

# Equation-Free function toolbox for Matlab/Octave

A. J. Roberts\*      John Maclean†      J. E. Bunder‡      et al.§

January 21, 2019

## 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

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Quick start</b>	<b>4</b>
<b>3</b>	<b>Projective integration of deterministic ODEs</b>	<b>7</b>

---

\*School of Mathematical Sciences, University of Adelaide, South Australia. <http://www.maths.adelaide.edu.au/anthony.roberts>, <http://orcid.org/0000-0001-8930-1552>

†School of Mathematical Sciences, University of Adelaide, South Australia. <http://www.adelaide.edu.au/directory/john.maclean>

‡School of Mathematical Sciences, University of Adelaide, South Australia. <mailto:judith.bunder@adelaide.edu.au>, <http://orcid.org/0000-0001-5355-2288>

§Appear here for your contribution.

<b>4 Patch scheme for given microscale discrete space system</b>	<b>29</b>
<b>A Create, document and test algorithms</b>	<b>69</b>
<b>B Aspects of developing a ‘toolbox’ for patch dynamics</b>	<b>72</b>

# 1 Introduction

This document is intended to be used in conjunction with the user manual. It contains line-by-line descriptions of the code in each function in the toolbox. For brief descriptions of each function, quick start guides, and many examples, see the user manual.

## *Subsubsection contents*

Users . . . . .	3
Blackbox scenario . . . . .	3
Contributors . . . . .	4

**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 micro-field variables could be in  $\mathbb{R}^p$  for any  $p = 1, 2, \dots, \infty$ .

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 Quick start

### *Section contents*

2.1 Cheat sheet: Projective Integration . . . . .	4
2.2 Cheat sheet: constructing patches . . . . .	6

This section may be used in conjunction with the many examples in later sections to help apply the toolbox functions to a particular problem, or to assist in distinguishing between the various functions.

### 2.1 Cheat sheet: Projective Integration

This section pertains to the Projective Integration (PI) methods of Section 3. The PI approach is to greatly accelerate computations of a system exhibiting multiple time scales.

The PI toolbox presents several ‘main’ functions that could separately be called to perform PI, as well as several optional wrapper functions that may be called. This section helps to distinguish between the top-level PI functions, and helps to tell which of the optional functions may be needed at a glance. For full details on each function refer to Section 3.

The cheat sheet consists of two flow charts. For an overview of constructing a PI simulation, see Figure 1. For a rough guide as to which of the top-level PI functions should be used, refer to Figure 2.

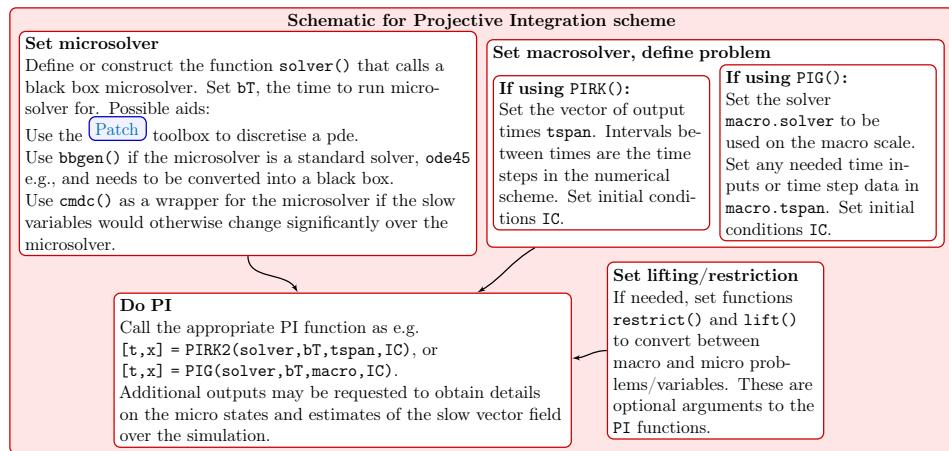


Figure 1

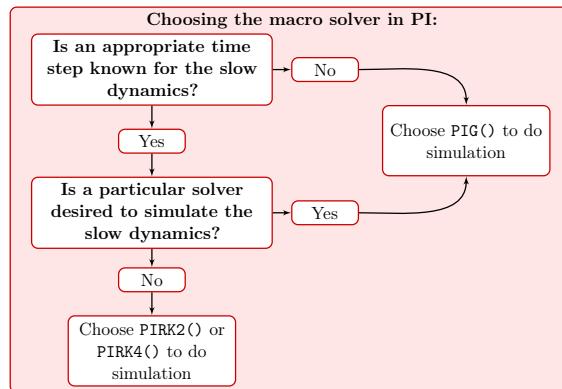


Figure 2

## 2.2 Cheat sheet: constructing patches

This section pertains to the Patch approach to discretising PDEs of Section 3.

The Patch toolbox requires that one configure patches, couple the patches and interface the coupled patches with a time integrator. For an overview of the chief functions involved and their interactions, see Figure 3.

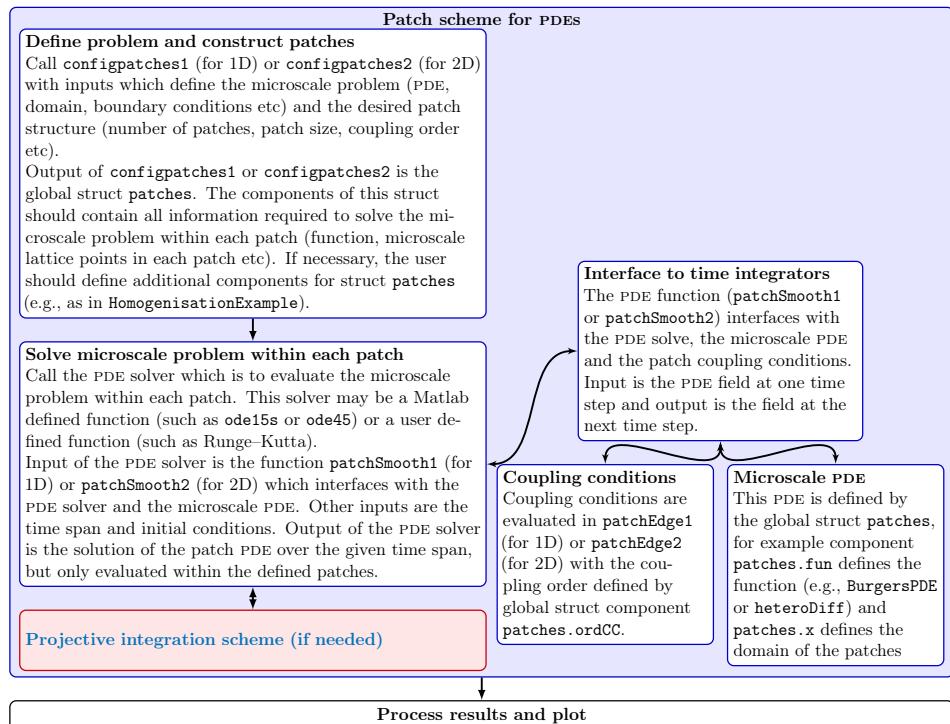


Figure 3

### 3 Projective integration of deterministic ODEs

*Subsubsection contents*

Scenario . . . . .	8
Main functions . . . . .	8
Minor functions . . . . .	8
3.1 PIRK2(): projective integration of second order accuracy . . . . .	9
Input . . . . .	9
Output . . . . .	10
3.2 egPIMM: Example projective integration of Michaelis–Menton kinetics . . . . .	11
3.2.1 Invoke projective integration . . . . .	12
Optional: request and plot the microscale bursts . . . . .	12
Optional: simulate backwards in time . . . . .	13
3.2.2 Code a burst of Michaelis–Menten enzyme kinetics . . . . .	14
3.3 PIG(): Projective Integration with a General macrosolver . . . . .	15
Input . . . . .	16
Output . . . . .	17
3.4 PIRK4(): projective integration of fourth order accuracy . . . . .	17
3.5 Example: PI using Runge–Kutta macrosolvers . . . . .	18
3.6 Example 2: PI using General macrosolvers . . . . .	21
3.7 Minor functions . . . . .	24
3.7.1 cdmc() . . . . .	24
Input . . . . .	24
Output . . . . .	24
3.7.2 bbgen() . . . . .	25
Input . . . . .	25
Output . . . . .	25
3.8 Explore: PI using constraint-defined manifold computing . . . . .	25
3.9 To do/discuss . . . . .	27

This section provides some good projective integration functions ([Gear & Kevrekidis 2003a,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 3.2](#) 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. Then `PIRK4()` is very similar to `PIRK2()`. Descriptions for the minor functions follow, and an example of the use of `cdmc()`.

### 3.1 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.

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

**Input** If there are no input arguments, then this function applies itself to the Michaelis–Menton example: see the code in ?? 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.

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

### *Subsection contents*

3.2.1	Invoke projective integration	12
3.2.2	Code a burst of Michaelis–Menton enzyme kinetics	14

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

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

As illustrated in [Figure 5](#), 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  $\epsilon$ .

### 3.2.1 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
```

First, [Section 3.2.2](#) 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).

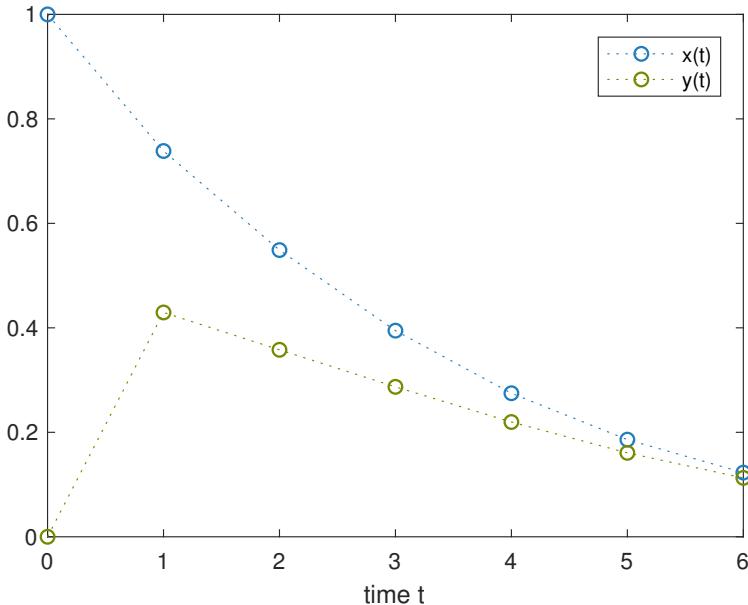
```
48 ts = 0:6
49 xs = PIRK2(@MMburst, 2*epsilon, ts, [1;0])
50 plot(ts,xs,'o:')
51 xlabel('time t'), legend('x(t)', 'y(t)')
52 pause(1)
```

[Figure 4](#) plots the macroscale results showing the long time decay of the Michaelis–Menten system on the slow manifold. [? \[§4\]](#) used this system as an example of their analysis of the convergence of Projective Integration.

**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 4](#)). To see the initial transient attraction to the slow manifold we plot some microscale data in [Figure 5](#). Two further output variables provide this microscale burst information.

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

**Figure 4:** Michaelis–Menton enzyme kinetics simulated with the projective integration of PIRK2(): macroscale samples.



[Figure 5](#) 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.

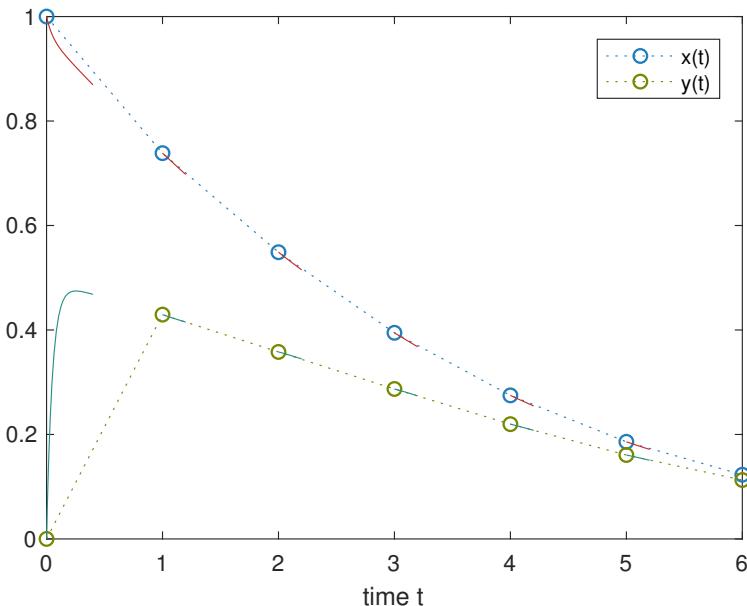
**Optional: simulate backwards in time** [Figure 6](#) shows that projective integration even simulates backwards in time along the slow manifold using short forward bursts. Such backwards macroscale simulations succeed despite the fast variable  $y(t)$ , when backwards in time, being viciously unstable. However, backwards integration appears to need longer bursts, here  $3\epsilon$ .

```

111 ts = 0:-1:-5
112 [xs,tMicro,xMicro] = PIRK2(@MMburst, 3*epsilon, ts, 0.2*[1;1]);
113 figure, plot(ts,xs,'o:',tMicro,xMicro)

```

**Figure 5:** 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.



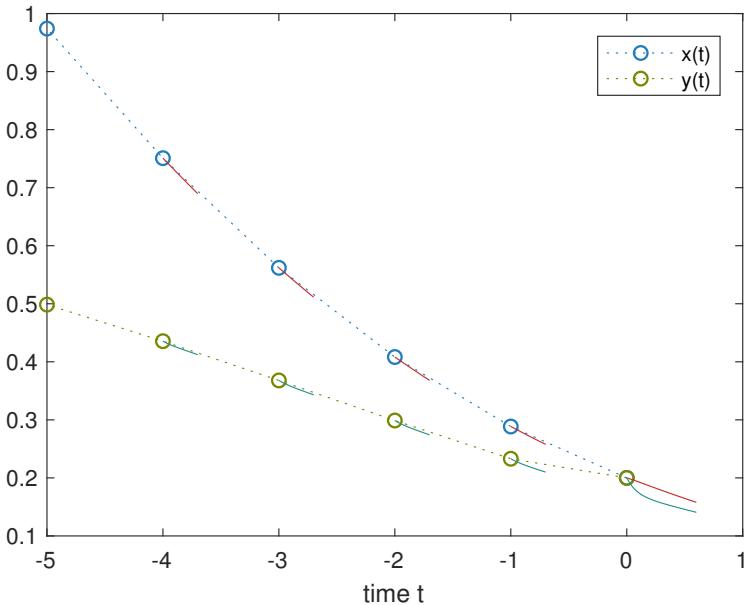
```
114 xlabel('time t'), legend('x(t)', 'y(t)')
```

### 3.2.2 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  $x = \mathbf{x}(1)$  and  $y = \mathbf{x}(2)$ . 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.

```
141 function [ts, xs] = MMBurst(ti, xi, bT)
142     global epsilon
143     dMMdt = @(t,x) [-x(1)+(x(1)+0.5)*x(2)
```

**Figure 6:** 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  $\epsilon = 0.1$ .



```

144      1/epsilon*( x(1)-(x(1)+1)*x(2) ) ];
145      [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
146  end

```

### 3.3 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.

### 3.4 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()`.

### 3.5 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.

```
12 clear
13 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,

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

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

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

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

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

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

```
41 dt = 0.001;
42 bT = 0.05;
```

As a rule of thumb, the time steps `dt` should satisfy  $dt \leq 1/|\text{fastband}(1)|$  and the time to simulate with each application of the microsolver, `micro.bT`, should be larger than or equal to  $1/|\text{fastband}(2)|$ . 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

```
49 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.

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

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

```
70 tSpan=0: 1 : 30;
71 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.

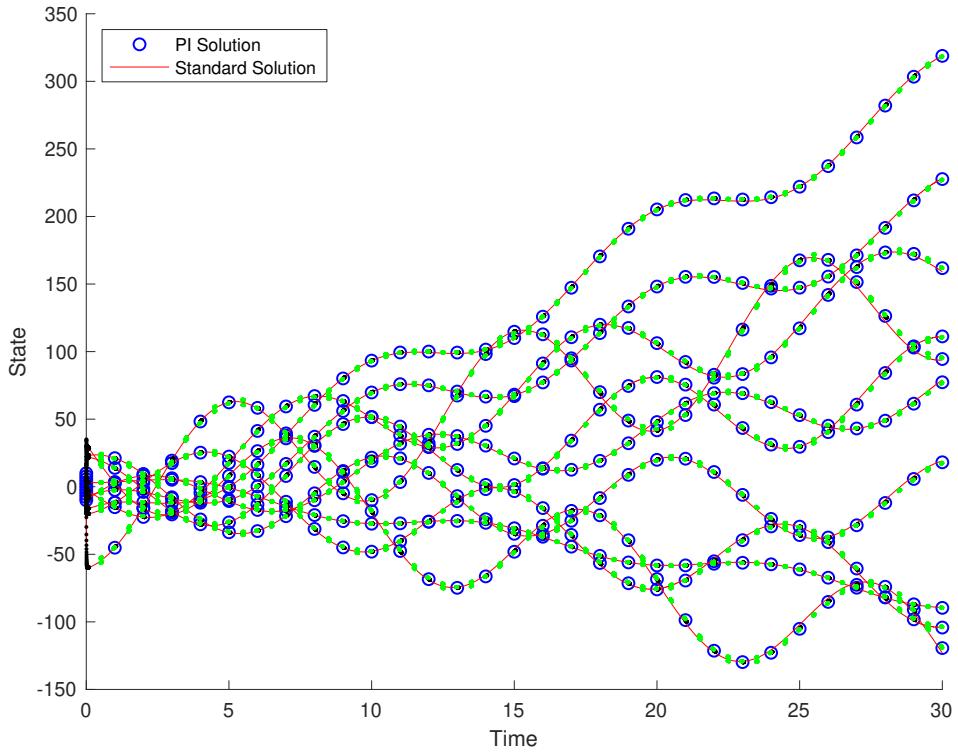
```
80 [x, tms, xms, rm] = PIRK4(solver, bT, tSpan, IC);
```

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

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

Figure 7 plots the output.

```
99 tmsr = rm.t; xmsr = rm.x;
100 clf()
```



**Figure 7:** Demonstration of PIRK4(). From initial conditions, the system rapidly transitions 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.

```

101 hold on
102 PI_sol=plot(tSpan,x,'bo');
103 std_sol=plot(tt,ode45x,'r');
104 plot(tms,xms,'k.');
105 plot(tmsr,xmsr,'g.');
106 legend([PI_sol(1),std_sol(1)],'PI Solution',...
107     'Standard Solution','Location','NorthWest')
108 xlabel('Time');
109 ylabel('State');

```

Save plot to a file.

```

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

```

### 3.6 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.

```
11 clear
```

Set time scale separation and model.

```

18 epsilon = 1e-4;
19 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.

```

26 solver = @(tIC, xIC,T) feval('ode45',f,[tIC tIC+T],xIC);
27 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.

```

34 IC = [1 3];
35 tspan=[0 40];
36 macro.tspan = tspan;
37 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.

```

45 tic
46 [t,x,tms,xms] = PIG(solver,bT,macro,IC);
47 tPI=toc;
48 fprintf(['PI took %f seconds, using ode45 as the ...
49      'macrosolver.\n'],tPI)
50 tic
51 [t45,xode45] = ode45(f,[tspan(1) tspan(end)],IC);
52 tODE45 = toc;
53
54 fprintf('Brute force ode45 took %f seconds.\n',tODE45)

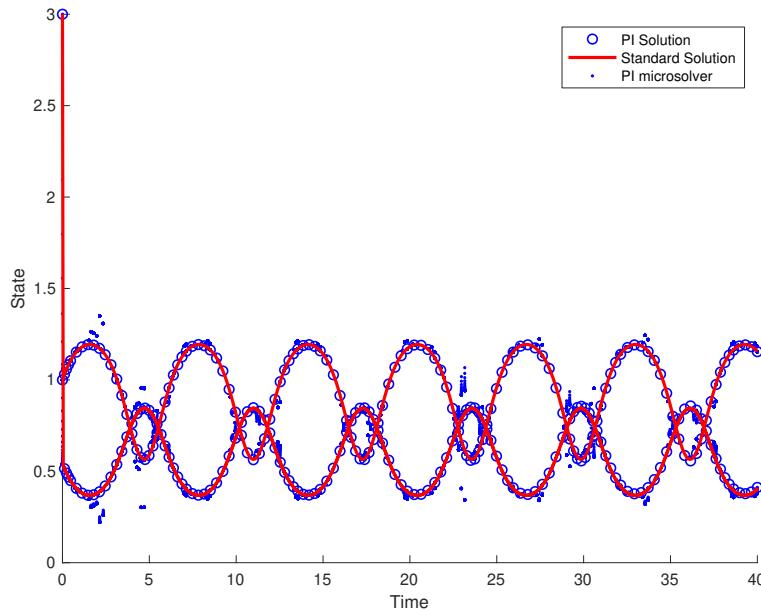
```

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

```

63 figure; set(gcf,'PaperPosition',[0 0 14 10])
64 hold on
65 PI=plot(tms,xms,'b.');
66 PI=plot(t,x,'bo');
67 ODE45=plot(t45,xode45,'r-','LineWidth',2);
68 legend([PI(1),ODE45(1),PI(1)],'PI Solution',...
69      'Standard Solution','PI microsolver')
70 xlabel('Time')
71 ylabel('State')
72 axis([0 40 0 3])
73
74 figure; set(gcf,'PaperPosition',[0 0 14 10])
75 hold on
76 PI2=plot(t,x,'bo');

```



**Figure 8:** Accurate simulation of a stiff nonautonomous system by PIG(). The microsolver is called on-the-fly by the macrosolver (here ode45).

```

77 ODE452=plot(t45,xode45,'r-','LineWidth',2);
78
79 legend([PI2(1),ODE452(1)],'PI Solution','Standard Solution')
80 xlabel('Time')
81 ylabel('State')
```

The output is plotted in Figure 8.

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 `cldmc()`; see Section 3.8.
- The mildly stiff problem in Example 3.5 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.

## 3.7 Minor functions

### 3.7.1 `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.

### 3.7.2 `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  $T$ .

## 3.8 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.

```
10 clear
```

Set a weak time scale separation and model.

```
17 epsilon = 1e-2;
18 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.

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

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

```
32 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.

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

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

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

Plot the output.

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

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

The output is plotted in Figure 9. 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**<sup>1</sup> 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.

### 3.9 To do/discuss

- 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 with a little fiddling. Then output at each user-requested coarse time is the end point of an application of the microsolver - better predictions for fast variables.
- Can maybe implement microsolvers that terminate a burst when the fast dynamics have settled using, for example, the ‘Events’ function handle in `ode23`.

---

<sup>1</sup>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.

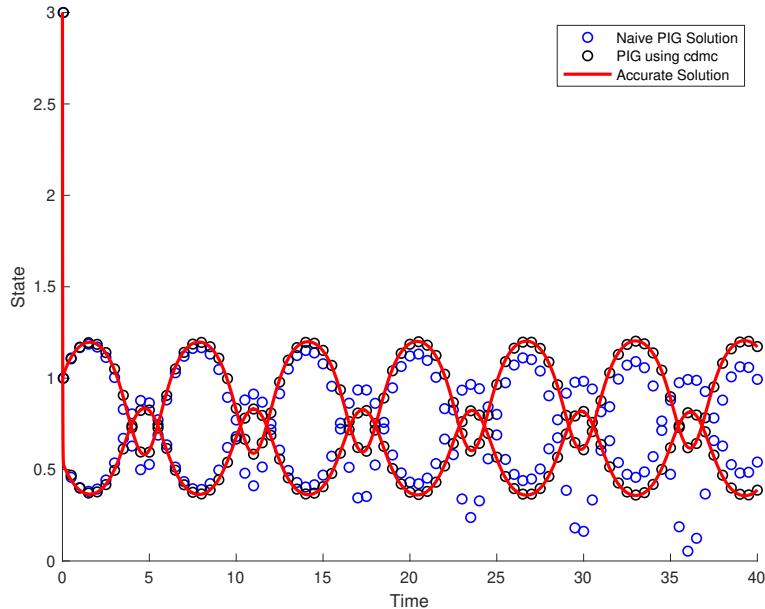


Figure 9: Accurate simulation of a weakly stiff nonautonomous system by PIG() using cdmc(), and an inaccurate solution using a naive application of PIG().

## 4 Patch scheme for given microscale discrete space system

*Subsubsection contents*

Quick start . . . . .	30
4.1 configPatches1(): configures spatial patches in 1D . . . . .	30
Input . . . . .	31
Output . . . . .	32
4.2 patchSmooth1(): interface to time integrators . . . . .	32
Input . . . . .	33
Output . . . . .	33
4.3 patchEdgeInt1(): sets edge values from interpolation over the macroscale . . . . .	33
Input . . . . .	34
Output . . . . .	34
4.4 BurgersExample: simulate Burgers' PDE on patches . . . . .	35
4.4.1 Script code to simulate a micro-scale space-time map . . . . .	35
Use projective integration . . . . .	37
4.4.2 burgersMap(): discretise the PDE microscale . . . . .	39
4.4.3 burgerBurst(): code a burst of the patch map . . . . .	39
4.5 HomogenisationExample: simulate heterogeneous diffusion in 1D . . . . .	40
4.5.1 Script to simulate via stiff or projective integration . . . . .	41
Conventional integration in time . . . . .	42
Use projective integration in time . . . . .	43
4.5.2 heteroDiff(): heterogeneous diffusion . . . . .	45
4.5.3 heteroBurst(): a burst of heterogeneous diffusion . . . . .	45
4.6 waterWaveExample: simulate a water wave PDE on patches . . . . .	46
4.6.1 Script code to simulate wave systems . . . . .	49
Conventional integration in time . . . . .	50
Use projective integration . . . . .	51
4.6.2 simpleWavePDE(): simple wave PDE . . . . .	51
4.6.3 waterWavePDE(): water wave PDE . . . . .	52
4.7 configPatches2(): configures spatial patches in 2D . . . . .	53

Input	54
Output	55
4.8 patchSmooth2(): interface to time integrators	55
Input	56
Output	57
4.9 patchEdgeInt2(): 2D patch edge values from 2D interpolation	57
Input	57
Output	58
4.10 wave2D: example of a wave on patches in 2D	58
4.10.1 Check on the linear stability of the wave PDE	59
4.10.2 Execute a simulation	60
4.10.3 Example of simple wave PDE inside patches	62
4.11 To do	63
4.12 Miscellaneous tests	63
4.12.1 patchEdgeInt1test: test the spectral interpolation	63
Test standard spectral interpolation	63
Now test spectral interpolation on staggered grid	65
Finish	66
4.13 patchEdgeInt2test: tests 2D spectral interpolation	66

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 structure 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 [????](#) for basic code that uses the provided functions to simulate Burgers' PDE and a nonlinear 'diffusion' PDE.

## 4.1 configPatches1(): configures spatial patches in 1D

*Subsubsection contents*

<code>Input</code>	31
<code>Output</code>	32

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

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

**Input** If invoked with no input arguments, then executes an example of simulating Burgers' PDE—see ?? 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  $\{-1, 0, \dots, 8\}$ .
- `ratio` (real) is the ratio of the half-width of a patch to the spacing of the patch mid-points: so `ratio = 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.
- `nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch. May be omitted. The default

is one (suitable for microscale lattices with only nearest neighbour interactions).

**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** × **nPatch** array of the regular spatial locations  $x_{ij}$  of the microscale grid points in every patch.
- **.nEdge** is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.

## 4.2 patchSmooth1(): interface to time integrators

*Subsubsection contents*

<b>Input</b>	33
<b>Output</b>	33

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.

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

*Subsubsection contents*

## Output . . . . . 34

Couples 1D patches across 1D space by computing their edge values from macroscale interpolation. This function is primarily used by `patchSmooth1` but is also useful for user graphics. 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`.

```
24 function u=patchEdgeInt1(u)
25 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()` which includes the following.
  - `.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 `.Cwtstl`

## Output

- `u` is `nSubP × nPatch × nVars` 2/3D array of the fields with edge values set by interpolation of patch centre-values.

## 4.4 BurgersExample: simulate Burgers' PDE on patches

### *Subsection contents*

4.4.1	Script code to simulate a micro-scale space-time map . . . . .	35
4.4.2	burgersMap(): discretise the PDE microscale . . . . .	39
4.4.3	burgerBurst(): code a burst of the patch map . . . . .	39

?? 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 10), 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.

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

1. configPatches1
2. burgerBurst  $\leftrightarrow$  patchSmooth1  $\leftrightarrow$  burgersMap
3. process results

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

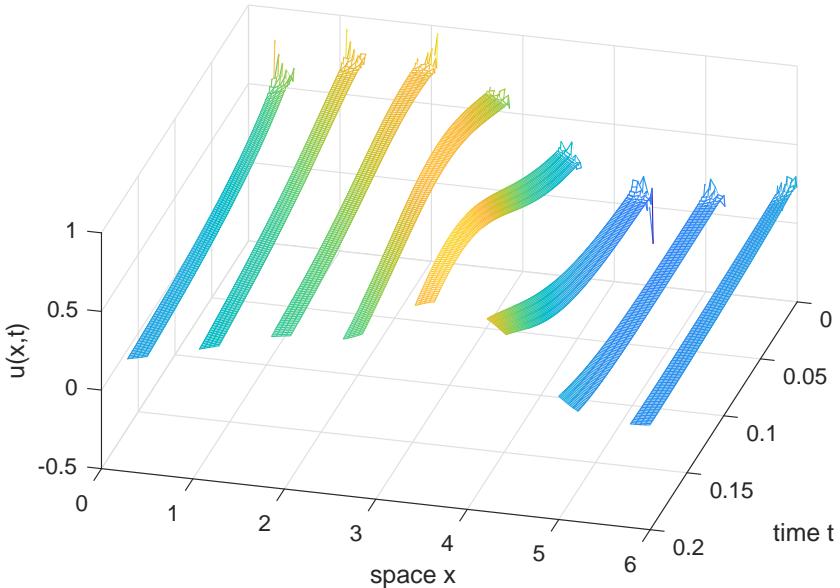
Establish global data struct for the Burgers' map (Section 4.4.2) solved on  $2\pi$ -periodic domain, with eight patches, each patch of half-size ratio 0.2, with seven points within each patch, and say fourth order interpolation provides edge-values that couple the patches.

```

47 clear all
48 global patches
49 nPatch = 8
50 ratio = 0.2
51 nSubP = 7
52 interpOrd = 4
53 Len = 2*pi
54 configPatches1(@burgersMap,[0 Len],nan,nPatch,interpOrd,ratio,nSubP);

```

**Figure 10:** a short time simulation of the Burgers' map (Section 4.4.2) on patches in space. It requires many very small time steps only just visible in this mesh.



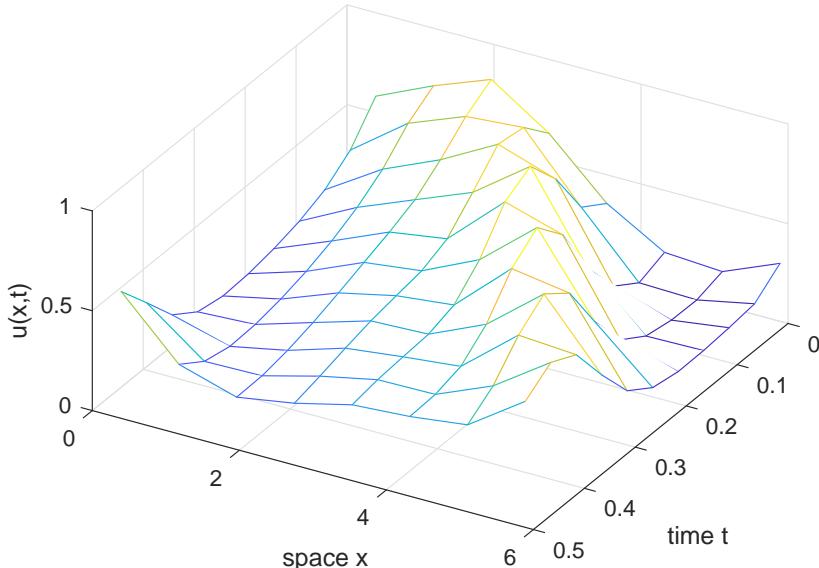
Set an initial condition, and simulate a burst of the micro-scale space-time map over a time 0.2 using the function `burgerBurst()` (Section 4.4.3).

```
62 u0 = 0.4*(1+sin(patches.x))+0.1*randn(size(patches.x));
63 [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.

```
71 figure(1),clf
72 xs = patches.x;  xs([1 end],:) = nan;
73 mesh(ts,xs(:,us'))
74 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
75 view(105,45)
76 set(gcf,'paperposition',[0 0 14 10])
```

**Figure 11:** macro-scale space-time field  $u(x, t)$  in a basic projective integration of the patch scheme applied to the micro-scale Burgers' map.



```
77 print('-depsc2','ps1BurgersMapU')
```

**Use projective integration** Around the micro-scale burst `burgerBurst()`, wrap the projective integration function `PIRK2()` of Section 3.1. Figure 11 shows the macroscale prediction of the patch centre values on macro-scale time-steps.

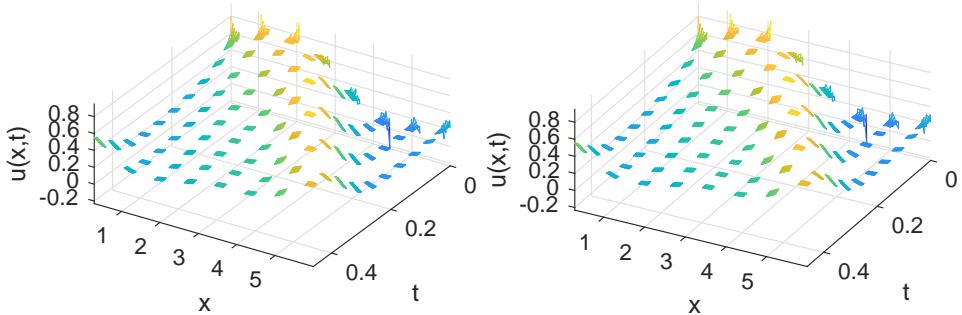
This second part of the script implements the following design.

1. configPatches1 (done in first part)
2. `PIRK2`  $\leftrightarrow$  `burgerBurst`  $\leftrightarrow$  `patchSmooth1`  $\leftrightarrow$  `burgersMap`
3. process results

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

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

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



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.

```

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

```

Plot the macroscale predictions of the mid-patch values to give the macroscale mesh of [Figure 11](#).

```

128 figure(2),clf
129 mid = (nSubP+1)/2;
130 mesh(ts,xs(mid,:),us(:,mid:nSubP:end)')
131 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
132 view(120,50)
133 set(gcf,'paperposition',[0 0 14 10])
134 print('-depsc2','ps1BurgersU')

```

Then plot the microscale mesh of the microscale bursts shown in [Figure 12](#) (a stereo pair). The details of the fine microscale mesh are almost invisible.

```

148 figure(3),clf
149 for k = 1:2, subplot(2,2,k)

```

```

150 mesh(tss, xs(:,uss'))
151 ylabel('x'), xlabel('t'), zlabel('u(x,t)')
152 axis tight, view(126-4*k,50)
153 end
154 set(gcf,'paperposition',[0 0 17 12])
155 print('-depsc2','ps1BurgersMicro')
```

#### 4.4.2 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).

```

172 function u = burgersMap(t,u,x)
173   dx = diff(x(2:3)); dt = dx^2/2;
174   i = 2:size(u,1)-1;
175   u(i,:) = u(i,:) +dt*( diff(u,2)/dx^2 ...
176     -20*u(i,:).*(u(i+1,:)-u(i-1,:))/(2*dx) );
177 end
```

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

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

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

```

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

Apply the microscale map over all time-steps in the burst, using `patchSmooth1` (Section 4.2) 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.

```

206 us = nan(ndt+1,numel(ui));
207 us(1,:) = reshape(ui,1,[]);
208 for j = 1:ndt
209     ui = patchSmooth1(ts(j),ui);
210     us(j+1,:) = reshape(ui,1,[]);
211 end

```

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

```

218 ts(ndt+1) = ti+bT;
219 us(ndt+1,:) = us(ndt,:)
220     + diff(ts(ndt:ndt+1))/dt*diff(us(ndt:ndt+1,:));
221 end

```

Fin.

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

### *Subsection contents*

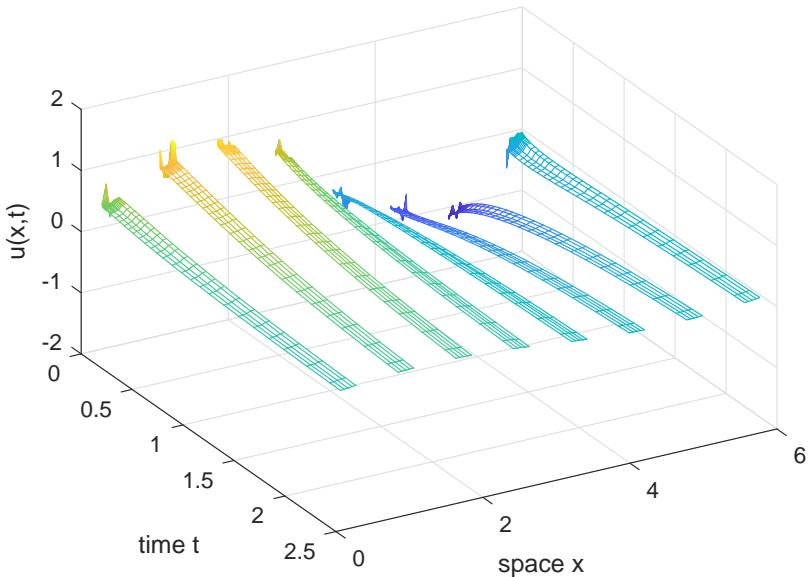
4.5.1	Script to simulate via stiff or projective integration . . .	41
4.5.2	heteroDiff(): heterogeneous diffusion . . . . .	45
4.5.3	heteroBurst(): a burst of heterogeneous diffusion . .	45

Figure 13 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 ? 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.

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

1. configPatches1
2. ode15s  $\leftrightarrow$  patchSmooth1  $\leftrightarrow$  heteroDiff

**Figure 13:** the diffusing field  $u(x, t)$  in the patch (gap-tooth) scheme applied to microscale heterogeneous diffusion.



### 3. process results

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

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

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

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

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

```
49 clear all
50 mPeriod = 3
```

```
51 cDiff = exp(randn(mPeriod,1))
52 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 ? showed is accurate). Quadratic (fourth-order) interpolation provides values for the inter-patch coupling conditions.

```
65 global patches
66 nPatch = 9
67 ratio = 0.2
68 nSubP = 2*mPeriod+1
69 Len = 2*pi;
70 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.

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

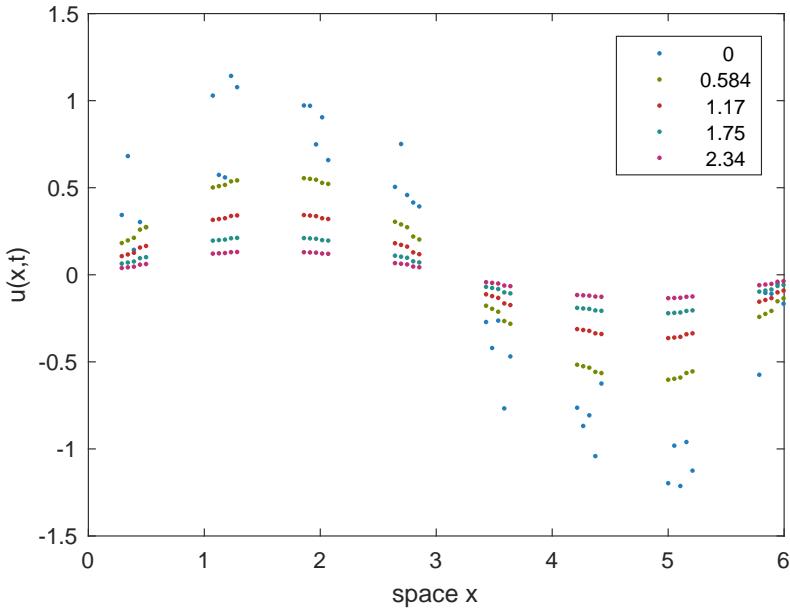
**Conventional integration in time** Set an initial condition, and here integrate forward in time using a standard method for stiff systems—because of the simplicity of linear problems this method works quite efficiently here. Integrate the interface **patchSmooth1** (Section 4.2) to the microscale differential equations.

```
92 %u0 = sin(patches.x)+0.2*randn(nSubP,nPatch);
93 u0 = exp(-2*(patches.x-Len/2).^2).*(1+0.1*rand(nSubP,nPatch));
94 [ts,ucts] = ode15s(@patchSmooth1, [0 2/cHomo], u0(:));
```

Plot the simulation in Figure 13.

```
100 figure(1),clf
101 xs = patches.x; xs([1 end],:) = nan;
102 mesh(ts,xs(:,ucts'), view(60,40)
103 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
```

**Figure 14:** field  $u(x, t)$  shows basic projective integration of patches of heterogeneous diffusion: different colours correspond to the times in the legend.



```
104 set(gcf, 'paperposition', [0 0 14 10])
105 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 3.1), of bursts of simulation from `heteroBurst` (Section 4.5.3), as illustrated by Figure 14.

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

1. `configPatches1` (done in first part)
2. `PIRK2`  $\leftrightarrow$  `heteroBurst`  $\leftrightarrow$  micro-integrator  $\leftrightarrow$  `patchSmooth1`  $\leftrightarrow$  `heteroDiff`

### 3. process results

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

```
134 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.

```
146 ts = linspace(0,2/cHomo,7)
147 bT = 3*( ratio*Len/nPatch )^2/cHomo
148 addpath('..../ProjInt','..../RKint')
149 [us,tss,uss] = PIRK2(@heteroBurst, bT, ts, u0(:));
```

Plot the macroscale predictions to draw [Figure 14](#).

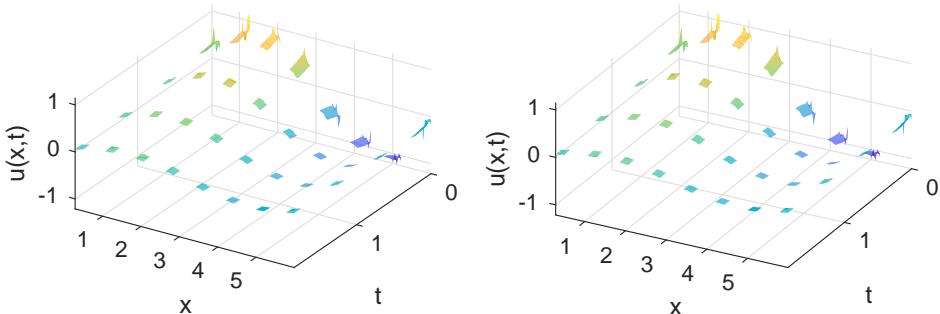
```
156 figure(2),clf
157 plot(xs(:,us,'.')
158 ylabel('u(x,t)'), xlabel('space x')
159 legend(num2str(ts',3))
160 set(gcf,'paperposition',[0 0 14 10])
161 print('-depsc2','ps1HomogenisationU')
```

Also plot a surface detailing the microscale bursts as shown in [Figure 15](#).

```
174 figure(3),clf
175 for k = 1:2, subplot(2,2,k)
176 surf(tss,xs(:,uss,'EdgeColor','none')
177 ylabel('x'), xlabel('t'), zlabel('u(x,t)')
178 axis tight, view(126-4*k,45)
179 end
180 set(gcf,'paperposition',[0 0 17 12])
181 print('-depsc2','ps1HomogenisationMicro')
```

End of the script.

Figure 15: stereo pair of the field  $u(x, t)$  during each of the microscale bursts used in the projective integration.



#### 4.5.2 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 4.2), 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**.<sup>2</sup>

```

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

```

#### 4.5.3 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:

---

<sup>2</sup>Use **bsxfun()** as pre-2017 Matlab versions may not support auto-replication.

- `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.

```

229 function [ts, ucts] = heteroBurst(ti, ui, bT)
230 switch 'rk2'
231 case '23', [ts,ucts] = ode23 (@patchSmooth1,[ti ti+bT],ui(:));
232 case '15s', [ts,ucts] = ode15s(@patchSmooth1,[ti ti+bT],ui(:));
233 case 'rk2', ts = linspace(ti,ti+bT,200)';
234             ucts = rk2int(@patchSmooth1,ts,ui(:));
235 end
236 end

```

Fin.

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

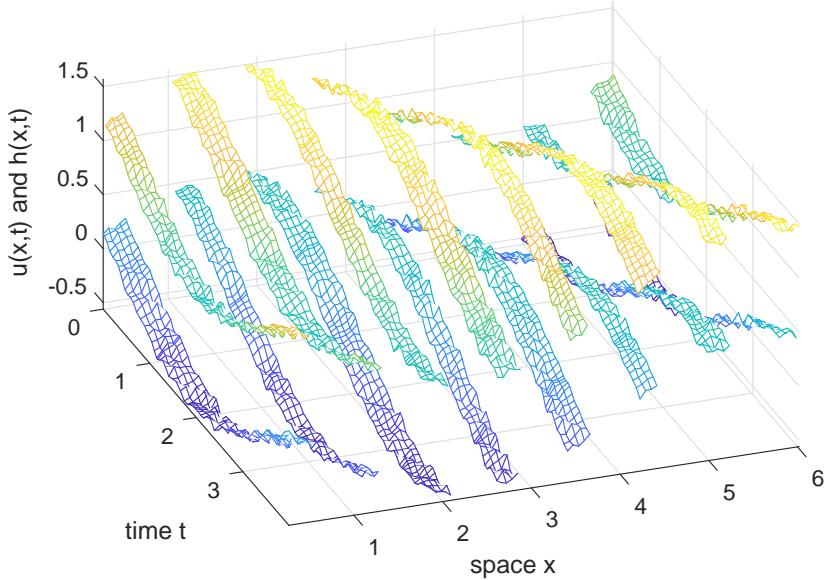
### *Subsection contents*

4.6.1	Script code to simulate wave systems . . . . .	49
4.6.2	<code>simpleWavePDE()</code> : simple wave PDE . . . . .	51
4.6.3	<code>waterWavePDE()</code> : water wave PDE . . . . .	52

Figure 16 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 4.6.2, may be adapted by a user to a wide variety of 1D wave and near-wave systems. For example, the differential equations of Section 4.6.3 describes the nonlinear microscale simulator of the nonlinear shallow water wave PDE derived from the Smagorinski model of turbulent flow (??).

**Figure 16:** 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.

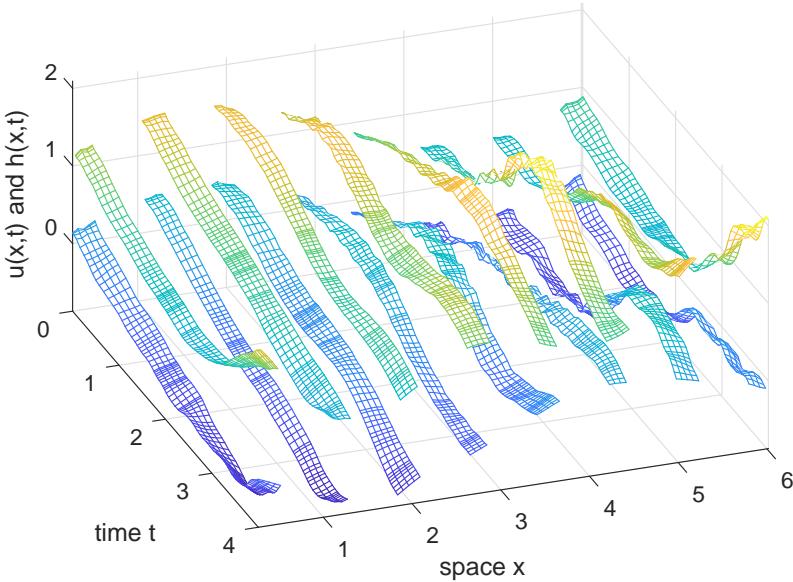


Often, wave-like systems are written in terms of two conjugate variables, for example, position and momentum density, electric and magnetic fields, and water depth  $h(x, t)$  and mean lateral velocity  $u(x, t)$  as herein. The approach developed in this section applies to any wave-like system in the form

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

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 4.6.3](#) 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 of fluid depth  $h(x, t)$  and depth-averaged lateral velocity  $u(x, t)$  the model

Figure 17: 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.



PDEs are

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

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

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 17 shows one simulation of this system—for the same initial condition as Figure 16.

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.

#### 4.6.1 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  $\leftrightarrow$  patchSmooth1  $\leftrightarrow$  simpleWavePDE
3. process results
4. ode15s  $\leftrightarrow$  patchSmooth1  $\leftrightarrow$  waterWavePDE
5. process results

Establish the global data struct **patches** 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).

```

71 clear all
72 global patches
73 nPatch = 8
74 ratio = 0.2
75 nSubP = 11 %of the form 4*n-1
76 Len = 2*pi;
77 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.

```

85 uPts = mod( bsxfun(@plus,(1:nSubP)',(1:nPatch)) ,2);
86 hPts = find(1-uPts);
87 uPts = find(uPts);
88 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).

```
96 U0 = nan(nSubP,nPatch);
97 U0(hPts) = 1+0.5*sin(patches.x(hPts));
98 U0(uPts) = 0+0.5*sin(patches.x(uPts));
99 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.

```
108 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.

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

Plot the simulation.

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

Print the output.

```
133 set(gcf,'paperposition',[0 0 14 10])
134 if k==1, print('-depsc2','ps1WaveCtsUH')
135 else print('-depsc2','ps1WaterWaveCtsUH')
136 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.

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

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

#### 4.6.2 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.

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

Compute the PDE derivatives at interior points of the patches.

```
246     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.

253      $ht(i,:) = -(U(i+1,:)-U(i-1,:))/(2\cdot dx);$

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.

259      $Ut(i,:) = -(U(i+1,:)-U(i-1,:))/(2\cdot dx);$

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

265      $Ut(\text{patches}.hPts) = ht(\text{patches}.hPts);$

266 end

#### 4.6.3 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()**.

278 function  $Ut = \text{waterWavePDE}(t, U, x)$

279     global patches

280      $rabs = @(\mathbf{u}) \sqrt{1e-4 + \mathbf{u}.^2};$

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

286      $dx = \text{diff}(x(2:3));$

287      $Ut = \text{nan}(\text{size}(U)); \quad ht = Ut;$

288      $i = 2:\text{size}(U,1)-1;$

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

294      $ii = i(2:\text{end}-1);$

295      $V = Ut;$

296      $V(ii,:) = (U(ii+1,:)+U(ii-1,:))/2;$

297      $V(1:2,:) = 2*U(2:3,:)-V(3:4,:);$

298      $V(\text{end}-1:\text{end},:) = 2*U(\text{end}-2:\text{end}-1,:)-V(\text{end}-3:\text{end}-2,:);$

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

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

Correspondingly estimate the terms in the momentum PDE:  $u$ -values in  $U_i$  and  $V_{i\pm 1}$ ; and  $h$ -values in  $V_i$  and  $U_{i\pm 1}$ .

310     $Ut(i,:) = -0.985*(U(i+1,:)-U(i-1,:))/(2*dx) \dots$

311        $-0.003*U(i,:).*rabs(U(i,:)./V(i,:)) \dots$

312        $-1.045*U(i,:).* (V(i+1,:)-V(i-1,:))/(2*dx) \dots$

313        $+0.26*rabs(V(i,:).*U(i,:)).*(V(i+1,:)-2*U(i,:)+V(i-1,:))/dx^2/2;$

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

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

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

326     $Ut(\text{patches}.hPts) = ht(\text{patches}.hPts);$

327    end

Fin.

## 4.7 configPatches2(): configures spatial patches in 2D

### *Subsubsection contents*

Input	.....	.....	54
Output	.....	.....	55

Makes the struct `patches` for use by the patch/gap-tooth time derivative function `patchSmooth2()`. `??` lists an example of its use.

```
17 function configPatches2(fun,Xlim,BCs,nPatch,ordCC,ratio,nSubP,nEdge)
18 global patches
```

**Input** If invoked with no input arguments, then executes an example of simulating a nonlinear diffusion PDE relevant to the lubrication flow of a thin layer of fluid—see `??` for the example code.

- `fun` is the name of the user function, `fun(t,u,x,y)`, that computes time derivatives (or time-steps) of quantities on the patches.
- `Xlim` array/vector giving the macro-space domain of the computation: patches are equi-spaced over the interior of the rectangle  $[Xlim(1), Xlim(2)] \times [Xlim(3), Xlim(4)]$ : if of length two, then use the same interval in both directions, otherwise `Xlim(1:4)` give the interval in each direction.
- `BCs` somehow will define the macroscale boundary conditions. Currently, `BCs` is ignored and the system is assumed macro-periodic in the domain.
- `nPatch` determines the number of equi-spaced spaced patches: if scalar, then use the same number of patches in both directions, otherwise `nPatch(1:2)` give the number in each direction.
- `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 {0}.
- `ratio` (real) is the ratio of the half-width of a patch to the spacing of the patch mid-points: so `ratio = 1/2` means the patches abut; and `ratio = 1` would be overlapping patches as in holistic discretisation: if scalar, then use the same ratio in both directions, otherwise `ratio(1:2)` give the ratio in each direction.
- `nSubP` is the number of equi-spaced microscale lattice points in each patch: if scalar, then use the same number in both directions, otherwise

**nSubP(1:2)** gives the number in each direction. Must be odd so that there is a central lattice point.

- **nEdge** is, for each patch, the number of edge values set by interpolation at the edge regions of each patch. May be omitted. The default is one (suitable for microscale lattices with only nearest neighbours. interactions).

**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,y)` 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—not yet implemented.
  - `.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(1) × nPatch(1)` array of the regular spatial locations  $x_{ij}$  of the microscale grid points in every patch.
  - `.y` is `nSubP(2) × nPatch(2)` array of the regular spatial locations  $y_{ij}$  of the microscale grid points in every patch.
  - `.nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.

## 4.8 patchSmooth2(): interface to time integrators

### *Subsubsection contents*

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 via the previously established global struct `patches`.

```
23 function dudt=patchSmooth2(t,u)
24 global patches
```

## Input

- `u` is a vector of length `prod(nSubP) · prod(nPatch) · nVars` where there are `nVars` field values at each of the points in the `nSubP(1) × nSubP(2) × nPatch(1) × nPatch(2)` grid.
- `t` is the current time to be passed to the user's time derivative function.
- `patches` a struct set by `configPatches2()` with the following information used here.
  - `.fun` is the name of the user's function `fun(t,u,x,y)` that computes the time derivatives on the patchy lattice. The array `u` has size `nSubP(1) × nSubP(2) × nPatch(1) × nPatch(2) × nVars`. Time derivatives must be computed into the same sized array, but herein the patch edge values are overwritten by zeros.
  - `.x` is `nSubP(1) × nPatch(1)` 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.
  - `.y` is similarly `nSubP(2) × nPatch(2)` array of the spatial locations  $y_{ij}$  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  $\text{prod}(\text{nSubP}) \cdot \text{prod}(\text{nPatch}) \cdot \text{nVars}$  vector of time derivatives, but with patch edge values set to zero.

## 4.9 patchEdgeInt2(): sets 2D patch edge values from 2D macroscale interpolation

*Subsubsection contents*

Input . . . . .	57
Output . . . . .	58

Couples 2D patches across 2D space by computing their edge values via macroscale interpolation. 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**.

```
20 function u=patchEdgeInt2(u)
21 global patches
```

## Input

- **u** is a vector of length  $\text{nx} \cdot \text{ny} \cdot \text{Nx} \cdot \text{Ny} \cdot \text{nVars}$  where there are **nVars** field values at each of the points in the  $\text{nx} \times \text{ny} \times \text{Nx} \times \text{Ny}$  grid on the  $\text{Nx} \times \text{Ny}$  array of patches.
- **patches** a struct set by **configPatches2()** which includes the following information.
  - **.x** is  $\text{nx} \times \text{Nx}$  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.

- `.y` is similarly  $\text{ny} \times \text{Ny}$  array of the spatial locations  $y_{ij}$  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 only  $\{0\}$ .
- `.Cwtsr` and `.Cwtss`—not yet used

## Output

- `u` is  $\text{nx} \times \text{ny} \times \text{Nx} \times \text{Ny} \times \text{nVars}$  array of the fields with edge values set by interpolation.

## 4.10 wave2D: example of a wave on patches in 2D

*Subsection contents*

4.10.1 Check on the linear stability of the wave PDE . . . . .	59
4.10.2 Execute a simulation . . . . .	60
4.10.3 Example of simple wave PDE inside patches . . . . .	62

For  $u(x, y, t)$ , test and simulate the simple wave PDE in 2D space:

$$\frac{\partial^2 u}{\partial t^2} = \nabla^2 u.$$

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

1. configPatches2
2. ode15s integrator  $\leftrightarrow$  patchSmooth2  $\leftrightarrow$  wavePDE
3. process results

Establish global patch data struct to interface with a function coding the wave PDE: to be solved on  $2\pi$ -periodic domain, with  $9 \times 9$  patches, spectral interpolation couples the patches, each patch of half-size ratio 0.25, and with  $5 \times 5$  points within each patch.

```

33 clear all, close all
34 global patches
35 nSubP = 5;
36 nPatch = 9;
37 configPatches2(@wavePDE, [-pi pi], nan, nPatch, 0, 0.1, nSubP);

```

#### 4.10.1 Check on the linear stability of the wave PDE

Set a zero equilibrium as basis. Then find the patch-interior points as the only ones to vary in order to construct the Jacobian.

```

48 disp('Check linear stability of the wave scheme')
49 uv0=zeros(nSubP,nSubP,nPatch,nPatch,2);
50 uv0([1 end],:,:,:,:)=nan;
51 uv0(:,[1 end],:,:,:,:)=nan;
52 i=find(~isnan(uv0));

```

Now construct the Jacobian. Since linear wave PDE, use large perturbations.

```

58 small=1;
59 jac=nan(length(i));
60 sizejac=size(jac)
61 for j=1:length(i)
62     uv=uv0(:);
63     uv(i(j))=uv(i(j))+small;
64     tmp=patchSmooth2(0,uv)/small;
65     jac(:,j)=tmp(i);
66 end

```

Now explore the eigenvalues a little: find the ten with the biggest real-part; if small enough, then the method may be good.

```

72 evals=eig(jac);
73 nEvals=length(evals)
74 [~,k]=sort(-abs(real(evals)));
75 evalsWithBiggestRealPart=evals(k(1:10))
76 if abs(real(evals(k(1))))>1e-4

```

```

77     warning('eigenvalue failure: real-part > 1e-4')
78     return, end

```

Check eigenvalues close to true waves of the PDE (not yet the micro-discretised equations).

```

85 kwave=0:(nPatch-1)/2;
86 freq=sort(reshape(sqrt(kwave'.^2+kwave.^2),1,[]));
87 freq= freq(diff([-1 freq])>1e-9);
88 freqerr=[freq; min(abs(imag(evals)-freq))]

```

#### 4.10.2 Execute a simulation

Set a Gaussian initial condition using auto-replication of the spatial grid: here **u0** and **v0** are in the form required for computation:  $n_x \times n_y \times N_x \times N_y$ .

```

101 x = reshape(patches.x,nSubP,1,[],1);
102 y = reshape(patches.y,1,nSubP,1,[]);
103 u0 = exp(-x.^2-y.^2);
104 v0 = zeros(size(u0));

```

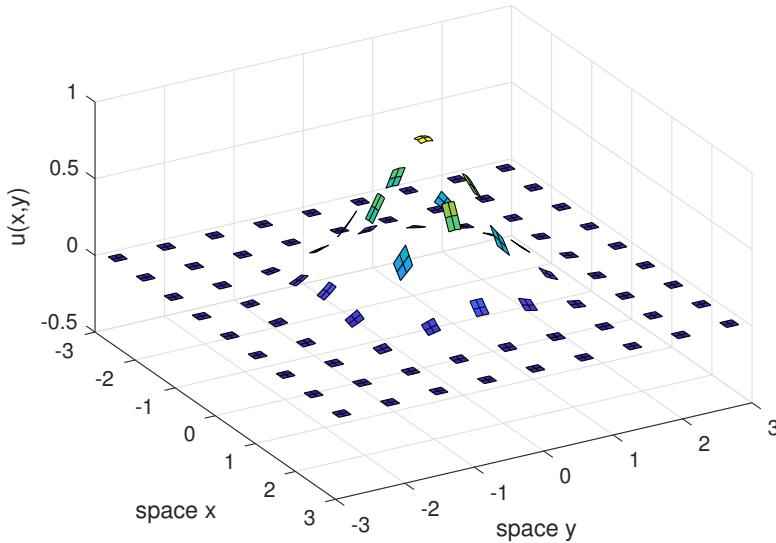
Initiate a plot of the simulation using only the microscale values interior to the patches: set  $x$  and  $y$ -edges to **nan** to leave the gaps. Start by showing the initial conditions of ?? while the simulation computes. To mesh/surf plot we need to ‘transpose’ to size  $n_x \times N_x \times n_y \times N_y$ , then reshape to size  $n_x \cdot N_x \times n_y \cdot N_y$ .

```

114 x=patches.x; y=patches.y;
115 x([1 end],:)=nan; y([1 end],:)=nan;
116 u = reshape(permute(u0,[1 3 2 4]), [numel(x) numel(y)]);
117 usurf = surf(x(:,y(:,u')));
118 axis([-3 3 -3 3 -0.5 1]), view(60,40)
119 xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
120 drawnow
121 set(gcf,'paperposition',[0 0 14 10])
122 print('-depsc','wave2Dic.eps')

```

**Figure 18:** initial field  $u(x, y, t)$  at time  $t = 0$  of the patch scheme applied to the simple wave PDE: [Figure 19](#) plots the computed field at time  $t = 6$ .



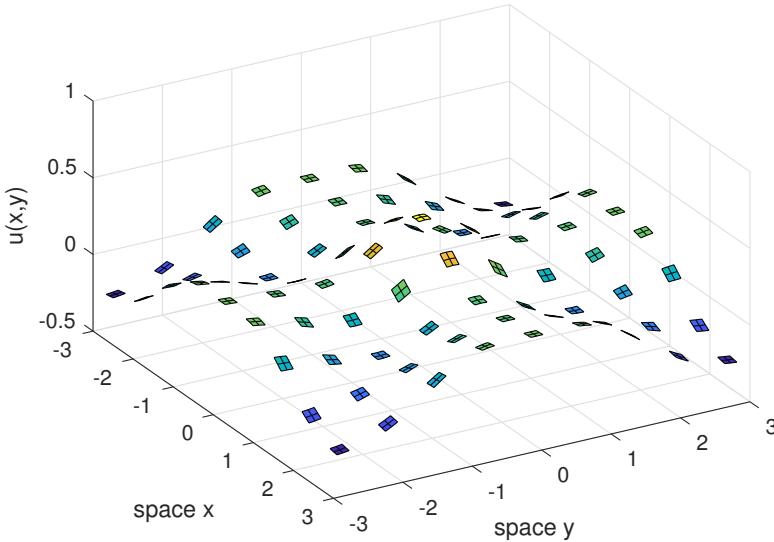
Integrate in time using standard functions.

```
136 disp('Wait while we simulate u_t=v, v_t=u_xx+u_yy')
137 [ts, uvs] = ode15s(@patchSmooth2, [0 1], [u0(:); v0(:)]);
```

Animate the computed simulation to end with [Figure 19](#). Subsample to plot at most 200 times.

```
144 di = ceil(length(ts)/200);
145 for i = [1:di:length(ts)-1 length(ts)]
146   uv = patchEdgeInt2(uvs(i,:));
147   uv = reshape(permute(uv,[1 3 2 4 5]), [numel(x) numel(y) 2]);
148   usurf.ZData = uv(:,:,1)';
149   title(['wave PDE on patches: time = ' num2str(ts(i))])
150   pause(0.1)
151 end
152 title('')
```

Figure 19: field  $u(x, y, t)$  at time  $t = 6$  of the patch scheme applied to the simple wave PDE with initial condition in Figure 18.



```
153 set(gcf,'paperposition',[0 0 14 10])
154 print('-depsc',['wave2Dt' num2str(ts(end)) '.eps'])
```

#### 4.10.3 Example of simple wave PDE inside patches

As a microscale discretisation of  $u_{tt} = \nabla^2(u)$ , so code  $\dot{u}_{ijkl} = v_{ijkl}$  and  $\ddot{v}_{ijkl} = \frac{1}{\delta x^2}(u_{i+1,j,k,l} - 2u_{i,j,k,l} + u_{i-1,j,k,l}) + \frac{1}{\delta y^2}(u_{i,j+1,k,l} - 2u_{i,j,k,l} + u_{i,j-1,k,l})$ .

```
174 function uvt = wavePDE(t,uv,x,y)
175 if ceil(t+1e-7)-t<2e-2, simTime=t, end %track progress
176 dx=diff(x(1:2)); dy=diff(y(1:2)); % micro-scale spacing
177 i=2:size(uv,1)-1; j=2:size(uv,2)-1; % interior patch-points
178 uvt = nan(size(uv)); % preallocate storage
179 uvt(i,j,:,:,:1) = uv(i,j,:,:,:2);
180 uvt(i,j,:,:,:2) = diff(uv(:,j,:,:,:1),2,1)/dx^2 ...
181 %+ diff(uv(i,:,:,:,1),2,2)/dy^2;
```

182 end

## 4.11 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.

## 4.12 Miscellaneous tests

### 4.12.1 patchEdgeInt1test: test the spectral interpolation

*Subsubsection contents*

Test standard spectral interpolation . . . . .	63
Now test spectral interpolation on staggered grid . . . . .	65
Finish . . . . .	66

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

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

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

```

24 for nPatch=5:10
25 nPatch=nPatch
26 Len=10*rand
27 ratio=0.5*rand
28 configPatches1(@sin,[0,Len],nan,nPatch,0,ratio,nSubP);
29 kMax=floor((nPatch-1)/2);

```

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

```

37 for k=-kMax:kMax
38     u0=exp(1i*k*patches.x*2*pi/Len);
39     ui=patchEdgeInt1(u0(:));
40     normError=norm(ui-u0);
41     if abs(normError)>5e-14
42         normError=normError
43         error(['failed single var interpolation k=' num2str(k)])
44     end
45 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.

```

54 for k=1:nPatch/2
55     u0=sin(k*patches.x*2*pi/Len);
56     v0=cos(k*patches.x*2*pi/Len);
57     uvi=patchEdgeInt1([u0(:);v0(:)]);
58     normuError=norm(uvi(:,1)-u0)*norm(u0(i0,:));
59     normvError=norm(uvi(:,2)-v0)*norm(v0(i0,:));
60     if abs(normuError)+abs(normvError)>2e-13
61         normuError=normuError, normvError=normvError
62         error(['failed double field interpolation k=' num2str(k)])
63     end
64 end

```

End the for-loop over various geometries.

```
71 end
```

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

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

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

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

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

```
100 fprintf('Single field-pair test.\n')
101 for k=-kMax:kMax
102     U0=nan(nSubP,nPatch);
103     U0(hPts)=rand*exp(+1i*k*patches.x(hPts)*2*pi/Len);
104     U0(uPts)=rand*exp(-1i*k*patches.x(uPts)*2*pi/Len);
105     Ui=patchEdgeInt1(U0(:));
106     normError=norm(Ui-U0);
107     if abs(normError)>5e-14
108         normError=normError
109         error(['failed single sys interpolation k=' num2str(k)])
110     end
111 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.

```

121 fprintf('Two field-pairs test.\n')
122 x0=patches.x((nSubP+1)/2,1);
123 patches.x=patches.x-x0;
124 for k=1:nPatch/4
125     U0=nan(nSubP,nPatch); V0=U0;
126     U0(hPts)=rand*sin(k*patches.x(hPts)*2*pi/Len);
127     U0(uPts)=rand*sin(k*patches.x(uPts)*2*pi/Len);
128     V0(hPts)=rand*cos(k*patches.x(hPts)*2*pi/Len);
129     V0(uPts)=rand*cos(k*patches.x(uPts)*2*pi/Len);
130     UVi=patchEdgeInt1([U0(:);V0(:)]);
131     normuError=norm(UVi(:,1:2:nPatch,1)-U0(:,1:2:nPatch))*norm(U0(i0,2:132
132         +norm(UVi(:,2:2:nPatch,1)-U0(:,2:2:nPatch))*norm(U0(i0,1:2:nPatch));
133     normuError=norm(UVi(:,1:2:nPatch,2)-V0(:,1:2:nPatch))*norm(V0(i0,2:134
134         +norm(UVi(:,2:2:nPatch,2)-V0(:,2:2:nPatch))*norm(V0(i0,1:2:nPatch));
135     if abs(normuError)+abs(normvError)>2e-13
136         normuError=normuError, normvError=normvError
137         error(['failed double field interpolation k=' num2str(k)])
138     end
139 end
End for-loop over patches
146 end

```

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

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

#### 4.13 patchEdgeInt2test: tests 2D spectral interpolation

Try 99 realisations of random tests.

```

11 clear all, close all
12 global patches
13 for realisation=1:99

```

Choose and configure random sized domains, random sub-patch resolution, random size-ratios, random number of periodic-patches.

```

19 Lx=1+3*rand, Ly=1+3*rand
20 nSubP=1+2*randi(3,1,2)
21 ratios=rand(1,2)/2
22 nPatch=2+randi(4,1,2)
23 configPatches2(@sin,[0 Lx 0 Ly],nan,nPatch,0,ratios,nSubP)

```

Choose a random number of fields, then generate trigonometric shape with random wavenumber and random phase shift.

```

29 nV=randi(3)
30 [nx,Nx]=size(patches.x);
31 [ny,Ny]=size(patches.y);
32 u0s=nan(nx,ny,Nx,Ny,nV);
33 for iV=1:nV
34     kx=randi([0 ceil((nPatch(1)-1)/2)])
35     ky=randi([0 ceil((nPatch(2)-1)/2)])
36     phix=pi*rand*(2*kx~=nPatch(1))
37     phiy=pi*rand*(2*ky~=nPatch(2))
38     % generate 2D array via auto-replication
39     u0=sin(2*pi*kx*patches.x(:)/Lx+phix) ...
40         .*sin(2*pi*ky*patches.y(:)'/Ly+phiy);
41     % reshape into 4D array
42     u0=reshape(u0,[nx Nx ny Ny]);
43     u0=permute(u0,[1 3 2 4]);
44     % store into 5D array
45     u0s(:,:,:,:,iV)=u0;
46 end

```

Copy and NaN the edges, then interpolate

```

52 u=u0s; u([1 end],:,:,:,:)=nan; u(:,[1 end],:,:,:,:)=nan;

```

```
53 u=patchEdgeInt2(u(:));
```

If there is an error in the interpolation then abort the script for checking:  
record parameter values and inform.

```
59 err=u-u0s;
60 normerr=norm(err(:))
61 if normerr>1e-12, error('2D interpolation failed'), end
62 end
```

## 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**.
- Create a LaTeX file **Dir/funs.tex**: establish as one LaTeX section that `\input{Dir/*.m}`s the files of the functions in the class, example scripts of use, and possibly test scripts, [Table 1](#).
- Each such **Dir/funs.tex** file is to be included from the main LaTeX file **Doc/equationFreeDoc.tex** so that people can most easily work on one section at a time:
  - put `\include{fun}` into **Doc/equationFreeDoc.tex**;
  - to include we have to use a soft link so at the command line in the directory **Doc** execute `ln -s ../Dir/funs.tex`<sup>3</sup>
- Each toolbox function is documented as a separate subsection, with tests and examples as separate subsections.
- Each function-subsection and test-subsection is to be created as a MATLAB/Octave **Dir/\*.m** file, say **Dir/fun1.m**, so that users simply invoke the function in MATLAB/Octave as usual by `fun1(...)`.

Some editors may need to be told that **fun1.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.

---

<sup>3</sup>Such soft links are necessary for at least my Mac osx and hopefully will work for other developers. Further, it has the advantage that auxiliary files are also located in the **Doc** directory.

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

```

1 % input *.m files for ... Author, date
2 %!TEX root = ../Doc/equationFreeDoc.tex
3 \section{...}
4 \label{sec:...}
5 \localtableofcontents
6 introduction...
7 \input{../Dir/fun.m}
8 \input{../Dir/funExample.m}
9 ...
10 \begin{body}
11 \subsection{To do}
12 ...
13 \subsection{Miscellaneous tests}
14 \input{../Dir/funTest.m}
15 ...
16 \end{body}
```

- 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)
```

Include with \includegraphics[scale=0.85]{filename}

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

```
1 % Short explanation for users typing "help fun"
2 % Author, date
3 %!TEX root = ../Doc/equationFreeDoc.tex
4 %{
5 \subsection{\texttt{...}: ...}
6 \label{sec:...}
7 \localtableofcontents
8 Overview LaTeX explanation.
9 \begin{matlab}
10 %}
11 function ...
12 %{
13 \end{matlab}
14 \paragraph{Input} ...
15 \paragraph{Output} ...
16 \begin{body}
17 Repeated as desired:
18 LaTeX between end-matlab and begin-matlab
19 \begin{matlab}
20 %}
21 Matlab code between \%} and \%{
22 %{
23 \end{matlab}
24 Concluding LaTeX before following final lines.
25 \end{body}
26 %}
```

## B Aspects of developing a ‘toolbox’ for patch dynamics

### *Section contents*

B.1 Macroscale grid . . . . .	72
B.2 Macroscale field variables . . . . .	72
B.3 Boundary and coupling conditions . . . . .	73
B.4 Mesotime communication . . . . .	74
B.5 Projective integration . . . . .	74
B.6 Lift to many internal modes . . . . .	75
B.7 Macroscale closure . . . . .	75
B.8 Exascale fault tolerance . . . . .	76
B.9 Link to established packages . . . . .	76

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  $\vec{U}(t) \in \mathbb{R}^{d_U}$  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 ([Bunder et al. 2016](#)). 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 ([Samaey et al. 2010](#)). 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. [Calderon \(2007\)](#) 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/momenta ([Roberts & Li 2006](#), 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 ([Gustafsson 1975, Svard & Nordstrom 2006](#))

## 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 ([Plimpton et al. 2016](#)). Second, will evaluate performance, issues, and then consider what other established packages are most promising.

## References

- Bunder, J., Roberts, A. J. & Kevrekidis, I. G. (2016), ‘Accuracy of patch dynamics with mesoscale temporal coupling for efficient massively parallel simulations’, *SIAM Journal on Scientific Computing* **38**(4), C335–C371.
- Calderon, C. P. (2007), ‘Local diffusion models for stochastic reacting systems: estimation issues in equation-free numerics’, *Molecular Simulation* **33**(9–10), 713–731.
- Gear, C. W. & Kevrekidis, I. G. (2003a), ‘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. (2003b), ‘Telescopic projective methods for parabolic differential equations’, *Journal of Computational Physics* **187**, 95–109.
- Givon, D., Kevrekidis, I. G. & Kupferman, R. (2006), ‘Strong convergence of projective integration schemes for singularly perturbed stochastic differential systems’, *Comm. Math. Sci.* **4**(4), 707–729.
- Gustafsson, B. (1975), ‘The convergence rate for difference approximations to mixed initial boundary value problems’, *Mathematics of Computation* **29**(10), 396–406.
- Hyman, J. M. (2005), ‘Patch dynamics for multiscale problems’, *Computing in Science & Engineering* **7**(3), 47–53.  
<http://scitation.aip.org/content/aip/journal/cise/7/3/10.1109/MCSE.2005.57>
- Kevrekidis, I. G., Gear, C. W. & Hummer, G. (2004), ‘Equation-free: the computer-assisted analysis of complex, multiscale systems’, *A. I. Ch. E. Journal* **50**, 1346–1354.
- Kevrekidis, I. G., Gear, C. W., Hyman, J. M., Kevrekidis, P. G., Runborg, O. & Theodoropoulos, K. (2003), ‘Equation-free, coarse-grained multiscale

- computation: enabling microscopic simulators to perform system level tasks', *Comm. Math. Sciences* **1**, 715–762.
- Kevrekidis, I. G. & Samaey, G. (2009), 'Equation-free multiscale computation: Algorithms and applications', *Annu. Rev. Phys. Chem.* **60**, 321–44.
- Liu, P., Samaey, G., Gear, C. W. & Kevrekidis, I. G. (2015), 'On the acceleration of spatially distributed agent-based computations: A patch dynamics scheme', *Applied Numerical Mathematics* **92**, 54–69.  
<http://www.sciencedirect.com/science/article/pii/S0168927414002086>
- Plimpton, S., Thompson, A., Shan, R., Moore, S., Kohlmeyer, A., Crozier, P. & Stevens, M. (2016), Large-scale atomic/molecular massively parallel simulator, Technical report, <http://lammps.sandia.gov>.
- Roberts, A. J. & Kevrekidis, I. G. (2007), 'General tooth boundary conditions for equation free modelling', *SIAM J. Scientific Computing* **29**(4), 1495–1510.
- Roberts, A. J. & Li, Z. (2006), 'An accurate and comprehensive model of thin fluid flows with inertia on curved substrates', *J. Fluid Mech.* **553**, 33–73.
- Samaey, G., Kevrekidis, I. G. & Roose, D. (2005), 'The gap-tooth scheme for homogenization problems', *Multiscale Modeling and Simulation* **4**, 278–306.
- Samaey, G., Roberts, A. J. & Kevrekidis, I. G. (2010), Equation-free computation: an overview of patch dynamics, in J. Fish, ed., 'Multiscale methods: bridging the scales in science and engineering', Oxford University Press, chapter 8, pp. 216–246.
- Samaey, G., Roose, D. & Kevrekidis, I. G. (2006), 'Patch dynamics with buffers for homogenization problems', *J. Comput Phys.* **213**, 264–287.
- Svard, M. & Nordstrom, J. (2006), 'On the order of accuracy for difference approximations of initial-boundary value problems', *Journal of Computational Physics* **218**, 333–352.