

Equation-Free function toolbox for Matlab/Octave:

Full Developers Manual

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

April 24, 2019

* 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.

Abstract

This ‘equation-free toolbox’ empowers the computer-assisted analysis of complex, multiscale systems. Its aim is to enable you to use microscopic simulators to perform system level tasks and analysis, because microscale simulations are often the best available description of a system. The methodology bypasses the derivation of macroscopic evolution equations by computing only short bursts of of the microscale simulator ([Kevrekidis & Samaey 2009](#), [Kevrekidis et al. 2004](#), [2003](#), e.g.), and often only computing on small patches of the spatial domain ([Roberts et al. 2014](#), e.g.). This suite of functions empowers users to start implementing such methods in their own applications.

Contents

1	Introduction	3
2	Projective integration of deterministic ODEs	5
2.1	Introduction	6
2.2	PIRK2(): projective integration of second-order accuracy . . .	7
2.3	egPIMM: Example projective integration of Michaelis–Menton kinetics	14
2.4	PIG(): Projective Integration via a General macroscale integrator	17
2.5	PIRK4(): projective integration of fourth-order accuracy . . .	24
2.6	cdmc(): constraint defined manifold computing	30
2.7	Example: PI using Runge–Kutta macrosolvers	31
2.8	Example: Projective Integration using General macrosolvers	34
2.9	Explore: Projective Integration using constraint-defined manifold computing	35
2.10	To do/discuss	37
3	Patch scheme for given microscale discrete space system	38
3.1	Introduction	39
3.2	configPatches1(): configures spatial patches in 1D	39
3.3	patchSmooth1(): interface to time integrators	45
3.4	patchEdgeInt1(): sets edge values from interpolation over the macroscale	46
3.5	homogenisationExample: simulate heterogeneous diffusion in 1D	50
3.6	BurgersExample: simulate Burgers’ PDE on patches	54
3.7	ensembleAverageExample: simulate an ensemble of solutions for heterogeneous diffusion in 1D	58
3.8	waterWaveExample: simulate a water wave PDE on patches .	63
3.9	configPatches2(): configures spatial patches in 2D	68
3.10	patchSmooth2(): interface to time integrators	74

3.11	<code>patchEdgeInt2()</code> : 2D patch edge values from 2D interpolation	75
3.12	<code>wave2D</code> : example of a wave on patches in 2D	79
3.13	To do	83
3.14	Miscellaneous tests	83
A	Create, document and test algorithms	87
B	Aspects of developing a ‘toolbox’ for patch dynamics	90
B.1	Macroscale grid	90
B.2	Macroscale field variables	90
B.3	Boundary and coupling conditions	91
B.4	Mesotime communication	91
B.5	Projective integration	91
B.6	Lift to many internal modes	92
B.7	Macroscale closure	92
B.8	Exascale fault tolerance	92
B.9	Link to established packages	93

1 Introduction

This Developers Manual contains complete descriptions of the code in each function in the toolbox, and of each example. For concise descriptions of each function, quick start guides, and some basic examples, see the User Manual.

Users Place the folder of this toolbox in a path searched by MATLAB/Octave. Then read the section(s) that documents the function of interest.

Quick start Maybe start by adapting one of the included examples. Many of the main functions include, at their beginning, example code of their use—code which is executed when the function is invoked without any arguments.

- To projectively integrate over time a multiscale, slow-fast, system of ODEs you could use `PIRK2()`, or `PIRK4()` for higher-order accuracy: adapt the Michaelis–Menten example at the beginning of `PIRK2.m` ([Section 2.2.2](#)).
- You may use forward bursts of simulation in order to simulate the slow dynamics backward in time, as in `egPIMM.m` ([Section 2.3](#)).
- To only resolve the slow dynamics in the projective integration, use lifting and restriction functions by adapting the singular perturbation ODE example at the beginning of `PIG.m` ([Section 2.4.2](#)).

Space-time systems Consider an evolving system over a large spatial domains when all you have is a microscale code. To efficiently simulate over the large domain, one can simulate in just small patches of the domain, appropriately coupled.

- In 1D adapt the code at the beginning of `configPatches1.m` for Burgers' PDE ([Section 3.2.2](#)), or the staggered patches of 1D water wave equations in `waterWaveExample.m` ([Section 3.8](#)).
- in 2D adapt the code at the beginning of `configPatches2.m` for non-linear diffusion ([Section 3.9.2](#)), or the regular patches of the 2D wave equation of `wave2D.m` ([Section 3.12](#)).
- The above two are for systems that have *smooth* spatial structures on the microscale: when the microscale is 'rough' with a known period (so far only in 1D), then adapt the example of `HomogenisationExample.m` ([Section 3.5](#)).

Blackbox scenarios Suppose that you have a *detailed and trustworthy* computational simulation of some problem of interest. Let's say the simulation is coded in terms of detailed (microscale) variable values $\vec{u}(t)$, in \mathbb{R}^p for some p , and evolving time t . The details \vec{u} could represent particles, agents, or states of a system. When the computation is too time consuming to simulate all the times of interest, then Projective Integration may be able to predict long-time dynamics, both forward and backward in time. In this case, provide your detailed computational simulation as a 'black box' to the Projective Integration functions of [Chapter 2](#).

In many scenarios, the problem of interest involves space or a 'spatial' lattice. Let's say that indices i correspond to 'spatial' coordinates $\vec{x}_i(t)$, which are often fixed: in lattice problems the positions \vec{x}_i would be fixed in time (unless employing a moving mesh on the microscale); however, in particle problems the positions would evolve. And suppose your detailed and trustworthy simulation is coded also in terms of micro-field variable values $\vec{u}_i(t) \in \mathbb{R}^p$ at time t . Often the detailed computational simulation is too expensive over all the desired spatial domain $\vec{x} \in \mathbb{X} \subset \mathbb{R}^d$. In this case, the toolbox functions of [Chapter 3](#) empower you to simulate on only small, well-separated, patches of space by appropriately coupling between patches your simulation code, as a 'black box', executing on each small patch. The computational savings may be enormous, especially if combined with projective integration.

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

MATLAB appears a good choice for a first version since it is widespread, efficient, supports various parallel modes, and development costs are reasonably low. Further it is built on BLAS and LAPACK so the cache and superscalar CPU are potentially well utilised. We aim to develop functions that work for MATLAB/Octave. [Appendix A](#) outlines some details for contributors.

2 Projective integration of deterministic ODEs

Chapter contents

2.1	Introduction	6
2.2	PIRK2(): projective integration of second-order accuracy . . .	7
2.2.1	Introduction	8
2.2.2	If no arguments, then execute an example	10
2.2.3	The projective integration code	11
2.2.4	If no output specified, then plot the simulation	14
2.3	egPIMM: Example projective integration of Michaelis–Menton kinetics	14
2.4	PIG(): Projective Integration via a General macroscale integrator	17
2.4.1	Introduction	18
2.4.2	If no arguments, then execute an example	20
2.4.3	The projective integration code	21
2.4.4	If no output specified, then plot the simulation	24
2.5	PIRK4(): projective integration of fourth-order accuracy . . .	24
2.5.1	Introduction	24
2.5.2	The projective integration code	27
2.5.3	If no output specified, then plot the simulation	30
2.6	cdmc(): constraint defined manifold computing	30
2.7	Example: PI using Runge–Kutta macrosolvers	31
2.8	Example: Projective Integration using General macrosolvers	34
2.9	Explore: Projective Integration using constraint-defined manifold computing	35
2.10	To do/discuss	37

2.1 Introduction

This section provides some good projective integration functions (Gear & Kevrekidis 2003*b,c*, Givon et al. 2006, Maclean & Gottwald 2015, Sieber et al. 2018, e.g.). The goal is to enable computationally expensive multiscale dynamic simulations/integrations to efficiently compute over very long time scales.

Quick start Section 2.2.2 shows the most basic use of a projective integration function. Section 2.3 shows how to code more variations of the introductory example of a long time simulation of the Michaelis–Menton multiscale system of differential equations. Then see Figures 2.1 and 2.2

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, microscale simulation of the complex system, and call such code a *microsolver*.

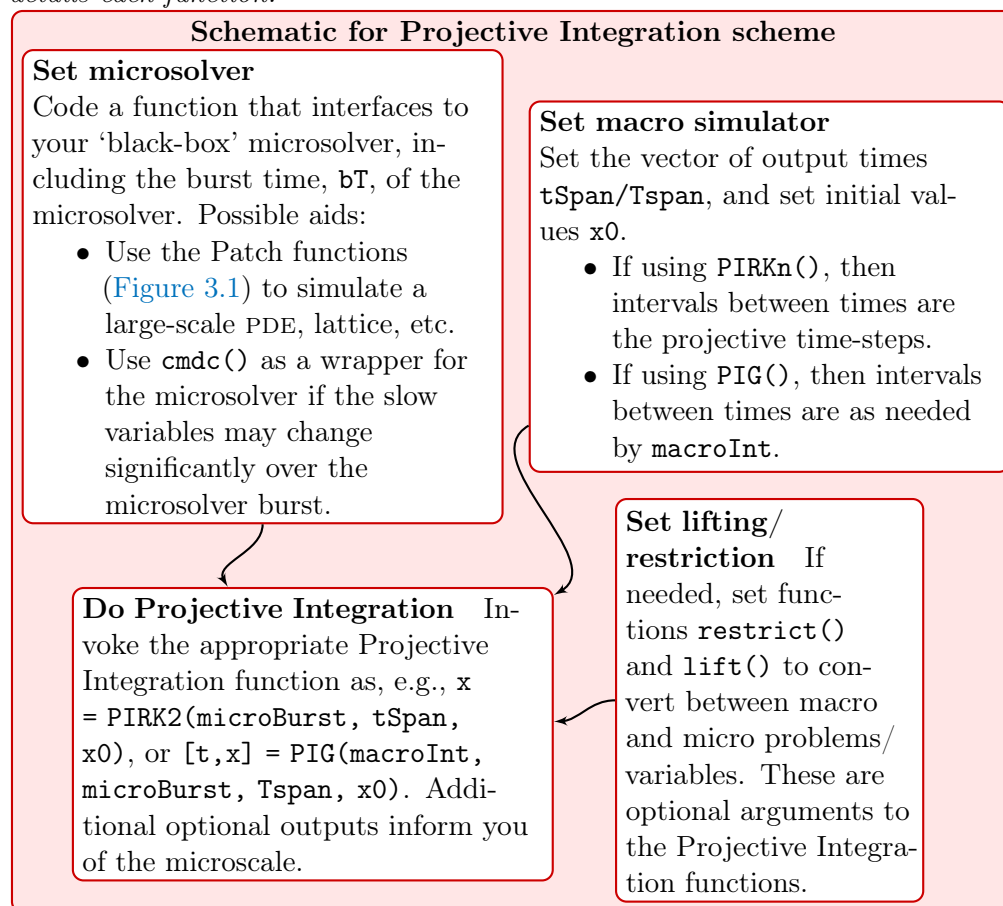
The Projective Integration section of this toolbox consists of several functions. Each function implements over a long-time scale a variant of a standard numerical method to simulate/integrate the emergent dynamics of the complex system. Each function has standardised inputs and outputs.

Main functions

- Projective Integration by second or fourth-order Runge–Kutta is implemented by `PIRK2()` or `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 method, `PIG()`. This function enables a Projective Integration implementation of any integration method over macroscale time-steps. It does not matter whether the method is a standard MATLAB/Octave algorithm, or one supplied by the user. `PIG()` should only be used directly in very stiff systems, less stiff systems additionally require `cdmc()`.
- *Constraint-defined manifold computing*, `cdmc()`, is a helper function, based on the method introduced in Gear et al. (2005*a*), that iteratively applies the *microsolver* and backward projection in time. The result is to project the fast variables close to the slow manifold, without advancing the current time by the burst time of the *microsolver*. This function reduces errors related to the simulation length of the *microsolver* in the `PIG` function. In particular, it enables `PIG()` to be used on problems that are not particularly stiff.

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

Figure 2.1: The Projective Integration method greatly accelerates simulation/integration of a system exhibiting multiple time scales. The Projective Integration [Chapter 2](#) presents several separate functions, as well as several optional wrapper functions that may be invoked. This chart overviews constructing a Projective Integration simulation, whereas [Figure 2.2](#) roughly guides which top-level Projective Integration functions should be used. [Chapter 2](#) fully details each function.



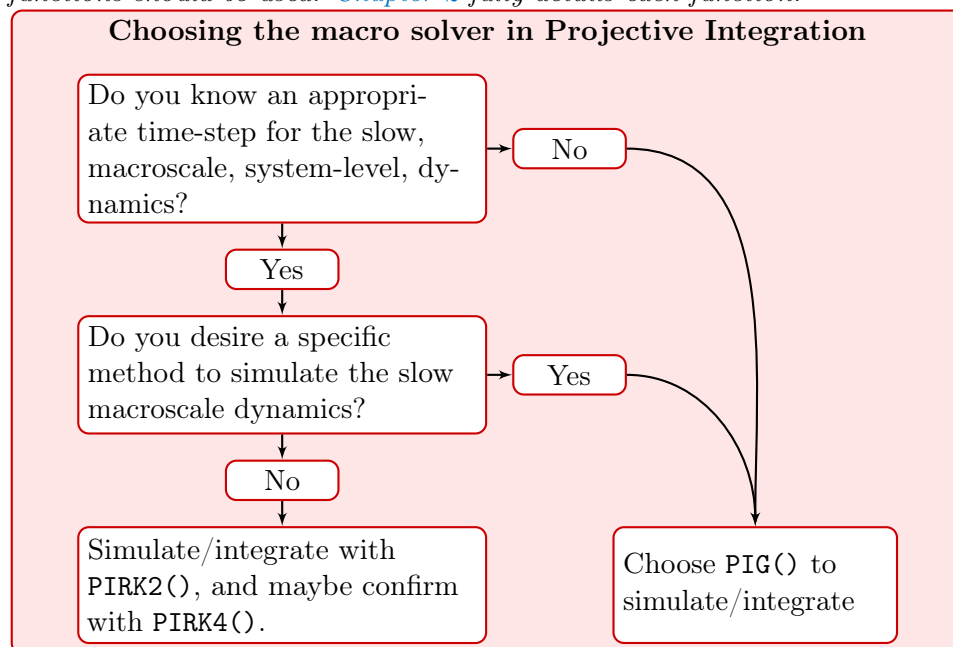
The following sections describe the `PIRK2()` and `PIG()` functions in detail, providing an example for each. The function `PIRK4()` is very similar to `PIRK2()`. Descriptions for the minor functions follow, and an example using `cdmc()`.

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

Section contents

2.2.1	Introduction	8
2.2.2	If no arguments, then execute an example	10
2.2.3	The projective integration code	11
2.2.4	If no output specified, then plot the simulation	14

Figure 2.2: The Projective Integration method greatly accelerates simulation/integration of a system exhibiting multiple time scales. In conjunction with Figure 2.1, this chart roughly guides which top-level Projective Integration functions should be used. Chapter 2 fully details each function.



2.2.1 Introduction

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

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

Input If there are no input arguments, then this function applies itself to the Michaelis–Menton example: see the code in Section 2.2.2 as a basic template of how to use.

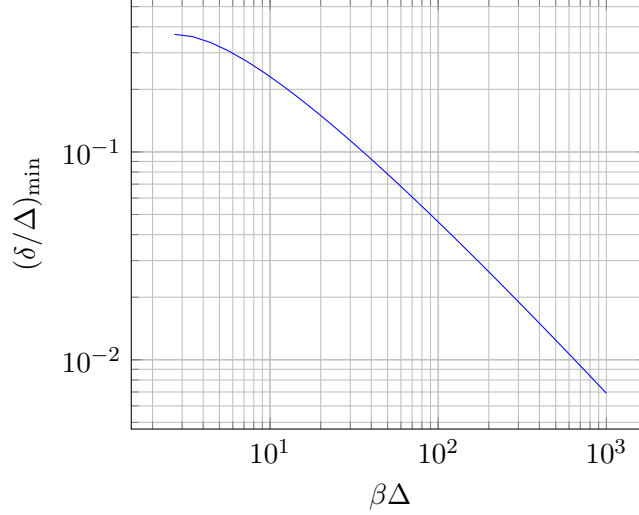
- `microBurst()`, a user-coded function that computes a short-time burst of the microscale simulation.

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

- Inputs: `tStart`, the start time of a burst of simulation; `xStart`, the row n -vector of the starting state; `bT`, *optional*, the total time to simulate in the burst—if your `microBurst()` determines the burst time, then replace `bT` in the argument list by `varargin`.
- Outputs: `tOut`, the column vector of solution times; and `xOut`, an array in which each *row* contains the system state at corresponding times.

- `tSpan` is an ℓ -vector of times at which the user requests output, of which the first element is always the initial time. `PIRK2()` does not

Figure 2.3: Need macroscale step Δ such that $|\alpha\Delta| \lesssim \sqrt{6\varepsilon}$ for given relative error ε and slow rate α , and then $\delta/\Delta \gtrsim \frac{1}{\beta\Delta} \log \beta\Delta$ determines the minimum required burst length δ for every given fast rate β .



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`: such `Nans` are carried in the simulation through to the output, and often represent boundaries/edges in spatial fields.
- `bT`, *optional*, either missing, or empty (`[]`), or a scalar: if a given scalar, then it is the length of the micro-burst simulations—the minimum amount of time needed for the microscale simulation to relax to the slow manifold; else if missing or `[]`, then `microBurst()` must itself determine the length of a burst.

```
70 if nargin<4, bT=[]; end
```

Choose a long enough burst length Suppose: firstly, you have some desired relative accuracy ε that you wish to achieve (e.g., $\varepsilon \approx 0.01$ for two digit accuracy); secondly, the slow dynamics of your system occurs at rate/frequency of magnitude about α ; and thirdly, the rate of *decay* of your fast modes are faster than the lower bound β (e.g., if three fast modes decay roughly like e^{-12t} , e^{-34t} , e^{-56t} then $\beta \approx 12$). Then set

1. a macroscale time-step, $\Delta = \text{diff}(\text{tSpan})$, such that $\alpha\Delta \approx \sqrt{6\varepsilon}$, and
2. a microscale burst length, $\delta = \text{bT} \gtrsim \frac{1}{\beta} \log(\beta\Delta)$, see Figure 2.3.

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(microBurst,tSpan,x0,bT)`.

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

- `tms`, optional, is an L dimensional column vector containing the microscale times within the burst simulations, each burst separated by `NaN`;
- `xms`, optional, is an $L \times n$ array of the corresponding microscale states—each rows is an accurate estimate of the state at the corresponding time `tms` and helps visualise details of the solution.
- `rm`, optional, a struct containing the ‘remaining’ applications of the `microBurst` 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* accurately approximate the macroscale dynamics.

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

2.2.2 If no arguments, then execute an example

```
175 if nargin==0
```

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

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

(encoded in function `MMburst()` in the next paragraph). With initial conditions $x(0) = 1$ and $y(0) = 0$, the following code computes and plots a solution over time $0 \leq t \leq 6$ for parameter $\epsilon = 0.05$. Since the rate of decay is $\beta \approx 1/\epsilon$ we choose a burst length $\epsilon \log(\Delta/\epsilon)$ as here the macroscale time-step $\Delta = 1$.

```
196 global MMepsilon
197 MMepsilon = 0.05
198 ts = 0:6
199 bT = MMepsilon*log((ts(2)-ts(1))/MMepsilon)
200 [x,tms,xms] = PIRK2(@MMburst, ts, [1;0], bT);
201 figure, plot(ts,x,'o:',tms,xms)
```

```

202 title('Projective integration of Michaelis--Menten enzyme kinetics')
203 xlabel('time t'), legend('x(t)','y(t)')

```

Upon finishing execution of the example, exit this function.

```

209 return
210 end%if no arguments

```

Code a burst of Michaelis–Menten enzyme kinetics Integrate a burst of length `bT` of the ODEs for the Michaelis–Menten enzyme kinetics at parameter ϵ inherited from above. Code ODEs in function `dMMdt` with variables $x = x(1)$ and $y = x(2)$. Starting at time `ti`, and state `xi` (row), we here simply use MATLAB/Octave’s `ode23/lsode` to integrate a burst in time.

```

15 function [ts, xs] = MMburst(ti, xi, bT)
16     global MMepsilon
17     dMMdt = @(t,x) [ -x(1)+(x(1)+0.5)*x(2)
18                     1/MMepsilon*( x(1)-(x(1)+1)*x(2) ) ];
19     if ~exist('OCTAVE_VERSION','builtin')
20         [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
21     else % octave version
22         [ts, xs] = odeOct(dMMdt, [ti ti+bT], xi);
23     end
24 end

8 function [ts,xs] = odeOct(dxdt,tSpan,x0)
9     if length(tSpan)>2, ts = tSpan;
10    else ts = linspace(tSpan(1),tSpan(end),21);
11    end
12    % mimic ode45 and ode23, but much slower for non-PI
13    lsode_options('integration method','non-stiff');
14    xs = lsode(@(x,t) dxdt(t,x),x0,ts);
15 end

```

2.2.3 The projective integration code

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

```

229 nT=length(tSpan);
230 x=nan(nT,length(x0));

```

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

```

238 nArgOut=nargout();
239 saveMicro = (nArgOut>1);
240 saveFullMicro = (nArgOut>3);
241 saveSvf = (nArgOut>4);

```

Run a preliminary application of the `microBurst` on the given initial state to help relax to the slow manifold. This is done in addition to the `microBurst` in

the main loop, because the initial state is often far from the attracting slow manifold. Require the user to input and output rows of the system state.

```
254 x0 = reshape(x0,1,[]);
255 [relax_t,relax_x0] = microBurst(tSpan(1),x0,bT);
```

Use the end point of this preliminary microBurst as the initial state for the loop of macro-steps.

```
263 tSpan(1) = relax_t(end);
264 x(1,:)=relax_x0(end,:);
```

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

```
274 if saveMicro
275     tms = cell(nT,1);
276     xms = cell(nT,1);
277     tms{1} = reshape(relax_t,[],1);
278     xms{1} = relax_x0;
279     if saveFullMicro
280         rm.t = cell(nT,1);
281         rm.x = cell(nT,1);
282         if saveSvf
283             svf.t = nan(2*nT-2,1);
284             svf.dx = nan(2*nT-2,length(x0));
285         end
286     end
287 end
```

Loop over the macroscale time-steps

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

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

```
304     if ~isempty(bT) && 2*abs(bT)>=abs(tSpan(jT)-T) && bT*(tSpan(jT)-T)>0
305         [t1,xm1] = microBurst(T, x(jT-1,:), tSpan(jT)-T);
306         x(jT,:) = xm1(end,:);
307         t2=nan; xm2=nan(1,size(xm1,2));
308         dx1=xm2; dx2=xm2;
309     else
```

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

```

320     [t1,xm1] = microBurst(T, x(jT-1,:), bT);
321     del = t1(end)-t1(end-1);

```

Check for round-off error.

```

327     xt=[reshape(t1(end-1:end),[],1) xm1(end-1:end,:)];
328     roundingTol=1e-8;
329     if norm(diff(xt))/norm(xt,'fro') < roundingTol
330         warning(['significant round-off error in 1st projection at T=' num2str(T)
331         end

```

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

```

340     Dt = tSpan(jT)-t1(end);
341     dx1 = (xm1(end,:)-xm1(end-1,:))/del;

```

Project along `dx1` to form an intermediate approximation of `x`; run another application of the `microBurst` and form a second estimate of the slow vector field (assuming the burst length is the same, or nearly so).

```

351     xint = xm1(end,:) + (Dt-(t1(end)-t1(1)))*dx1;
352     [t2,xm2] = microBurst(T+Dt, xint, bT);
353     del = t2(end)-t2(end-1);
354     dx2 = (xm2(end,:)-xm2(end-1,:))/del;

```

Check for round-off error.

```

360     xt=[reshape(t2(end-1:end),[],1) xm2(end-1:end,:)];
361     if norm(diff(xt))/norm(xt,'fro') < roundingTol
362         warning(['significant round-off error in 2nd projection at T=' num2str(T)
363         end

```

Use the weighted average of the estimates of the slow vector field to take a macro-step.

```

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

```

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

```

379     end

```

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

```

389     if saveMicro
390         tms{jT} = [reshape(t1,[],1); nan];
391         xms{jT} = [xm1; nan(1,size(xm1,2))];

```

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

```

399         if saveFullMicro
400             rm.t{jT} = [reshape(t2,[],1); nan];
401             rm.x{jT} = [xm2; nan(1,size(xm2,2))];

```

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

```

410         if saveSvf
411             svf.t(2*jT-3:2*jT-2) = [t1(end); t2(end)];
412             svf.dx(2*jT-3:2*jT-2,:) = [dx1; dx2];
413         end
414     end
415 end

```

End the main loop over all the macro-steps.

```

421 end

```

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

```

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

```

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

```

438 if saveMicro
439     tms = cell2mat(tms);
440     xms = cell2mat(xms);
441     if saveFullMicro
442         rm.t = cell2mat(rm.t);
443         rm.x = cell2mat(rm.x);
444     end
445 end

```

2.2.4 If no output specified, then plot the simulation

```

453 if nArgOut==0
454     figure, plot(tSpan,x,'o:')
455     title('Projective Simulation with PIRK2')
456 end

```

This concludes `PIRK2()`.

```

463 end

```

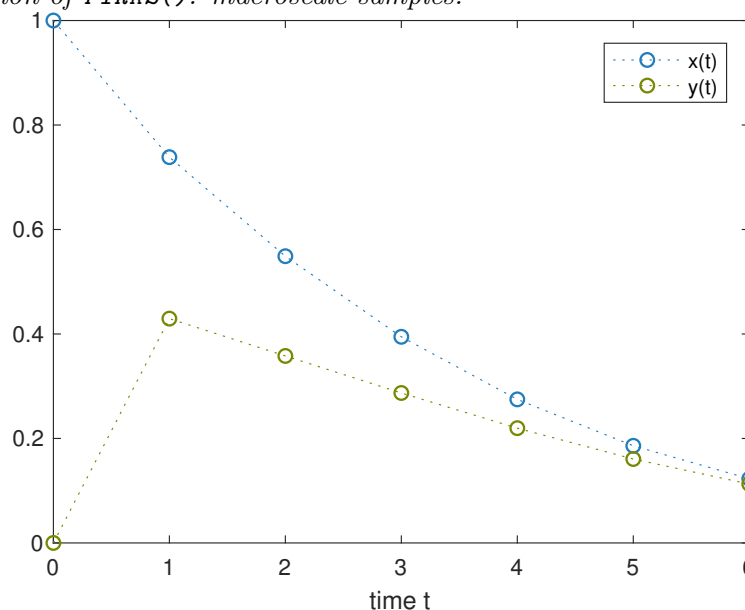
2.3 egPIMM: Example projective integration of Michaelis–Menton kinetics

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

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

(encoded in function `MMburst()` below). As illustrated by [Figure 2.5](#), 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 ϵ .

Figure 2.4: Michaelis–Menten enzyme kinetics simulated with the projective integration of `PIRK2()`: macroscale samples.



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

```

31 clear all, close all
32 global MMepsilon
33 MMepsilon = 0.1

```

First, the end of this section 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 2ϵ , and starting from the initial condition for the Michaelis–Menten system, at time $t = 0$, of $(x, y) = (1, 0)$ (off the slow manifold).

```

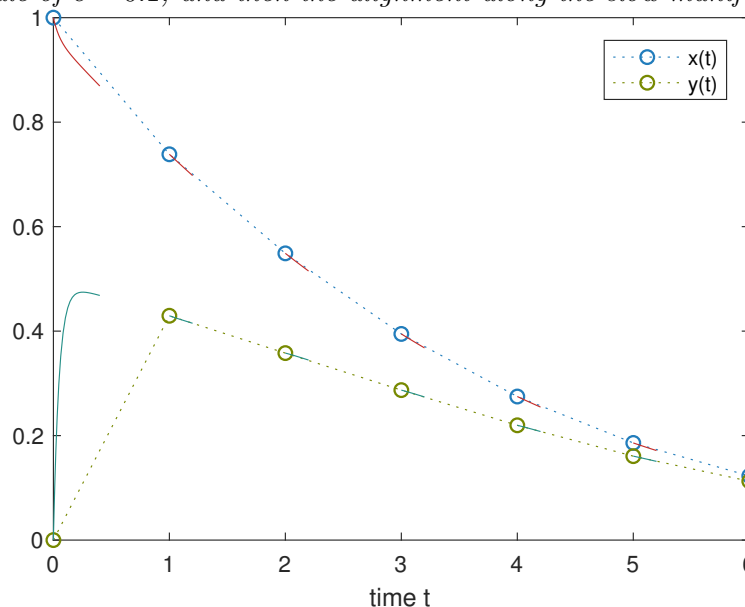
48 ts = 0:6
49 xs = PIRK2(@MMburst, ts, [1;0], 2*MMepsilon)
50 plot(ts,xs,'o:')
51 xlabel('time t'), legend('x(t)','y(t)')
52 pause(1)

```

Figure 2.4 plots the macroscale results showing the long time decay of the Michaelis–Menten system on the slow manifold. Sieber et al. (2018) [§4] used this system as an example of their analysis of the convergence of Projective Integration.

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 2.4). In order to see the initial transient attraction to the slow manifold we plot some microscale data in Figure 2.5. Two further output

Figure 2.5: 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.



variables provide this microscale burst information.

```

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

```

Figure 2.5 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 (hence other schemes which ‘freeze’ slow variables are less accurate).

Simulate backward in time Figure 2.6 shows that projective integration even simulates backward in time along the slow manifold using short forward bursts (Gear & Kevrekidis 2003a). Such backward macroscale simulations succeed despite the fast variable $y(t)$, when backward in time, being viciously unstable. However, backward integration appears to need longer bursts, here 3ϵ .

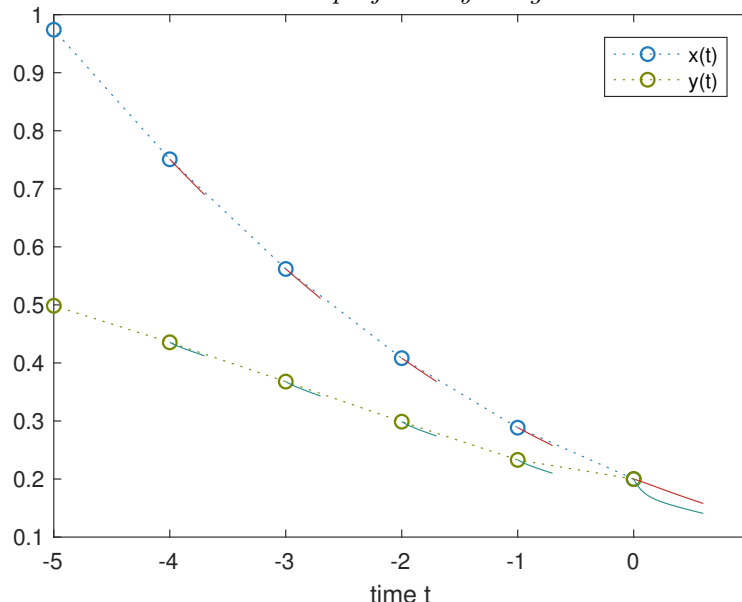
```

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

```

Code a burst of Michaelis–Menten enzyme kinetics Integrate a burst of length `bT` of the ODEs for the Michaelis–Menten enzyme kinetics at parameter ϵ inherited from above. Code ODEs in function `dMMdt` with variables

Figure 2.6: Michaelis–Menten enzyme kinetics at $\epsilon = 0.1$ simulated backward with the projective integration of `PIRK2()`: the microscale bursts show the short forward simulations used to projectively integrate backward in time.



$x = x(1)$ and $y = x(2)$. Starting at time t_i , and state x_i (row), we here simply use MATLAB/Octave's `ode23/lsode` to integrate a burst in time.

```

15 function [ts, xs] = MMburst(ti, xi, bT)
16     global MMepsilon
17     dMMdt = @(t,x) [ -x(1)+(x(1)+0.5)*x(2)
18                     1/MMepsilon*( x(1)-(x(1)+1)*x(2) ) ];
19     if ~exist('OCTAVE_VERSION','builtin')
20         [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
21     else % octave version
22         [ts, xs] = odeOct(dMMdt, [ti ti+bT], xi);
23     end
24 end

8 function [ts,xs] = odeOct(dxdt,tSpan,x0)
9     if length(tSpan)>2, ts = tSpan;
10    else ts = linspace(tSpan(1),tSpan(end),21);
11    end
12    % mimic ode45 and ode23, but much slower for non-PI
13    lsode_options('integration method','non-stiff');
14    xs = lsode(@(x,t) dxdt(t,x),x0,ts);
15 end

```

2.4 PIG(): Projective Integration via a General macroscale integrator

Section contents

2.4.1 Introduction 18

2.4.2	If no arguments, then execute an example	20
2.4.3	The projective integration code	21
2.4.4	If no output specified, then plot the simulation	24

2.4.1 Introduction

This is a Projective Integration scheme when the macroscale integrator is any specified coded method. The advantage is that one may use MATLAB/Octave's inbuilt integration functions, with all their sophisticated error control and adaptive time-stepping, to do the macroscale integration/simulation.

By default, for the microscale simulations `PIG()` uses 'constraint-defined manifold computing', `cdmc()` (Section 2.6). This algorithm, initiated by Gear et al. (2005b), uses a backward projection so that the simulation time is unchanged after running the microscale simulator.

```

29 function [T,X,tms,xms,svf] = PIG(macroInt,microBurst,Tspan,x0 ...
30                                ,restrict,lift,cdmcFlag)

```

Inputs:

- `macroInt()`, the numerical method that the user wants to apply on a slow-time macroscale. Either specify a standard MATLAB/Octave integration function (such as 'ode23' or 'ode45'), or code your own integration function using standard arguments. That is, if you code your own, then it must be

$$[Ts,Xs] = \text{macroInt}(F,Tspan,X0)$$

where

- function $F(T,X)$ notionally evaluates the time derivatives $d\vec{X}/dt$ at any time;
- `Tspan` is either the macro-time interval, or the vector of macroscale times at which macroscale values are to be returned; and
- `X0` are the initial values of \vec{X} at time `Tspan(1)`.

Then the i th row of `Xs`, `Xs(i,:)`, is to be the vector $\vec{X}(t)$ at time $t = Ts(i)$. Remember that in `PIG()` the function $F(T,X)$ is to be estimated by Projective Integration.

- `microBurst()` is a function that produces output from the user-specified code for a burst of microscale simulation. The function must internally specify/decide how long a burst it is to use. Usage

$$[tbs,xbs] = \text{microBurst}(tb0,xb0)$$

Inputs: `tb0` is the start time of a burst; `xb0` is the n -vector microscale state at the start of a burst.

Outputs: `tbs`, the vector of solution times; and `xbs`, the corresponding microscale states.

- **Tspan**, a vector of macroscale times at which the user requests output. The first element is always the initial time. If **macroInt** reports adaptively selected time steps (e.g., **ode45**), then **Tspan** consists of an initial and final time only.
- **x0**, the n -vector of initial microscale values at the initial time **Tspan**(1).

Optional Inputs: **PIG()** allows for none, two or three additional inputs after **x0**. If you distinguish distinct microscale and macroscale states and your aim is to do Projective Integration on the macroscale only, then lifting and restriction functions must be provided to convert between them. Usage **PIG(...,restrict,lift)**:

- **restrict(x)**, a function that takes an input high-dimensional, n -D, microscale state \vec{x} and computes the corresponding low-dimensional, N -D, macroscale state \vec{X} ;
- **lift(X,xApprox)**, a function that converts an input low-dimensional, N -D, macroscale state \vec{X} to a corresponding high-dimensional, n -D, microscale state \vec{x} , given that **xApprox** is a recently computed microscale state on the slow manifold.

Either both **restrict()** and **lift()** are to be defined, or neither. If neither are defined, then they are assumed to be identity functions, so that $N=n$ in the following.

If desired, the default constraint-defined manifold computing microsolver may be disabled, via **PIG(...,restrict,lift,cdmcFlag)**

- **cdmcFlag**, *any* seventh input to **PIG()**, will disable **cdmc()**, e.g., the string '**cdmc off**'.

If the **cdmcFlag** is to be set without using a **restrict()** or **lift()** function, then use empty matrices **[]** for the restrict and lift functions.

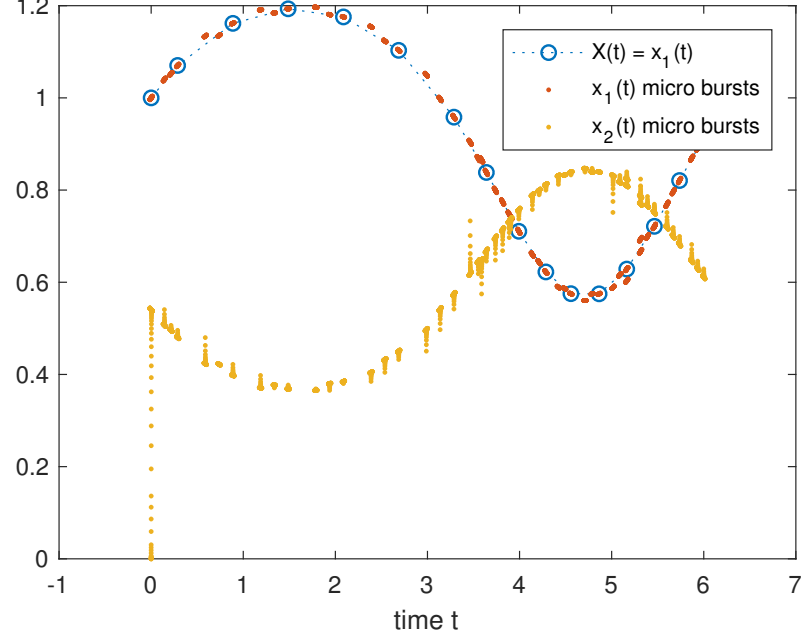
Output Between zero and five outputs may be requested. If there are no output arguments specified, then a plot is drawn of the computed solution **X** versus **T**. Most often you would store the first two output results of **PIG()**, via say **[T,X] = PIG(...)**.

- **T**, an L -vector of times at which **macroInt** produced results.
- **X**, an $L \times N$ array of the computed solution: the i th row of **X**, **X(i,:)**, is to be the macro-state vector $\vec{X}(t)$ at time $t = T(i)$.

However, microscale details of the underlying Projective Integration computations may be helpful, and so **PIG()** provides some optional outputs of the microscale bursts, via **[T,X,tms,xms] = PIG(...)**

- **tms**, optional, is an ℓ -dimensional column vector containing microscale times with bursts, each burst separated by **NaN**;
- **xms**, optional, is an $\ell \times n$ array of the corresponding microscale states.

Figure 2.7: Projective Integration by *PIG* of the example system (2.1) with $\epsilon = 10^{-3}$ (Section 2.4.2). The macroscale solution $X(t)$ is represented by just the blue circles. The microscale bursts are the microscale states $(x_1(t), x_2(t)) = (\text{red, yellow})$ dots.



In some contexts it may be helpful to see directly how Projective Integration approximates a reduced slow vector field, via `[T,X,tms,xms,svf] = PIG(...)` in which

- `svf`, optional, a struct containing the Projective Integration estimates of the slow vector field.
 - `svf.T` is a \hat{L} -dimensional column vector containing all times at which the microscale simulation data is extrapolated to form an estimate of $d\vec{x}/dt$ in `macroInt()`.
 - `svf.dX` is a $\hat{L} \times N$ array containing the estimated slow vector field.

If `macroInt()` is, for example, the forward Euler method (or the Runge–Kutta method), then $\hat{L} = L$ (or $\hat{L} = 4L$).

2.4.2 If no arguments, then execute an example

178 `if nargin==0`

As a basic example, consider a microscale system of the singularly perturbed system of differential equations

$$\frac{dx_1}{dt} = \cos(x_1) \sin(x_2) \cos(t) \quad \text{and} \quad \frac{dx_2}{dt} = \frac{1}{\epsilon} [\cos(x_1) - x_2]. \quad (2.1)$$

The macroscale variable is $X(t) = x_1(t)$, and the evolution dX/dt is unclear. With initial condition $X(0) = 1$, the following code computes and plots a solution of the system (2.1) over time $0 \leq t \leq 6$ for parameter

$\epsilon = 10^{-3}$ (Figure 2.7). Whenever needed by `microBurst()`, the microscale system (2.1) is initialised ('lifted') using $x_2(t) = x_2^{\text{approx}}$ (yellow dots in Figure 2.7).

First we code the right-hand side function of the microscale system (2.1) of ODEs.

```
211 epsilon = 1e-3;
212 dxdt=@(t,x) [ cos(x(1))*sin(x(2))*cos(t)
213               ( cos(x(1))-x(2) )/epsilon ];
```

Second, we code microscale bursts, here using the standard `ode45()`. We choose a burst length $2\epsilon \log(1/\epsilon)$ as the rate of decay is $\beta \approx 1/\epsilon$ and we do not know the macroscale time-step invoked by `macroInt()`, so blithely assume $\Delta \leq 1$ and then double the usual formula for safety.

```
225 bT = 2*epsilon*log(1/epsilon)
226 if ~exist('OCTAVE_VERSION','builtin')
227     micB='ode45'; else micB='rk2Int'; end
228 microBurst = @(tb0, xb0) feval(micB,dxdt,[tb0 tb0+bT],xb0);
```

Third, code functions to convert between macroscale and microscale states.

```
235 restrict = @(x) x(1);
236 lift = @(X,xApprox) [X; xApprox(2)];
```

Fourth, invoke PIG to use MATLAB/Octave's `ode23/lode`, say, on the macroscale slow evolution. Integrate the micro-bursts over $0 \leq t \leq 6$ from initial condition $\vec{x}(0) = (1, 0)$. You could set `Tspan=[0 -6]` to integrate backward in macroscale time with forward microscale bursts (Gear & Kevrekidis 2003a).

```
247 Tspan = [0 6];
248 x0 = [1;0];
249 if ~exist('OCTAVE_VERSION','builtin')
250     macInt='ode23'; else macInt='odeOct'; end
251 [Ts,Xs,tms,xms] = PIG(macInt,microBurst,Tspan,x0,restrict,lift);
```

Plot output of this projective integration.

```
257 figure, plot(Ts,Xs,'o:',tms,xms,'.')
258 title('Projective integration of singularly perturbed ODE')
259 xlabel('time t')
260 legend('X(t) = x_1(t)', 'x_1(t) micro bursts', 'x_2(t) micro bursts')
```

Upon finishing execution of the example, exit this function.

```
266 return
267 end%if no arguments
```

2.4.3 The projective integration code

If no lifting/restriction functions are provided, then assign them to be the identity functions.

```

284 if nargin < 5 || isempty(restrict)
285     lift=@(X,xApprox) X;
286     restrict=@(x) x;
287 end

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

295 nArgOut = nargout();
296 saveMicro = (nArgOut>2);
297 saveSvf = (nArgOut>4);

    Find the number of time-steps at which output is expected, and the number
    of variables.

305 nT = length(Tspan)-1;
306 nx = length(x0);
307 nX = length(restrict(x0));

    Reformulate the microsolver to use cdmc(), unless flagged otherwise. The
    result is that the solution from microBurst will terminate at the given initial
    time.

317 if nargin<7
318     microBurst = @(t,x) cdmc(microBurst,t,x);
319 else
320     warning(['A ' class(cdmcFlag) ' seventh input to PIG().'\...
321         ' PIG will not use constraint-defined manifold computing.'])
322 end

    Execute a preliminary application of the microBurst on the initial state. This
    is done in addition to the microBurst in the main loop, because the initial
    state is often far from the attracting slow manifold.

334 [relaxT,x0MicroRelax] = microBurst(Tspan(1),x0);
335 xMicroLast = x0MicroRelax(end,:).';
336 X0Relax = restrict(xMicroLast);

    Update the initial time.

343 Tspan(1) = relaxT(end);

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

355 if saveMicro
356     tms=cell(nT+1,1); xms=cell(nT+1,1);
357     n=1;
358     tms{n} = reshape(relaxT,[],1);
359     xms{n} = x0MicroRelax;
360
361     if saveSvf

```



```

362         svf.T = cell(nT+1,1);
363         svf.dX = cell(nT+1,1);
364     else
365         svf = [];
366     end
367 else
368     tms = []; xms = []; svf = [];
369 end

```

Define a function of macro simulation The idea of PIG() is to use the output from the `microBurst()` to approximate an unknown function $F(t, X)$ that computes $d\vec{X}/dt$. This approximation is then used in the system/user-defined ‘coarse solver’ `macroInt()`. The approximation is computed in the function

```

382 function [dXdt]=PIFun(t,X)

```

Run a `microBurst` from the given macroscale initial values.

```

388     x = lift(X,xMicroLast);
389     [tTmp,xMicroTmp] = microBurst(t,reshape(x,[],1));
390     xMicroLast = xMicroTmp(end,:).';

```

Compute the standard Projective Integration approximation of the slow vector field.

```

397     X2 = restrict(xMicroTmp(end,:));
398     X1 = restrict(xMicroTmp(end-1,:));
399     dt = tTmp(end)-tTmp(end-1);
400     dXdt = (X2 - X1).'/dt;

```

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

```

407     if saveMicro
408         n=n+1;
409         tms{n} = [reshape(tTmp,[],1); nan];
410         xms{n} = [xMicroTmp; nan(1,nx)];
411         if saveSvf
412             svf.T{n-1} = t;
413             svf.dX{n-1} = dXdt;
414         end
415     end
416 end% PIFun function

```

Invoke the macroscale integration Integrate PIF() with the user-specified simulator `macroInt()`. For some reason, in MATLAB/Octave we need to use a one-line function, PIF, that invokes the above macroscale function, PIFun. We also need to use `feval` because `macroInt()` has multiple outputs.

```

429 PIF = @(t,x) PIFun(t,x);
430 [T,X] = feval(macroInt,PIF,Tspan,X0Relax. ');

```

Overwrite $X(1,:)$ and $T(1)$, which a user expects to be $X0$ and $Tspan(1)$ respectively, with the given initial conditions.

```
439 X(1,:) = restrict(x0);
440 T(1) = Tspan(1);
```

Concatenate all the additional requested outputs into arrays.

```
447 if saveMicro
448     tms = cell2mat(tms);
449     xms = cell2mat(xms);
450     if saveSvf
451         svf.T = cell2mat(svf.T);
452         svf.dX = cell2mat(svf.dX);
453     end
454 end
```

2.4.4 If no output specified, then plot the simulation

```
462 if nArgOut==0
463     figure, plot(T,X,'o:')
464     title('Projective Simulation via PIG')
465 end
```

This concludes `PIG()`.

```
473 end
```

2.5 PIRK4(): projective integration of fourth-order accuracy

Section contents

2.5.1	Introduction	24
2.5.2	The projective integration code	27
2.5.3	If no output specified, then plot the simulation	30

2.5.1 Introduction

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

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

See [Section 2.2](#) as the inputs and outputs are the same as `PIRK2()`.

If no arguments, then execute an example

```
29 if nargin==0
```

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

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

With initial conditions $x(0) = y(0) = 0.2$, the following code uses forward time bursts in order to integrate backwards in time to $t = -5$. It plots the computed solution over time $-5 \leq t \leq 0$ for parameter $\epsilon = 0.1$. Since the rate of decay is $\beta \approx 1/\epsilon$ we choose a burst length $\epsilon \log(|\Delta|/\epsilon)$ as here the macroscale time-step $\Delta = -1$.

```

50 global MMepsilon
51 MMepsilon = 0.1
52 ts = 0:-1:-5
53 bT = MMepsilon*log(abs(ts(2)-ts(1))/MMepsilon)
54 [x,tms,xms,rm,svf] = PIRK4(@MMburst, ts, 0.2*[1;1], bT);
55 figure, plot(ts,x,'o:',tms,xms)
56 xlabel('time t'), legend('x(t)','y(t)')
57 title('Backwards-time projective integration of Michaelis--Menton')

```

Upon finishing execution of the example, exit this function.

```

63 return
64 end%if no arguments

```

Code a burst of Michaelis–Menton enzyme kinetics Integrate a burst of length `bT` of the ODEs for the Michaelis–Menton enzyme kinetics at parameter ϵ inherited from above. Code ODEs in function `dMMdt` with variables $x = x(1)$ and $y = x(2)$. Starting at time `ti`, and state `xi` (row), we here simply use MATLAB/Octave’s `ode23/lsode` to integrate a burst in time.

```

15 function [ts, xs] = MMburst(ti, xi, bT)
16     global MMepsilon
17     dMMdt = @(t,x) [ -x(1)+(x(1)+0.5)*x(2)
18                     1/MMepsilon*( x(1)-(x(1)+1)*x(2) ) ];
19     if ~exist('OCTAVE_VERSION','builtin')
20         [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
21     else % octave version
22         [ts, xs] = odeOct(dMMdt, [ti ti+bT], xi);
23     end
24 end

8 function [ts,xs] = odeOct(dxdt,tSpan,x0)
9     if length(tSpan)>2, ts = tSpan;
10     else ts = linspace(tSpan(1),tSpan(end),21);
11     end
12     % mimic ode45 and ode23, but much slower for non-PI
13     lsode_options('integration method','non-stiff');
14     xs = lsode(@(x,t) dxdt(t,x),x0,ts);
15 end

```

Input If there are no input arguments, then this function applies itself to the Michaelis–Menton example: see the code in [Section 2.2.2](#) as a basic template of how to use.

- `microBurst()`, a user-coded function that computes a short-time burst of the microscale simulation.

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

- Inputs: `tStart`, the start time of a burst of simulation; `xStart`, the row n -vector of the starting state; `bT`, *optional*, the total time to simulate in the burst—if your `microBurst()` determines the burst time, then replace `bT` in the argument list by `varargin`.
- Outputs: `tOut`, the column vector of solution times; and `xOut`, an array in which each *row* contains the system state at corresponding times.

- `tSpan` is an ℓ -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: such Nans are carried in the simulation through to the output, and often represent boundaries/edges in spatial fields.
- `bT`, *optional*, either missing, or empty (`[]`), or a scalar: if a given scalar, then it is the length of the micro-burst simulations—the minimum amount of time needed for the microscale simulation to relax to the slow manifold; else if missing or `[]`, then `microBurst()` must itself determine the length of a burst.

124 `if nargin<4, bT=[]; end`

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(microBurst,tSpan,x0,bT)`.

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

- `tms`, *optional*, is an L dimensional column vector containing the microscale times within the burst simulations, each burst separated by NaN;
- `xms`, *optional*, is an $L \times n$ array of the corresponding microscale states—each rows is an accurate estimate of the state at the corresponding time `tms` and helps visualise details of the solution.

- **rm**, optional, a struct containing the ‘remaining’ applications of the microBurst 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* accurately approximate the macroscale dynamics.

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

2.5.2 The projective integration code

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

```
194 nT = length(tSpan);
195 x = nan(nT,length(x0));
```

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

```
203 nArgOut = nargout();
204 saveMicro = (nArgOut>1);
205 saveFullMicro = (nArgOut>3);
206 saveSvf = (nArgOut>4);
```

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

```
219 x0 = reshape(x0,1,[]);
220 [relax_t,relax_x0] = microBurst(tSpan(1),x0,bT);
```

Use the end point of the micro-burst as the initial state for the macroscale time-steps.

```
228 tSpan(1) = relax_t(end);
229 x(1,:) = relax_x0(end,:);
```

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

```

239 if saveMicro
240     tms = cell(nT,1);
241     xms = cell(nT,1);
242     tms{1} = reshape(relax_t,[],1);
243     xms{1} = relax_x0;
244     if saveFullMicro
245         rm.t = cell(nT,1);
246         rm.x = cell(nT,1);
247         if saveSvf
248             svf.t = nan(4*nT-4,1);
249             svf.dx = nan(4*nT-4,length(x0));
250         end
251     end
252 end

```

Loop over the macroscale time-steps

```

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

```

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

```

270     if ~isempty(bT) && 4*abs(bT)>=abs(tSpan(jT)-T) && bT*(tSpan(jT)-T)>0
271         [t1,xm1] = microBurst(T, x(jT-1,:), tSpan(jT)-T);
272         x(jT,:) = xm1(end,:);
273         t2=nan; xm2=nan(1,size(xm1,2));
274         t3=nan; t4=nan; xm3=xm2; xm4 = xm2; dx1=xm2; dx2=xm2;
275     else

```

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

```

286     [t1,xm1] = microBurst(T, x(jT-1,:), bT);
287     del = t1(end)-t1(end-1);

```

Check for round-off error.

```

293     xt = [reshape(t1(end-1:end),[],1) xm1(end-1:end,:)];
294     roundingTol = 1e-8;
295     if norm(diff(xt))/norm(xt,'fro') < roundingTol
296         warning(['significant round-off error in 1st projection at T=' num2str(T)
297     end

```

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

```

306     Dt = tSpan(jT)-t1(end);
307     dx1 = (xm1(end,:)-xm1(end-1,:))/del;

```

Assume burst times are the same length for this macro-step, or effectively so (recall that `bT` may be empty as it may be only coded and known in `microBurst()`).

```
316 abT = t1(end)-t1(1);
```

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

```
327 xint = xm1(end,:) + (Dt/2-abT)*dx1;
328 [t2,xm2] = microBurst(T+Dt/2, xint, bT);
329 del = t2(end)-t2(end-1);
330 dx2 = (xm2(end,:)-xm2(end-1,:))/del;
```

```
331
332 xint = xm1(end,:) + (Dt/2-abT)*dx2;
333 [t3,xm3] = microBurst(T+Dt/2, xint, bT);
334 del = t3(end)-t3(end-1);
335 dx3 = (xm3(end,:)-xm3(end-1,:))/del;
```

```
336
337 xint = xm1(end,:) + (Dt-abT)*dx3;
338 [t4,xm4] = microBurst(T+Dt, xint, bT);
339 del = t4(end)-t4(end-1);
340 dx4 = (xm4(end,:)-xm4(end-1,:))/del;
```

Check for round-off error.

```
346 xt = [reshape(t2(end-1:end),[],1) xm2(end-1:end,:)];
347 if norm(diff(xt))/norm(xt,'fro') < roundingTol
348     warning(['significant round-off error in 2nd projection at T=' num2str(T)
349     end
```

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

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

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

```
365     end
```

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

```
375     if saveMicro
376         tms{jT} = [reshape(t1,[],1); nan];
377         xms{jT} = [xm1; nan(1,size(xm1,2))];
```

If saving all microscale data, then repeat for the remaining applications of the micro-burst.

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

```

387             reshape(t3,[],1); nan;...
388             reshape(t4,[],1); nan];
389         rm.x{jT} = [xm2; nan(1,size(xm2,2));...
390                   xm3; nan(1,size(xm2,2));...
391                   xm4; nan(1,size(xm2,2))];

```

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

```

400         if saveSvf
401             svf.t(4*jT-7:4*jT-4) = [t1(end); t2(end); t3(end); t4(end)];
402             svf.dx(4*jT-7:4*jT-4,:) = [dx1; dx2; dx3; dx4];
403         end
404     end
405 end

```

End of the main loop of all macro-steps.

```

411 end

Overwrite x(1,:) with the specified initial state tSpan(1).
420 x(1,:) = reshape(x0,1,[]);

```

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

```

428 if saveMicro
429     tms = cell2mat(tms);
430     xms = cell2mat(xms);
431     if saveFullMicro
432         rm.t = cell2mat(rm.t);
433         rm.x = cell2mat(rm.x);
434     end
435 end

```

2.5.3 If no output specified, then plot the simulation

```

443 if nArgOut==0
444     figure, plot(tSpan,x,'o:')
445     title('Projective Simulation with PIRK4')
446 end

```

This concludes PIRK4().

```

453 end

```

2.6 cdmc(): constraint defined manifold computing

The function `cdmc()` iteratively applies the given micro-burst and then projects backward to the initial time. 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.


```
17 function [ts, xs] = cdmc(microBurst, t0, x0)
```

Input

- `microBurst()`, a black-box micro-burst function suitable for Projective Integration. See any of `PIRK2()`, `PIRK4()`, or `PIG()` for a description of `microBurst()`.
- `t0`, an initial time.
- `x0`, an initial state vector.

Output

- `ts`, a vector of times.
- `xs`, an array of state estimates produced by `microBurst()`.

This function is a wrapper for the micro-burst. For instance if the problem of interest is a dynamical system that is not too stiff, and which is simulated by the micro-burst function `sol(t,x)`, one would invoke `cdmc()` by defining

```
cdmcSol = @(t,x) cdmc(sol,t,x) |
```

and thereafter use `cdmcSol()` in place of `sol()` as the `microBurst` in any Projective Integration scheme. The original `microBurst sol()` could create large errors if used in the `PIG()` scheme, but the output via `cdmc()` should not.

Begin with a standard application of the micro-burst. Need `feval` as `microBurst` has multiple outputs.

```
56 [t1,x1] = feval(microBurst,t0,x0);
57 bT = t1(end)-t1(1);
```

Project backwards to before the initial time, then simulate just one burst forward to obtain a simulation burst that ends at the original `t0`.

```
66 dxdt = (x1(end,:) - x1(end-1,:))/(t1(end) - t1(end-1));
67 x0 = x1(end,:)-2*bT*dxdt;
68 t0 = t1(1)-bT;
69 [t2,x2] = feval(microBurst,t0,x0.');
```

Return both sets of output(?), although only `(t2,x2)` should be used in Projective Integration—maybe safer to return only `(t2,x2)`.

```
77 ts = [t1(:); t2(:)];
78 xs = [x1; x2];
```

2.7 Example: PI using Runge–Kutta macrosolvers

This script demonstrates the `PIRK4()` scheme that uses a Runge–Kutta macrosolver, applied to simple linear systems with some slow and fast directions.

Clear workspace and set a seed.

```

15 clear
16 rand('seed',1) % albeit discouraged in Matlab
17 global dxdt

```

The majority of this example involves setting up details for the microsolver. We use a simple function `gen_linear_system()` that outputs a function $f(t, x) = A\vec{x} + \vec{b}$, where matrix 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,

```

32 fastband = [-5e2; -1e2];

```

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

```

39 slowband = [-0.002; 0.002];

```

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

```

46 dxdt = gen_linear_system(7,3,fastband,slowband);

```

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

```

56 tSpan = 0:1:20;
57 x0 = linspace(-10,10,10)';

```

We implement the PI scheme, saving the coarse states in `x`, the ‘trusted’ applications of the microsolver in `tms` and `xms`, and the additional applications of the microsolver in `rm` (the second, third and fourth outputs are optional).

```

70 [x, tms, xms, rm] = PIRK4(@linearBurst, tSpan, x0);

```

To verify, we also compute the trajectories using a standard integrator.

```

77 if ~exist('OCTAVE_VERSION','builtin')
78     [tt,xode] = ode45(dxdt,tSpan([1,end]),x0);
79 else % octave version
80     tt = linspace(tSpan(1),tSpan(end),101);
81     xode = lsode(@(x,t) dxdt(t,x),x0,tt);
82 end

```

Figure 2.8 plots the output.

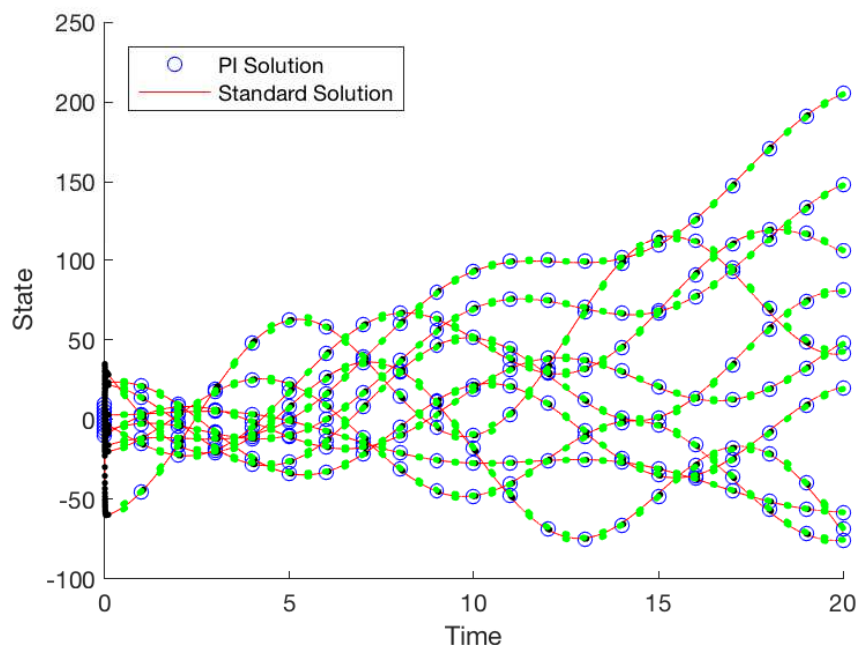
```

98 clf()
99 hold on
100 PI_sol=plot(tSpan,x,'bo');
101 std_sol=plot(tt,xode,'r');
102 plot(tms,xms,'k.', rm.t,rm.x,'g. ');
103 legend([PI_sol(1),std_sol(1)],'PI Solution',...
104        'Standard Solution','Location','NorthWest')
105 xlabel('Time'), ylabel('State')

```

Save plot to a file.

Figure 2.8: 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.



```

111 if ~exist('OCTAVE_VERSION','builtin')
112 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
113 print('-depsc2','PIRKexample')
114 end

```

A micro-burst simulation Used by `PIRKexample.m`. Code the micro-burst function using simple Euler steps. As a rule of thumb, the time-steps Δt should satisfy $\Delta t \leq 1/|\text{fastband}(1)|$ and the time to simulate with each application of the microsolver, `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

```

17 function [ts, xs] = linearBurst(ti, xi, varargin)
18 global dxdt
19 dt = 0.001;
20 ts = ti+(0:dt:0.05)';
21 nts = length(ts);
22 xs = NaN(nts,length(xi));
23 xs(1,:)=xi;
24 for k=2:nts
25     xi = xi + dt*dxdt(ts(k),xi.').';
26     xs(k,:)=xi;
27 end
28 end

```

2.8 Example: Projective Integration using General macrosolvers

In this example the Projective Integration-General scheme is applied to a singularly perturbed ordinary differential equation. The aim is to use a standard non-stiff numerical integrator, such as `ode45()`, on the slow, long-time macroscale. For this stiff system, `PIG()` is an order of magnitude faster than ordinary use of `ode45`.

```
18 clear all, close all
```

Set time scale separation and the underlying ODEs:

$$\frac{dx_1}{dt} = \cos x_1 \sin x_2 \cos t, \quad \frac{dx_2}{dt} = \frac{1}{\epsilon}(-x_2 + \cos x_1).$$

```
30 epsilon = 1e-4;
31 dxdt=@(t,x) [ cos(x(1))*sin(x(2))*cos(t)
32               (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.

```
41 bT = epsilon*log(1/epsilon);
42 if ~exist('OCTAVE_VERSION','builtin')
43     micB='ode45'; else micB='rk2Int'; end
44 microBurst = @(tb0, xb0) feval(micB,dxdt,[tb0 tb0+bT],xb0);
```

Set initial conditions, and the time to be covered by the macrosolver.

```
52 x0 = [1 0.9];
53 tSpan = [0 5];
```

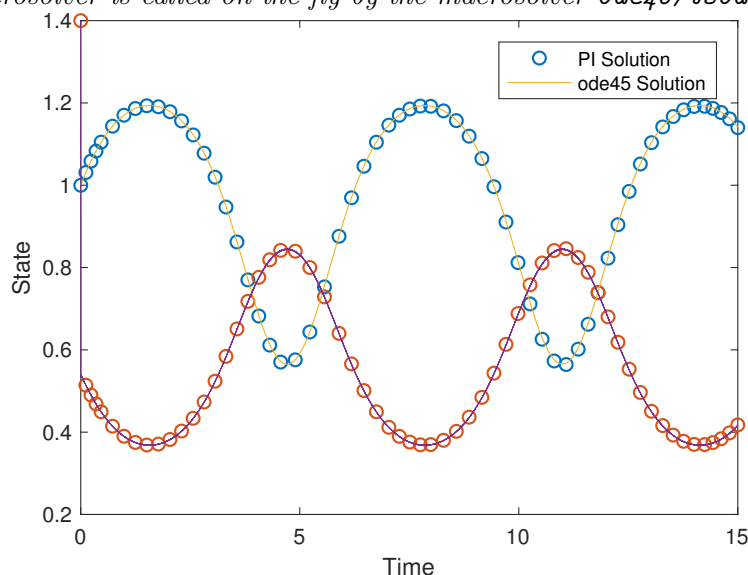
Now time and integrate the above system over `tSpan` using `PIG()` and, for comparison, a brute force implementation of `ode45/lode`. Report the time taken by each method (in seconds).

```
62 if ~exist('OCTAVE_VERSION','builtin')
63     macInt='ode45'; else macInt='odeOct'; end
64 tic
65 [ts,xs,tms,xms] = PIG(macInt,microBurst,tSpan,x0);
66 secsPIGusingODEasMacro = toc
67 tic
68 [tClassic,xClassic] = feval(macInt,dxdt,tSpan,x0);
69 secsODEalone = toc
```

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

```
79 figure
80 h = plot(ts,xs,'o', tClassic,xClassic,'-', tms,xms,'.');
81 legend(h(1:2:5),'Pro Int method','classic method','PI microsolver')
82 xlabel('Time'), ylabel('State')
83
84 figure
85 h = plot(ts,xs,'o', tClassic,xClassic,'-');
```

Figure 2.9: Accurate simulation of a stiff nonautonomous system by `PIG()`. The microsolver is called on-the-fly by the macrosolver `ode45/lsode`.



```

86 legend(h([1 3]),'Pro Int method','classic method')
87 xlabel('Time'), ylabel('State')
88 if ~exist('OCTAVE_VERSION','builtin')
89 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
90 %print('-depsc2','PIGExample')
91 end

```

Figure 2.9 plots the output.

- The problem may be made more stiff or less stiff by changing the time-scale separation parameter $\epsilon = \text{epsilon}$. The compute time of `PIG()` is almost independent of ϵ , whereas that of `ode45()` is proportional to $1/\epsilon$.

If the problem is ‘semi-stiff’ (larger ϵ), then `PIG()`’s default of using `cdmc()` avoids nonsense (Section 2.9).

- The stiff but low dimensional problem in this example may be solved efficiently by a standard stiff solver (e.g., `ode15s()`). The real advantage of the Projective Integration schemes is in high dimensional stiff problems, that are not efficiently solved by most standard methods.

2.9 Explore: Projective Integration using constraint-defined manifold computing

In this example the Projective Integration-General scheme is applied to a singularly perturbed ordinary differential equation in which the time scale separation is not large. The results demonstrate the value of the default `cdmc()` wrapper for the microsolver.

```

16 clear all, close all

```

Set a weak time scale separation, and the underlying ODEs:

$$\frac{dx_1}{dt} = \cos x_1 \sin x_2 \cos t, \quad \frac{dx_2}{dt} = \frac{1}{\epsilon}(-x_2 + \cos x_1).$$

```

28 epsilon = 0.01;
29 dxdt=@(t,x) [ cos(x(1))*sin(x(2))*cos(t)
30               (cos(x(1))-x(2))/epsilon ];

```

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

```

39 bT = epsilon*log(1/epsilon);
40 if ~exist('OCTAVE_VERSION','builtin')
41     micB='ode45'; else micB='rk2Int'; end
42 microBurst = @(tb0, xb0) feval(micB,dxdt,[tb0 tb0+bT],xb0);

```

Set initial conditions, and the time to be covered by the macrosolver.

```

50 x0 = [1 0];
51 tSpan=0:0.5:15;

```

Simulate using PIG(), first without the default treatment of `cdmc` for the microsolver and second with. Generate a trusted solution using standard numerical methods.

```

62 if ~exist('OCTAVE_VERSION','builtin')
63     macInt='ode45'; else macInt='odeOct'; end
64 [nt,nx] = PIG(macInt,microBurst,tSpan,x0,[],[],'no cdmc');
65 [ct,cx] = PIG(macInt,microBurst,tSpan,x0);
66 [tClassic,xClassic] = feval(macInt,dxdt,tSpan,x0);

```

Figure 2.10 plots the output.

```

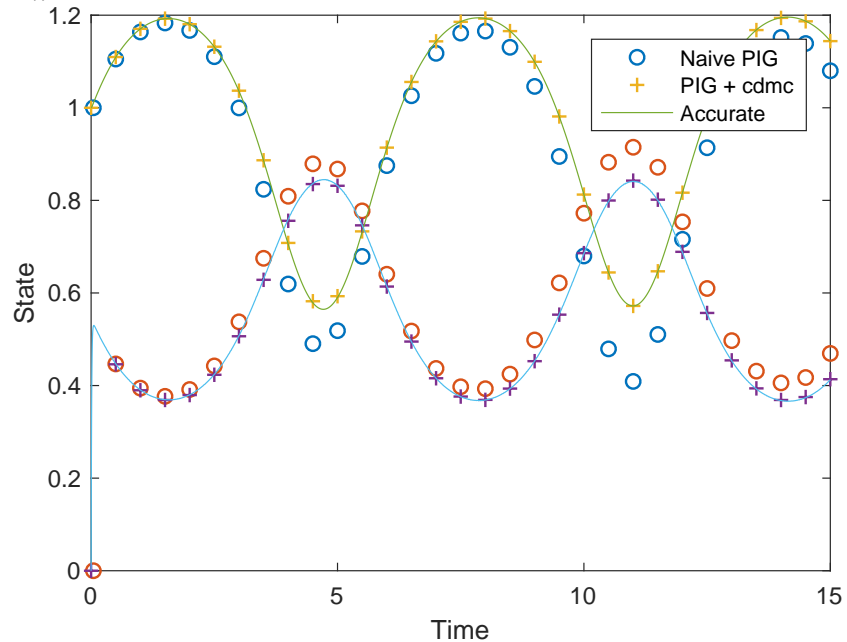
83 figure
84 h = plot(nt,nx,'rx', ct,cx,'bo', tClassic,xClassic,'-');
85 legend(h(1:2:5),'Naive PIG','PIG + cdmc','Accurate')
86 xlabel('Time'), ylabel('State')
87 if ~exist('OCTAVE_VERSION','builtin')
88     set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
89     %print('-depsc2','PIGExplore')
90 end

```

A source of error in the standard PIG() scheme is the finite length of each burst, `bT`. This computes a time derivative at a time that is significantly different to that requested by standard coded schemes. Set `bT` to `20*epsilon` or `50*epsilon`¹ to worsen the error in both schemes. This example reflects a general principle: most Projective Integration schemes incur a global error term proportional to the burst time of the microsolver and independent of the order of the microsolver. The PIRKn() schemes are written to eliminate this error, but PIG() works with any user-defined macrosolver and cannot reduce this error, except by using the function `cdmc()`, its default.

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

Figure 2.10: Accurate simulation of a weakly stiff non-autonomous system by $PIG()$ using $cdmc()$, and an inaccurate solution using a naive application of $PIG()$.



2.10 To do/discuss

- Implement lifting and restriction for $PIRK_n()$ functions.
- 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 maybe implement microsolvers that terminate a burst when the fast dynamics have settled using, for example, the ‘Events’ function handle in `ode23`.
- Need projective integration of systems with fast oscillations, perhaps by DMD.
- Need projective integration for stochastic systems.

3 Patch scheme for given microscale discrete space system

Chapter contents

3.1	Introduction	39
3.2	<code>configPatches1()</code> : configures spatial patches in 1D	39
3.2.1	Introduction	41
3.2.2	If no arguments, then execute an example	42
3.2.3	The code to make patches and interpolation	43
3.3	<code>patchSmooth1()</code> : interface to time integrators	45
3.3.1	Introduction	45
3.4	<code>patchEdgeInt1()</code> : sets edge values from interpolation over the macroscale	46
3.4.1	Introduction	46
3.5	<code>homogenisationExample</code> : simulate heterogeneous diffusion in 1D	50
3.5.1	Script to simulate via stiff or projective integration	51
3.5.2	<code>heteroDiff()</code> : heterogeneous diffusion	53
3.5.3	<code>heteroBurst()</code> : a burst of heterogeneous diffusion	54
3.6	<code>BurgersExample</code> : simulate Burgers' PDE on patches	54
3.6.1	Script code to simulate a microscale space-time map	55
3.6.2	<code>burgersMap()</code> : discretise the PDE microscale	57
3.6.3	<code>burgerBurst()</code> : code a burst of the patch map	58
3.7	<code>ensembleAverageExample</code> : simulate an ensemble of solutions for heterogeneous diffusion in 1D	58
3.7.1	Script to simulate via stiff or projective integration	59
3.8	<code>waterWaveExample</code> : simulate a water wave PDE on patches	63
3.8.1	Script code to simulate wave systems	64
3.8.2	<code>idealWavePDE()</code> : ideal wave PDE	66
3.8.3	<code>waterWavePDE()</code> : water wave PDE	67
3.9	<code>configPatches2()</code> : configures spatial patches in 2D	68
3.9.1	Introduction	68

3.9.2	If no arguments, then execute an example	69
3.9.3	The code to make patches	72
3.10	patchSmooth2(): interface to time integrators	74
3.10.1	Introduction	74
3.11	patchEdgeInt2(): 2D patch edge values from 2D interpolation	75
3.12	wave2D: example of a wave on patches in 2D	79
3.12.1	Check on the linear stability of the wave PDE	80
3.12.2	Execute a simulation	81
3.12.3	wavePDE(): Example of simple wave PDE inside patches	83
3.13	To do	83
3.14	Miscellaneous tests	83
3.14.1	patchEdgeInt1test: test the spectral interpolation . .	83
3.14.2	patchEdgeInt2test: tests 2D spectral interpolation .	85

3.1 Introduction

The patch scheme applies to spatio-temporal systems where the spatial domain is larger than what can be computed in reasonable time in a given complicated microscale code. In the scheme we compute the microscale details 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 resulting macroscale predictions were generally proved to be consistent with the microscale dynamics, to some specified order of accuracy, in a series of papers: 1D-space dissipative systems (Roberts & Kevrekidis 2007, Bunder et al. 2017); 2D-space dissipative systems (Roberts et al. 2014); and 1D-space wave-like systems (Cao & Roberts 2016b).

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

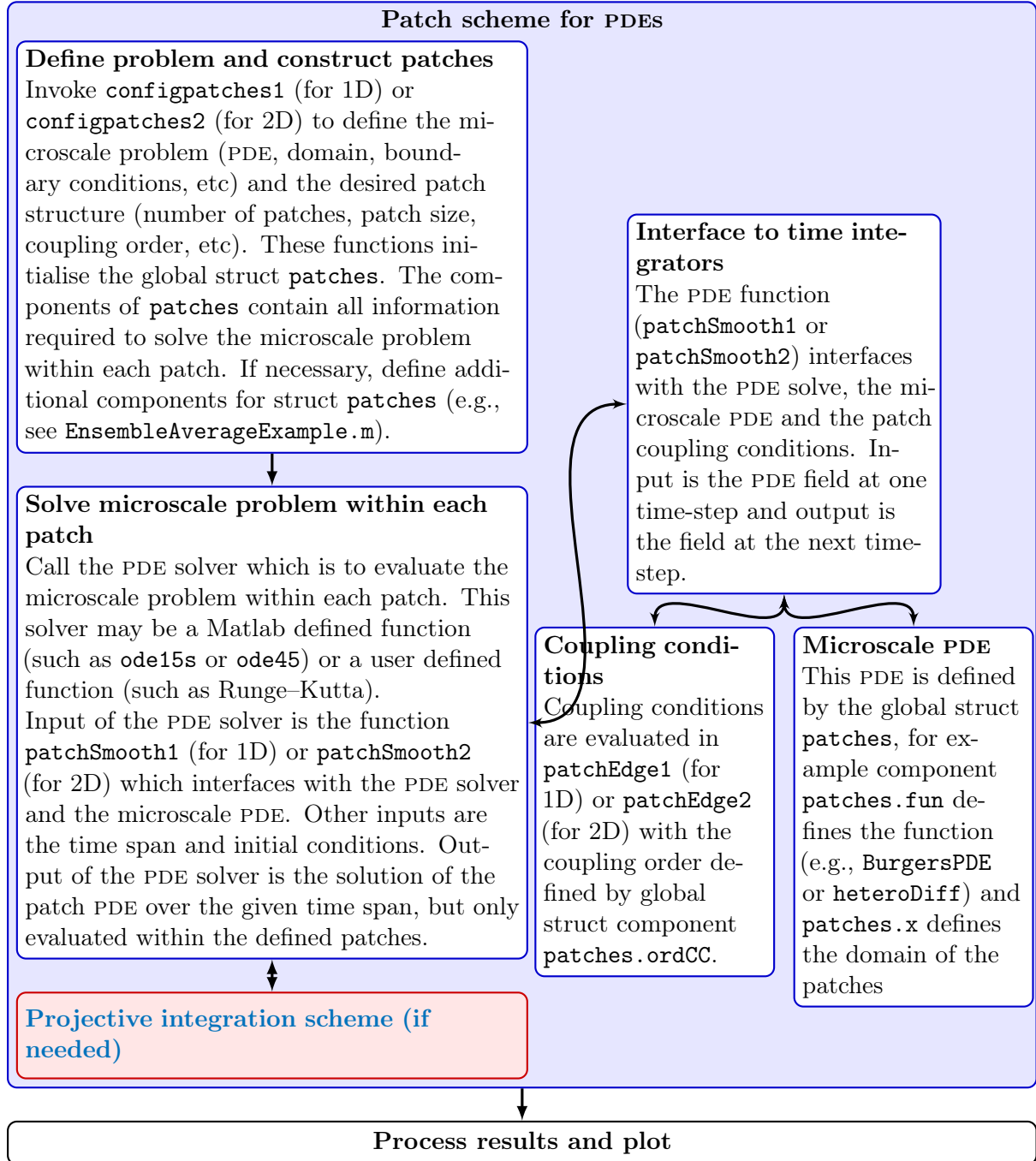
Quick start See Sections 3.2.2 and 3.9.2 which respectively list example basic code that uses the provided functions to simulate the 1D Burgers' PDE, and a 2D nonlinear 'diffusion' PDE. Then see Figure 3.1.

3.2 configPatches1(): configures spatial patches in 1D

Section contents

3.2.1	Introduction	41
3.2.2	If no arguments, then execute an example	42

Figure 3.1: The Patch methods, [Chapter 3](#), accelerate simulation/integration of multiscale systems with interesting spatial (or network) structure/patterns. The patch methods use your given microsimulators whether coded from PDEs, lattice systems, or agent/particle microscale simulators. The patch functions require that a user configure the patches, and interface the coupled patches with a time integrator/simulator. This chart overviews the main functions involved and their interrelationships.



3.2.3 The code to make patches and interpolation 43

3.2.1 Introduction

Makes the struct `patches` for use by the patch/gap-tooth time derivative function `patchSmooth1()`. [Section 3.2.2](#) lists an example of its use.

```

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

```

Input If invoked with no input arguments, then executes an example of simulating Burgers' PDE—see [Section 3.2.2](#) 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 ≥ -1 .
- `ratio` (real) is the ratio of the half-width of a patch to the spacing of the patch mid-points: so `ratio` = $\frac{1}{2}$ means the patches abut; and `ratio` = 1 is overlapping patches as in holistic discretisation.
- `nSubP` is the number of equi-spaced microscale lattice points in each patch. Must be odd so that there is a central lattice point.
- `nEdge`, optional, 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:macro` 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.

3.2.2 If no arguments, then execute an example

```
100 if nargin==0
```

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

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

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

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

Set an initial condition, with some randomness, and simulate in time using a standard stiff integrator and the interface function `patchsmooth1()` (Section 3.3).

```
128 u0=0.3*(1+sin(patches.x))+0.1*randn(size(patches.x));
129 if ~exist('OCTAVE_VERSION','builtin')
130 [ts,uets] = ode15s( @patchSmooth1,[0 0.5],u0(:));
131 else % octave version
132 [ts,uets] = ode0ets(@patchSmooth1,[0 0.5],u0(:));
133 end
```

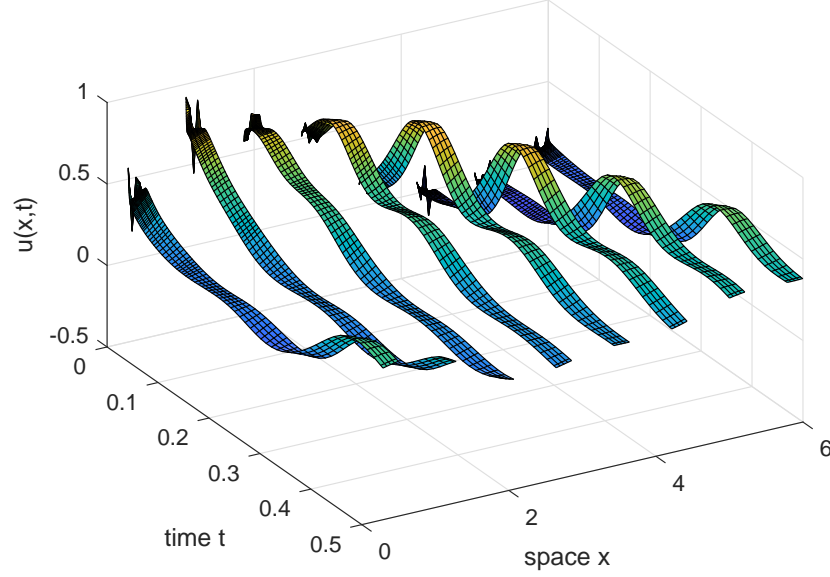
Plot the simulation using only the microscale values interior to the patches: set x -edges to `nan` to leave the gaps. Figure 3.2 illustrates an example simulation in time generated by the patch scheme applied to Burgers' PDE.

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

Upon finishing execution of the example, exit this function.

```
158 return
159 end%if no arguments
```

Figure 3.2: field $u(x,t)$ of the patch scheme applied to Burgers' PDE.
Example of Burgers PDE on patches in space



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

```

12 function ut=BurgersPDE(t,u,x)
13     dx=diff(x(1:2)); % microscale spacing
14     i=2:size(u,1)-1; % interior points in patches
15     ut=nan(size(u)); % preallocate storage
16     ut(i,:)=diff(u,2)/dx^2 ...
17         -30*u(i,:).*(u(i+1,:)-u(i-1,:))/(2*dx);
18 end

```

By default, do not ensemble average.

```

171 patches.EnsAve = 0;

```

3.2.3 The code to make patches and interpolation

Set one edge-value to compute by interpolation if not specified by the user. Store in the struct.

```

181 if nargin<8, nEdge=1; end
182 if nEdge>1, error('multi-edge-value interp not yet implemented'), end
183 if 2*nEdge+1>nSubP, error('too many edge values requested'), end
184 patches.nEdge=nEdge;

```

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

```

191 patches.fun=fun;

```

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

```

199 if (ordCC<-1) | ~(floor(ordCC)==ordCC)
200     error('ordCC out of allowed range integer>-2')
201 end

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

208 patches.alt=mod(ordCC,2);
209 ordCC=ordCC+patches.alt;
210 patches.ordCC=ordCC;

    Check for staggered grid and periodic case.

216 if patches.alt && (mod(nPatch,2)==1)
217     error('Require an even number of patches for staggered grid')
218 end

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

226 patches.Cwtsr=zeros(ordCC,1);
227 if patches.alt % eqn (7) in \cite{Cao2014a}
228     patches.Cwtsr(1:2:ordCC)=[1 ...
229         cumprod((ratio^2-(1:2:(ordCC-2)).^2)/4)./ ...
230         factorial(2*(1:(ordCC/2-1)))];
231     patches.Cwtsr(2:2:ordCC)=[ratio/2 ...
232         cumprod((ratio^2-(1:2:(ordCC-2)).^2)/4)./ ...
233         factorial(2*(1:(ordCC/2-1))+1)*ratio/2];
234 else %
235     patches.Cwtsr(1:2:ordCC)=(cumprod(ratio^2- ...
236         (((1:(ordCC/2))-1)).^2)./factorial(2*(1:(ordCC/2))-1)/ratio);
237     patches.Cwtsr(2:2:ordCC)=(cumprod(ratio^2- ...
238         (((1:(ordCC/2))-1)).^2)./factorial(2*(1:(ordCC/2))));
239 end
240 patches.Cwtsl=(-1).^((1:ordCC)'+patches.alt).*patches.Cwtsr;

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

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

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

257 if mod(nSubP,2)==0, error('configPatches1: nSubP must be odd'), end
258 i0=(nSubP+1)/2;
259 dx=ratio*DX/(i0-1);
260 patches.x=bsxfun(@plus,dx*(-i0+1:i0-1)',X); % micro-grid
261 end% function

    Fin.

```

3.3 patchSmooth1(): interface to time integrators

Section contents

3.3.1 Introduction	45
------------------------------	----

3.3.1 Introduction

To simulate in time with spatial patches we often need to interface a user's 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` (Section 3.2).

```

25 function dudt=patchSmooth1(t,u)
26 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, although 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 microscale.

Output

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

Reshape the fields `u` as a 2/3D-array, and sets the edge values from macroscale interpolation of centre-patch values. Section 3.4 describes `patchEdgeInt1()`.

```

76 u=patchEdgeInt1(u);

```

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

```

86 dudt=patches.fun(t,u,patches.x);
87 dudt([1 end],:,:)=0;
88 dudt=reshape(dudt,[],1);

```

Fin.

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

Section contents

3.4.1 Introduction 46

3.4.1 Introduction

Couples 1D patches across 1D space by computing their edge values from macroscale interpolation of either the mid-patch value or the patch-core average. This function is primarily used by `patchSmooth1()` but is also useful for user graphics. A spatially discrete system could be integrated in time via the patch or gap-tooth scheme (Roberts & Kevrekidis 2007). Assumes that the core averages are in some sense *smooth* so that these averages are sensible macroscale variables. Then patch edge values are determined by macroscale interpolation of the core averages (Bunder et al. 2017). Communicate patch-design variables via the global struct `patches`.

```

27 function u=patchEdgeInt1(u)
28 global patches

```

Input

- `u` is a vector of length $nSubP \cdot nPatch \cdot nVars$ where there are $nVars$ field values at each of the points in the $nSubP \times nPatch$ grid.
- `patches` a struct set by `configPatches1()` which includes the following.
 - `.x` is $nSubP \times nPatch$ 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 microscale.
 - `.ordCC` is order of interpolation integer ≥ -1 .
 - `.alt` in $\{0,1\}$ is one for staggered grid (alternating) interpolation.
 - `.Cwtsr` and `.Cwtsl` define the coupling.

Output

- `u` is $nSubP \times nPatch \times nVars$ 2/3D array of the fields with edge values set by interpolation of patch core averages.

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


```

66 [nSubP,nPatch] = size(patches.x);
67 nVars = round(numel(u)/numel(patches.x));
68 if numel(u)~=nSubP*nPatch*nVars
69     nSubP=nSubP, nPatch=nPatch, nVars=nVars, sizeu=size(u)
70 end
71 u = reshape(u,nSubP,nPatch,nVars);

```

Compute lattice sizes from inside the patches as the edge values may be NaNs, etc.

```

78 dx = patches.x(3,1)-patches.x(2,1);
79 DX = patches.x(2,2)-patches.x(2,1);

```

If the user has not defined the patch core, then we assume it to be a single point in the middle of the patch. For `patches.nCore` $\neq 1$ the half width ratio is reduced, as described by [Bunder et al. \(2017\)](#).

```

88 if ~isfield(patches,'nCore')
89     patches.nCore = 1;
90 end
91 r = dx*(nSubP-1)/2/DX*(nSubP - patches.nCore)/(nSubP - 1);

```

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

```

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

```

Calculate centre of each patch and the surrounding core (`nSubP` and `nCore` are both odd).

```

109 i0 = round((nSubP+1)/2);
110 c = round((patches.nCore-1)/2);

```

Lagrange interpolation gives patch-edge values Consequently, compute centred differences of the patch core averages for the macro-interpolation of all fields. Assumes the domain is macro-periodic.

```

120 if patches.ordCC>0 % then non-spectral interpolation
121     if patches.EnsAve
122         uCore = sum(mean(u((i0-c):(i0+c),j,:),3),1)';
123         dmu = zeros(patches.ordCC,nPatch);
124     else
125         uCore = reshape(sum(u((i0-c):(i0+c),j,:),1),nPatch,nVars);
126         dmu = zeros(patches.ordCC,nPatch,nVars);
127     end;
128     if patches.alt % use only odd numbered neighbours
129         dmu(1,.,.) = (uCore(jp,.)+uCore(jm,.))/2; % \mu
130         dmu(2,.,.) = (uCore(jp,.)-uCore(jm,.)); % \delta
131         jp = jp(jp); jm = jm(jm); % increase shifts to \pm 2
132     else % standard
133         dmu(1,j,.) = (uCore(jp,.)-uCore(jm,.))/2; % \mu\delta

```

```

134     dmu(2,j,:) = (uCore(jp,:)-2*uCore(j,:)+uCore(jm,:))/2; % \delta^2
135     end% if odd/even

```

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

```

143     for k = 3:patches.ordCC
144         dmu(k,,:,) = dmu(k-2,jp,:)-2*dmu(k-2,j,:)+dmu(k-2,jm,:);
145     end

```

Interpolate macro-values to be Dirichlet edge values for each patch ([Roberts & Kevrekidis 2007](#), [Bunder et al. 2017](#)), using weights computed in `configPatches1()`. Here interpolate to specified order.

```

154     if patches.EnsAve
155         u(nSubP,j,:) = repmat(uCore(j)'*(1-patches.alt) ...
156             +sum(bsxfun(@times,patches.Cwtsr,dmu)), [1,1,nVars]) ...
157             -sum(u((nSubP-patches.nCore+1):(nSubP-1),:,:),1);
158         u(1,j,:) = repmat(uCore(j)'*(1-patches.alt) ...
159             +sum(bsxfun(@times,patches.Cwtsl,dmu)), [1,1,nVars]) ...
160             -sum(u(2:patches.nCore,:,:),1);
161     else
162         u(nSubP,j,:) = uCore(j,:)*(1-patches.alt) ...
163             + reshape(-sum(u((nSubP-patches.nCore+1):(nSubP-1),j,:),1) ...
164                 +sum(bsxfun(@times,patches.Cwtsr,dmu)), nPatch,nVars);
165         u(1,j,:) = uCore(j,:)*(1-patches.alt) ...
166             + reshape(-sum(u(2:patches.nCore,j,:),1) ...
167                 +sum(bsxfun(@times,patches.Cwtsl,dmu)), nPatch,nVars);
168     end;

```

Case of spectral interpolation Assumes the domain is macro-periodic. As the macroscale fields are N -periodic, the macroscale Fourier transform writes the centre-patch values as $U_j = \sum_k C_k e^{ik2\pi j/N}$. Then the edge-patch values $U_{j\pm r} = \sum_k C_k e^{ik2\pi/N(j\pm r)} = \sum_k C'_k e^{ik2\pi j/N}$ where $C'_k = C_k e^{ikr2\pi/N}$. For `nPatch` patches we resolve ‘wavenumbers’ $|k| < \text{nPatch}/2$, so set row vector $\mathbf{k}s = k2\pi/N$ for ‘wavenumbers’ $k = (0, 1, \dots, k_{\max}, -k_{\max}, \dots, -1)$ for odd N , and $k = (0, 1, \dots, k_{\max}, \pm(k_{\max} + 1), -k_{\max}, \dots, -1)$ for even N .

```

186     else% spectral interpolation

```

Deal with staggered grid by doubling the number of fields and halving the number of patches (`configPatches1()` tests that there are an even number of patches). Then the patch-ratio is effectively halved. The patch edges are near the middle of the gaps and swapped.

```

196     if patches.alt % transform by doubling the number of fields
197         v = nan(size(u)); % currently to restore the shape of u
198         u = cat(3,u(:,1:2:nPatch,:),u(:,2:2:nPatch,:));
199         altShift = reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
200         iV = [nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
201         r = r/2; % ratio effectively halved
202         nPatch = nPatch/2; % halve the number of patches

```

```

203     nVars = nVars*2;    % double the number of fields
204     else % the values for standard spectral
205         altShift = 0;
206         iV = 1:nVars;
207     end

```

Now set wavenumbers.

```

213     kMax = floor((nPatch-1)/2);
214     ks = 2*pi/nPatch*(mod((0:nPatch-1)+kMax,nPatch)-kMax);

```

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

```

221     if imag(u(i0,:,:))==0, uclean=@(u) real(u);
222     else                    uclean=@(u) u;
223     end

```

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

```

232     Ck = fft(u(i0,:,:));
233     if mod(nPatch,2)==0
234         Czz = Ck(1,nPatch/2+1,:)/nPatch;
235         Ck(1,nPatch/2+1,:) = 0;
236     end

```

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

```

244     u(nSubP,:,iV) = uclean(ifft(bsxfun(@times,Ck ...
245         ,exp(1i*bsxfun(@times,ks,altShift+r)))));
246     u( 1,:,iV) = uclean(ifft(bsxfun(@times,Ck ...
247         ,exp(1i*bsxfun(@times,ks,altShift-r)))));

```

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

```

253     if mod(nPatch,2)==0
254         cosr = cos(pi*(altShift+r+(0:nPatch-1)));
255         u(nSubP,:,iV) = u(nSubP,:,iV)+uclean(bsxfun(@times,Czz,cosr));
256         cosr = cos(pi*(altShift-r+(0:nPatch-1)));
257         u( 1,:,iV) = u( 1,:,iV)+uclean(bsxfun(@times,Czz,cosr));
258     end

```

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

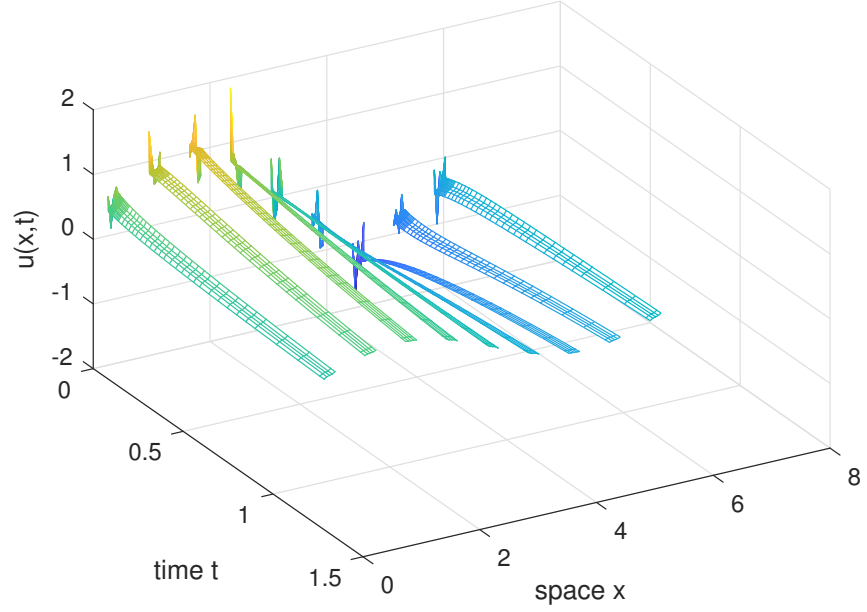
```

265     if patches.alt
266         nVars = nVars/2;  nPatch = 2*nPatch;
267         v(:,1:2:nPatch,:) = u(:, :, 1:nVars);
268         v(:,2:2:nPatch,:) = u(:, :, nVars+1:2*nVars);
269         u = v;
270     end
271     end% if spectral

```

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

Figure 3.3: the diffusing field $u(x, t)$ in the patch (gap-tooth) scheme applied to microscale heterogeneous diffusion (Section 3.5).



3.5 homogenisationExample: simulate heterogeneous diffusion in 1D on patches

Section contents

3.5.1	Script to simulate via stiff or projective integration . .	51
3.5.2	heteroDiff(): heterogeneous diffusion	53
3.5.3	heteroBurst(): a burst of heterogeneous diffusion . .	54

Figure 3.3 shows an example simulation in time generated by the patch scheme function applied to heterogeneous diffusion. That such simulations of heterogeneous diffusion makes valid predictions was established by Bunder et al. (2017) who proved that the scheme is accurate when the number of points in a patch is one more than a multiple of the microscale periodicity.

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

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

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

$$\dot{u}_i = [c_{i-1/2}(u_{i-1} - u_i) + c_{i+1/2}(u_{i+1} - u_i)]/dx^2. \quad (3.1)$$

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

3.5.1 Script to simulate via stiff or projective integration

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

```
50 clear all
51 mPeriod = 3
52 cDiff = exp(randn(mPeriod,1))
53 cHomo = 1/mean(1./cDiff)
```

Establish global data struct `patches` for heterogeneous diffusion solved on 2π -periodic domain, with nine patches, each patch of half-size 0.2. A user can add information to `patches` in order to communicate to the time derivative function. Quadratic (fourth-order) interpolation `ordCC = 4` provides values for the inter-patch coupling conditions. The odd integer `patches.nCore = 3` defines the size of the patch core (this must be larger than zero and less than `nSubP`), where a core of size zero indicates that the value in the centre of the patch gives the macroscale. The introduction of a finite width core requires a redefinition of the half-patch ratio, as described by [Bunder et al. \(2017\)](#). We evaluate the patch coupling by interpolating the core.

```
73 global patches
74 nPatch = 9
75 ratio = 0.2
76 nSubP = 2*mPeriod+1
77 Len = 2*pi;
78 ordCC = 4;
79 configPatches1(@heteroDiff,[0 Len],nan,nPatch ...
80               ,ordCC,ratio,nSubP);
```

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

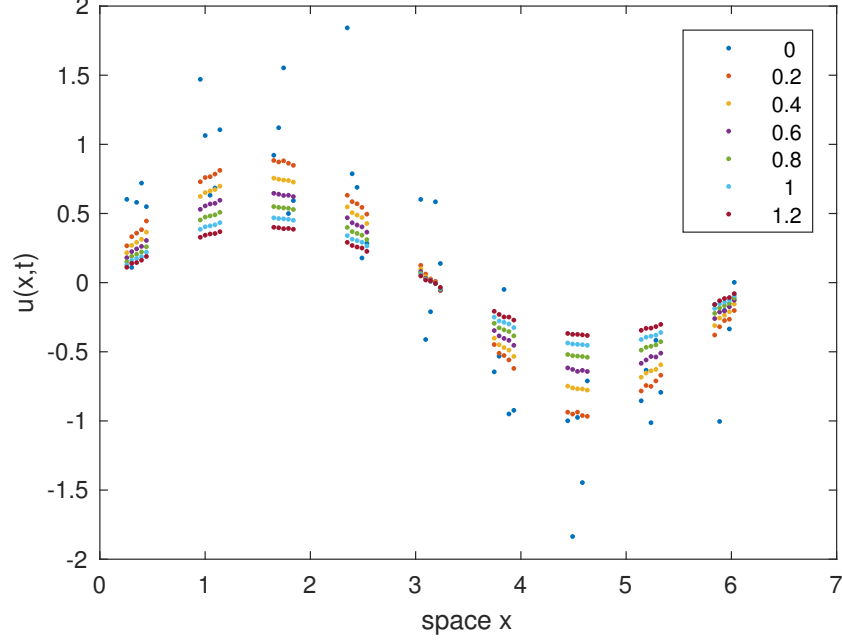
```
89 patches.c= repmat(cDiff,(nSubP-1)/mPeriod,1);
```

For comparison: conventional integration in time Set an initial condition, and here integrate forward in time using a standard method for stiff systems—because of the simplicity of linear problems this method works quite efficiently here. Integrate the interface `patchSmooth1` ([Section 3.3](#)) to the microscale differential equations.

```
102 u0 = sin(patches.x)+0.4*randn(nSubP,nPatch);
103 if ~exist('OCTAVE_VERSION','builtin')
104 [ts,ucls] = ode15s(@patchSmooth1, [0 2/cHomo], u0(:));
105 else % octave version
106 [ts,ucls] = ode0cls(@patchSmooth1, [0 2/cHomo], u0(:));
107 end
108 ucls = reshape(ucls,length(ts),length(patches.x(:)),[]);
```

Plot the simulation in [Figure 3.3](#).

Figure 3.4: field $u(x, t)$ shows basic projective integration of patches of heterogeneous diffusion: different colours correspond to the times in the legend. This field solution displays some fine scale heterogeneity due to the heterogeneous diffusion.



```

115 figure(1),clf
116 xs = patches.x; xs([1 end],:) = nan;
117 mesh(ts,xs(:,ucts'), view(60,40)
118 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
119 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
120 %print('-depsc2','homogenisationCtsU')

```

Use projective integration in time Now take `patchSmooth1`, the interface to the time derivatives, and wrap around it the projective integration `PIRK2` (Section 2.2), of bursts of simulation from `heteroBurst` (Section 3.5.3), as illustrated by Figure 3.4.

This second part of the script implements the following design, where the micro-integrator could be, for example, `ode45` 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.

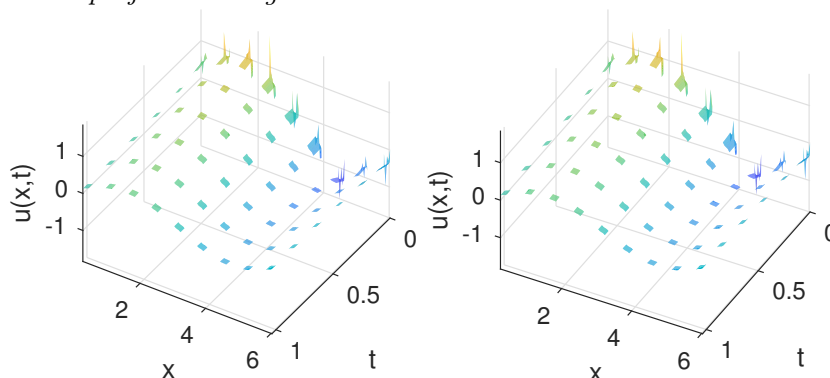
```

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

```

Set the desired macro- and microscale time-steps over the time domain: the macroscale step is in proportion to the effective mean diffusion time on the

Figure 3.5: stereo pair of the field $u(x,t)$ during each of the microscale bursts used in the projective integration.



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.

```

164 ts = linspace(0,2/cHomo,7)
165 bT = 3*( ratio*Len/nPatch )^2/cHomo
166 addpath(' ../ProjInt')
167 [us,tss,uss] = PIRK2(@heteroBurst, ts, u0(:), bT);

```

Plot the macroscale predictions to draw [Figure 3.4](#).

```

174 figure(2),clf
175 plot(xs(:),us','.')
176 ylabel('u(x,t)'), xlabel('space x')
177 legend(num2str(ts',3))
178 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
179 %print('-depsc2','homogenisationU')

```

Also plot a surface detailing the microscale bursts as shown in [Figure 3.5](#).

```

192 figure(3),clf
193 for k = 1:2, subplot(1,2,k)
194     surf(tss,xs(:),uss', 'EdgeColor','none')
195     ylabel('x'), xlabel('t'), zlabel('u(x,t)')
196     axis tight, view(126-4*k,45)
197 end
198 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
199 %print('-depsc2','homogenisationMicro')

```

End of the script.

3.5.2 heteroDiff(): heterogeneous diffusion

This function codes the lattice heterogeneous diffusion inside the patches. For 2D input arrays u and x (via edge-value interpolation of `patchSmooth1`, [Section 3.3](#)), computes the time derivative (3.2) 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`.

```

20 function ut = heteroDiff(t,u,x)
21     global patches
22     dx = diff(x(2:3)); % space step
23     i = 2:size(u,1)-1; % interior points in a patch
24     ut = nan(size(u)); % preallocate output array
25     ut(i, :, :) = diff(patches.c.*diff(u))/dx^2;
26 end% function

```

3.5.3 heteroBurst(): a burst of heterogeneous diffusion

This code integrates in time the derivatives computed by `heteroDiff` from within the patch coupling of `patchSmooth1`. Try `ode23` or `rk2Int`, although `ode45` may give smoother results.

```

15 function [ts, ucts] = heteroBurst(ti, ui, bT)
16     if ~exist('OCTAVE_VERSION','builtin')
17         [ts,ucts] = ode23( @patchSmooth1,[ti ti+bT],ui(:));
18     else % octave version
19         [ts,ucts] = rk2Int(@patchSmooth1,[ti ti+bT],ui(:));
20     end
21 end

```

Fin.

3.6 BurgersExample: simulate Burgers' PDE on patches

Section contents

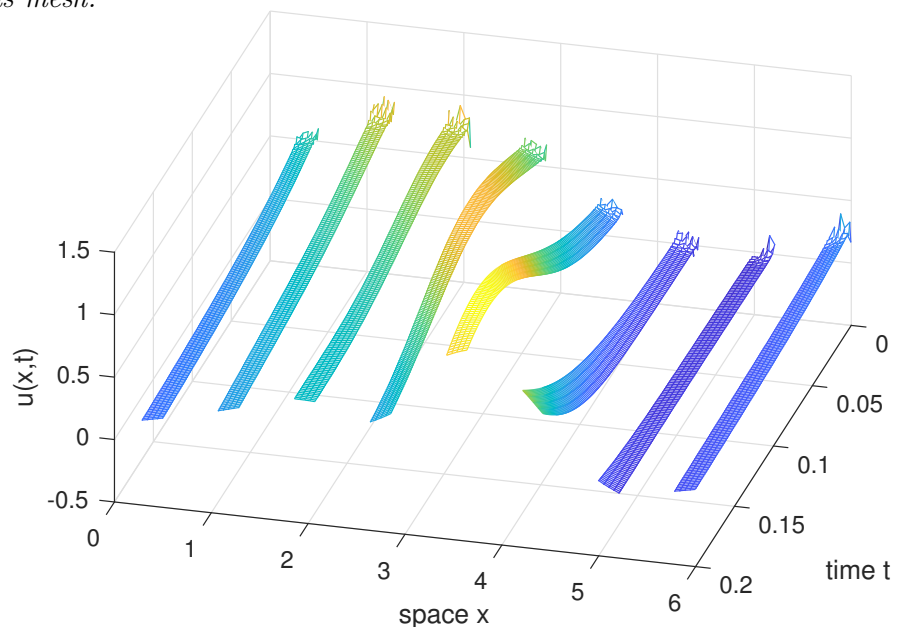
3.6.1	Script code to simulate a microscale space-time map	55
	Alternatively use projective integration	56
3.6.2	<code>burgersMap()</code> : discretise the PDE microscale	57
3.6.3	<code>burgerBurst()</code> : code a burst of the patch map	58

Figure 3.2 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 3.6), a map that happens to be derived as a microscale 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 (left-right arrows denote function recursion).

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

Figure 3.6: a short time simulation of the Burgers' map (Section 3.6.2) on patches in space. It requires many very small time-steps only just visible in this mesh.



3.6.1 Script code to simulate a microscale space-time map

Establish global data struct for the Burgers' map (Section 3.6.2) solved on 2π -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.

```

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

```

Set an initial condition, and simulate a burst of the microscale space-time map over a time 0.2 using the function `burgerBurst()` (Section 3.6.3).

```

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

```

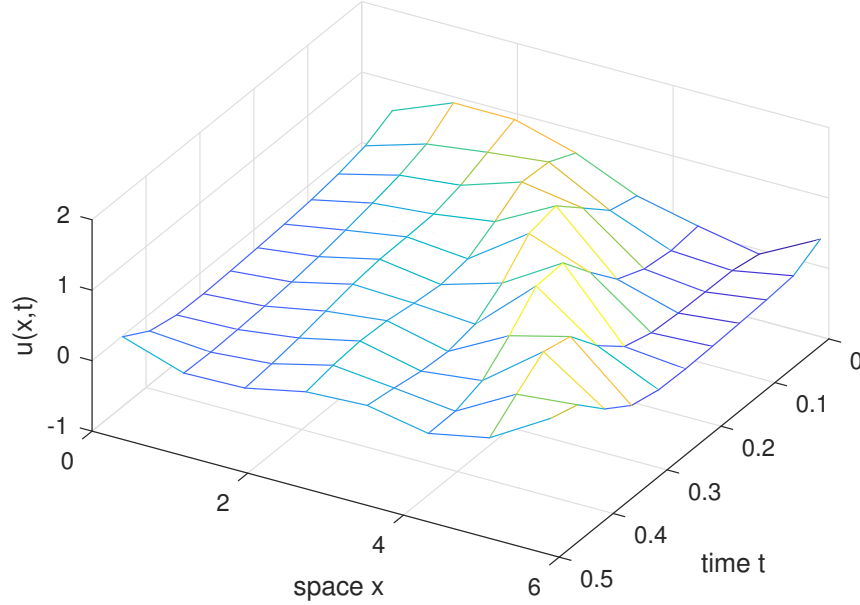
Plot the simulation. Use only the microscale values interior to the patches by setting the edges to `nan` in order to leave gaps.

```

72 figure(1),clf
73 xs = patches.x; xs([1 end],:) = nan;
74 mesh(ts,xs(:,),us')
75 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')

```

Figure 3.7: macroscale space-time field $u(x,t)$ in a basic projective integration of the patch scheme applied to the microscale Burgers' map.



```
76 view(105,45)
```

Save the plot to file to form [Figure 3.6](#).

```
82 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
83 %print('-depsc2','BurgersMapU')
```

Alternatively use projective integration

Around the microscale burst `burgerBurst()`, wrap the projective integration function `PIRK2()` of [Section 2.2](#). [Figure 3.7](#) shows the macroscale prediction of the patch centre values on macroscale 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

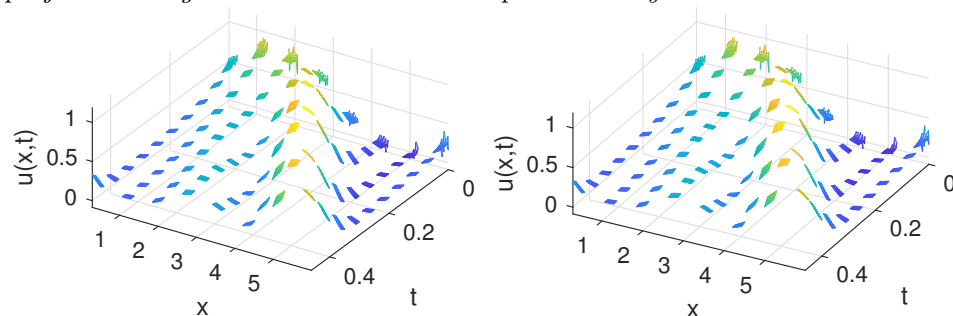
Mark that edge-values of patches are not to be used in the projective extrapolation by setting initial values to NaN.

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

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

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

Figure 3.8: the field $u(x,t)$ during each of the microscale bursts used in the projective integration. View this stereo pair cross-eyed.



Plot and save the macroscale predictions of the mid-patch values to give the macroscale mesh-surface of Figure 3.7 that shows a progressing wave solution.

```

136 figure(2),clf
137 midP = (nSubP+1)/2;
138 mesh(ts,xs(midP,:),us(:,midP:nSubP:end)')
139 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
140 view(120,50)
141 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
142 %print('-depsc2','BurgersU')

```

Then plot and save the microscale mesh of the microscale bursts shown in Figure 3.8 (a stereo pair). The details of the fine microscale mesh are almost invisible.

```

157 figure(3),clf
158 for k = 1:2, subplot(2,2,k)
159     mesh(tss,xs(:),uss')
160     ylabel('x'),xlabel('t'),zlabel('u(x,t)')
161     axis tight, view(126-4*k,50)
162 end
163 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
164 %print('-depsc2','BurgersMicro')

```

3.6.2 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).

```

14 function u = burgersMap(t,u,x)
15     dx = diff(x(2:3));
16     dt = dx^2/2;
17     i = 2:size(u,1)-1;
18     u(i,:) = u(i,:) +dt*( diff(u,2)/dx^2 ...
19         -20*u(i,:).*(u(i+1,:)-u(i-1,:))/(2*dx) );
20 end

```

3.6.3 burgerBurst(): code a burst of the patch map

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

First find and set the number of microscale time-steps.

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

Use `patchSmooth1()` ([Section 3.3](#)) to apply the microscale map over all time-steps in the burst. The `patchSmooth1()` interface provides the interpolated edge-values of each patch. Store the results in rows to be consistent with ODE and projective integrators.

```
29 us = nan(ndt+1,numel(ui));
30 us(1,:) = reshape(ui,1,[]);
31 for j = 1:ndt
32     ui = patchSmooth1(ts(j),ui);
33     us(j+1,:) = reshape(ui,1,[]);
34 end
```

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

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

Fin.

3.7 ensembleAverageExample: simulate an ensemble of solutions for heterogeneous diffusion in 1D on patches

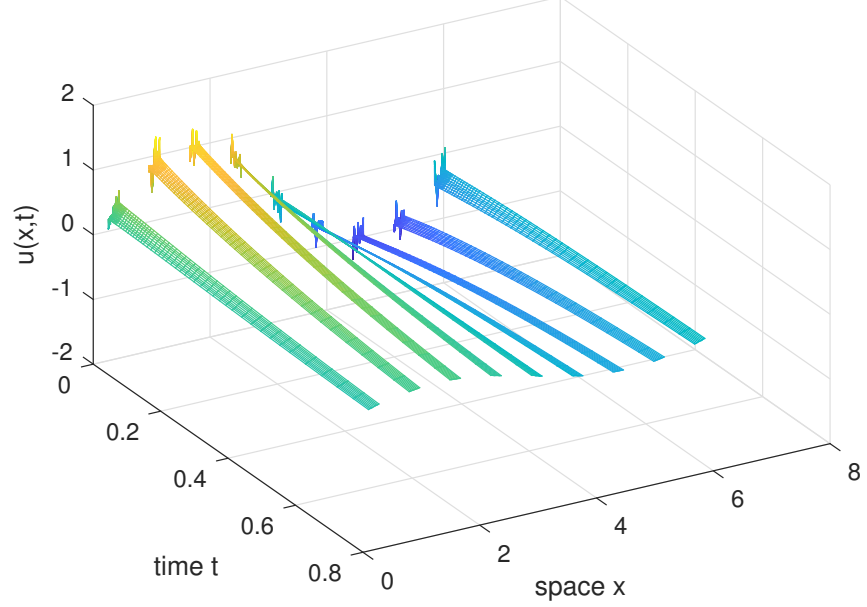
Section contents

3.7.1 Script to simulate via stiff or projective integration . . . 59

This example is an extension of the homogenisation example of [Section 3.5](#) for heterogeneous diffusion. In cases where the periodicity of the heterogeneous diffusion is known, then [Section 3.5](#) provides a efficient patch dynamics simulation. However, if the diffusion is not completely known or is stochastic, then we cannot choose ideal patch and core sizes as described by [Bunder et al. \(2017\)](#) and applied in [Section 3.5](#). In this case, [Bunder et al. \(2017\)](#) recommend constructing an ensemble of diffusivity configurations and then computing an ensemble of field solutions, finally averaging over the ensemble of fields to obtain the ensemble averaged field solution.

For a first comparison, we present a very similar example to that presented in [Section 3.5](#), but whereas [Section 3.5](#) simulates using only one diffusivity

Figure 3.9: the diffusing field $u(x,t)$ in the patch (gap-tooth) scheme applied to microscale heterogeneous diffusion with an ensemble average. The ensemble average smooths out the heterogeneous diffusion.



configuration, here we simulate over an ensemble. For example, [Figure 3.9](#) is similar to [Figure 3.3](#), but the former is an ensemble average of an ensemble of eight different simulations with different diffusivity configurations and the latter is simulated from just one diffusivity configuration. The main difference between these two is that the average over the ensemble removes any heterogeneity in the solution.

Much of this script is similar to that of [Section 3.5](#), but with some additions to manage the ensemble. The first part of the script implements the following gap-tooth scheme (left-right arrows denote function recursion).

1. configPatches1
2. ode15s \leftrightarrow patchSmooth1 \leftrightarrow heteroDiff
3. process results

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

$$\dot{u}_i = [c_{i-1/2}(u_{i-1} - u_i) + c_{i+1/2}(u_{i+1} - u_i)]/dx^2. \quad (3.2)$$

In this 1D space, the macroscale, homogenised, effective diffusion should be the harmonic mean of these coefficients. But we do not have full knowledge of these coefficients.

3.7.1 Script to simulate via stiff or projective integration

Say we only know four diffusivities in our diffusion problem, as defined here (which are the same as those given in [Section 3.5](#)).

```

75 clear all
76 mPeriod = 4
77 rand('seed',1);
78 c = exp(4*rand(mPeriod,1))
79 cHomo = 1/mean(1./c)

```

The chosen parameters are the same as [Section 3.5](#), but here we also introduce the Boolean `patches.EnsAve` which determines whether or not we construct an ensemble average of diffusivity configurations. Setting `patches.EnsAve=0` simulates the same problem as in [Section 3.5](#).

```

91 global patches
92 nPatch = 9
93 ratio = 0.2
94 nSubP = 11
95 Len = 2*pi;
96 ordCC = 4;
97 patches.nCore = 3;
98 patches.ratio = ratio*(nSubP - patches.nCore)/(nSubP - 1);
99 configPatches1(@heteroDiff,[0 Len],nan,nPatch ...
100     ,ordCC,patches.ratio,nSubP);
101 patches.EnsAve = 1;

```

In the case of ensemble averaging, `nVars` is the size of the ensemble (for the case of no ensemble averaging `nVars` is the number of different field variables, which in this example is `nVars = 1`) and we use the ensemble described by [Bunder et al. \(2017\)](#) which includes all reflected and translated configurations of `patches.c`. We must increase the size of the diffusivity matrix to $(nSubP-1) \times nPatch \times nVars$.

```

116 patches.c = c((mod(round(patches.x(1:(end-1),:)) ...
117     /(patches.x(2)-patches.x(1))-0.5),mPeriod)+1));
118 if patches.EnsAve
119     nVars = mPeriod+(mPeriod>2)*mPeriod;
120     patches.c = repmat(patches.c,[1,1,nVars]);
121     for sx = 2:mPeriod
122         patches.c(:,:,sx) = circshift( ...
123             patches.c(:,:,sx-1),[sx-1,0]);
124     end;
125     if nVars>2
126         patches.c(:,:,mPeriod+1:end) = flipud( ...
127             patches.c(:,:,1:mPeriod));
128     end;
129 end

```

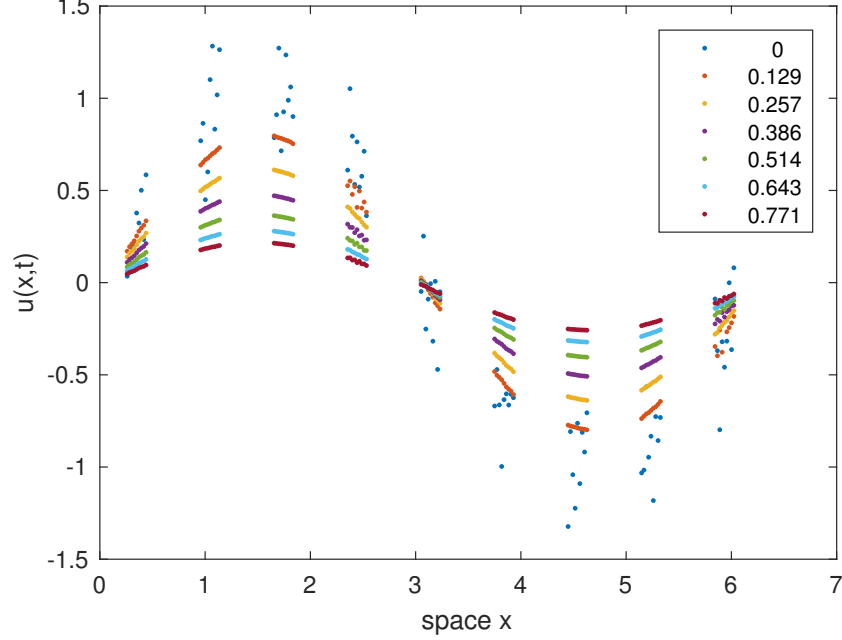
Conventional integration in time Set an initial condition, and here integrate forward in time using a standard method for stiff systems. Integrate the interface `patchSmooth1` ([Section 3.3](#)) to the microscale differential equations.

```

140 u0 = sin(patches.x)+0.2*randn(nSubP,nPatch);
141 if patches.EnsAve

```

Figure 3.10: field $u(x,t)$ shows basic projective integration of patches of heterogeneous diffusion with an ensemble average: different colours correspond to the times in the legend. Once transients have decayed, this field solution is smooth due to the ensemble average.



```

142     u0 = repmat(u0,[1,1,nVars]);
143 end
144 if ~exist('OCTAVE_VERSION','builtin')
145     [ts,ucts] = ode15s( @patchSmooth1, [0 2/cHomo], u0(:));
146 else % octave version is slower
147     [ts,ucts] = ode0cts(@patchSmooth1, [0 2/cHomo], u0(:));
148 end
149 ucts = reshape(ucts,length(ts),length(patches.x(:)),[]);

```

Plot the ensemble averaged simulation in [Figure 3.9](#).

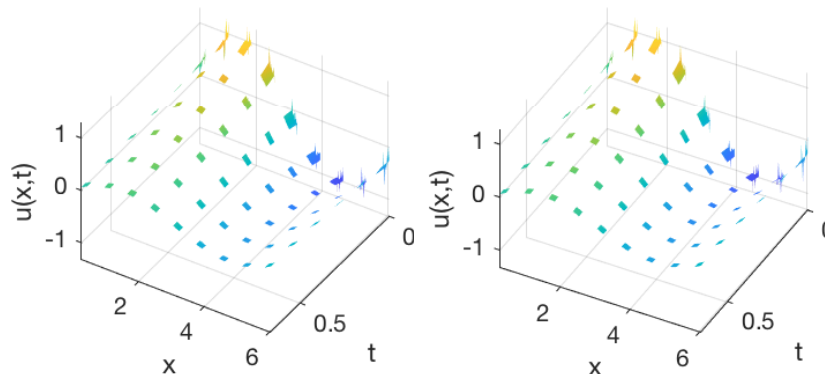
```

157 if patches.EnsAve % calculate the ensemble average
158     uctsAve = mean(ucts,3);
159 else
160     uctsAve = ucts;
161 end
162 figure(1),clf
163 xs = patches.x; xs([1 end],:) = nan;
164 mesh(ts,xs(:),uctsAve', view(60,40)
165 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
166 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
167 %print('-depsc2','ensAveExCtsU')

```

Use projective integration in time Now take `patchSmooth1`, the interface to the time derivatives, and wrap around it the projective integration `PIRK2` ([Section 2.2](#)), of bursts of simulation from `heteroBurst` ([Section 3.5.3](#)),

Figure 3.11: stereo pair of ensemble averaged fields $u(x,t)$ during each of the microscale bursts used in the projective integration.



as illustrated by Figure 3.10. The rest of this code follows that of Section 3.5, but as we now evaluate an ensemble of field solutions, our final step is always an ensemble average.

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

```
194 disp('Now start Projective Integration')
195 u0([1 end],:) = nan;
```

Set the desired macro- and microscale time-steps over the time domain.

```
202 ts = linspace(0,2/cHomo,7)
203 bT = 3*( ratio*Len/nPatch )^2/cHomo
204 addpath(' ../ProjInt')
205 [us,tss,uss] = PIRK2(@heteroBurst, ts, u0(:), bT);
```

Plot an average of the ensemble of macroscale predictions to draw Figure 3.10.

```
212 usAve = mean(reshape(us,size(us,1),length(xs(:)),nVars),3);
213 ussAve = mean(reshape(uss,length(tss),length(xs(:)),nVars),3);
214 figure(2),clf
215 plot(xs(:),usAve','.')
216 ylabel('u(x,t)'), xlabel('space x')
217 legend(num2str(ts',3))
218 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
219 %print('-depsc2','ensAveExU')
```

Also plot a surface detailing the ensemble average microscale bursts as shown Figure 3.11.

```
234 figure(3),clf
235 for k = 1:2, subplot(1,2,k)
236     surf(tss,xs(:),ussAve', 'EdgeColor','none')
237     ylabel('x'), xlabel('t'), zlabel('u(x,t)')
238     axis tight, view(126-4*k,45)
239 end
```



```

240 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
241 %print('-depsc2','ensAveExMicro')

```

End of the script.

Sections 3.5.2 and 3.5.3 list the functions used here.

Fin.

3.8 waterWaveExample: simulate a water wave PDE on patches

Section contents

3.8.1	Script code to simulate wave systems	64
3.8.2	idealWavePDE(): ideal wave PDE	66
3.8.3	waterWavePDE(): water wave PDE	67

Figure 3.12 shows an example simulation in time generated by the patch scheme function applied to an ideal wave PDE (Cao & Roberts 2013). The inter-patch coupling is realised by spectral interpolation of the mid-patch values to the patch edges.

This approach, based upon the differential equations coded in Section 3.8.2, may be adapted by a user to a wide variety of 1D wave and wave-like systems. For example, the differential equations of Section 3.8.3 describes the nonlinear microscale simulator of the nonlinear shallow water wave PDE derived from the Smagorinski model of turbulent flow (Cao & Roberts 2012, 2016a).

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 longitudinal velocity $u(x, t)$ as herein. The approach developed in this section applies to any wave-like system in the form

$$\frac{\partial h}{\partial t} = -c_1 \frac{\partial u}{\partial x} + f_1[h, u] \quad \text{and} \quad \frac{\partial u}{\partial t} = -c_2 \frac{\partial h}{\partial x} + f_2[h, u], \quad (3.3)$$

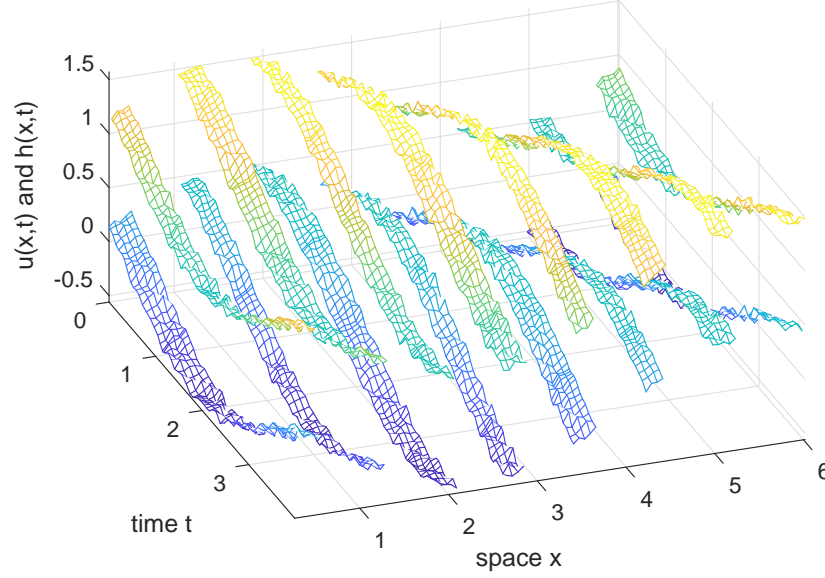
where the brackets indicate that the nonlinear functions f_ℓ may involve various spatial derivatives of the fields $h(x, t)$ and $u(x, t)$. For example, Section 3.8.3 encodes a nonlinear Smagorinski model of turbulent shallow water (Cao & Roberts 2012, 2016a, e.g.) along an inclined flat bed: let x measure position along the bed and in terms of fluid depth $h(x, t)$ and depth-averaged longitudinal velocity $u(x, t)$ the model PDEs are

$$\frac{\partial h}{\partial t} = -\frac{\partial(hu)}{\partial x}, \quad (3.4a)$$

$$\frac{\partial u}{\partial t} = 0.985 \left(\tan \theta - \frac{\partial h}{\partial x} \right) - 0.003 \frac{u|u|}{h} - 1.045u \frac{\partial u}{\partial x} + 0.26h|u| \frac{\partial^2 u}{\partial x^2}, \quad (3.4b)$$

where $\tan \theta$ is the slope of the bed. Equation (3.4a) represents conservation of the fluid. The momentum PDE (3.4b) represents the effects of

Figure 3.12: water depth $h(x,t)$ (above) and velocity field $u(x,t)$ (below) of the gap-tooth scheme applied to the ideal wave PDE (3.3), linearised. The microscale random component to the initial condition has long lasting effects on the simulation—but the macroscale wave still propagates.



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 3.13 shows one simulation of this system—for the same initial condition as Figure 3.12.

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

3.8.1 Script code to simulate wave systems

This script implements the following gap-tooth scheme (left-right arrows denote function recursion).

1. configPatches1, and add micro-information
2. ode15s \leftrightarrow patchSmooth1 \leftrightarrow idealWavePDE
3. process results
4. ode15s \leftrightarrow patchSmooth1 \leftrightarrow waterWavePDE
5. process results

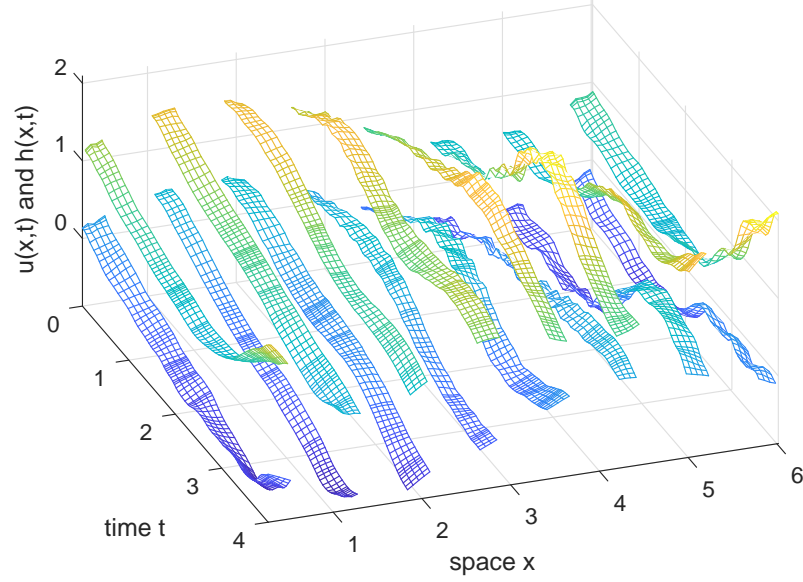
Establish the global data struct **paches** for the PDEs (3.3) (linearised) solved on 2π -periodic domain, with eight patches, each patch of half-size ratio 0.2, with eleven points within each patch, and spectral interpolation (-1) to provide edge-values of the inter-patch coupling conditions.

```

115 clear all
116 global patches
117 nPatch = 8
118 ratio = 0.2

```

Figure 3.13: 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 (3.4). The microscale random initial component decays where the water speed is non-zero due to ‘turbulent’ dissipation.



```

119 nSubP = 11 %of the form 4*n-1
120 Len = 2*pi;
121 configPatches1(@idealWavePDE,[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.

```

131 uPts = mod( bsxfun(@plus,(1:nSubP)',(1:nPatch)) ,2);
132 hPts = find(1-uPts);
133 uPts = find(uPts);
134 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 microscale wave noise).

```

145 U0 = nan(nSubP,nPatch);
146 U0(hPts) = 1+0.5*sin(patches.x(hPts));
147 U0(uPts) = 0+0.5*sin(patches.x(uPts));
148 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 ideal wave and the water wave equations in the one loop.

```

158 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.

```

166 if ~exist('OCTAVE_VERSION','builtin')
167     [ts,Ucts] = ode15s( @patchSmooth1,[0 4],U0(:));
168     ts = ts(1:5:end);
169     Ucts = Ucts(1:5:end,:);
170 else % octave version is slower
171     [ts,Ucts] = ode0cts(@patchSmooth1,[0 4],U0(:));
172 end
    Plot the simulation.

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

```

Save the plot to file.

```

189 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
190 % if k==1, print('-depsc2','ps1WaveCtsUH')
191 % else print('-depsc2','ps1WaterWaveCtsUH')
192 % end

```

For the second time through the loop, change to the Smagorinski turbulence model (3.4) of shallow water flow, keeping other parameters and the initial condition the same.

```

202 patches.fun = @waterWavePDE;
203 end

```

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

3.8.2 idealWavePDE(): ideal wave PDE

This function codes the staggered lattice equation inside the patches for the ideal wave PDE system $h_t = -u_x$ and $u_t = -h_x$. Here code for a staggered microscale grid, index i , of staggered macroscale patches, index j : 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.

```

23 function Ut = idealWavePDE(t,U,x)
24     global patches
25     dx = diff(x(2:3));
26     Ut = nan(size(U)); ht = Ut;

```

Compute the PDE derivatives at interior points of the patches.

```
32    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.

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

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

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

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

```
61    Ut(patches.hPts) = ht(patches.hPts);
```

```
62    end
```

3.8.3 waterWavePDE(): water wave PDE

This function codes the staggered lattice equation inside the patches for the nonlinear wave-like PDE system (3.4). Also, regularise the absolute value appearing in the PDEs via the one-line function `rabs()`.

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

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

```
25    dx = diff(x(2:3));
26    Ut = nan(size(U)); ht = Ut;
27    i = 2:size(U,1)-1;
```

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

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

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

```
46    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 $U_{i\pm 1}$.

```

54     Ut(i,:) = -0.985*(U(i+1,:)-U(i-1,:))/(2*dx) ...
55             -0.003*U(i,:).*rabs(U(i,:)./V(i,:)) ...
56             -1.045*U(i,:).*(V(i+1,:)-V(i-1,:))/(2*dx) ...
57             +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} .

```

73     Ut(patches.hPts) = ht(patches.hPts);
74 end

```

Fin.

3.9 configPatches2(): configures spatial patches in 2D

Section contents

3.9.1	Introduction	68
3.9.2	If no arguments, then execute an example	69
3.9.3	The code to make patches	72

3.9.1 Introduction

Makes the struct `patches` for use by the patch/gap-tooth time derivative function `patchSmooth2()`. [Section 3.9.2](#) lists an example of its use.

```

19 function configPatches2(fun,Xlim,BCs,nPatch,ordCC,ratio,nSubP,nEdge)
20 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 [Section 3.9.2](#) 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 distributed equi-spaced over the interior of the rectangle $[\text{Xlim}(1), \text{Xlim}(2)] \times [\text{Xlim}(3), \text{Xlim}(4)]$: if `Xlim` is of length two, then use the same interval in both directions.

- **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 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** = $\frac{1}{2}$ 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.

3.9.2 If no arguments, then execute an example

123 **if nargin==0**

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

1. configPatches2
2. ode15s integrator \leftrightarrow patchSmooth2 \leftrightarrow user's nonDiffPDE
3. process results

Establish global patch data struct to interface with a function coding a nonlinear 'diffusion' PDE: to be solved on 6×4 -periodic domain, with 9×7 patches, spectral interpolation (0) couples the patches, each patch of half-size ratio 0.25, and with 5×5 points within each patch. [Roberts et al. \(2014\)](#) established that this scheme is consistent with the PDE (as the patch spacing decreases).

```

145 nSubP = 5;
146 configPatches2(@nonDiffPDE,[-3 3 -2 2], nan, [9 7], 0, 0.25, nSubP);

```

Set a Gaussian initial condition using auto-replication of the spatial grid.

```

153 x = reshape(patches.x,nSubP,1,[],1);
154 y = reshape(patches.y,1,nSubP,1,[]);
155 u0 = exp(-x.^2-y.^2);
156 u0 = u0.*(0.9+0.1*rand(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.

```

164 figure(1), clf
165 x = patches.x; y = patches.y;
166 x([1 end],:) = nan; y([1 end],:) = nan;

```

Start by showing the initial conditions of [Figure 3.14](#) while the simulation computes.

```

173 u = reshape(permute(u0,[1 3 2 4]), [numel(x) numel(y)]);
174 hsurf = surf(x(:),y(:),u');
175 axis([-3 3 -3 3 -0.001 1]), view(60,40)
176 legend('time = 0','Location','north')
177 xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')

```

Save the initial condition to file for [Figure 3.14](#).

```

183 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
184 %print('-depsc2','configPatches2ic')

```

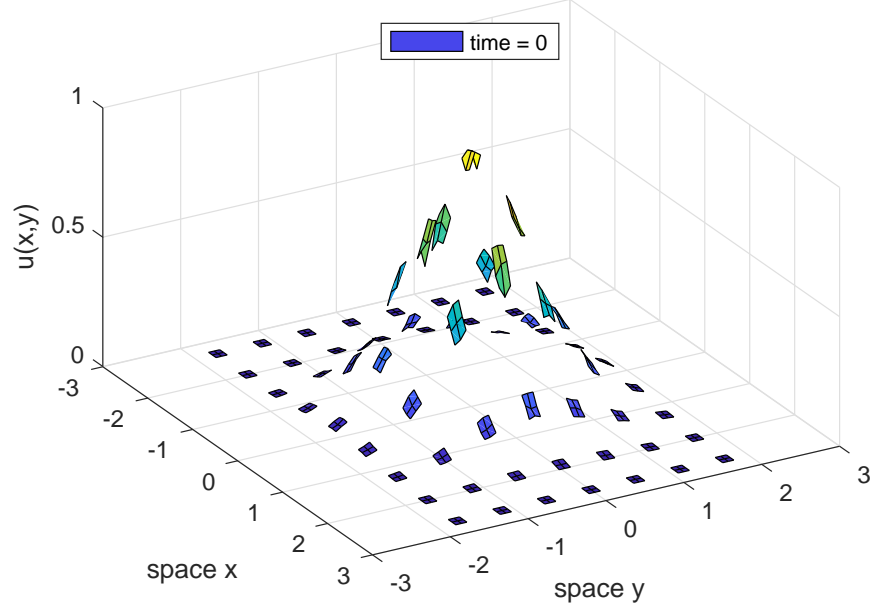
Integrate in time using standard functions.

```

198 disp('Wait while we simulate h_t=(h^3)_xx+(h^3)_yy')
199 drawnow
200 if ~exist('OCTAVE_VERSION','builtin')
201     [ts,ucts] = ode15s( @patchSmooth2,[0 3],u0(:));
202 else % octave version is quite slow
203     lsode_options('absolute tolerance',1e-4);
204     lsode_options('relative tolerance',1e-4);

```


Figure 3.14: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to a nonlinear ‘diffusion’ PDE: Figure 3.15 plots the computed field at time $t = 3$.



```

205         [ts,ucts] = odeOcts(@patchSmooth2,[0 1],u0(:));
206     end

    Animate the computed simulation to end with Figure 3.15.

213     for i = 1:length(ts)
214         u = patchEdgeInt2(ucts(i,:));
215         u = reshape(permute(u,[1 3 2 4]), [numel(x) numel(y)]);
216         set(hsurf,'ZData', u');
217         legend(['time = ' num2str(ts(i),2)])
218         pause(0.1)
219     end
220     %print('-depsc2','configPatches2t3')

```

Upon finishing execution of the example, exit this function.

```

235     return
236 end%if no arguments

```

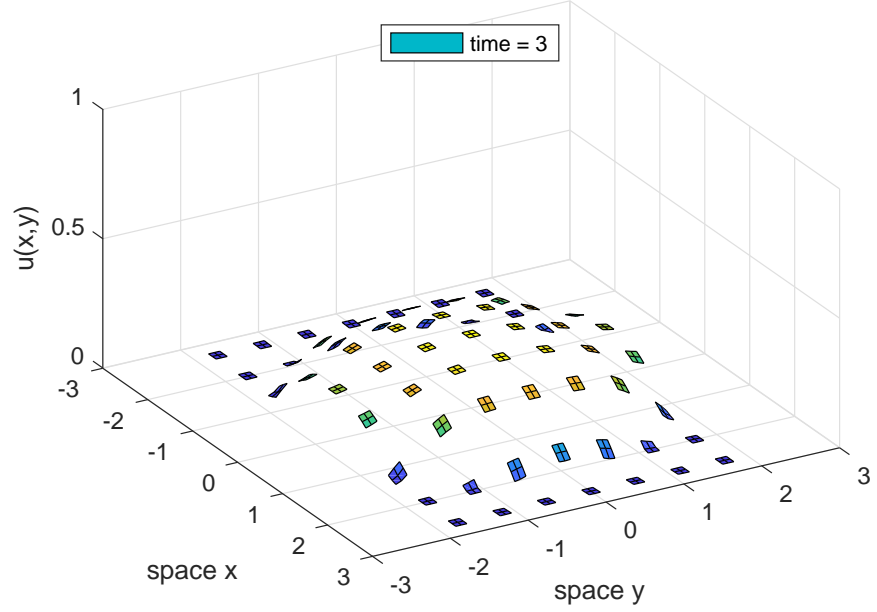
Example of nonlinear diffusion PDE inside patches As a microscale discretisation of $u_t = \nabla^2(u^3)$, code $\dot{u}_{ijkl} = \frac{1}{\delta x^2}(u_{i+1,j,k,l}^3 - 2u_{i,j,k,l}^3 + u_{i-1,j,k,l}^3) + \frac{1}{\delta y^2}(u_{i,j+1,k,l}^3 - 2u_{i,j,k,l}^3 + u_{i,j-1,k,l}^3)$.

```

13 function ut = nonDiffPDE(t,u,x,y)
14     dx = diff(x(1:2)); dy = diff(y(1:2)); % microscale spacing
15     i = 2:size(u,1)-1; j = 2:size(u,2)-1; % interior points in patches
16     ut = nan(size(u)); % preallocate storage
17     ut(i,j, :, :) = diff(u(:,j, :, :).^3,2,1)/dx^2 ...
18         +diff(u(i, :, :, :).^3,2,2)/dy^2;

```

Figure 3.15: field $u(x,y,t)$ at time $t = 3$ of the patch scheme applied to a nonlinear ‘diffusion’ PDE with initial condition in Figure 3.14.



19 end

3.9.3 The code to make patches

Initially duplicate parameters as needed.

```

254 if numel(Xlim)==2, Xlim = repmat(Xlim,1,2); end
255 if numel(nPatch)==1, nPatch = repmat(nPatch,1,2); end
256 if numel(ratio)==1, ratio = repmat(ratio,1,2); end
257 if numel(nSubP)==1, nSubP = repmat(nSubP,1,2); end

```

Set one edge-value to compute by interpolation if not specified by the user.
Store in the struct.

```

265 if nargin<8, nEdge = 1; end
266 if nEdge>1, error('multi-edge-value interp not yet implemented'), end
267 if 2*nEdge+1>nSubP, error('too many edge values requested'), end
268 patches.nEdge = nEdge;

```

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

```

277 patches.fun = fun;

```

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

```

286 if ~ismember(ordCC,[0])
287     error('ordCC out of allowed range [0]')
288 end

```

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

```

295 patches.alt = mod(ordCC,2);
296 ordCC = ordCC+patches.alt;
297 patches.ordCC = ordCC;

```

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

```

313 ratio = ratio(:)'; % force to be row vector
314 if patches.alt % eqn (7) in \cite{Cao2014a}
315     patches.Cwtsr = [1
316         ratio/2
317         (-1+ratio.^2)/8
318         (-1+ratio.^2).*ratio/48
319         (9-10*ratio.^2+ratio.^4)/384
320         (9-10*ratio.^2+ratio.^4).*ratio/3840
321         (-225+259*ratio.^2-35*ratio.^4+ratio.^6)/46080
322         (-225+259*ratio.^2-35*ratio.^4+ratio.^6).*ratio/645120 ];
323 else %
324     patches.Cwtsr = [ratio
325         ratio.^2/2
326         (-1+ratio.^2).*ratio/6
327         (-1+ratio.^2).*ratio.^2/24
328         (4-5*ratio.^2+ratio.^4).*ratio/120
329         (4-5*ratio.^2+ratio.^4).*ratio.^2/720
330         (-36+49*ratio.^2-14*ratio.^4+ratio.^6).*ratio/5040
331         (-36+49*ratio.^2-14*ratio.^4+ratio.^6).*ratio.^2/40320 ];
332 end
333 patches.Cwtsr = patches.Cwtsr(1:ordCC,:);
334 % should avoid this next implicit auto-replication
335 patches.Cwtsl = (-1).^((1:ordCC)'-patches.alt).*patches.Cwtsr;

```

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

```

344 X = linspace(Xlim(1),Xlim(2),nPatch(1)+1);
345 X = X(1:nPatch(1))+diff(X)/2;
346 DX = X(2)-X(1);
347 Y = linspace(Xlim(3),Xlim(4),nPatch(2)+1);
348 Y = Y(1:nPatch(2))+diff(Y)/2;
349 DY = Y(2)-Y(1);

```

Construct the microscale in each patch, assuming Dirichlet patch edges, and a half-patch length of $\text{ratio}(1) \cdot \text{DX}$ and $\text{ratio}(2) \cdot \text{DY}$.

```

357 nSubP = nSubP(:)'; % force to be row vector
358 if mod(nSubP,2)==[0 0], error('configPatches2: nSubP must be odd'), end
359 i0 = (nSubP(1)+1)/2;
360 dx = ratio(1)*DX/(i0-1);
361 patches.x = bsxfun(@plus,dx*(-i0+1:i0-1)',X); % micro-grid
362 i0 = (nSubP(2)+1)/2;
363 dy = ratio(2)*DY/(i0-1);

```

```

364 patches.y = bsxfun(@plus,dy*(-i0+1:i0-1)',Y); % micro-grid
365 end% function

```

Fin.

3.10 patchSmooth2(): interface to time integrators

Section contents

3.10.1 Introduction 74

3.10.1 Introduction

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`.

```

25 function dudt = patchSmooth2(t,u)
26 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 microscale.
 - `.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 microscale.

Output

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

Reshape the fields `u` as a 4/5D-array, and sets the edge values from macroscale interpolation of centre-patch values. [Section 3.11](#) describes `patchEdgeInt2()`.

```
84 u = patchEdgeInt2(u);
```

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

```
94 dudt = patches.fun(t,u,patches.x,patches.y);
95 dudt([1 end],:,:,:) = 0;
96 dudt(:,[1 end],:,:) = 0;
97 dudt = reshape(dudt,[],1);
```

Fin.

3.11 `patchEdgeInt2()`: sets 2D patch edge values from 2D macroscale interpolation

Section contents

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`.

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

Input

- `u` is a vector of length `nx · ny · Nx · Ny · nVars` where there are `nVars` field values at each of the points in the `nx × ny × Nx × Ny` grid on the `Nx × Ny` array of patches.
- `patches` a struct set by `configPatches2()` which includes the following information.
 - `.x` is `nx × 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 microscale.
 - `.y` is similarly `ny × 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 microscale.
 - `.ordCC` is order of interpolation, currently only `{0}`.

– .Cwtsr and .Cwtsl—not yet used

Output

- u is $n_x \times n_y \times N_x \times N_y \times n_{\text{Vars}}$ array of the fields with edge values set by interpolation.

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

```

76 [ny,Ny] = size(patches.y);
77 [nx,Nx] = size(patches.x);
78 nVars = round(numel(u)/numel(patches.x)/numel(patches.y));
79 if numel(u) ~= nx*ny*Nx*Ny*nVars
80     nSubP=[nx ny], nPatch=[Nx Ny], nVars=nVars, sizeu=size(u)
81 end
82 u = reshape(u,[nx ny Nx Ny nVars]);

```

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

```

92 dx = patches.x(3,1)-patches.x(2,1);
93 DX = patches.x(2,2)-patches.x(2,1);
94 rx = dx*(nx-1)/2/DX;
95 dy = patches.y(3,1)-patches.y(2,1);
96 DY = patches.y(2,2)-patches.y(2,1);
97 ry = dy*(ny-1)/2/DY;

```

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

```

108 %i=1:Nx; ip=mod(i,Nx)+1; im=mod(j-2,Nx)+1;
109 %j=1:Ny; jp=mod(j,Ny)+1; jm=mod(j-2,Ny)+1;

```

The centre of each patch (as n_x and n_y are odd) is at

```

116 i0 = round((nx+1)/2);
117 j0 = round((ny+1)/2);

```

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

```

127 if patches.ordCC>0 % then non-spectral interpolation
128 error('non-spectral interpolation not yet implemented')
129 dmu=nan(patches.ordCC,nPatch,nVars);
130 % if patches.alt % use only odd numbered neighbours
131 % dmu(1, :, :)=(u(i0,jp,:)+u(i0,jm,:))/2; % \mu
132 % dmu(2, :, :) = u(i0,jp,:)-u(i0,jm,:); % \delta

```

```

133 %   jp=jp(jp); jm=jm(jm); % increase shifts to \pm 2
134 % else % standard
135     dmu(1, :, :) = (u(i0, jp, :) - u(i0, jm, :)) / 2; % \mu \delta
136     dmu(2, :, :) = (u(i0, jp, :) - 2*u(i0, j, :) + u(i0, jm, :)); % \delta^2
137 % end% if odd/even

```

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

```

145     for k=3:patches.ordCC
146         dmu(k, :, :) = dmu(k-2, jp, :) - 2*dmu(k-2, j, :) + dmu(k-2, jm, :);
147     end

```

Interpolate macro-values to be Dirichlet edge values for each patch ([Roberts & Kevrekidis 2007](#)), using weights computed in `configPatches2()`. Here interpolate to specified order.

```

155     u(nSubP, j, :) = u(i0, j, :)*(1-patches.alt) ...
156         + sum(bsxfun(@times, patches.Cwtsr, dmu));
157     u(1, j, :) = u(i0, j, :)*(1-patches.alt) ...
158         + sum(bsxfun(@times, patches.Cwtsl, dmu));

```

Case of spectral interpolation Assumes the domain is macro-periodic. We interpolate in terms of the patch index j , say, not directly in space. As the macroscale fields are N -periodic in the patch index j , the macroscale Fourier transform writes the centre-patch values as $U_j = \sum_k C_k e^{ik2\pi j/N}$. Then the edge-patch values $U_{j\pm r} = \sum_k C_k e^{ik2\pi/N(j\pm r)} = \sum_k C'_k e^{ik2\pi j/N}$ where $C'_k = C_k e^{ikr2\pi/N}$. For N patches we resolve ‘wavenumbers’ $|k| < N/2$, so set row vector $\mathbf{ks} = k2\pi/N$ for ‘wavenumbers’ $k = (0, 1, \dots, k_{\max}, -k_{\max}, \dots, -1)$ for odd N , and $k = (0, 1, \dots, k_{\max}, \pm(k_{\max} + 1) - k_{\max}, \dots, -1)$ for even N .

```

180 else% spectral interpolation

```

Deal with staggered grid by doubling the number of fields and halving the number of patches (`configPatches2` tests there are an even number of patches). Then the patch-ratio is effectively halved. The patch edges are near the middle of the gaps and swapped.

```

190 % if patches.alt % transform by doubling the number of fields
191 % error('staggered grid not yet implemented')
192 % v=nan(size(u)); % currently to restore the shape of u
193 % u=cat(3,u(:,1:2:nPatch,:),u(:,2:2:nPatch,:));
194 % altShift=reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
195 % iV=[nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
196 % r=r/2; % ratio effectively halved
197 % nPatch=nPatch/2; % halve the number of patches
198 % nVars=nVars*2; % double the number of fields
199 % else % the values for standard spectral
200     altShift = 0;
201     iV = 1:nVars;
202 % end

```

Now set wavenumbers in the two directions. In the case of even N these compute the +-case for the highest wavenumber zig-zag mode, $k = (0, 1, \dots, k_{\max}, +(k_{\max} + 1) - k_{\max}, \dots, -1)$.

```

211     kMax = floor((Nx-1)/2);
212     krx = rx*2*pi/Nx*(mod((0:Nx-1)+kMax,Nx)-kMax);
213     kMay = floor((Ny-1)/2);
214     kry = ry*2*pi/Ny*(mod((0:Ny-1)+kMay,Ny)-kMay);

```

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

```

221     if imag(u(i0,j0,:,:))==0, uclean = @(u) real(u);
222                     else uclean = @(u) u; end

```

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

```

231     Ck = fft2(squeeze(u(i0,j0,:,:)));

```

The inverse Fourier transform gives the edge values via a shift a fraction \mathbf{rx}/\mathbf{ry} to the next macroscale grid point. Initially preallocate storage for all the IFFTs that we need to cater for the zig-zag modes when there are an even number of patches in the directions.

```

242     nFTx = 2-mod(Nx,2);
243     nFTy = 2-mod(Ny,2);
244     unj = nan(1,ny,Nx,Ny,nVars,nFTx*nFTy);
245     u1j = nan(1,ny,Nx,Ny,nVars,nFTx*nFTy);
246     uin = nan(nx,1,Nx,Ny,nVars,nFTx*nFTy);
247     ui1 = nan(nx,1,Nx,Ny,nVars,nFTx*nFTy);

```

Loop over the required IFFTs.

```

253     iFT = 0;
254     for iFTx = 1:nFTx
255         for iFTy = 1:nFTy
256             iFT = iFT+1;

```

First interpolate onto x -limits of the patches. (It may be more efficient to product exponentials of vectors, instead of exponential of array—only for $N > 100$. Can this be vectorised further??)

```

265     for jj = 1:ny
266         ks = (jj-j0)*2/(ny-1)*kry; % fraction of kry along the edge
267         unj(1,jj,:,: ,iV,iFT) = ifft2( bsxfun(@times,Ck ...
268             ,exp(1i*bsxfun(@plus,altShift+krx',ks))));
269         u1j(1,jj,:,: ,iV,iFT) = ifft2( bsxfun(@times,Ck ...
270             ,exp(1i*bsxfun(@plus,altShift-krx',ks))));
271     end

```

Second interpolate onto y -limits of the patches.

```

277     for i = 1:nx
278         ks = (i-i0)*2/(nx-1)*krx; % fraction of krx along the edge

```



```

279     uin(i,1,:,:iV,iFT) = ifft2( bsxfun(@times,Ck ...
280         ,exp(1i*bsxfun(@plus,ks',altShift+kry))));
281     ui1(i,1,:,:iV,iFT) = ifft2( bsxfun(@times,Ck ...
282         ,exp(1i*bsxfun(@plus,ks',altShift-kry))));
283 end

```

When either direction have even number of patches then swap the zig-zag wavenumber to the conjugate.

```

290 if nFTy==2, kry(Ny/2+1) = -kry(Ny/2+1); end
291 end% iFTy-loop
292 if nFTx==2, krx(Nx/2+1) = -krx(Nx/2+1); end
293 end% iFTx-loop

```

Put edge-values into the u -array, using `mean()` to treat a zig-zag mode as cosine. Enforce reality when appropriate via `uclean()`.

```

301 if numel(size(unj))>5
302     u(end,:,:iV) = uclean( mean(unj,6) );
303     u( 1 ,,:,:iV) = uclean( mean(u1j,6) );
304     u(:,end,:,:iV) = uclean( mean(uin,6) );
305     u(:, 1 ,,:,:iV) = uclean( mean(ui1,6) );
306 else
307     u(end,:,:iV) = uclean( unj );
308     u( 1 ,,:,:iV) = uclean( u1j );
309     u(:,end,:,:iV) = uclean( uin );
310     u(:, 1 ,,:,:iV) = uclean( ui1 );
311 end

```

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

```

318 %if patches.alt
319 % nVars=nVars/2; nPatch=2*nPatch;
320 % v(:,1:2:nPatch,:)=u(:,1:nVars);
321 % v(:,2:2:nPatch,:)=u(:,nVars+1:2*nVars);
322 % u=v;
323 %end
324 end% if spectral
325 end% function patchEdgeInt2

```

Fin, returning the 4/5D array of field values with interpolated edges.

3.12 wave2D: example of a wave on patches in 2D

Section contents

3.12.1 Check on the linear stability of the wave PDE	80
3.12.2 Execute a simulation	81
3.12.3 wavePDE(): Example of simple wave PDE inside patches	83

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 (left-right arrows denote 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 2π -periodic domain, with 9×9 patches, spectral interpolation (0) couples the patches, each patch of half-size ratio 0.25, and with 5×5 points within each patch.

```

34 clear all, close all
35 global patches
36 nSubP = 5;
37 nPatch = 9;
38 configPatches2(@wavePDE,[-pi pi], nan, nPatch, 0, 0.25, nSubP);

```

3.12.1 Check on the linear stability of the wave PDE

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

```

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

```

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

```

61 small = 1;
62 jac = nan(length(i));
63 sizeJacobian = size(jac)
64 for j = 1:length(i)
65     uv = uv0(:);
66     uv(i(j)) = uv(i(j))+small;
67     tmp = patchSmooth2(0,uv)/small;
68     jac(:,j) = tmp(i);
69 end

```

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

```

77 evals = eig(jac);
78 nEvals = length(evals)
79 [~,k] = sort(-abs(real(evals)));
80 evalsWithBiggestRealPart = evals(k(1:10))
81 if abs(real(evals(k(1))))>1e-4

```

```

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

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

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

```

3.12.2 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: $n_x \times n_y \times N_x \times N_y$.

```

108 x = reshape(patches.x,nSubP,1,[],1);
109 y = reshape(patches.y,1,nSubP,1,[]);
110 u0 = exp(-x.^2-y.^2);
111 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 [Figure 3.14](#) 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 $n_x \cdot N_x \times n_y \cdot N_y$.

```

123 x = patches.x; y = patches.y;
124 x([1 end],:) = nan; y([1 end],:) = nan;
125 u = reshape(permute(u0,[1 3 2 4]), [numel(x) numel(y)]);
126 usurf = surf(x(:),y(:),u');
127 axis([-3 3 -3 3 -0.5 1]), view(60,40)
128 xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
129 legend('time = 0','Location','north')
130 drawnow
131 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
132 %print('-depsc','wave2Dic')

```

Integrate in time using standard functions.

```

145 disp('Wait while we simulate u_t=v, v_t=u_xx+u_yy')
146 if ~exist('OCTAVE_VERSION','builtin')
147     [ts,uvs] = ode15s( @patchSmooth2,[0 2],[u0(:);v0(:)]);
148 else % octave version is slower
149     [ts,uvs] = ode0cts(@patchSmooth2,[0 1],[u0(:);v0(:)]);
150 end

```

Animate the computed simulation to end with [Figure 3.17](#). Because of the very small time-steps, subsample to plot at most 100 times.

```

158 di = ceil(length(ts)/100);
159 for i = [1:di:length(ts)-1 length(ts)]
160     uv = patchEdgeInt2(uvs(i,:));

```

Figure 3.16: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to the simple wave PDE: Figure 3.17 plots the computed field at time $t = 2$.

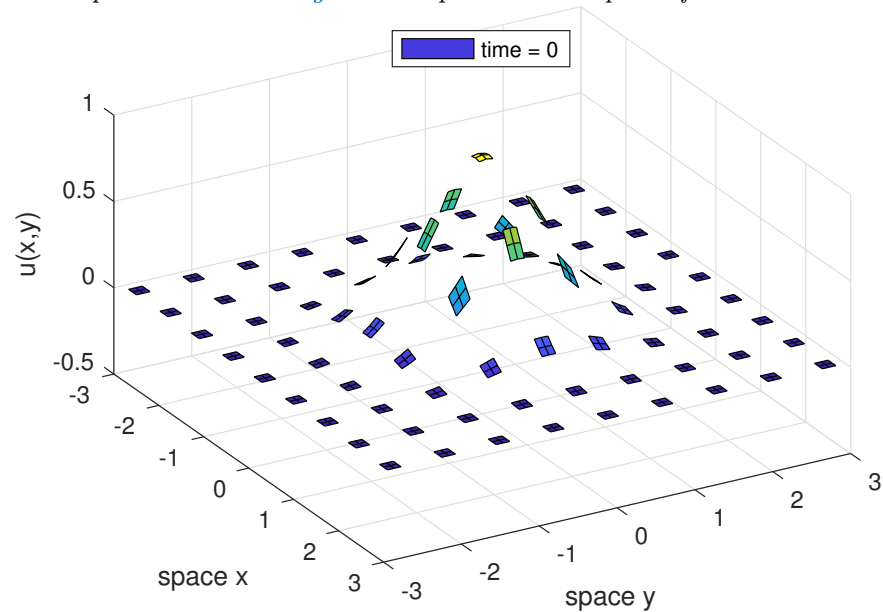
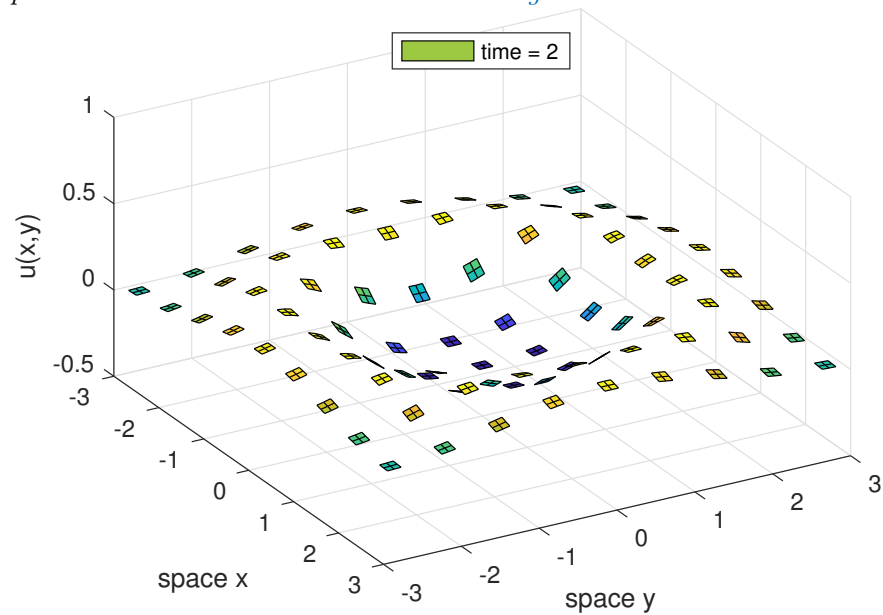


Figure 3.17: field $u(x, y, t)$ at time $t = 2$ of the patch scheme applied to the simple wave PDE with initial condition in Figure 3.16.



```

161     uv = reshape(permute(uv,[1 3 2 4 5]), [numel(x) numel(y) 2]);
162     set(usurf,'ZData', uv(:,:,1));
163     legend(['time = ' num2str(ts(i),2)])
164     pause(0.1)
165 end
166 %print('-depsc', ['wave2Dt' num2str(ts(end))])

```

3.12.3 wavePDE(): 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 $\dot{v}_{ijkl} = \frac{1}{\delta x^2}(u_{i+1,j,k,l} - 2u_{i,j,k,l} + u_{i-1,j,k,l}) + \frac{1}{\delta y^2}(u_{i,j+1,k,l} - 2u_{i,j,k,l} + u_{i,j-1,k,l})$.

```

14 function uvt = wavePDE(t,uv,x,y)
15     if ceil(t+1e-7)-t<2e-2, simTime = t, end %track progress
16     dx = diff(x(1:2)); dy = diff(y(1:2)); % microscale spacing
17     i = 2:size(uv,1)-1; j = 2:size(uv,2)-1; % interior patch-points
18     uvt = nan(size(uv)); % preallocate storage
19     uvt(i,j,:,:1) = uv(i,j,:,:2);
20     uvt(i,j,:,:2) = diff(uv(:,j,:,:1),2,1)/dx^2 ...
21                     +diff(uv(i,:,:,:),2,2)/dy^2;
22 end

```

3.13 To do

- Testing needs to be quantitative.
- more than two space dimensions??
- Heterogeneous microscale via averaging regions—but I suspect should be separated from simple homogenisation
- Parallel processing versions.
- ??
- Adapt to maps in micro-time? Surely easy, just an example.

3.14 Miscellaneous tests

3.14.1 patchEdgeInt1test: test the spectral interpolation

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

```

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.

```

101 fprintf('Single field-pair test.\n')
102 for k=-kMax:kMax
103     U0=nan(nSubP,nPatch);
104     U0(hPts)=rand*exp(+1i*k*patches.x(hPts)*2*pi/Len);
105     U0(uPts)=rand*exp(-1i*k*patches.x(uPts)*2*pi/Len);
106     Ui=patchEdgeInt1(U0(:));
107     normError=norm(Ui-U0);
108     if abs(normError)>5e-14
109         normError=normError
110         error(['failed single sys interpolation k=' num2str(k)])
111     end
112 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.

```

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

```

Finish If no error messages, then all OK.

```

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

```

3.14.2 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

```

Appendix 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 folders in the repository, say `Dir`.
- Create a LaTeX file `Dir/funs.tex`: establish as one LaTeX chapter that `\input{Dir/*.m}`s the files of the functions in the class, example scripts of use, and possibly test scripts, [Table A.1](#).
- Each such `Dir/funs.tex` file is to be included from the main LaTeX file `Doc/docBody.tex` so that people can most easily work on one chapter at a time:
 - put `\include{funs}` into `Doc/docBody.tex`;
 - to include, create a ‘link’ file `Doc/funs.tex` whose only active content is the command `\input{../Dir/funs.tex}`;
 - in `Doc/docBody.tex` modify the `\graphicspath` command to include `{../Dir/Figs}`.
- Each toolbox function is documented as a separate section, within its chapter, with tests and examples as separate sections.
- Each function-section and test-section 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 A.2](#) gives the template for the `Dir/*.m` function-sections. The format for a example/test-section is similar.
- Any figures from examples should be generated and then saved for later inclusion with the following (which finally works properly for MATLAB 2017+)

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

If it is a suitable replacement for an existing graphic, then move it into the `Dir/Figs` folder. Include such a graphic into the LaTeX document with (do *not* postfix with `.eps` or `.pdf`)

```
\includegraphics[scale=0.9]{filename}
```

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

```

1 % input *.m files for ... Author, date
2 %!TEX root = ../Doc/eqnFreeDevMan.tex
3 \chapter{...}
4 \label{ch:...}
5 \localtableofcontents
6 \section{Introduction}
7 introduction...
8 \input{../Dir/fun.m} % prefix associated files with 'fun'
9 \input{../Dir/funExample.m}
10 ...
11 \begin{devMan}
12 \section{To do}
13 ...
14 \section{Miscellaneous tests}
15 \input{../Dir/funTest.m}
16 ...
17 \end{devMan}

```

- In figures and other graphics, do *not* resize/scale fixed width constructs: instead use `\linewidth` to configure large-scale layout, `em` for small-widths, and `ex` for small-heights.
- For every function, generally include at the start of the function a simple example of its use. The example is only to be executed when the function is invoked with no input arguments (`if nargin==0`).

When appropriate, if a function is invoked with no output arguments (`if nargout==0`), then draw some reasonable graph of the results.

- In all MATLAB/Octave code, prefer camel case for variable names (not underscores).
- When a function is ‘finalised’, wrap (most) of the lines to be no more than 60 characters so that readers looking at the source can read the plain text reasonably.
- In the documentation (e.g., [Higham 1998](#), Ch. 4): write actively, not passively (e.g., avoid “-tion” words, and “is/are verbed” phrases); avoid wishy-washy “can”; use the present tense; cross-reference precisely; avoid useless padding such as “note that”; and so on.

Table A.2: template for a function-section Dir/*.m file.

```

1  % Short explanation for users typing "help fun"
2  % Author, date
3  %!TEX root = ../Doc/eqnFreeDevMan.tex
4  %{
5  \section{\texttt{...}: ...}
6  \label{sec:...}
7  \localtableofcontents
8  \subsection{Introduction}
9  Overview LaTeX explanation.
10 \begin{matlab}
11 %}
12 function ...
13 %{
14 \end{matlab}
15 \paragraph{Input} ...
16 \paragraph{Output} ...
17 \begin{devMan}
18 Repeated as desired:
19 LaTeX between end-matlab and begin-matlab
20 \begin{matlab}
21 %}
22 Matlab code between %} and %{
23 %{
24 \end{matlab}
25 Concluding LaTeX before following final lines.
26 \end{devMan}
27 %}

```

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

Chapter contents

B.1	Macroscale grid	90
B.2	Macroscale field variables	90
B.3	Boundary and coupling conditions	91
B.4	Mesotime communication	91
B.5	Projective integration	91
B.6	Lift to many internal modes	92
B.7	Macroscale closure	92
B.8	Exascale fault tolerance	92
B.9	Link to established packages	93

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 $\vec{U}(t) \in \mathbb{R}^{d_{\vec{U}}}$ 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, $d_{\vec{J}}$, 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 $d_{\vec{u}} - d_{\vec{J}}$ data at each position in the patch: this is lifting. The first methodology is to first guess, then run repeated short bursts with reinitialisation, until the simulation reaches a slow manifold. Then run the real simulation.

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

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

B.7 Macroscale closure

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

Bibliography

- Bunder, J. E., Roberts, A. J. & Kevrekidis, I. G. (2017), ‘Good coupling for the multiscale patch scheme on systems with microscale heterogeneity’, *J. Computational Physics* **337**, 154–174.
- Bunder, J., Roberts, A. J. & Kevrekidis, I. G. (2016), ‘Accuracy of patch dynamics with mesoscale temporal coupling for efficient massively parallel simulations’, *SIAM Journal on Scientific Computing* **38**(4), C335–C371.
- Calderon, C. P. (2007), ‘Local diffusion models for stochastic reacting systems: estimation issues in equation-free numerics’, *Molecular Simulation* **33**(9–10), 713–731.
- Cao, M. & Roberts, A. J. (2012), Modelling 3d turbulent floods based upon the smagorinski large eddy closure, in P. A. Brandner & B. W. Pearce, eds, ‘18th Australasian Fluid Mechanics Conference’.
<http://people.eng.unimelb.edu.au/imarusic/proceedings/18/70%20-%20Cao.pdf>
- Cao, M. & Roberts, A. J. (2013), Multiscale modelling couples patches of wave-like simulations, in S. McCue, T. Moroney, D. Mallet & J. Bunder, eds, ‘Proceedings of the 16th Biennial Computational Techniques and Applications Conference, CTAC-2012’, Vol. 54 of *ANZIAM J.*, pp. C153–C170.
- Cao, M. & Roberts, A. J. (2016a), ‘Modelling suspended sediment in environmental turbulent fluids’, *J. Engrg. Maths* **98**(1), 187–204.
- Cao, M. & Roberts, A. J. (2016b), ‘Multiscale modelling couples patches of nonlinear wave-like simulations’, *IMA J. Applied Maths.* **81**(2), 228–254.
- Gear, C. W., Kaper, T. J., Kevrekidis, I. G. & Zagaris, A. (2005a), ‘Projecting to a slow manifold: singularly perturbed systems and legacy codes’, *SIAM J. Applied Dynamical Systems* **4**(3), 711–732.
<http://www.siam.org/journals/siads/4-3/60829.html>
- Gear, C. W., Kaper, T. J., Kevrekidis, I. G. & Zagaris, A. (2005b), ‘Projecting to a slow manifold: Singularly perturbed systems and legacy codes’, *SIAM Journal on Applied Dynamical Systems* **4**(3), 711–732.
- Gear, C. W. & Kevrekidis, I. G. (2003a), ‘Computing in the past with forward integration’, *Phys. Lett. A* **321**, 335–343.
- Gear, C. W. & Kevrekidis, I. G. (2003b), ‘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. (2003c), ‘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.
- Higham, N. J. (1998), *Handbook of writing for the mathematical sciences*, 2nd edition edn, SIAM.
- 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>
- Maclean, J. & Gottwald, G. A. (2015), ‘On convergence of higher order schemes for the projective integration method for stiff ordinary differential equations’, *Journal of Computational and Applied Mathematics* **288**, 44–69.
<http://www.sciencedirect.com/science/article/pii/S0377042715002149>
- 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.

- Roberts, A. J., MacKenzie, T. & Bunder, J. (2014), ‘A dynamical systems approach to simulating macroscale spatial dynamics in multiple dimensions’, *J. Engineering Mathematics* **86**(1), 175–207.
<http://arxiv.org/abs/1103.1187>
- 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.
- Sieber, J., Marschler, C. & Starke, J. (2018), ‘Convergence of Equation-Free Methods in the Case of Finite Time Scale Separation with Application to Deterministic and Stochastic Systems’, *SIAM Journal on Applied Dynamical Systems* **17**(4), 2574–2614.
- 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.