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 equi-spaced 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 with the following components.

- .fun is the name of the user's function fun(u,t,x) that computes the time derivatives (or steps) on the patchy lattice.
- .ordCC is the specified order of inter-patch coupling.
- .alt is true for interpolation using only odd neighbouring patches as for staggered grids, and false for the usual case of all neighbour coupling.
- .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.
- .x is nSubP $\times$ nPatch array of the regular spatial locations $x_{ij}$ of the microscale grid points in every patch.

### 3.1.1 If no arguments, then execute an example

```
79 if nargin==0
```

The code here shows one way to get started: a user's script may have the following three steps (arrows indicate function recursion).

1. configPatches1
2. ode15s integrator $\leftrightarrow$ patchSmooth1 $\leftrightarrow$ user's burgersPDE
3. process results

Establish global patch data struct to interface with a function coding Burgers' PDE: to be solved on $2\pi$-periodic domain, with eight patches, spectral interpolation couples the patches, each patch of half-size ratio 0.2, and with seven points within each patch.

```
configPatches1(@BurgersPDE,[0 2*pi], nan, 8, 0, 0.2, 7);
```

Set an initial condition, and integrate in time using standard functions.

```
u0=0.3*(1+sin(patches.x))+0.1*randn(size(patches.x));
[ts,ucts]=ode15s(@patchSmooth1,[0 0.5],u0(:));
```

Plot the simulation using only the microscale values interior to the patches: set $x$-edges to `nan` to leave the gaps. Figure 7 illustrates an example simulation in time generated by the patch scheme applied to Burgers' PDE.

```
figure(1),clf
patches.x([1 end],:)=nan;
surf(ts,patches.x(:),ucts'), view(60,40)
title('Example of Burgers PDE on patches in space')
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
```

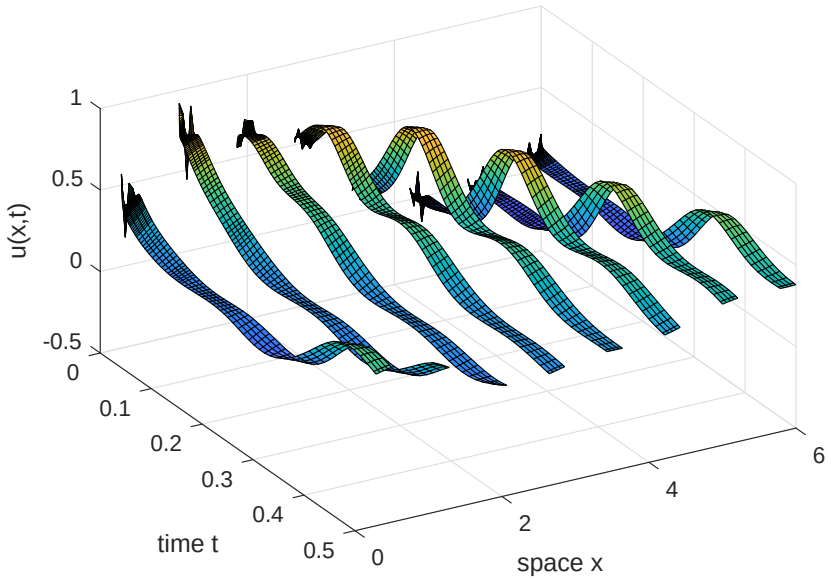
Upon finishing execution of the example, exit this function.

```
return
end%if no arguments
```

**Example of Burgers PDE inside patches** As a microscale discretisation of $u_t = u_{xx} - 30uu_x$, code $\dot{u}_{ij} = \frac{1}{\delta x^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) - 30u_{ij}\frac{1}{2\delta x}(u_{i+1,j} - u_{i-1,j})$.

```
function ut=BurgersPDE(t,u,x)
  dx=diff(x(1:2));  % micro-scale spacing
  i=2:size(u,1)-1;  % interior points in patches
  ut=nan(size(u));  % preallocate storage
  ut(i,:)=diff(u,2)/dx^2 ...
    -30*u(i,:).*(u(i+1,:)-u(i-1,:))/(2*dx);
end
```

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




### 3.1.2 The code to make patches

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

```
165 patches.fun=fun;
```

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

```
174 if ~ismember(ordCC,[-1:8])
175     error('ordCC out of allowed range [-1:8]')
176 end
```

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

```
183 patches.alt=mod(ordCC,2);
184 ordCC=ordCC+patches.alt;
```

```
185 patches.ordCC=ordCC;
```

Check for staggered grid and periodic case.

```
191   if patches.alt & (mod(nPatch,2)==1)
192     error('Require an even number of patches for staggered grid')
193   end
```

Might as well precompute the weightings for the interpolation of field values for coupling. (Could sometime extend to coupling via derivative values.)

```
201 if patches.alt  % eqn (7) in \cite{Cao2014a}
202   patches.Cwtsr=[1
203     ratio/2
204     (-1+ratio^2)/8
205     (-1+ratio^2)*ratio/48
206     (9-10*ratio^2+ratio^4)/384
207     (9-10*ratio^2+ratio^4)*ratio/3840
208     (-225+259*ratio^2-35*ratio^4+ratio^6)/46080
209     (-225+259*ratio^2-35*ratio^4+ratio^6)*ratio/645120 ];
210 else %
211   patches.Cwtsr=[ratio
212     ratio^2/2
213     (-1+ratio^2)*ratio/6
214     (-1+ratio^2)*ratio^2/24
215     (4-5*ratio^2+ratio^4)*ratio/120
216     (4-5*ratio^2+ratio^4)*ratio^2/720
217     (-36+49*ratio^2-14*ratio^4+ratio^6)*ratio/5040
218     (-36+49*ratio^2-14*ratio^4+ratio^6)*ratio^2/40320 ];
219 end
220 patches.Cwtsr=patches.Cwtsr(1:ordCC);
221 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.

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

```
232 DX=X(2)-X(1);
```

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

```
240 if mod(nSubP,2)==0, error('configPatches1: nSubP must be odd'), end
241 i0=(nSubP+1)/2;
242 dx=ratio*DX/(i0-1);
243 patches.x=bsxfun(@plus,dx*(-i0+1:i0-1)',X); % micro-grid
244 end% function
```

Fin.

## 3.2 patchSmooth1(): interface to time integrators

To simulate in time with spatial patches we often need to interface a users time derivative function with time integration routines such as `ode15s` or `PIRK2`. This function provides an interface. It 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. Communicate patch-design variables to this function using the previously established global struct `patches`.

```
23 function dudt=patchSmooth1(t,u)
24 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 `configPatches1()` with the following information 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 herein the patch edge values are 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 an equi-spaced lattice on both macro- and micro-scales.

**Output**

- `dudt` is `nSubP` · `nPatch` · `nVars` vector of time derivatives, but with patch edge values set to zero.

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

```
67 u=patchEdgeInt1(u);
```

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

```
77 dudt=patches.fun(t,u,patches.x);
78 dudt([1 end],:,:)=0;
79 dudt=reshape(dudt,[],1);
```

Fin.

## 3.3 `patchEdgeInt1()`: sets edge values from interpolation over the macroscale

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. Communicate patch-design variables via the global struct patches.

```
22 function u=patchEdgeInt1(u)
23 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.
- patches a struct set by configPatches1() with the following information.
  - .fun is the name of the user's function fun(t,u,x) that computes the time derivatives (or time-steps) 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 are 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 an equi-spaced 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 size.

```
[nSubP,nPatch]=size(patches.x);
nVars=round(numel(u)/numel(patches.x));
if numel(u)~=nSubP*nPatch*nVars
  nSubP=nSubP, nPatch=nPatch, nVars=nVars, sizeu=size(u)
  end
u=reshape(u,nSubP,nPatch,nVars);
```

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.

```
dx=patches.x(3,1)-patches.x(2,1);
DX=patches.x(2,2)-patches.x(2,1);
r=dx*(nSubP-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.

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

The centre of each patch (as `nSubP` is odd) is at

```
i0=round((nSubP+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.

```
if patches.ordCC>0 % then non-spectral interpolation
  dmu=nan(patches.ordCC,nPatch,nVars);
  if patches.alt % use only odd numbered neighbours
    dmu(1,:,:)=(u(i0,jp,:)+u(i0,jm,:))/2; % \mu
    dmu(2,:,:)= u(i0,jp,:)-u(i0,jm,:); % \delta
    jp=jp(jp); jm=jm(jm); % increase shifts to \pm2
  else % standard
```

```
119     dmu(1,:,:)=(u(i0,jp,:)-u(i0,jm,:))/2; % \mu\delta
120     dmu(2,:,:)=(u(i0,jp,:)-2*u(i0,j,:)+u(i0,jm,:)); % \delta^2
121   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).

```
129   for k=3:patches.ordCC
130     dmu(k,:,:)=dmu(k-2,jp,:)-2*dmu(k-2,j,:)+dmu(k-2,jm,:);
131   end
```

Interpolate macro-values to be Dirichlet edge values for each patch (Roberts & Kevrekidis 2007), using weights computed in `configPatches1()` . Here interpolate to specified order.

```
139   u(nSubP,j,:)=u(i0,j,:)*(1-patches.alt) ...
140     +sum(bsxfun(@times,patches.Cwtsr,dmu));
141   u( 1,j,:)=u(i0,j,:)*(1-patches.alt) ...
142     +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 `nPatch` patches we resolve 'wavenumbers' $|k| <$ `nPatch`$/2$, so set row vector `ks` $= k2\pi/N$ for 'wavenumbers' $k = (0, 1, \ldots, k_{\max}, -k_{\max}, \ldots, -1)$.

```
159   else% spectral interpolation
```

Deal with staggered grid by doubling the number of fields and halving the number of patches (`configPatches1` 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.

```
169   if patches.alt % transform by doubling the number of fields
170     v=nan(size(u)); % currently to restore the shape of u
171     u=cat(3,u(:,1:2:nPatch,:),u(:,2:2:nPatch,:));
172     altShift=reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
```

```
173     iV=[nVars+1:2*nVars 1:nVars]; % scatter interp to alternate fiel
174     r=r/2;           % ratio effectively halved
175     nPatch=nPatch/2; % halve the number of patches
176     nVars=nVars*2;   % double the number of fields
177   else % the values for standard spectral
178     altShift=0;
179     iV=1:nVars;
180   end
```

Now set wavenumbers.

```
186   kMax=floor((nPatch-1)/2);
187   ks=2*pi/nPatch*(mod((0:nPatch-1)+kMax,nPatch)-kMax);
```

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

```
194   if imag(u(i0,:,:))==0, uclean=@(u) real(u);
195     else                 uclean=@(u) u;
196     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.

```
205   Ck=fft(u(i0,:,:));
206   if mod(nPatch,2)==0
207     Czz=Ck(1,nPatch/2+1,:)/nPatch;
208     Ck(1,nPatch/2+1,:)=0;
209   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.

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

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

```
226   if mod(nPatch,2)==0
227     cosr=cos(pi*(altShift+r+(0:nPatch-1)));
228     u(nSubP,:,iV)=u(nSubP,:,iV)+uclean(bsxfun(@times,Czz,cosr));
229     cosr=cos(pi*(altShift-r+(0:nPatch-1)));
230     u( 1,:,iV)=u( 1,:,iV)+uclean(bsxfun(@times,Czz,cosr));
231   end
```

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

```
238 if patches.alt
239   nVars=nVars/2;  nPatch=2*nPatch;
240   v(:,1:2:nPatch,:)=u(:,:,1:nVars);
241   v(:,2:2:nPatch,:)=u(:,:,nVars+1:2*nVars);
242   u=v;
243 end
244 end% if spectral
```

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

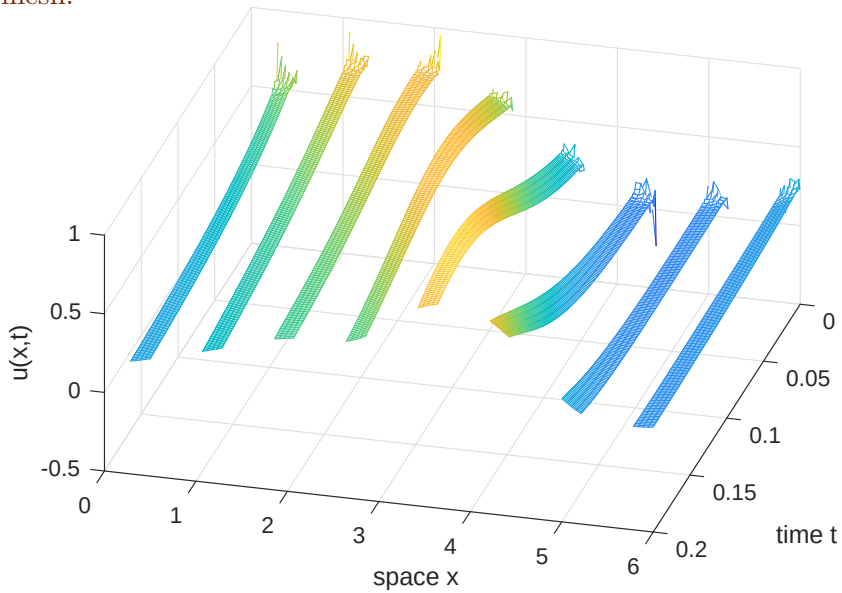
## 3.4 BurgersExample: simulate Burgers' PDE on patches

*Subsection contents*

3.4.1 Script code to simulate a micro-scale space-time map . 56
3.4.2 burgersMap(): discretise the PDE microscale . . . . . 59
3.4.3 burgerBurst(): code a burst of the patch map . . . . 60

Figure 7 shows an example simulation in time generated by the patch scheme function applied to Burgers' PDE. This code similarly applies the Equation-Free functions to a microscale space-time map (Figure 8), a map that happens to be derived as a micro-scale space-time discretisation of Burgers' PDE. Then this example applies projective integration to simulate further in time.

Figure 8: a short time simulation of the Burgers' map (Section 3.4.2) on patches in space. It requires many very small time steps only just visible in this mesh.




### 3.4.1 Script code to simulate a micro-scale space-time map

This first part of the script implements the following gap-tooth scheme (arrows indicate function recursion).

1. configPatches1
2. burgerBurst ↔ patchSmooth1 ↔ burgersMap
3. process results

Establish global data struct for the Burgers' map (Section 3.4.2) 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 interpolation provides edge-values that couple the patches.

```
45 clear all
```

```
global patches
nPatch = 8
ratio = 0.2
nSubP = 7
interpOrd = 4
Len = 2*pi
configPatches1(@burgersMap,[0 Len],nan,nPatch,interpOrd,ratio,nSubP)
```

Set an initial condition, and simulate a burst of the micro-scale space-time map over a time 0.2 using the function `burgerBurst()` (Section 3.4.3).

```
u0 = 0.4*(1+sin(patches.x))+0.1*randn(size(patches.x));
[ts,us] = burgerBurst(0,u0,0.2);
```

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

```
figure(1),clf
xs = patches.x;  xs([1 end],:) = nan;
mesh(ts,xs(:),us')
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
view(105,45)
set(gcf,'paperposition',[0 0 14 10])
print('-depsc2','ps1BurgersMapU')
```

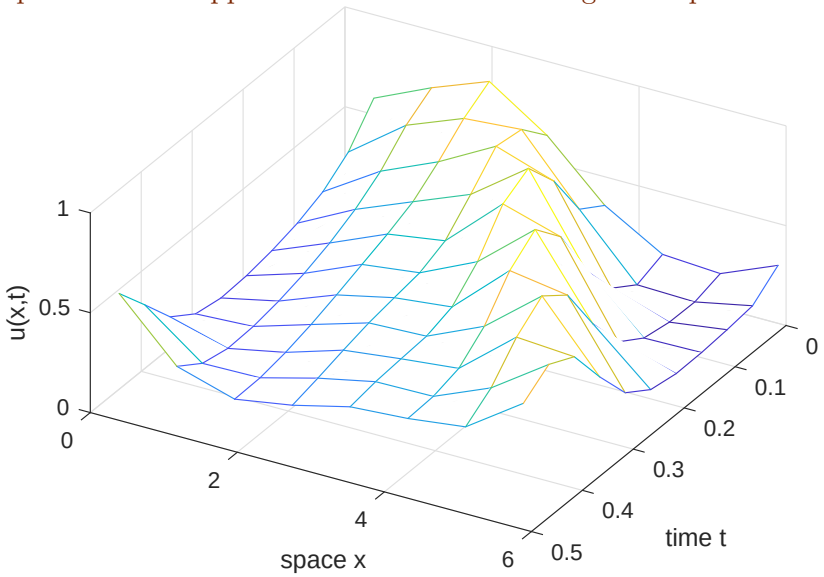
**Use projective integration** Around the micro-scale burst `burgerBurst()`, wrap the projective integration function `PIRK2()` of Section 2.2. Figure 9 shows the macroscale prediction of the patch centre values on macro-scale time-steps.

This second part of the script implements the following design.

1. configPatches1 (done in first part)
2. PIRK2 ↔ burgerBurst ↔ patchSmooth1 ↔ burgersMap
3. process results

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

Figure 9: macro-scale space-time field $u(x,t)$ in a basic projective integration of the patch scheme applied to the micro-scale Burgers' map.




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

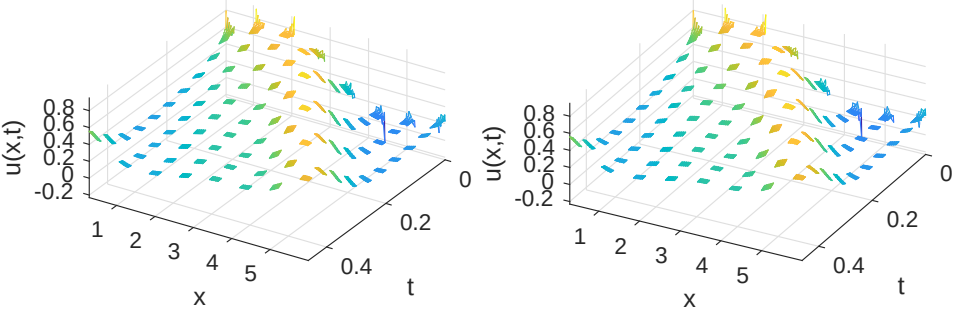
Set the desired macro-scale time-steps, and micro-scale burst length over the time domain. Then projectively integrate in time using `PIRK2()` which is (roughly) second-order accurate in the macro-scale time-step.

```
116 ts = linspace(0,0.5,11);
117 bT = 3*(ratio*Len/nPatch/(nSubP/2-1))^2
118 addpath('../ProjInt')
119 [us,tss,uss] = PIRK2(@burgerBurst,bT,ts,u0(:));
```

Plot the macroscale predictions of the mid-patch values to give the macroscale mesh of Figure 9.

```
126 figure(2),clf
127 mid = (nSubP+1)/2;
128 mesh(ts,xs(mid,:),us(:,mid:nSubP:end)')
```

Figure 10: the field $u(x,t)$ during each of the microscale bursts used in the projective integration. View this stereo pair cross-eyed.




```
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
view(120,50)
set(gcf,'paperposition',[0 0 14 10])
print('-depsc2','ps1BurgersU')
```

Then plot the microscale mesh of the microscale bursts shown in Figure 10 (a stereo pair). The details of the fine microscale mesh are almost invisible.

```
figure(3),clf
for k = 1:2, subplot(2,2,k)
  mesh(tss,xs(:),uss')
  ylabel('x'),xlabel('t'),zlabel('u(x,t)')
  axis tight, view(126-4*k,50)
end
set(gcf,'paperposition',[0 0 17 12])
print('-depsc2','ps1BurgersMicro')
```

### 3.4.2 burgersMap(): discretise the PDE microscale

This function codes the microscale Euler integration map of the lattice differential equations inside the patches. Only the patch-interior values mapped (patchSmooth1 overrides the edge-values anyway).

```
function u = burgersMap(t,u,x)
  dx = diff(x(2:3));   dt = dx^2/2;
  i = 2:size(u,1)-1;
  u(i,:) = u(i,:) +dt*( diff(u,2)/dx^2 ...
     -20*u(i,:).*(u(i+1,:)-u(i-1,:))/(2*dx) );
end
```

### 3.4.3 burgerBurst(): code a burst of the patch map

```
function [ts, us] = burgerBurst(ti, ui, bT)
```

First find and set the number of micro-scale time-steps.

```
  global patches
  dt = diff(patches.x(2:3))^2/2;
  ndt = ceil(bT/dt -0.2);
  ts = ti+(0:ndt)'*dt;
```

Apply the microscale map over all time-steps in the burst, using patchSmooth1 (Section 3.2) as the interface that provides the interpolated edge-values of each patch. Store the results in rows to be consistent with ODE and projective integrators.

```
  us = nan(ndt+1,numel(ui));
  us(1,:) = reshape(ui,1,[]);
  for j = 1:ndt
    ui = patchSmooth1(ts(j),ui);
    us(j+1,:) = reshape(ui,1,[]);
  end
```

Linearly interpolate (extrapolate) to get the field values at the final time of the burst. Then return.

```
  ts(ndt+1) = ti+bT;
  us(ndt+1,:) = us(ndt,:) ...
    + diff(ts(ndt:ndt+1))/dt*diff(us(ndt:ndt+1,:));
end
```

Fin.

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

*Subsection contents*

3.5.1 Script to simulate via stiff or projective integration . . 61
3.5.2 heteroDiff(): heterogeneous diffusion . . . . . 66
3.5.3 heteroBurst(): a burst of heterogeneous diffusion . . 66

Figure 11 shows an example simulation in time generated by the patch scheme function applied to heterogeneous diffusion. That such simulations of heterogeneous diffusion makes valid predictions was established by Bunder et al. (2017) who proved that the scheme is accurate when the number of points in a patch is one more than an even multiple of the microscale periodicity.

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

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

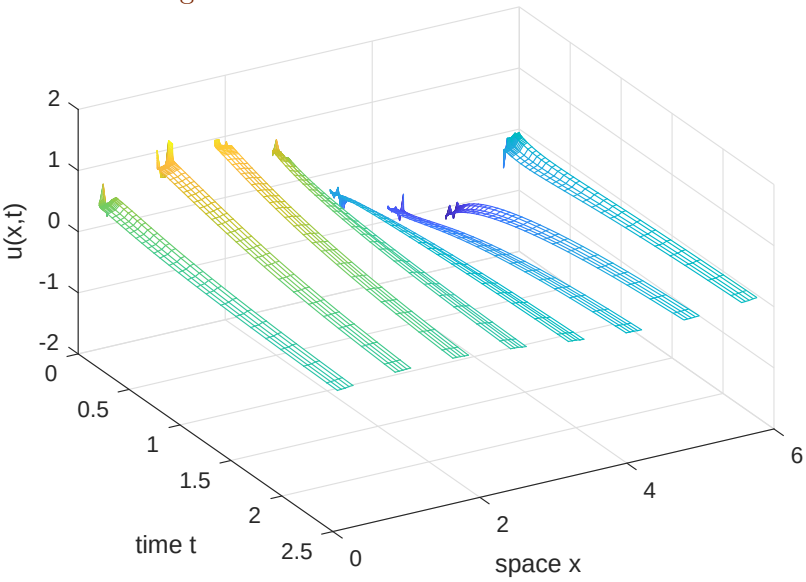
In this 1D space, the macroscale, homogenised, effective diffusion should be the harmonic mean of these coefficients.

### 3.5.1 Script to simulate via stiff or projective integration

This first part of the script implements the following gap-tooth scheme (arrows indicate function recursion).

1. configPatches1
2. ode15s ↔ patchSmooth1 ↔ heteroDiff
3. process results

Figure 11: the diffusing field $u(x,t)$ in the patch (gap-tooth) scheme applied to microscale heterogeneous diffusion.




Set the desired microscale periodicity, and microscale diffusion coefficients (with subscripts shifted by a half).

```
clear all
mPeriod = 3
cDiff = exp(randn(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 0.2, 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). Quadratic (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;
configPatches1(@heteroDiff,[0 Len],nan,nPatch,4,ratio,nSubP);
```

A user can add information to the global data struct `patches` in order to communicate to the time derivative function. Here include the diffusivity coefficients, replicated to fill up a patch.

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

**Conventional integration in time** Set an initial condition, and here integrate forward in time using a standard method for stiff systems—because of the simplicity of linear problems this method works quite efficiently here. Integrate the interface `patchSmooth1` (Section 3.2) to the microscale differential equations.

```
u0 = sin(patches.x)+0.2*randn(nSubP,nPatch);
[ts,ucts] = ode15s(@patchSmooth1, [0 2/cHomo], u0(:));
```

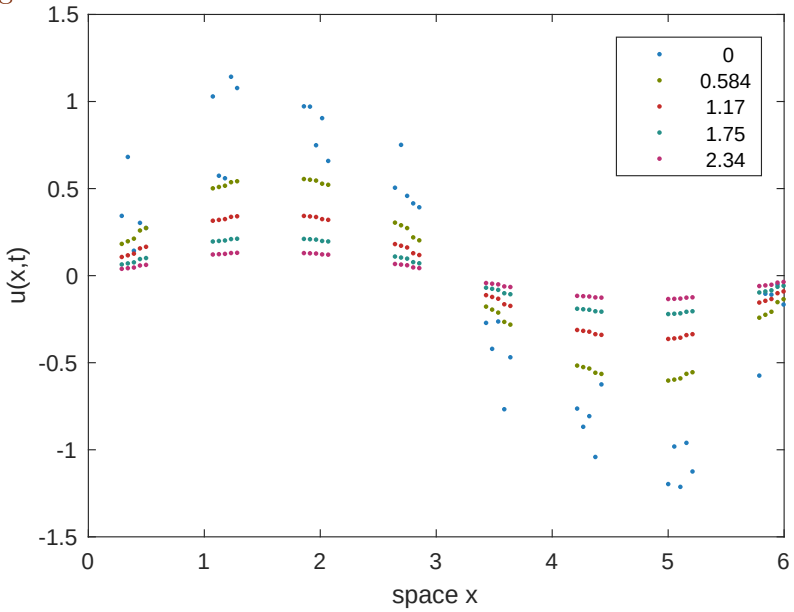
Plot the simulation in Figure 11.

```
figure(1),clf
xs = patches.x;  xs([1 end],:) = nan;
mesh(ts,xs(:),ucts'),  view(60,40)
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
set(gcf,'paperposition',[0 0 14 10])
print('-depsc2','ps1HomogenisationCtsU')
```

**Use projective integration in time** Now take `patchSmooth1`, the interface to the time derivatives, and wrap around it the projective integration `PIRK2` (Section 2.2), of bursts of simulation from `heteroBurst` (Section 3.5.3), as illustrated by Figure 12.

This second part of the script implements the following design, where the micro-integrator could be, for example, `ode23` or `rk2int`.

Figure 12: field $u(x,t)$ shows basic projective integration of patches of heterogeneous diffusion.



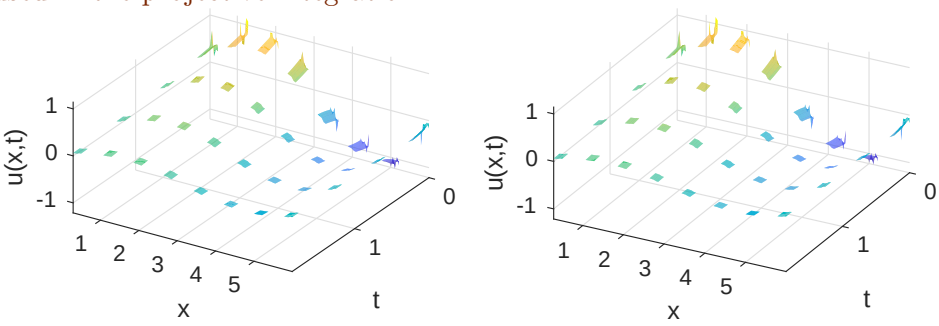

1. configPatches1 (done in first part)
2. PIRK2 ↔ heteroBurst ↔ micro-integrator ↔ patchSmooth1 ↔ heteroDiff
3. process results

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

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

Set the desired macro- and micro-scale time-steps over the time domain: the macroscale step is in proportion to the effective mean diffusion time on the macroscale; the burst time is proportional to the intra-patch effective diffusion time; and lastly, the microscale time-step is proportional to the diffusion time between adjacent points in the microscale lattice.

Figure 13: stereo pair of the field $u(x,t)$ during each of the microscale bursts used in the projective integration.




```
ts = linspace(0,2/cHomo,5)
bT = 3*( ratio*Len/nPatch )^2/cHomo
addpath('../ProjInt','../RKint')
[us,tss,uss] = PIRK2(@heteroBurst, bT, ts, u0(:));
```

Plot the macroscale predictions to draw Figure 12.

```
figure(2),clf
plot(xs(:),us',':')
ylabel('u(x,t)'), xlabel('space x')
legend(num2str(ts',3))
set(gcf,'paperposition',[0 0 14 10])
print('-depsc2','ps1HomogenisationU')
```

Also plot a surface detailing the microscale bursts as shown in Figure 13.

```
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(126-4*k,45)
end
set(gcf,'paperposition',[0 0 17 12])
print('-depsc2','ps1HomogenisationMicro')
```

End of the script.

### 3.5.2 heteroDiff(): heterogeneous diffusion

This function codes the lattice heterogeneous diffusion inside the patches. For 2D input arrays u and x (via edge-value interpolation of patchSmooth1, Section 3.2), computes the time derivative (1) at each point in the interior of a patch, output in ut. The column vector (or possibly array) of diffusion coefficients $c_i$ have previously been stored in struct patches.[2]

```
function ut = heteroDiff(t,u,x)
  global patches
  dx = diff(x(2:3)); % space step
  i = 2:size(u,1)-1; % interior points in a patch
  ut = nan(size(u)); % preallocate output array
  ut(i,:) = diff(bsxfun(@times,patches.c,diff(u)))/dx^2 ;
end% function
```

### 3.5.3 heteroBurst(): a burst of heterogeneous diffusion

This code integrates in time the derivatives computed by heteroDiff from within the patch coupling of patchSmooth1. Try three possibilities:

- ode23 generates 'noise' that is unsightly at best and may be ruinous;
- ode15s does not cater for the NaNs in some components of u;
- rk2int simple specified step integrator behaves consistently, and so appears best.

```
function [ts, ucts] = heteroBurst(ti, ui, bT)
  switch 'rk2'
  case '23',  [ts,ucts] = ode23 (@patchSmooth1,[ti ti+bT],ui(:));
  case '15s', [ts,ucts] = ode15s(@patchSmooth1,[ti ti+bT],ui(:));
  case 'rk2', ts = linspace(ti,ti+bT,100)';
```

[2] Use bsxfun() as pre-2017 Matlab versions may not support auto-replication.

```
229                 ucts = rk2int(@patchSmooth1,ts,ui(:));
230   end
231 end
```

Fin.

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

*Subsection contents*

3.6.1 Script code to simulate wave systems . . . . . 69
3.6.2 simpleWavePDE(): simple wave PDE . . . . . 72
3.6.3 waterWavePDE(): water wave PDE . . . . . 73

Figure 14 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 spectral interpolation to the patch edges of the mid-patch values.

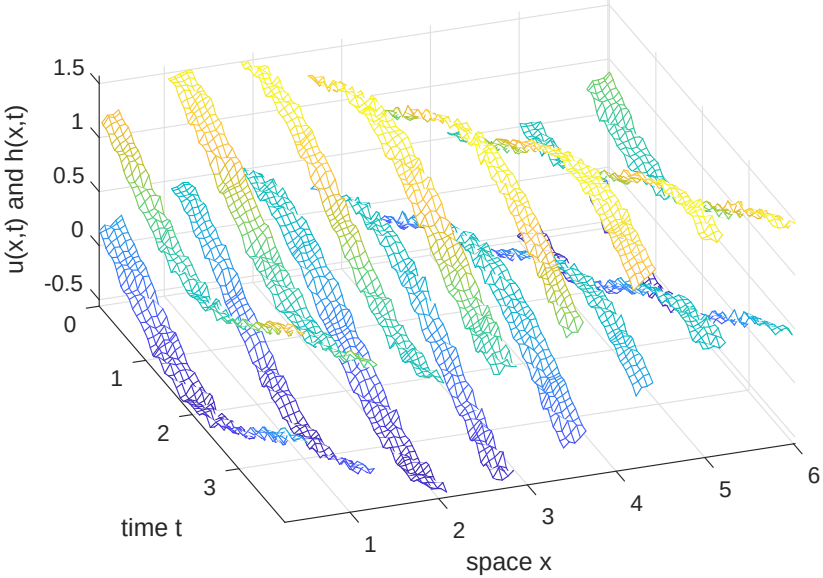
This approach, based upon the differential equations coded in Section 3.6.2, may be adapted by a user to a wide variety of 1D wave and near-wave systems. For example, the differential equations of Section 3.6.3 describes the nonlinear microscale simulator of the nonlinear shallow water wave PDE derived from the Smagorinski model of turbulent flow (**??**).

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 developed in this section applies to any wave-like system in the form

$$\frac{\partial h}{\partial t} = -c_1 \frac{\partial u}{\partial x} + f_1[h,u] \quad \text{and} \quad \frac{\partial u}{\partial t} = -c_2 \frac{\partial h}{\partial x} + f_2[h,u], \tag{2}$$

where the brackets indicate that the nonlinear functions $f_\ell$ may involve various spatial derivatives of the fields $h(x,t)$ and $u(x,t)$. For example, Section 3.6.3 encodes a nonlinear Smagorinski model of turbulent shallow water (**??**, e.g.) along an inclined flat bed: let $x$ measure position along the bed and in terms

Figure 14: water depth $h(x,t)$ (above) and velocity field $u(x,t)$ (below) of the gap-tooth scheme applied to the simple wave PDE (2), linearised. The micro-scale random component to the initial condition has long lasting effects on the simulation—but the macroscale wave still propagates.



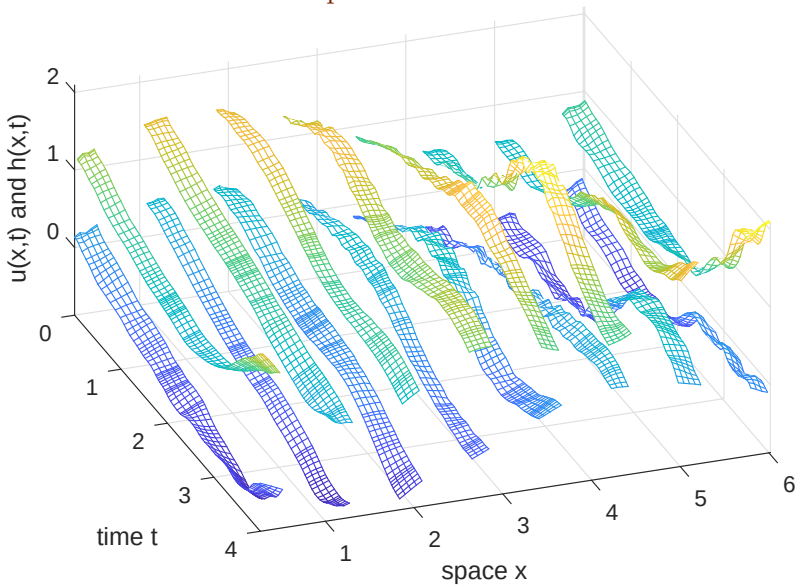

of fluid depth $h(x,t)$ and depth-averaged lateral velocity $u(x,t)$ the model PDEs are

$$\frac{\partial h}{\partial t} = -\frac{\partial (hu)}{\partial x}\,, \tag{3a}$$

$$\frac{\partial u}{\partial t} = 0.985\left(\tan\theta - \frac{\partial h}{\partial x}\right) - 0.003\frac{u|u|}{h} - 1.045u\frac{\partial u}{\partial x} + 0.26h|u|\frac{\partial^2 u}{\partial x^2}\,, \tag{3b}$$

where $\tan\theta$ is the slope of the bed. Equation (3a) represents conservation of the fluid. The momentum PDE (3b) 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 15 shows one simulation of this system—for the same initial condition as Figure 14.

Figure 15: water depth $h(x,t)$ (above) and velocity field $u(x,t)$ (below) of the gap-tooth scheme applied to the Smagorinski shallow water wave PDEs (3). The micro-scale random initial component decays where the water speed is non-zero due to 'turbulent' dissipation.




For such wave systems, let's implement a staggered microscale grid and staggered macroscale patches as introduced by **?** in their Figures 3 and 4, respectively.

### 3.6.1 Script code to simulate wave systems

This script implements the following gap-tooth scheme (arrows indicate function recursion).

1. configPatches1, and add micro-information
2. ode15s ↔ patchSmooth1 ↔ simpleWavePDE
3. process results
4. ode15s ↔ patchSmooth1 ↔ waterWavePDE

5. process results

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

```
clear all
global patches
nPatch = 8
ratio = 0.2
nSubP = 11 %of the form 4*n-1
Len = 2*pi;
configPatches1(@simpleWavePDE,[0 Len],nan,nPatch,-1,ratio,nSubP);
```

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

```
uPts = mod( bsxfun(@plus,(1:nSubP)',(1:nPatch)) ,2);
hPts = find(1-uPts);
uPts = find(uPts);
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 (possibly with some optional micro-scale wave noise).

```
U0 = nan(nSubP,nPatch);
U0(hPts) = 1+0.5*sin(patches.x(hPts));
U0(uPts) = 0+0.5*sin(patches.x(uPts));
U0 = U0+0.02*randn(nSubP,nPatch);
```

**Conventional integration in time** Integrate in time using standard MATLAB/Octave stiff integrators. Here do the two cases of the simple wave

and the water wave equations in the one loop.

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

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

```
  [ts,Ucts] = ode15s(@patchSmooth1,[0 4],U0(:));
  ts = ts(1:5:end);
  Ucts = Ucts(1:5:end,:);
```

Plot the simulation.

```
  figure(k),clf
  xs = patches.x;  xs([1 end],:) = nan;
  mesh(ts,xs(hPts),Ucts(:,hPts)'),hold on
  mesh(ts,xs(uPts),Ucts(:,uPts)'),hold off
  xlabel('time t'), ylabel('space x'), zlabel('u(x,t) and h(x,t)')
  axis tight, view(70,45)
```

Print the output.

```
  set(gcf,'paperposition',[0 0 14 10])
  if k==1, print('-depsc2','ps1WaveCtsUH')
  else print('-depsc2','ps1WaterWaveCtsUH')
  end
```

For the second time through the loop, change to the Smagorinski turbulence model (3) of shallow water flow, keeping other parameters and the initial condition the same.

```
  patches.fun = @waterWavePDE;
end
```

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

### 3.6.2 `simpleWavePDE()`: 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.

```
236 function Ut = simpleWavePDE(t,U,x)
237   global patches
238   dx = diff(x(2:3));
239   Ut = nan(size(U));  ht = Ut;
```

Compute the PDE derivatives at interior points of the patches.

```
245   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.

```
252   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.

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

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

```
264   Ut(patches.hPts) = ht(patches.hPts);
265 end
```

### 3.6.3 waterWavePDE(): water wave PDE

This function codes the staggered lattice equation inside the patches for the nonlinear wave-like PDE system (3). Also, regularise the absolute value appearing the the PDEs via the one-line function rabs().

```
277 function Ut = waterWavePDE(t,U,x)
278   global patches
279   rabs = @(u) sqrt(1e-4+u.^2);
```

As before, set the micro-grid spacing, reserve space for time derivatives, and index the patch-interior points of the micro-grid.

```
285   dx = diff(x(2:3));
286   Ut = nan(size(U));  ht = Ut;
287   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.

```
293   ii = i(2:end-1);
294   V = Ut;
295   V(ii,:) = (U(ii+1,:)+U(ii-1,:))/2;
296   V(1:2,:) = 2*U(2:3,:)-V(3:4,:);
297   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.

```
303   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 $\mathtt{U}_{i\pm1}$.

```
309   Ut(i,:) = -0.985*(U(i+1,:)-U(i-1,:))/(2*dx) ...
310     -0.003*U(i,:).*rabs(U(i,:)./V(i,:)) ...
311     -1.045*U(i,:).*(V(i+1,:)-V(i-1,:))/(2*dx) ...
312     +0.26*rabs(V(i,:).*U(i,:)).*(V(i+1,:)-2*U(i,:)+V(i-1,:))/dx^2/2;
```

where the mysterious division by two in the second 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 $\dot{h}_{ij}$.

```
  Ut(patches.hPts) = ht(patches.hPts);
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? Surely easy, just an example.

## 3.8 Miscellaneous tests

### 3.8.1 `patchEdgeInt1test`: test the 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
configPatches1(@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(:)]);
```

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

End the for-loop over various geometries.

```
end
```

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

```
for nPatch=6:2:20
nPatch=nPatch
ratio=0.5*rand
nSubP=3; % of form 4*N-1
Len=10*rand
configPatches1(@simpleWavepde,[0,Len],nan,nPatch,-1,ratio,nSubP);
kMax=floor((nPatch/2-1)/2)
```

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

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

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

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

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

# A 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.
%}
```

# B Aspects of developing a 'toolbox' for patch dynamics

*Section contents*

B.1 Macroscale grid . . . . . 81
B.2 Macroscale field variables . . . . . 81
B.3 Boundary and coupling conditions . . . . . 82
B.4 Mesotime communication . . . . . 83
B.5 Projective integration . . . . . 83
B.6 Lift to many internal modes . . . . . 84
B.7 Macroscale closure . . . . . 84
B.8 Exascale fault tolerance . . . . . 85
B.9 Link to established packages . . . . . 85

This appendix documents sketchy further thoughts on aspects of the development.

## B.1 Macroscale grid

The patches are to be distributed on a macroscale grid: the $j$th patch 'centred' at position $\vec{X}_j \in \mathbb{X}$. In principle the patches could move, but let's keep them fixed in the first version. The simplest macroscale grid will be rectangular (`meshgrid`), but we plan to allow a deformed grid to secondly cater for boundary fitting to quite general domain shapes $\mathbb{X}$. And plan to later allow for more general interconnect networks for more topologies in application.

## B.2 Macroscale field variables

The researcher/practitioner has to know an appropriate set of macroscale field variables $\vec{U}(t) \in \mathbb{R}^{d_{\bar{U}}}$ for each patch. For example, first they might be a simple average over a core of a patch of all of the micro-field variables;

second, they might be a subset of the average micro-field variables; and third in general the macro-variables might be a nonlinear function of the micro-field variables (such as temperature is the average speed squared). The core might be just one point, or a sizeable fraction of the patch.

The mapping from microscale variable to macroscale variables is often termed the restriction.

In practice, users may not choose an appropriate set of macro-variables, so will eventually need to code some diagnostic to indicate a failure of the assumed closure.

## B.3 Boundary and coupling conditions

The physical domain boundary conditions are distinct from the conditions coupling the patches together. Start with physical boundary conditions of periodicity in the macroscale.

Second, assume the physical boundary conditions are that the macro-variables are known at macroscale grid points around the boundary. Then the issue is to adjust the interpolation to cater for the boundary presence and shape. The coupling conditions for the patches should cater for the range of Robin-like boundary conditions, from Dirichlet to Neumann. Two possibilities arise: direct imposition of the coupling action (Roberts & Kevrekidis 2007), or control by the action.

Third, assume that some of the patches have some edges coincident with the boundary of the macroscale domain $\mathbb{X}$, and it is on these edges that macroscale physical boundary conditions are applied. Then the interpolation from the core of these edge patches is the same as the second case of prescribed boundary macro-variables. An issue is that each boundary patch should be big enough to cater for any spatial boundary layers transitioning from the applied boundary condition to the interior slow evolution.

Alternatively, we might have the physical boundary condition constrain the interpolation between patches.

Often microscale simulations are easiest to write when 'periodic' in microscale space. To cater for this we should also allow a control at perhaps the quartiles of a micro-periodic simulator.

## B.4 Mesotime communication

Since communication limits large scale parallelism, a first step in reducing communication will be to implement only updating the coupling conditions when necessary. Error analysis indicates that updating on times longer the microscale times and shorter than the macroscale times can be effective (**?**). Implementations can communicate one or more derivatives in time, as well as macroscale variables.

At this stage we can effectively parallelise over patches: first by simply using Matlab's `parfor`. Probably not using a GPU as we probably want to leave GPUs for the black box to utilise within each patch.

## B.5 Projective integration

To take macroscale time steps, invoke several possible projective integration schemes: simple Euler projection, Heun-like method, etc (**?**). Need to decide how long a microscale burst needs to be.

Should not need an implicit scheme as the fast dynamics are meant to be only in the micro variables, and the slow dynamics only in the macroscale variables. However, it could be that the macroscale variables have fast oscillations and it is only the amplitude of the oscillations that are slow. Perhaps need to detect and then fix or advise.

A further stage is to implement a projective integration scheme for stochastic macroscale variables: this is important because the averaging over a core of microscale roughness will almost invariably have at least some stochastic legacy effect. **?** did some useful research on stochastic projective intergration.

## B.6 Lift to many internal modes

In most problems the number of macroscale variables at any given position in space, $d_{\vec{U}}$, is less than the number of microscale variables at a position, $d_{\vec{u}}$; often much less (Kevrekidis & Samaey 2009, e.g.). In this case, every time we start a patch simulation we need to provide $d_{\vec{u}} - d_{\vec{U}}$ data at each position in the patch: this is lifting. The first methodology is to first guess, then run repeated short bursts with reinitialisation, until the simulation reaches a slow manifold. Then run the real simulation.

If the time taken to reach a local quasi-equilibrium is too long, then it is likely that the macroscale closure is bad and the macroscale variables need to be extended.

A second step is to cater for cases where the slow manifold is stochastic or is surrounded by fast waves: when it is hard to detect the slow manifold, or the slow manifold is not attractive.

## B.7 Macroscale closure

In some circumstances a researcher/practitioner will not code the appropriately set of macroscale variables for a complete closure of the macroscale. For example, in thin film fluid dynamics at low Reynolds number the only macroscale variable is the fluid depth; however, at higher Reynolds number, circa ten, the inertia of the fluid becomes important and the macroscale variables must additionally include a measure of the mean lateral velocity/momentum (**?**, e.g.).

At some stage we need to detect any flaw in the closure, and perhaps suggest additional appropriate macroscale variables, or at least their characteristics. Indeed, a poor closure and a stochastic slow manifold are really two faces of the same problem: the problem is that the chosen macroscale variables do not have a unique evolution in terms of themselves. A good resolution of the issue will account for both faces.

## B.8 Exascale fault tolerance

Matlab is probably not an appropriate vehicle to deal with real exascale faults. However, we should cater by coding procedures for fault tolerance and testing them at least synthetically. Eventually provide hooks to a user routine to be invoked under various potential scenarios. The nature of fault tolerant algorithms will vary depending upon the scenario, even assuming that each patch burst is executed on one CPU (or closely coupled CPUs): if there are much more CPUs than patches, then maybe simply duplicate all patch simulations; if much less CPUs than patches, then an asynchronous scheduling of patch bursts should effectively cater for recomputation of failed bursts; if comparable CPUs to patches, then more subtle action is needed.

Once mesotime communication and projective integration is provided, a recomputation approach to intermittent hardware faults should be effective because we then have the tools to restart a burst from available macroscale data. Should also explore proceeding with a lower order interpolation that misses the faulty burst—because an isolated lower order interpolation probably will not affect the global order of error (it does not in approximating some boundary conditions (**??**)

## B.9 Link to established packages

Several molecular/particle/agent based codes are well developed and used by a wide community of researchers. Plan to develop hooks to use some such codes as the microscale simulators on patches. First, plan to connect to LAMMPS (**?**). Second, will evaluate performance, issues, and then consider what other established packages are most promising.

# 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**.

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.

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.

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

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

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

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