Equation-Free function toolbox for Matlab/Octave: Full Developers Manual

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

February 27, 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. The methodology bypasses the derivation of macroscopic evolution equations by using only short bursts of microscale simulations which are often the best available description of a system (Kevrekidis & Samaey 2009, Kevrekidis et al. 2004, 2003, e.g.). This suite of functions should empower users to start implementing such methods—but so far we have only just started.

Contents

1 Introduction

Subsection contents

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

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

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

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

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

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

2 Quick start

Chapter contents

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

2.1 Cheat sheet: Projective Integration

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

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

The cheat sheet consists of two flow charts. Figure 2.1 overviews constructing a PI simulation. Figure 2.2 roughly guides which of the top-level PI functions should be used.

2.2 Cheat sheet: constructing patches

This section pertains to the Patch approach, Chapter 4, to solving PDEs, lattice systems, or agent/particle microscale simulators.

The Patch toolbox requires that one configure patches, couple the patches and interface the coupled patches with a time integrator. Figure 2.3 overviews the chief functions involved and their interactions.

Figure 2.1: these figures appear confusing to a newbie???? and we must *not* resize fixed width constructs. Use linewidth for large-scale layout scaling, em for small-widths, and ex for small-heights.

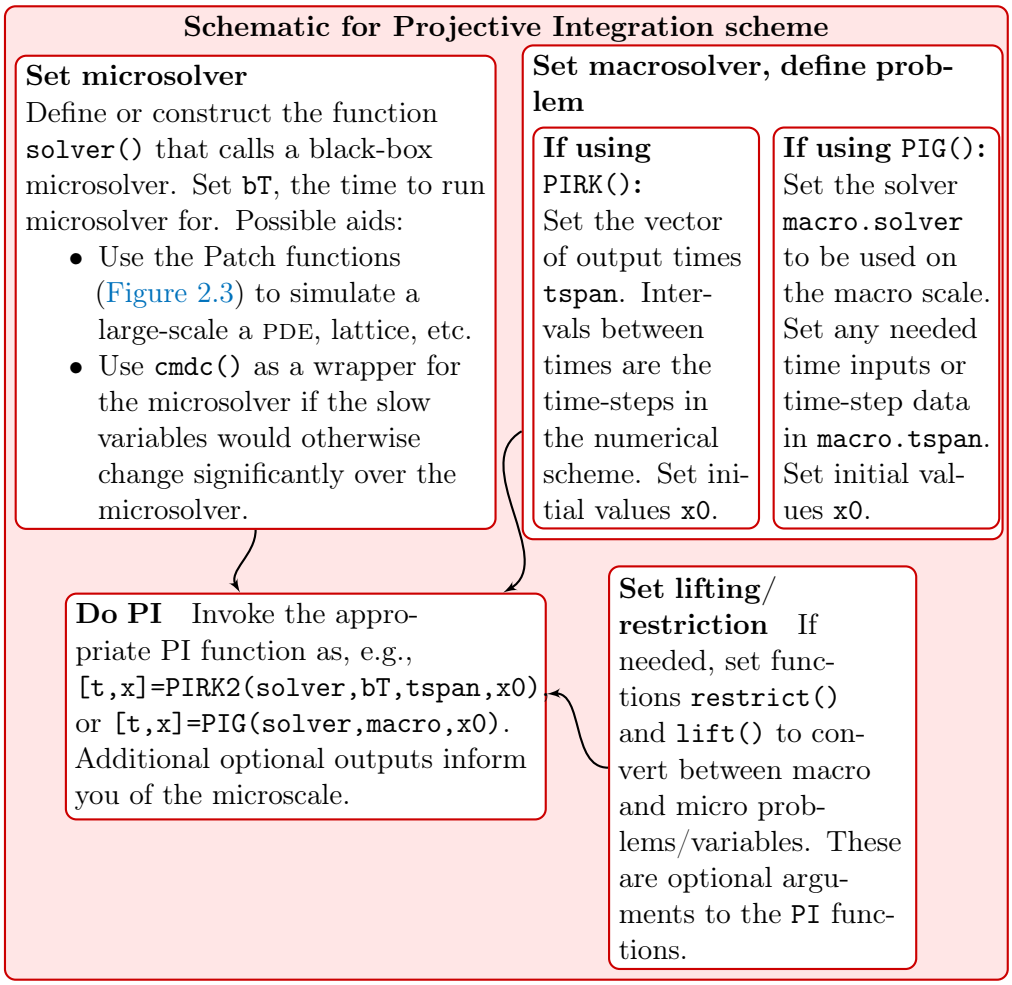


Figure 2.2

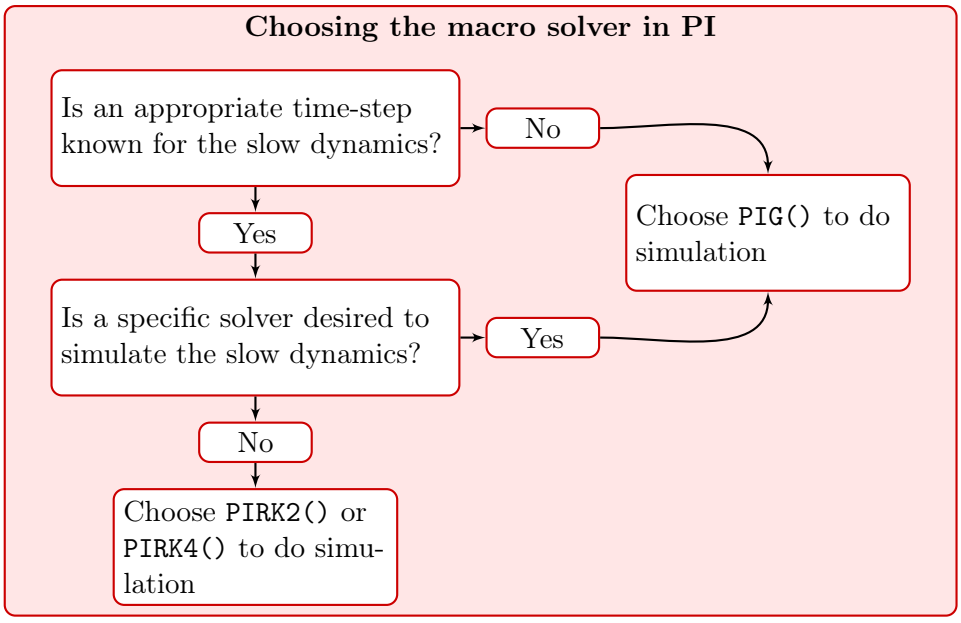
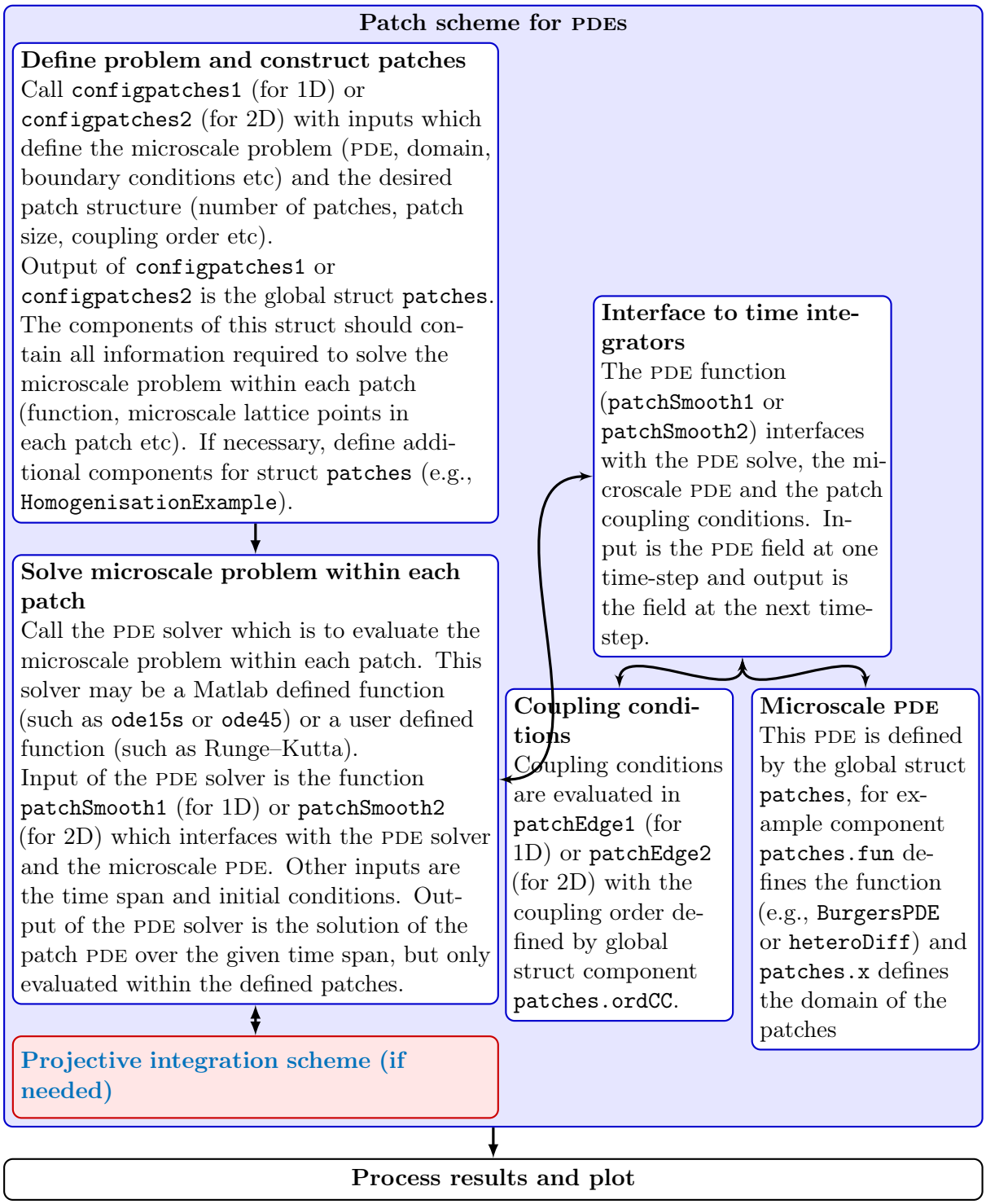

Choosing the macro solver in PI
Is an appropriate time-step known for the slow dynamics?
No
Choose PIG() to do simulation
Yes
Is a specific solver desired to simulate the slow dynamics?
Yes
No
Choose PIRK2() or PIRK4() to do simulation

Figure 2.3


Patch scheme for PDEs
Define problem and construct patches
Call configpatches1 (for 1D) or configpatches2 (for 2D) with inputs which define the microscale problem (PDE, domain, boundary conditions etc) and the desired patch structure (number of patches, patch size, coupling order etc).
Output of configpatches1 or configpatches2 is the global struct patches. The components of this struct should contain all information required to solve the microscale problem within each patch (function, microscale lattice points in each patch etc). If necessary, define additional components for struct patches (e.g., HomogenisationExample).
Solve microscale problem within each patch
Call the PDE solver which is to evaluate the microscale problem within each patch. This solver may be a Matlab defined function (such as ode15s or ode45) or a user defined function (such as Runge–Kutta).
Input of the PDE solver is the function patchSmooth1 (for 1D) or patchSmooth2 (for 2D) which interfaces with the PDE solver and the microscale PDE. Other inputs are the time span and initial conditions. Output of the PDE solver is the solution of the patch PDE over the given time span, but only evaluated within the defined patches.
Projective integration scheme (if needed)
Interface to time integrators
The PDE function (patchSmooth1 or patchSmooth2) interfaces with the PDE solve, the microscale PDE and the patch coupling conditions. Input is the PDE field at one time-step and output is the field at the next time-step.
Coupling conditions
Coupling conditions are evaluated in patchEdge1 (for 1D) or patchEdge2 (for 2D) with the coupling order defined by global struct component patches.ordCC.
Microscale PDE
This PDE is defined by the global struct patches, for example component patches.fun defines the function (e.g., BurgersPDE or heteroDiff) and patches.x defines the domain of the patches
Process results and plot

3 Projective integration of deterministic ODEs

Subsection contents

This section provides some good projective integration functions (Gear & Kevrekidis 2003*a*,*b*, Givon et al. 2006, Sieber et al. 2018, e.g.). The goal is to enable computationally expensive dynamic simulations to be run over long time scales.

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

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

Main functions

- Projective Integration by second or fourth-order Runge–Kutta, `PIRK2()` and `PIRK4()` respectively. These schemes are suitable for precise simulation of the slow dynamics, provided the time period spanned by an application of the microsolver is not too large.
- Projective Integration with a General solver, `PIG()`. This function enables a Projective Integration implementation of any solver with macroscale time-steps. It does not matter whether the solver is a standard Matlab or Octave algorithm, or one supplied by the user. As explored in later examples, `PIG()` should only be used in very stiff systems.
- 'Constraint-defined manifold computing', `cdmc()`. This helper function, based on the method introduced in **?**, iteratively applies the microsolver and projects the output backwards in time. The result is to constrain the fast variables close to the slow manifold, without advancing the current time by the duration of an application of the microsolver. This function can be used to reduce errors related to the simulation length of the microsolver in either the `PIRK` or `PIG` functions. In particular, it enables `PIG()` to be used on problems that are not particularly stiff.

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

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

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

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

```
18 function [x, tms, xms, rm, svf] = PIRK2(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 3.1.1 as a basic template of how to use.

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

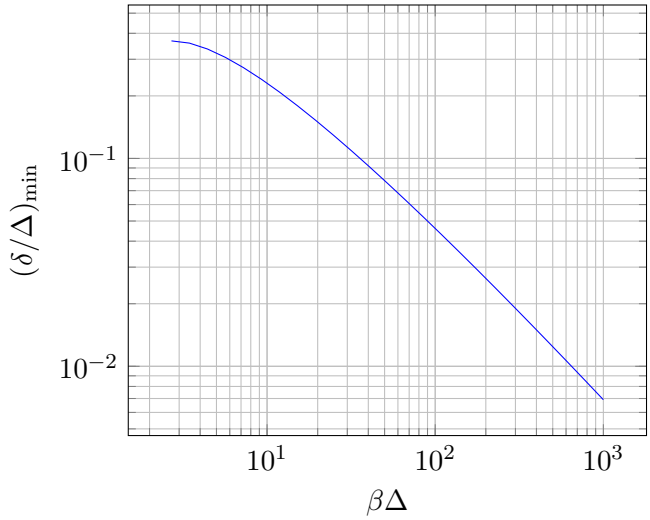
  `[tOut, xOut] = microBurst(tStart, xStart, bT)`

  - 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 `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`: they are included in the simulation and output, and often represent boundaries in space 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 computed burst.

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

Figure 3.1: 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 given fast rate β.



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

1. a macroscale time-step, $\Delta =$ `diff(tSpan)`, such that $\alpha\Delta \approx \sqrt{6\varepsilon}$, and
2. a microscale burst length, $\delta =$ `bT` $\gtrsim \frac{1}{\beta}\log(\beta\Delta)$ (see Figure 3.1).

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 two to four optional outputs of the microscale bursts.

- `tms`, optional, is an L dimensional column vector containing microscale times of burst simulations, each burst separated by `NaN`;

- `xms`, optional, is an $L \times n$ array of the corresponding microscale states—this data is an accurate simulation of the state and may help visualise more details of the solution.

- `rm`, optional, a struct containing the 'remaining' applications of the 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 in general resemble the true 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.

3.1.1 If no arguments, then execute an example

```
171 if nargin==0
```

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

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

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

```
191 epsilon = 0.05
192 ts = 0:6
193 bT = epsilon*log((ts(2)-ts(1))/epsilon)
194 [x,tms,xms] = PIRK2(@MMburst, ts, [1;0], bT);
195 figure, plot(ts,x,'o:',tms,xms)
196 title('Projective integration of Michaelis--Menten enzyme kinetics')
197 xlabel('time t'), legend('x(t)','y(t)')
```

Upon finishing execution of the example, exit this function.

```
203 return
204 end%if no arguments
```

Example function code for a burst of ODEs Integrate a burst of length `bT` of the ODEs for the Michaelis–Menten enzyme kinetics at parameter 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 `ode23` to integrate in time.

```
218 function [ts, xs] = MMburst(ti, xi, bT)
219     dMMdt = @(t,x) [ -x(1)+(x(1)+0.5)*x(2)
220           1/epsilon*( x(1)-(x(1)+1)*x(2) ) ];
221     [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
222 end
```

3.1.2 The projective integration code

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

```
239 nT=length(tSpan);
240 x=nan(nT,length(x0));
```

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

```
248 nArgs=nargout();
249 saveMicro = (nArgs>1);
250 saveFullMicro = (nArgs>3);
251 saveSvf = (nArgs>4);
```

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

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

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

```
273 tSpan(1) = relax_t(end);
274 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.

```
284 if saveMicro
285     tms = cell(nT,1);
286     xms = cell(nT,1);
287     tms{1} = reshape(relax_t,[],1);
288     xms{1} = relax_x0;
289     if saveFullMicro
290         rm.t = cell(nT,1);
291         rm.x = cell(nT,1);
292         if saveSvf
293             svf.t = nan(2*nT-2,1);
294             svf.dx = nan(2*nT-2,length(x0));
295         end
296     end
297 end
```

Loop over the macroscale time-steps

```
for jT = 2:nT
    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.

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

Run the first application of the 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.

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

Check for round-off error.

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

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

```
    Dt = tSpan(jT)-t1(end);
    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).

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

Check for round-off error.

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

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

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

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

```
389     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 `NaN`s.

```
399     if saveMicro
400         tms{jT} = [reshape(t1,[],1); nan];
401         xms{jT} = [xm1; nan(1,size(xm1,2))];
```

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

```
409         if saveFullMicro
410             rm.t{jT} = [reshape(t2,[],1); nan];
411             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.

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

Terminate the main loop:

```
431 end
```

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

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

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

```
448 if saveMicro
449     tms = cell2mat(tms);
450     xms = cell2mat(xms);
451     if saveFullMicro
452         rm.t = cell2mat(rm.t);
453         rm.x = cell2mat(rm.x);
454     end
455 end
```

3.1.3 If no output specified, then plot simulation

```
463 if nArgs==0
464     figure, plot(tSpan,x,'o:')
```

```
465     title('Projective Simulation with PIRK2')
466 end
```

This concludes `PIRK2()`.

```
473 end
```

3.2 `PIG()`: Projective Integration via a General macroscale integrator

This is an approximate Projective Integration scheme when the macroscale integrator is any coded scheme. 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 simulation.

By default, `PIG()` uses 'constraint-defined manifold computing' for the microscale simulations. This algorithm, developed in Gear et al. (2005), uses a backwards projection so that the simulation time is unchanged after running the microscale simulator. The implementation is `cdmc()`, described in Section 3.3.3.

```
23 function [t,x,tms,xms,svf] = PIG(macroInt,microBurst,tSpan,x0,...
24                                  restrict,lift,cdmcFlag)
```

Inputs:

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

  ```
  [ts,xs] = 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 times at which a macroscale value is to be returned; and `x0` are the initial values of $\vec{x}$ at time `tSpan(1)`. Then the ith *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 know how long a burst it is to use. Usage

  ```
  [tbs,xbs] = microBurst(tb0,xb0)
  ```

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

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

- `tSpan`, a vector of times at which the user requests output, of which the first element is always the initial time. If `macroInt` can adaptively select time steps (e.g., `ode45`), then `tSpan` can consist of an initial and final time only.
- `x0`, the N-vector of initial values at the initial time `tSpan(1)`.

Optional Inputs: `PIG()` allows for 0, 2 or 3 additional inputs after `x0`. If there are distinct microscale and macroscale states and the aim is to do Projective Integration on the macroscale only, then functions must be provided to convert between them. Usage `PIG(...,restrict,lift)`

- `restrict()`, a function that takes an input n-dimensional microscale state and returns an N-dimensional macroscale state.
- `lift()`, a function that converts an input N-dimensional macroscale state to an n-dimensional microscale state.

Either both, or neither, of `restrict()` and `lift()` must be defined. If neither are used, then `N=n` in the following descriptions of the output.
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()`. For clarity it is suggested to use a string, e.g. `cdmcFlag = 'flag cdmc off'`.

If the `cdmcFlag` should be set without using a `restrict()` or `lift()` function, use empty arguments `[]` to the restrict and lift inputs.

Output Between 0 and 5 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 ith *row* of `x`, `x(i,:)`, is to be the 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 e.g. `[t,x,tms,xms] = PIG(...)`

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

In some contexts it may be helpful to see directly how PI approximates a reduced slow vector field, via `[t,x,tms,xms,svf] = PIG(...)`

- `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 e.g. the forward Euler method (or the Runge-Kutta method), then $\hat{L} = L$ (or $\hat{L} = 4L$).

3.2.1 If no arguments, then execute an example

```
141 if nargin==0
```

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

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

The macroscale variable is $\vec{x}(t) = x_1(t)$, and the evolution of $d\vec{x}/dt$ is notionally unclear. With initial condition $\vec{x}(0) = 1$, the following code computes and plots a solution of the system over time $0 \le t \le 6$ for parameter $\epsilon = 10^{-3}$. The microscale system will be initialised ('`lift`ed') using $x_2(t) = 1/2$ whenever needed by `microBurst()`.

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

```
158 epsilon = 1e-3;
159 dxdt=@(t,x) [ cos(x(1))*sin(x(2))*cos(t)
160              ( cos(x(1))-x(2) )/epsilon ];
```

Second, we code microscale bursts, here using the standard `ode23()`. 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 $\Delta \le 1$ and then double the usual formula for safety.

```
172 bT = 2*epsilon*log(1/epsilon)
173 microBurst = @(tb0,xb0) ode23(dxdt,[tb0 tb0+bT],xb0);
```

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

```
179 restrict = @(x) x(1);
180 lift = @(x) [x; 0.5];
```

Fourth, invoke `PIG` to use `ode45()`, say, on the macroscale slow evolution. Integrate the micro-bursts over $0 \le t \le 6$ from initial condition $\vec{x} = (1, 0)$. (You could set `tSpan=[0 -6]` to integrate backwards in time with forward bursts.)

```
191 tSpan = [0 6];
192 [ts,xs,tms,xms] = PIG('ode45',microBurst,tSpan,1,restrict,lift);
```

Plot output of this projective integration.

```
198 figure, plot(ts,xs,'o:',tms,xms,'.')
199 title('Projective integration of singularly perturbed ODE')
```

```
200 xlabel('time t')
201 legend('x_1(t)','x_1(t) micro bursts', 'x_2(t) micro bursts')
```

Upon finishing execution of the example, exit this function.

```
207 return
208 end%if no arguments
```

3.2.2 The projective integration code

If no lifting/restriction functions were set, assign them.

```
222 if nargin < 5 || isempty(restrict)
223     lift=@(x) x;
224     restrict=@(x) x;
225 end
```

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

```
233 nArgs=nargout();
234 saveMicro = (nArgs>2);
235 saveSvf = (nArgs>4);
```

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

```
243 nT=length(tSpan)-1;
244 nx = length(lift(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.

```
252 if nargin<7
253     microBurst = @(t,x) cdmc(microBurst,t,x);
254 else
255 disp(['A ' class(cdmcFlag) ' was 7th input to PIG.'...
256  ' PIG will not use constraint-defined manifold computing.'])
257 end
```

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

```
269 x0 = reshape(x0,[],1);
270 [relax_t,x0_micro_relax] = microBurst(tSpan(1),lift(x0));
271 x0_relax = restrict(x0_micro_relax(end,:));
```

Update the initial time.

```
278 tSpan(1) = relax_t(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. Note that 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.

```
290 if saveMicro
291     tms=cell(nT+1,1); xms=cell(nT+1,1);
292     n=1;
293     tms{n} = reshape(relax_t,[],1);
294     xms{n} = x0_micro_relax;
295 
296     if saveSvf
297         svf.t = cell(nT+1,1);
298         svf.dx = cell(nT+1,1);
299     else
300         svf = [];
301     end
302 else
303     tms = []; xms = []; svf = [];
304 end
```

The idea of `PIG()` is to use the output from the microBurst to approximate an unknown function `ff(t,x)`, that describes the slow dynamics. This approximation is then used in the system/user-defined 'coarse solver' `macroInt()`. The approximation is described in

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

Run a microBurst from the given initial conditions.

```
322   [t_tmp,x_micro_tmp] = microBurst(tt,reshape(lift(xx),[],1));
```

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

```
329   x2 = restrict(x_micro_tmp(end,:));
330   x1 = restrict(x_micro_tmp(end-1,:));
331   del = t_tmp(end)-t_tmp(end-1);
332   dx = ( x2 - x1 ).'/del;
```

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

```
339   if saveMicro
340       n=n+1;
341       tms{n} = [reshape(t_tmp,[],1); nan];
342       xms{n} = [x_micro_tmp; nan(1,nx)];
343       if saveSvf
344           svf.t{n-1} = tt;
345           svf.dx{n-1} = dx;
346       end
347   end
348 end% PI projection
```

Define the approximate slow vector field according to Projective Integration.

```
357 PIf=@(t,x) genProjection(t,x);
```

Integrate `PIf()` with the user-specified simulator `macroInt()`.

```
366 [t,x]=feval(macroInt,PIf,tSpan,x0_relax.');
```

Overwrite `x(1,:)` and `t(1)`, which the user expect to be `x0` and `tSpan(1)` respectively, with the given initial conditions.

```
375 x(1,:) = x0.';
376 t(1) = tSpan(1);
```

Concatenate all the additional requested outputs into arrays.

```
383 if saveMicro
384     tms = cell2mat(tms);
385     xms = cell2mat(xms);
386     if saveSvf
387         svf.t = cell2mat(svf.t);
388         svf.dx = cell2mat(svf.dx);
389     end
390 end
```

3.2.3 If no output specified, then plot simulation.

```
399 if nArgs==0
400     figure, plot(t,x,'o:')
401     title('Projective Simulation via PIG')
402 end
```

This concludes `PIG()`.

```
410 end
```

3.3 `PIRK4()`: projective integration of fourth-order accuracy

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

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

See Section 3.1 as the inputs and outputs are the same as `PIRK2()`.

If no arguments, then execute an example

```
26 if nargin==0
```

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

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

With initial conditions $x(0) = y(0) = 0.2$, the following code uses forward time bursts in order to integrate backwards in time to $t = -5$. It plots the computed solution over time $-5 \le t \le 0$ 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 $\Delta = -1$.

```
epsilon = 0.1
ts = 0:-1:-5
bT = epsilon*log(abs(ts(2)-ts(1))/epsilon)
[x,tms,xms,rm,svf] = PIRK4(@MMburst, ts, 0.2*[1;1], bT);
figure, plot(ts,x,'o:',tms,xms)
xlabel('time t'), legend('x(t)','y(t)')
title('Backwards-time projective integration of Michaelis--Menten')
```

Upon finishing execution of the example, exit this function.

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

Example function code for a burst of ODEs Integrate a burst of length `bT` of the ODEs for the Michaelis–Menten enzyme kinetics at parameter 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 `ode23` to integrate in time.

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

Input

- `microBurst()`, a function that produces output from the user-specified code for 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 `microBurst()` determines `bT`, 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. `PIRK4()` does not use adaptive time-stepping; the macroscale time-steps are (nearly) the steps between elements of `tSpan`.

- `x0` is an n-vector of initial values at the initial time `tSpan(1)`. Elements of `x0` may be `NaN`: they are included in the simulation and output, and often represent boundaries in space fields.
- `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 computed burst.

```
127 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 = PIRK4(microBurst,tSpan,x0,bT)`.

  However, microscale details of the underlying Projective Integration computations may be helpful. `PIRK4()` provides two to four optional outputs of the microscale bursts.
- `tms`, optional, is an L dimensional column vector containing microscale times of burst simulations, each burst separated by `NaN`;
- `xms`, optional, is an $L \times n$ array of the corresponding microscale states—this data is an accurate simulation of the state and may help visualise more details of the solution.
- `rm`, optional, a struct containing the 'remaining' applications of the micro-burst required by the Projective Integration method during the calculation of the macrostep:
  - `rm.t` is a column vector of microscale times; and
  - `rm.x` is the array of corresponding burst states.

  The states `rm.x` do not have the same physical interpretation as those in `xms`; the `rm.x` are required in order to estimate the slow vector field during the calculation of the Runge–Kutta increments, and do not in general resemble the true dynamics.
- `svf`, optional, a struct containing the Projective Integration estimates of the slow vector field.
  - `svf.t` is a 4ℓ dimensional column vector containing all times at which the Projective Integration scheme is extrapolated along micro-burst data to form a macrostep.
  - `svf.dx` is a $4\ell \times n$ array containing the estimated slow vector field.

3.3.1 The projective integration code

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

```
191 nT=length(tSpan);
192 x=nan(nT,length(x0));
```

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

```
200 nArgs=nargout();
201 saveMicro = (nArgs>1);
202 saveFullMicro = (nArgs>3);
203 saveSvf = (nArgs>4);
```

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

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

Use the end point of the micro-burst as the initial conditions.

```
225 tSpan(1) = relax_t(end);
226 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.

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

Loop over the macroscale time-steps

```
257 for jT = 2:nT
258     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.

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

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

Check for round-off error.

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

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

```
303     Dt = tSpan(jT)-t1(end);
304     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()`).

```
312 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.

```
323     xint = xm1(end,:) + (Dt/2-abT)*dx1;
324     [t2,xm2] = microBurst(T+Dt/2, xint, bT);
325     del = t2(end)-t2(end-1);
326     dx2 = (xm2(end,:)-xm2(end-1,:))/del;
327
328     xint = xm1(end,:) + (Dt/2-abT)*dx2;
329     [t3,xm3] = microBurst(T+Dt/2, xint, bT);
330     del = t3(end)-t3(end-1);
331     dx3 = (xm3(end,:)-xm3(end-1,:))/del;
332
333     xint = xm1(end,:) + (Dt-abT)*dx3;
```

```
334     [t4,xm4] = microBurst(T+Dt, xint, bT);
335     del = t4(end)-t4(end-1);
336     dx4 = (xm4(end,:)-xm4(end-1,:))/del;
```

Check for round-off error.

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

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

```
353     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.

```
361     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 `NaN`s.

```
371     if saveMicro
372         tms{jT} = [reshape(t1,[],1); nan];
373         xms{jT} = [xm1; nan(1,size(xm1,2))];
```

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

```
381         if saveFullMicro
382             rm.t{jT} = [reshape(t2,[],1); nan;...
383                         reshape(t3,[],1); nan;...
384                         reshape(t4,[],1); nan];
385             rm.x{jT} = [xm2; nan(1,size(xm2,2));...
386                         xm3; nan(1,size(xm2,2));...
387                         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.

```
396             if saveSvf
397                 svf.t(4*jT-7:4*jT-4) = [t1(end); t2(end); t3(end); t4(end)];
398                 svf.dx(4*jT-7:4*jT-4,:) = [dx1; dx2; dx3; dx4];
399             end
400         end
401     end
```

Terminate the main loop:

```
407 end
```

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

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

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

```
424 if saveMicro
425     tms = cell2mat(tms);
426     xms = cell2mat(xms);
427     if saveFullMicro
428         rm.t = cell2mat(rm.t);
429         rm.x = cell2mat(rm.x);
430     end
431 end
```

3.3.2 If no output specified, then plot simulation

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

This concludes `PIRK4()`.

```
449 end
```

3.3.3 `cdmc()`

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

```
13 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

Output

- `ts`, a vector of times. `tout(end)` will equal `t`.
- `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 can be simulated by the microBurst `sol(t,x,T)`, one would define

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

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

Begin with a standard application of the micro-burst.

```
41 [t1,x1] = feval(microBurst,t0,x0);
42 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`.

```
50 dxdt = (x1(end,:) - x1(end-1,:))/(t1(end,:) - t1(end-1,:));
51 x0 = x1(end,:)-2*bT*dxdt;
52 t0 = t1(1)-bT;
53 [t2,x2] = feval(microBurst,t0,x0.');
```

Return both sets of output, though only (t2,x2) will be used in PI.

```
59 ts = [t1; t2];
60 xs = [x1; x2];
```

3.4 Example: PI using Runge–Kutta macrosolvers

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

Clear workspace and set a seed.

```
14 clear
15 rng(1)
16 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,

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

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

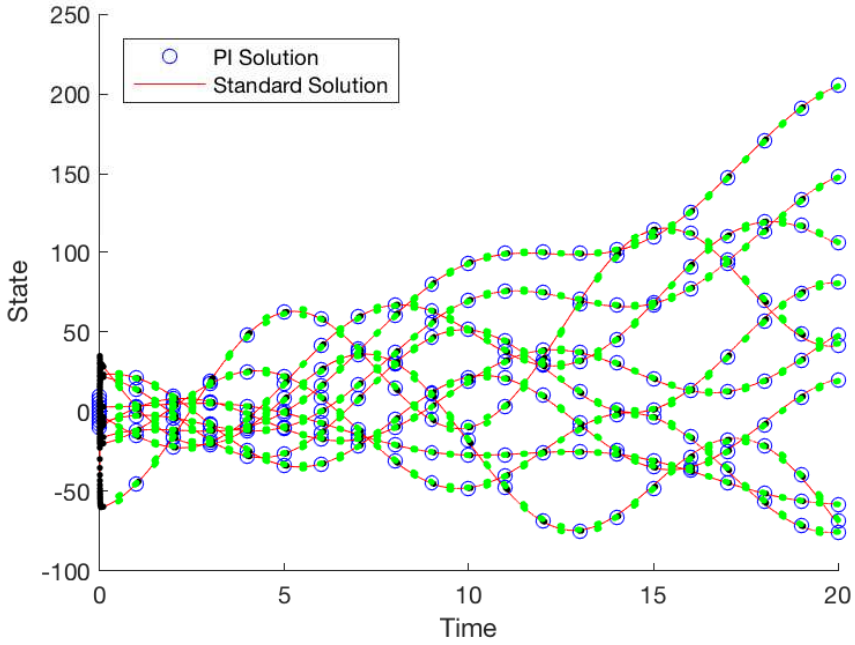
```
38 slowband = [-0.002; 0.002];
```

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

```
45 dxdt = gen_linear_system(7,3,fastband,slowband);
```

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

Figure 3.2: Demonstration of PIRK4(). From initial conditions, the system rapidly trannsitions to an attracting invariant manifold. The PI solution accurately tracks the evolution of the variables over time while requiring only a fraction of the computations of the standard solver.




```
55 tSpan = 0: 1 : 20;
56 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).

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

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

```
76 [tt,ode45x] = ode45(dxdt,tSpan([1,end]),x0);
```

Figure 3.2 plots the output.

```
92 clf()
93 hold on
94 PI_sol=plot(tSpan,x,'bo');
95 std_sol=plot(tt,ode45x,'r');
96 plot(tms,xms,'k.', rm.t,rm.x,'g.');
97 legend([PI_sol(1),std_sol(1)],'PI Solution',...
98     'Standard Solution','Location','NorthWest')
99 xlabel('Time'), ylabel('State')
```

Save plot to a file.

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

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

```
119 function [ts, xs] = linearBurst(ti, xi, varargin)
120 global dxdt
121 dt = 0.001;
122 ts = ti+(0:dt:0.05)';
123 nts = length(ts);
124 xs = NaN(nts,length(xi));
125 xs(1,:)=xi;
126 for k=2:nts
127     xi = xi + dt*dxdt(ts(k),xi.').';
128     xs(k,:)=xi;
129 end
130 end
```

3.5 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`.

```
16 clear all, close all
```

Set time scale separation and model.

```
23 epsilon = 1e-4;
24 dxdt=@(t,x) [ cos(x(1))*sin(x(2))*cos(t)
25              (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.

```
34 bT = epsilon*log(1/epsilon);
35 microBurst = @(tb0, xb0) ode45(dxdt,[tb0 tb0+bT],xb0);
```

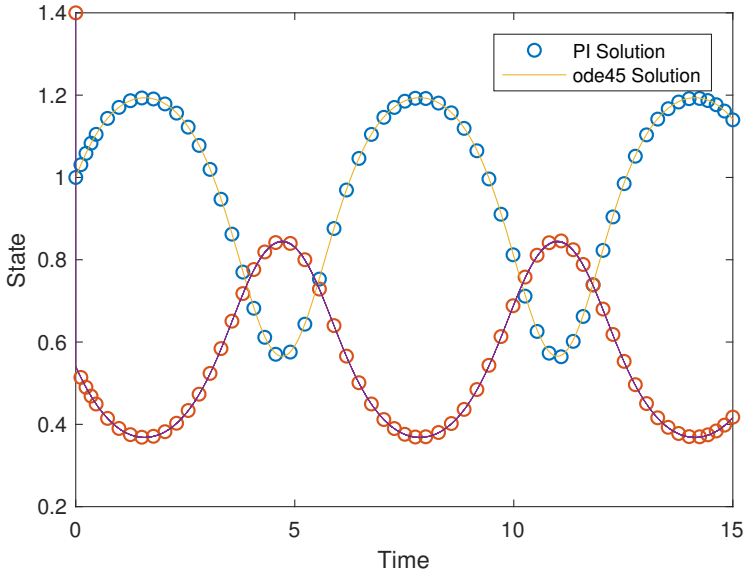
Set initial conditions, and the time to be covered by the macrosolver.

```
43 x0 = [1 1.4];
44 tSpan=[0 15];
```

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

```
53 tic
54 [ts,xs,tms,xms] = PIG('ode45',microBurst,tSpan,x0);
```

Figure 3.3: Accurate simulation of a stiff nonautonomous system by PIG(). The microsolver is called on-the-fly by the macrosolver `ode45`.




```
tPIGusingODE45asMacro = toc
tic
[t45,x45] = ode45(dxdt,tSpan,x0);
tODE45alone = toc
```

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

```
figure
h = plot(ts,xs,'o', t45,x45,'-', tms,xms,'.');
legend(h(1:2:5),'PI Solution','ode45 Solution','PI microsolver')
xlabel('Time'), ylabel('State')

figure
h = plot(ts,xs,'o', t45,x45,'-');
legend(h([1 3]),'PI Solution','ode45 Solution')
xlabel('Time'), ylabel('State')
set(gcf,'PaperPosition',[0 0 14 10]), print('-depsc2','PIGExample')
```

Figure 3.3 plots the output.

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

  But if the problem is insufficiently stiff (larger ϵ), then `PIG()` produces nonsense. This nonsense is overcome by `cdmc()` (Section 3.6).

- The mildly stiff problem in Section 3.4 may be efficiently solved by a standard solver (e.g., `ode45()`). The stiff but low dimensional problem in this example can be solved efficiently by a standard stiff solver (e.g.,

ode15s()). The real advantage of the Projective Integration schemes is in high dimensional stiff problems, that cannot be efficiently solved by most standard methods.

3.6 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.

```
14 clear all, close all
```

Set a weak time scale separation, and model.

```
21 epsilon = 0.01;
22 dxdt=@(t,x) [ cos(x(1))*sin(x(2))*cos(t)
23              (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.

```
32 bT = epsilon*log(1/epsilon);
33 microBurst = @(tb0,xb0) ode45(dxdt,[tb0 tb0+bT],xb0);
```

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

```
41 x0 = [1 0];
42 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.

```
52 [nt,nx] = PIG('ode45',microBurst,tSpan,x0,[],[],'no cdmc');
53 [ct,cx] = PIG('ode45',microBurst,tSpan,x0);
54 [t45,x45] = ode45(dxdt,tSpan([1 end]),x0);
```

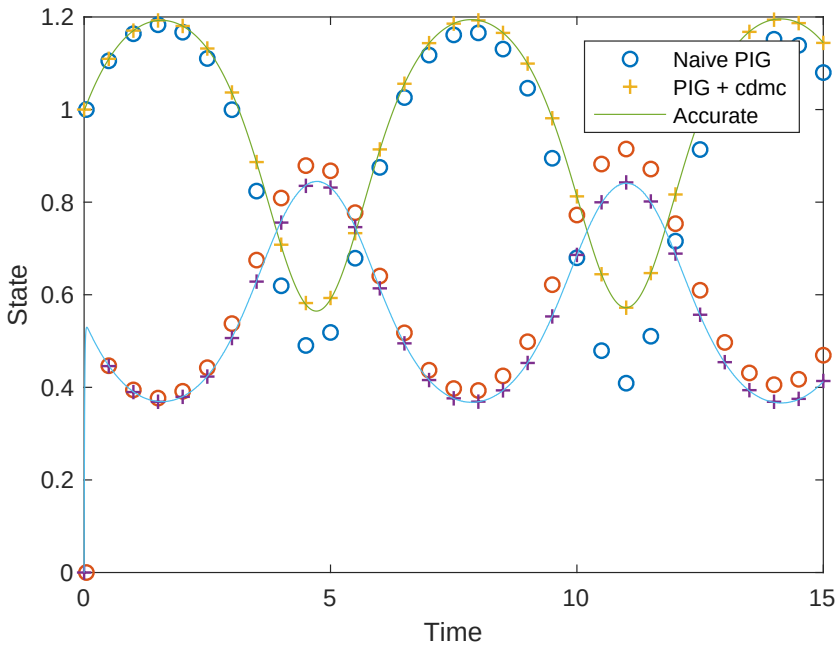
Figure 3.4 plots the output.

```
70 figure
71 h = plot(nt,nx,'rx', ct,cx,'bo', t45,x45,'-');
72 legend(h(1:2:5),'Naive PIG','default: PIG + cdmc','Accurate')
73 xlabel('Time'), ylabel('State')
74 set(gcf,'PaperPosition',[0 0 14 10]), print('-depsc2','PIGExplore')
```

The source of the error in the standard PIG() scheme is the burst length bT, that is significant on the slow time scale. Set bT to 20*epsilon or 50*epsilon[1] to worsen the error in both schemes. This example reflects a general principle, that most Projective Integration schemes will incur a global error term which is proportional to the simulation time of the microsolver

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



and independent of the order of the microsolver. The `PIRK()` schemes have been written to minimise, if not eliminate entirely, this error, but by design `PIG()` works with any user-defined macrosolver and cannot reduce this error. The function `cdmc()` reduces this error term by attempting to mimic the microsolver without advancing time.

3.7 To do/discuss

- could implement Projective Integration by 'arbitrary' Runge–Kutta scheme; that is, by having the user input a particular Butcher table—surely only specialists would be interested
- can 'reverse' the order of projection and microsolver applications with a little fiddling. Then output at each user-requested coarse time is the end point of an application of the microsolver - better predictions for fast variables.
- Can maybe implement microsolvers that terminate a burst when the fast dynamics have settled using, for example, the 'Events' function handle in ode23.

4 Patch scheme for given microscale discrete space system

Subsection contents

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

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

Quick start See Sections 4.1.1 and 4.7.1 which list example basic code that uses the provided functions to simulate 1D Burgers' PDE and a 2D nonlinear 'diffusion' PDE.

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

Subsection contents

Makes the struct `patches` for use by the patch/gap-tooth time derivative function `patchSmooth1()`. Section 4.1.1 lists an example of its use.

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

Input If invoked with no input arguments, then executes an example of simulating Burgers' PDE—see Section 4.1.1 for the example code.

- `fun` is the name of the user function, `fun(t,u,x)`, that computes time derivatives (or time-steps) of quantities on the patches.
- `Xlim` give the macro-space domain of the computation: patches are equi-spaced over the interior of the interval [`Xlim(1)`, `Xlim(2)`].
- `BCs` somehow will define the macroscale boundary conditions. Currently, `BCs` is ignored and the system is assumed macro-periodic in the domain.
- `nPatch` is the number of equi-spaced spaced patches.
- `ordCC` is the 'order' of interpolation across empty space of the macroscale mid-patch values to the edge of the patches for inter-patch coupling: currently must be ≥ -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` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch. May be omitted. The default is one (suitable for microscale lattices with only nearest neighbour interactions).

Output The *global* struct `patches` is created and set with the following components.

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

4.1.1 If no arguments, then execute an example

```
96 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).

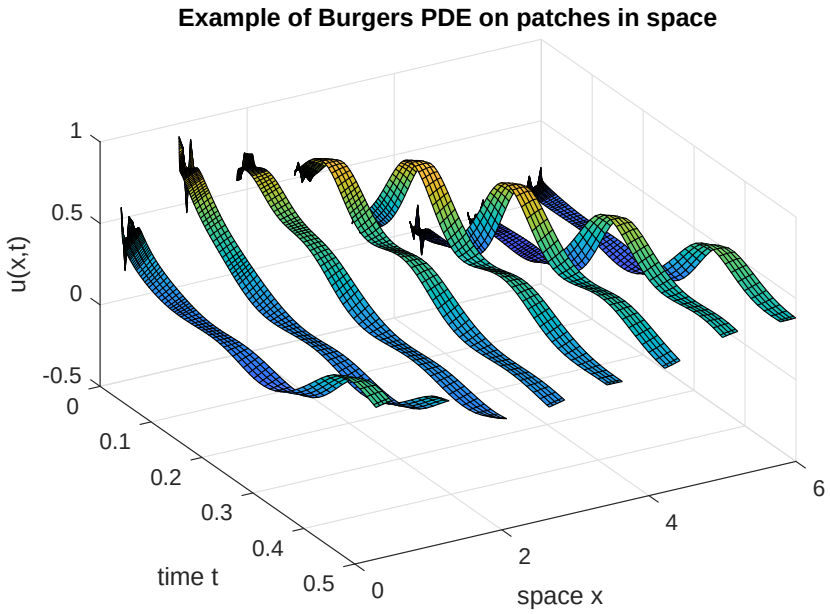
1. configPatches1
2. ode15s integrator ↔ patchSmooth1 ↔ user's burgersPDE
3. process results

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.

```
115 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 4.2).

Figure 4.1: field $u(x,t)$ of the patch scheme applied to Burgers' PDE.




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

Plot the simulation using only the microscale values interior to the patches: set x-edges to `nan` to leave the gaps. Figure 4.1 illustrates an example simulation in time generated by the patch scheme applied to Burgers' PDE.

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

Upon finishing execution of the example, exit this function.

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

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

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

This hack needs to be resolved: AJR, 2019-02-26

```
178 patches.EnsAve = 0;
```

4.1.2 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.

```
188 if nargin<8, nEdge=1; end
189 if nEdge>1, error('multi-edge-value interp not yet implemented'), end
190 if 2*nEdge+1>nSubP, error('too many edge values requested'), end
191 patches.nEdge=nEdge;
```

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

```
198 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.

```
206 if (ordCC<-1) | ~(floor(ordCC)==ordCC)
207     error('ordCC out of allowed range integer>-2')
208 end
```

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

```
215 patches.alt=mod(ordCC,2);
216 ordCC=ordCC+patches.alt;
217 patches.ordCC=ordCC;
```

Check for staggered grid and periodic case.

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

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

```
233 patches.Cwtsr=zeros(ordCC,1);
234 if patches.alt  % eqn (7) in \cite{Cao2014a}
235     patches.Cwtsr(1:2:ordCC)=[1 ...
236       cumprod((ratio^2-(1:2:(ordCC-2)).^2)/4)./ ...
237       factorial(2*(1:(ordCC/2-1)))];
238     patches.Cwtsr(2:2:ordCC)=[ratio/2 ...
239       cumprod((ratio^2-(1:2:(ordCC-2)).^2)/4)./ ...
240       factorial(2*(1:(ordCC/2-1))+1)*ratio/2];
241 else %
242     patches.Cwtsr(1:2:ordCC)=(cumprod(ratio^2- ...
243       (((1:(ordCC/2))-1)).^2)./factorial(2*(1:(ordCC/2))-1)/ratio);
244     patches.Cwtsr(2:2:ordCC)=(cumprod(ratio^2- ...
245       (((1:(ordCC/2))-1)).^2)./factorial(2*(1:(ordCC/2))));
246 end
247 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.

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

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

```
264 if mod(nSubP,2)==0, error('configPatches1: nSubP must be odd'), end
265 i0=(nSubP+1)/2;
266 dx=ratio*DX/(i0-1);
267 patches.x=bsxfun(@plus,dx*(-i0+1:i0-1)',X); % micro-grid
268 end% function
```

Fin.

4.2 `patchSmooth1()`: interface to time integrators

Subsection contents

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 4.1).

```
23 function dudt=patchSmooth1(t,u)
24 global patches
```

Input

- `u` is a vector of length `nSubP` · `nPatch` · `nVars` where there are `nVars` field values at each of the points in the `nSubP` × `nPatch` grid.
- `t` is the current time to be passed to the user's time derivative function.
- `patches` a struct set by `configPatches1()` with the following information used here.
  - `.fun` is the name of the user's function `fun(t,u,x)` that computes the time derivatives on the patchy lattice. The array `u` has size `nSubP` × `nPatch` × `nVars`. Time derivatives must be computed into the same sized array, 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 microscales.

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 4.3 describes `patchEdgeInt1()`.

```
74 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.

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

Fin.

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

Subsection contents

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`.

```
25 function u=patchEdgeInt1(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.
- `patches` a struct set by `configPatches1()` which includes the following.
  - `.x` is `nSubP` × `nPatch` array of the spatial locations x_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.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` × `nPatch` × `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.

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

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

```
76 dx = patches.x(3,1)-patches.x(2,1);
77 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).

```
86 if ~isfield(patches,'nCore')
87     patches.nCore = 1;
88 end
89 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.

```
100 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).

```
107 i0 = round((nSubP+1)/2);
108 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.

```
118 if patches.ordCC>0 % then non-spectral interpolation
119   if patches.EnsAve
120     uCore = sum(mean(u((i0-c):(i0+c),j,:),3),1)';
121     dmu = zeros(patches.ordCC,nPatch);
122   else
123     uCore = reshape(sum(u((i0-c):(i0+c),j,:),1),nPatch,nVars);
124     dmu = zeros(patches.ordCC,nPatch,nVars);
125   end;
126   if patches.alt % use only odd numbered neighbours
127     dmu(1,:,:) = (uCore(jp,:)+uCore(jm,:))/2; % \mu
128     dmu(2,:,:) = (uCore(jp,:)-uCore(jm,:)); % \delta
129     jp = jp(jp); jm = jm(jm); % increase shifts to \pm2
130   else % standard
131     dmu(1,j,:) = (uCore(jp,:)-uCore(jm,:))/2; % \mu\delta
132     dmu(2,j,:) = (uCore(jp,:)-2*uCore(j,:)+uCore(jm,:))/2; % \delta^2
133   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).

```
141   for k = 3:patches.ordCC
142     dmu(k,:,:) = dmu(k-2,jp,:)-2*dmu(k-2,j,:)+dmu(k-2,jm,:);
143   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.

```
152   if patches.EnsAve
153     u(nSubP,j,:) = repmat(uCore(j)'*(1-patches.alt) ...
154       +sum(bsxfun(@times,patches.Cwtsr,dmu)),[1,1,nVars]) ...
155       -sum(u((nSubP-patches.nCore+1):(nSubP-1),:,:),1);
156     u(1,j,:) = repmat(uCore(j)'*(1-patches.alt) ...
157       +sum(bsxfun(@times,patches.Cwtsl,dmu)),[1,1,nVars]) ...
158       -sum(u(2:patches.nCore,:,:),1);
159   else
160     u(nSubP,j,:) = uCore(j,:)*(1-patches.alt) ...
161       + reshape(-sum(u((nSubP-patches.nCore+1):(nSubP-1),j,:),1) ...
162       +sum(bsxfun(@times,patches.Cwtsr,dmu)),nPatch,nVars);
163     u(1,j,:) = uCore(j,:)*(1-patches.alt) ...
164       +reshape(-sum(u(2:patches.nCore,j,:),1)  ...
165       +sum(bsxfun(@times,patches.Cwtsl,dmu)),nPatch,nVars);
166   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' $|k| <$ `nPatch`$/2$, so set row vector `ks` $= k2\pi/N$ for 'wavenumbers' $k = (0, 1, \ldots, k_{\max}, -k_{\max}, \ldots, -1)$ for odd N, and $k = (0, 1, \ldots, k_{\max}, \pm(k_{\max}+1), -k_{\max}, \ldots, -1)$ for even N.

```
184 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.

```
194   if patches.alt % transform by doubling the number of fields
195     v = nan(size(u)); % currently to restore the shape of u
196     u = cat(3,u(:,1:2:nPatch,:),u(:,2:2:nPatch,:));
197     altShift = reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
198     iV = [nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
199     r = r/2;           % ratio effectively halved
200     nPatch = nPatch/2; % halve the number of patches
201     nVars = nVars*2;   % double the number of fields
202   else % the values for standard spectral
203     altShift = 0;
204     iV = 1:nVars;
205   end
```

Now set wavenumbers.

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

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

```
219   if imag(u(i0,:,:))==0, uclean=@(u) real(u);
220     else                 uclean=@(u) u;
221     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.

```
230   Ck = fft(u(i0,:,:));
231   if mod(nPatch,2)==0
232     Czz = Ck(1,nPatch/2+1,:)/nPatch;
233     Ck(1,nPatch/2+1,:) = 0;
234   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.

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

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

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

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

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

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

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

4.4 `BurgersExample`: simulate Burgers' PDE on patches

Section contents

Figure 4.1 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 4.2), 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).

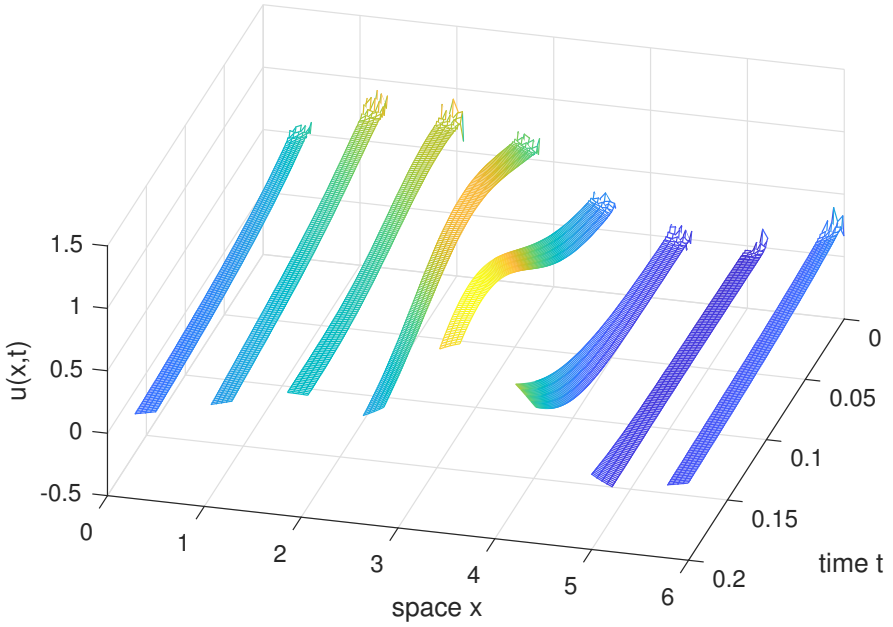
1. configPatches1
2. burgerBurst ↔ patchSmooth1 ↔ burgersMap
3. process results

4.4.1 Script code to simulate a microscale space-time map

Establish global data struct for the Burgers' map (Section 4.4.2) 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
```

Figure 4.2: a short time simulation of the Burgers' map (Section 4.4.2) on patches in space. It requires many very small time-steps only just visible in this mesh.




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

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

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

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

```
figure(1),clf
xs = patches.x;  xs([1 end],:) = nan;
mesh(ts,xs(:),us')
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
view(105,45)
```

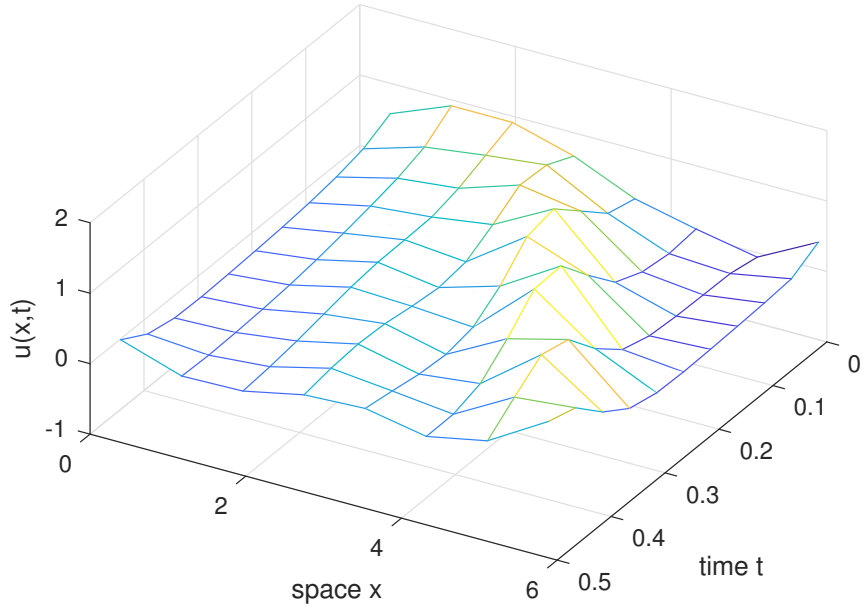
Save the plot to file to form Figure 4.2.

```
set(gcf,'paperposition',[0 0 14 10])
print('-depsc2','BurgersMapU')
```

Alternatively use projective integration

Around the microscale burst `burgerBurst()`, wrap the projective integration function `PIRK2()` of Section 3.1. Figure 4.3 shows the macroscale prediction

Figure 4.3: macroscale space-time field $u(x,t)$ in a basic projective integration of the patch scheme applied to the microscale Burgers' map.



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 ↔ burgerBurst ↔ patchSmooth1 ↔ 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;
```

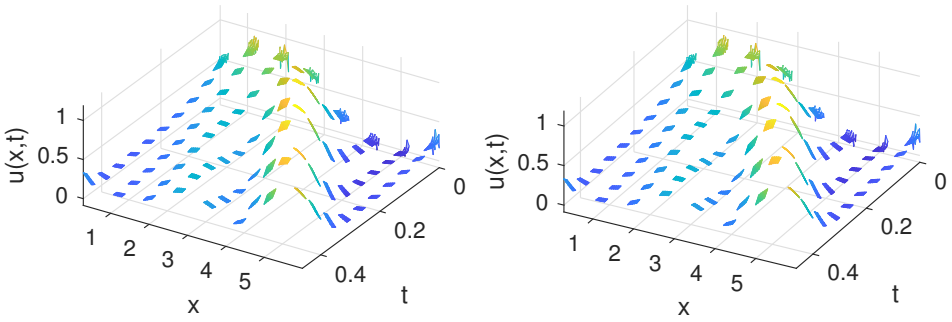
Set the desired macroscale time-steps, and microscale burst length over the time domain. Then projectively integrate in time using `PIRK2()` which is (roughly) second-order accurate in the macroscale time-step.

```
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);
```

Plot and save the macroscale predictions of the mid-patch values to give the macroscale mesh-surface of Figure 4.3 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,'paperposition',[0 0 14 10])
142 print('-depsc2','BurgersU')
```

Figure 4.4: the field $u(x,t)$ during each of the microscale bursts used in the projective integration. View this stereo pair cross-eyed.



Then plot and save the microscale mesh of the microscale bursts shown in Figure 4.4 (a stereo pair). The details of the fine microscale mesh are almost invisible.

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

4.4.2 `burgersMap()`: discretise the PDE microscale

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

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

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

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

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

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

Use `patchSmooth1()` (Section 4.2) 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.

```
216   us = nan(ndt+1,numel(ui));
217   us(1,:) = reshape(ui,1,[]);
218   for j = 1:ndt
219     ui = patchSmooth1(ts(j),ui);
220     us(j+1,:) = reshape(ui,1,[]);
221   end
```

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

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

Fin.

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

Section contents

Figures 4.5 and 4.6 show example simulations 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 minus the number of points in the core is an even multiple of the microscale periodicity. We present two different methods of obtaining a macroscale solution. One method uses the given heterogeneous diffusion, which produces a solution which has microscale roughness (Figure 4.5). The other method constructs an ensemble of heterogeneous diffusion and produces an ensemble average solution which has a smooth microscale (Figure 4.6).

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

$$\dot{u}_i = [c_{i-1/2}(u_{i-1} - u_i) + c_{i+1/2}(u_{i+1} - u_i)]/dx^2. \tag{4.1}$$

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

Figure 4.5: the diffusing field $u(x,t)$ in the patch (gap-tooth) scheme applied to microscale heterogeneous diffusion with no ensemble average. The heterogeneous diffusion results in a smilarly heterogeneous field solution.

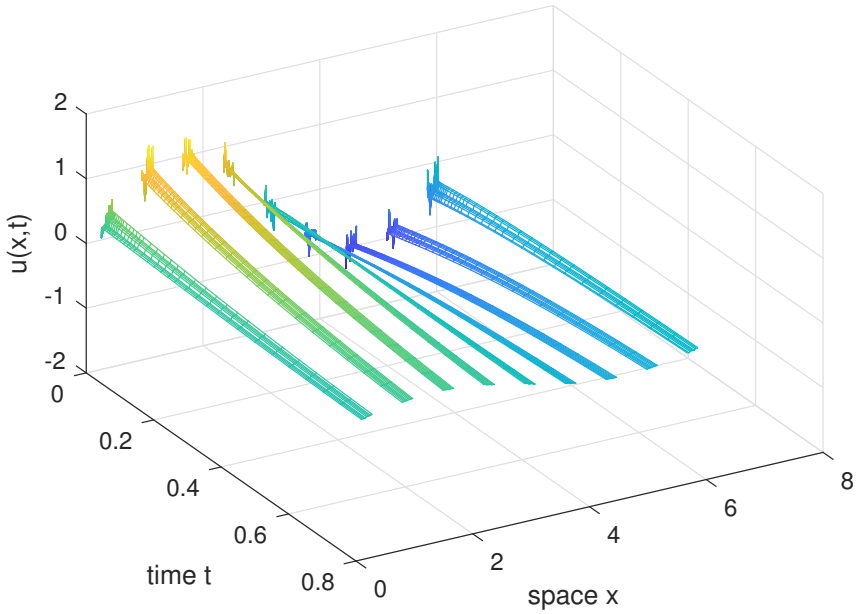
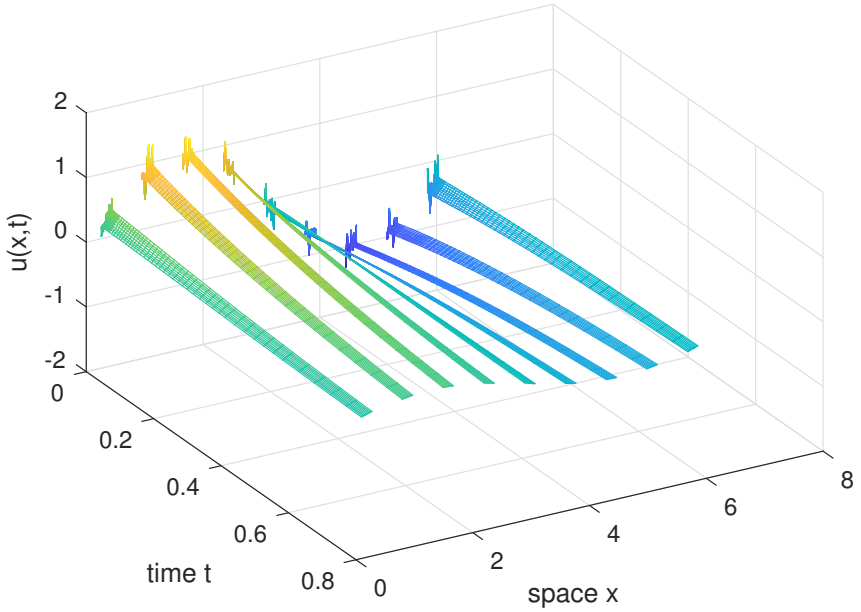


Figure 4.6: the diffusing field $u(x,t)$ in the patch (gap-tooth) scheme applied to microscale heterogeneous diffusion with an ensemble average. The ensemble average smooths out the heterogeneous diffusion.



4.5.1 Script to simulate via stiff or projective integration

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

```
clear all
mPeriod = 4
rng('default'); rng(1);
cDiff = exp(4*rand(mPeriod,1))
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). The Boolean `patches.EnsAve` determines whether or not we apply ensemble averaging of diffusivity configurations. We evaluate the patch coupling by interpolating the core.

```
global patches
nPatch = 9
ratio = 0.2
nSubP = 11
Len = 2*pi;
ordCC=4;
patches.nCore=3;
patches.ratio = ratio*(nSubP - patches.nCore)/(nSubP - 1);
configPatches1(@heteroDiff,[0 Len],nan,nPatch, ...
 ordCC,patches.ratio,nSubP);
patches.EnsAve = 1;
```

A (`nSubP-1`) $\times$ `nPatch` matrix defines the diffusivity coefficients within each patch. In the case of ensemble averaging, `nVars` becomes 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.cDiff`. With ensemble averaging we must increase the size of the diffusivity matrix to (`nSubP-1`) $\times$ `nPatch` $\times$ `nVars`.

```
patches.cDiff = cDiff((mod(round(patches.x(1:(end-1),:) ...
  /(patches.x(2)-patches.x(1))-0.5),mPeriod)+1));
if patches.EnsAve
  if mPeriod>2
    nVars=2*mPeriod;
  else
    nVars=mPeriod;
```

```
126   end
127   patches.cDiff=repmat(patches.cDiff,[1,1,nVars]);
128   for sx=2:mPeriod
129     patches.cDiff(:,:,sx)=circshift( ...
130       patches.cDiff(:,:,sx-1),[sx-1,0]);
131    end;
132    if nVars>2
133      patches.cDiff(:,:,(mPeriod+1):end)=flipud( ...
134        patches.cDiff(:,:,1:mPeriod));
135    end;
136 end
```

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

```
149 u0 = sin(patches.x)+0.2*randn(nSubP,nPatch);
150 %u0 = exp(-2*(patches.x-Len/2).^2).*(1+0.1*rand(nSubP,nPatch));
151 if patches.EnsAve
152   u0 = repmat(u0,[1,1,nVars]);
153 end
154 [ts,ucts] = ode15s(@patchSmooth1, [0 2/cHomo], u0(:));
155 ucts=reshape(ucts,length(ts),length(patches.x(:)),[]);
```

Plot the simulation in Figure 4.5 (with no ensemble average) or Figure 4.6 (with an ensemble average). If we have calculated an ensemble of field solutions, then we must first take the ensemble average.

```
166 if patches.EnsAve % calculate the ensemble average
167   uctsAve=mean(ucts,3);
168 else
169   uctsAve=ucts;
170 end
171 figure(1),clf
172 xs = patches.x;  xs([1 end],:) = nan;
173 mesh(ts,xs(:),uctsAve'),  view(60,40)
174 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
175 set(gcf,'PaperUnits','centimeters');
176 set(gcf,'PaperPosition',[0 0 14 10]);
177 if patches.EnsAve
178   print('-depsc2','HomogenisationCtsUEnsAve')
179 else
180   print('-depsc2','HomogenisationCtsU')
181 end
```

Use projective integration in time Now take `patchSmooth1`, the interface to the time derivatives, and wrap around it the projective integration

Figure 4.7: field $u(x,t)$ shows basic projective integration of patches of heterogeneous diffusion with no ensemble average: different colours correspond to the times in the legend. This field solution displays some fine scale heterogeneity due to the heterogeneous diffusion.

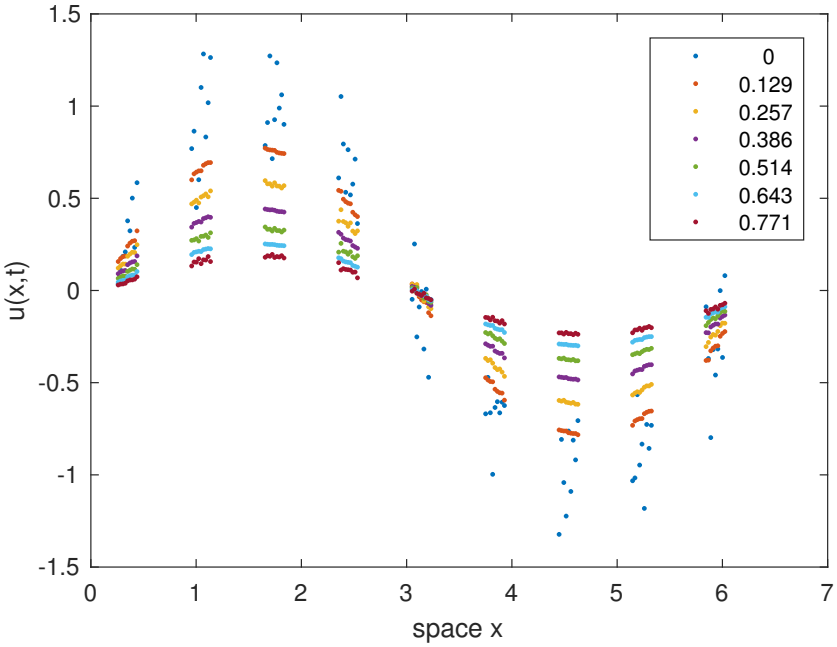
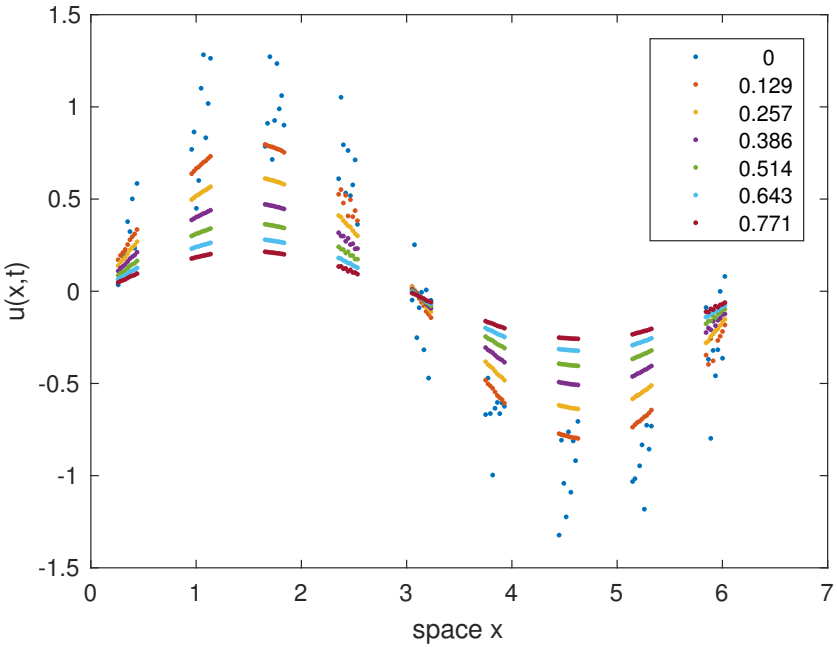


Figure 4.8: field $u(x,t)$ shows basic projective integration of patches of heterogeneous diffusion with ensemble average: different colours correspond to the times in the legend. Once transients have decayed, this field solution is smooth due to the ensemble average.



`PIRK2` (Section 3.1), of bursts of simulation from `heteroBurst` (Section 4.5.3), as illustrated by Figures 4.7 and 4.8.

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 ↔ heteroBurst ↔ micro-integrator ↔ patchSmooth1 ↔ heteroDiff
3. process results

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

```
225 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 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.

```
237 ts = linspace(0,2/cHomo,7)
238 bT = 3*( ratio*Len/nPatch )^2/cHomo
239 addpath('../ProjInt','../SandpitPlay/RKint')
240 [us,tss,uss] = PIRK2(@heteroBurst, ts, u0(:), bT);
```

Plot the macroscale predictions to draw Figure 4.7 or Figure 4.8. If we have calculated an ensemble of field solutions, then we must first take the ensemble average.

```
250 if patches.EnsAve % calculate the ensemble average
251   usAve=mean(reshape(us,size(us,1),length(xs(:)),nVars),3);
252   ussAve=mean(reshape(uss,length(tss),length(xs(:)),nVars),3);
253 else
254   usAve=us;
255   ussAve=uss;
256 end
257 figure(2),clf
258 plot(xs(:),usAve','.')
259 ylabel('u(x,t)'), xlabel('space x')
260 legend(num2str(ts',3))
261 set(gcf,'PaperUnits','centimeters');
262 set(gcf,'PaperPosition',[0 0 14 10]);
263 if patches.EnsAve
264   print('-depsc2','HomogenisationUEnsAve')
265 else
266   print('-depsc2','HomogenisationU')
267 end
```

Also plot a surface detailing the microscale bursts as shown in Figure 4.9 or Figure 4.10.

Figure 4.9: stereo pair of the field $u(x,t)$ during each of the microscale bursts used in the projective integration with no ensemble averaging.

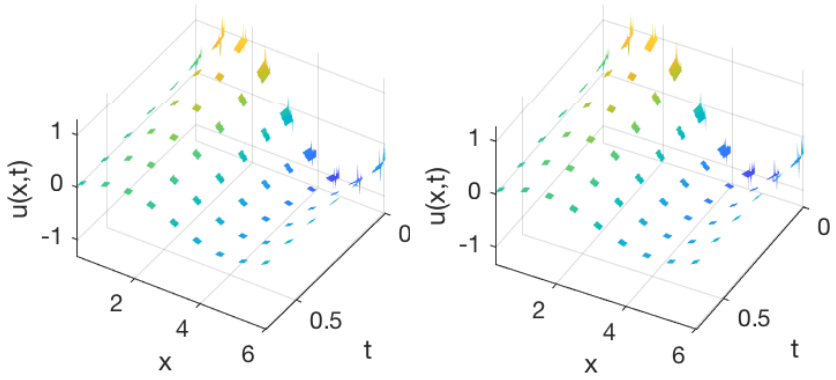
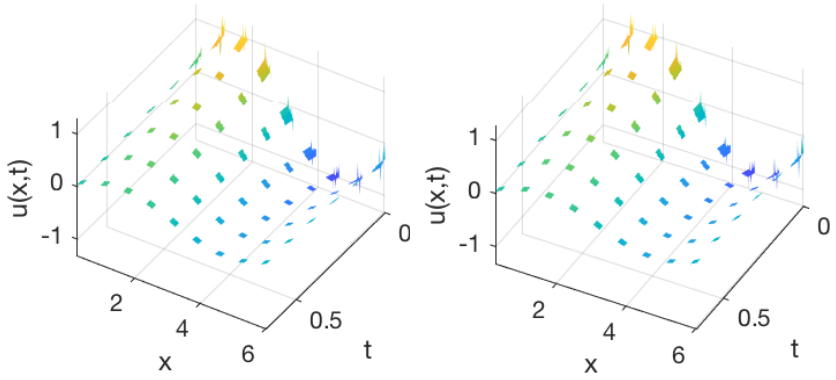


Figure 4.10: stereo pair of the field $u(x,t)$ during each of the microscale bursts used in the projective integration with ensemble averaging.




```
figure(3),clf
for k = 1:2, subplot(1,2,k)
  surf(tss,xs(:),ussAve',  'EdgeColor','none')
  ylabel('x'), xlabel('t'), zlabel('u(x,t)')
  axis tight, view(126-4*k,45)
end
set(gcf,'PaperUnits','centimeters');
set(gcf,'PaperPosition',[0 0 14 6]);
if patches.EnsAve
  print('-depsc2','HomogenisationMicroEnsAve')
else
  print('-depsc2','HomogenisationMicro')
end
```

End of the script.

4.5.2 `heteroDiff()`: heterogeneous diffusion

This function codes the lattice heterogeneous diffusion inside the patches. For 2D input arrays `u` and `x` (via edge-value interpolation of `patchSmooth1`, Section 4.2), computes the time derivative (4.1) at each point in the interior of a patch, output in `ut`. The column vector (or possibly array) of diffusion coefficients c_i have previously been stored in struct `patches`.

```
320 function ut = heteroDiff(t,u,x)
321   global patches
322   dx = diff(x(2:3)); % space step
323   i = 2:size(u,1)-1; % interior points in a patch
324   ut = nan(size(u)); % preallocate output array
325   ut(i,:,:) = diff(patches.cDiff.*diff(u))/dx^2; %- abs(u(i,:,:)).*u(i,:,:).^
326 end% function
```

4.5.3 `heteroBurst()`: a burst of heterogeneous diffusion

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

- `ode23` generates 'noise' that is unsightly at best and may be ruinous;
- `ode45` is similar to `ode23`, but with reduced noise;
- `ode15s` does not cater for the NaNs in some components of `u`;
- `rk2int` simple specified step integrator, but may require inefficiently small time-steps.

```
348 function [ts, ucts] = heteroBurst(ti, ui, bT)
349   switch '45'
350   case '23',  [ts,ucts] = ode23(@patchSmooth1,[ti ti+bT],ui(:));
351   case '45',  [ts,ucts] = ode45(@patchSmooth1,[ti ti+bT],ui(:));
352   case '15s', [ts,ucts] = ode15s(@patchSmooth1,[ti ti+bT],ui(:));
353   case 'rk2', ts = linspace(ti,ti+bT,200)';
354               ucts = rk2int(@patchSmooth1,ts,ui(:));
355   end
356 end
```

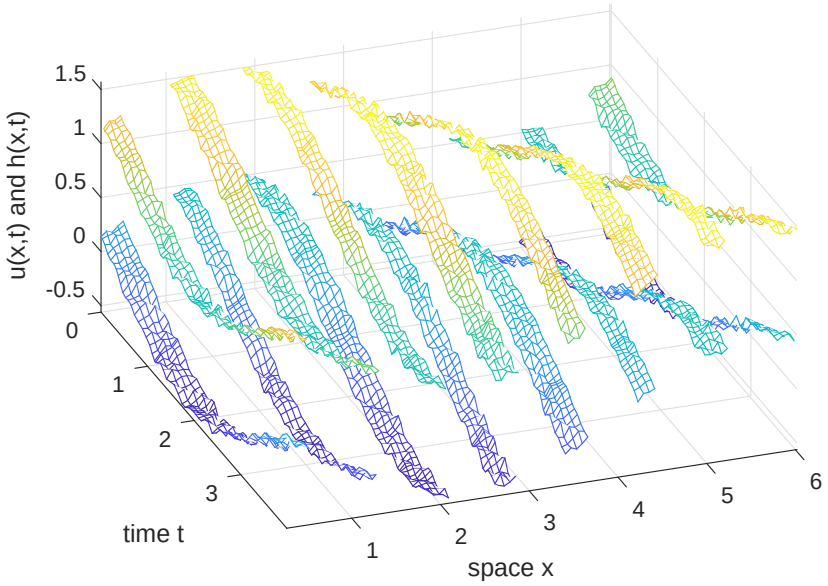
Fin.

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

Section contents

Figure 4.11 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.

Figure 4.11: water depth $h(x,t)$ (above) and velocity field $u(x,t)$ (below) of the gap-tooth scheme applied to the ideal wave PDE (4.2), linearised. The microscale random component to the initial condition has long lasting effects on the simulation—but the macroscale wave still propagates.



This approach, based upon the differential equations coded in Section 4.6.2, may be adapted by a user to a wide variety of 1D wave and wave-like systems. For example, the differential equations of Section 4.6.3 describes the nonlinear microscale simulator of the nonlinear shallow water wave PDE derived from the Smagorinski model of turbulent flow (Cao & Roberts 2012, 2016*a*).

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

$$\frac{\partial h}{\partial t} = -c_1\frac{\partial u}{\partial x} + f_1[h,u] \quad \text{and} \quad \frac{\partial u}{\partial t} = -c_2\frac{\partial h}{\partial x} + f_2[h,u], \tag{4.2}$$

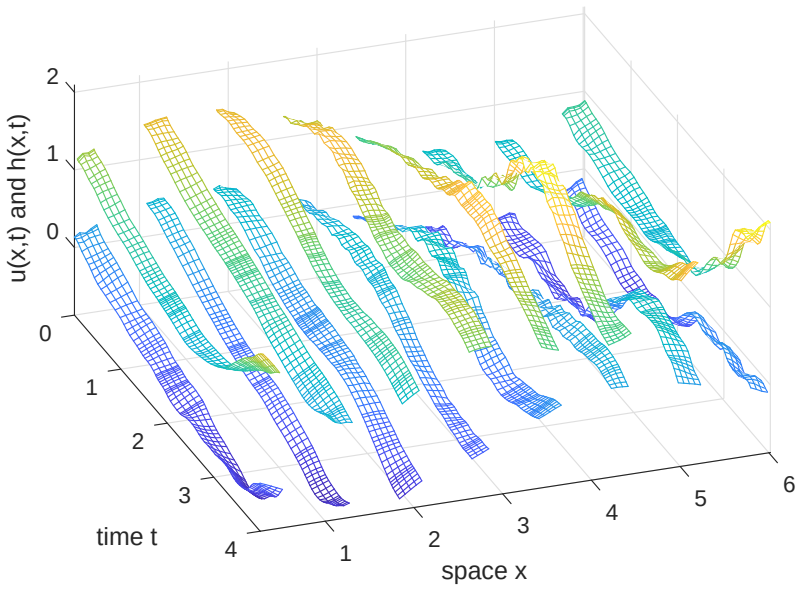
where the brackets indicate that the nonlinear functions f_ℓ may involve various spatial derivatives of the fields $h(x,t)$ and $u(x,t)$. For example, Section 4.6.3 encodes a nonlinear Smagorinski model of turbulent shallow water (Cao & Roberts 2012, 2016*a*, e.g.) along an inclined flat bed: let x measure position along the bed and in terms of fluid depth $h(x,t)$ and depth-averaged longitudinal velocity $u(x,t)$ the model PDEs are

$$\frac{\partial h}{\partial t} = -\frac{\partial (hu)}{\partial x}\,, \tag{4.3a}$$

$$\frac{\partial u}{\partial t} = 0.985\left(\tan\theta - \frac{\partial h}{\partial x}\right) - 0.003\frac{u|u|}{h} - 1.045u\frac{\partial u}{\partial x} + 0.26h|u|\frac{\partial^2 u}{\partial x^2}\,, \tag{4.3b}$$

where $\tan\theta$ is the slope of the bed. Equation (4.3a) represents conservation of the fluid. The momentum PDE (4.3b) represents the effects of

Figure 4.12: 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 (4.3). The microscale random initial component decays where the water speed is non-zero due to 'turbulent' dissipation.



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 4.12 shows one simulation of this system—for the same initial condition as Figure 4.11.

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

4.6.1 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 ↔ patchSmooth1 ↔ idealWavePDE
3. process results
4. ode15s ↔ patchSmooth1 ↔ waterWavePDE
5. process results

Establish the global data struct `paches` for the PDEs (4.2) (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
```

```
nSubP = 11 %of the form 4*n-1
Len = 2*pi;
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.

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

Set an initial condition of a progressive wave, and check evaluation of the time derivative. The capital letter `U` denotes an array of values merged from both u and h fields on the staggered grids (possibly with some optional microscale wave noise).

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

Conventional integration in time Integrate in time using standard MATLAB/Octave stiff integrators. Here do the two cases of the ideal wave and the water wave equations in the one loop.

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

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

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

Plot the simulation.

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

Save the plot to file.

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

For the second time through the loop, change to the Smagorinski turbulence model (4.3) of shallow water flow, keeping other parameters and the initial

condition the same.

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

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

4.6.2 `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.

```
297 function Ut = idealWavePDE(t,U,x)
298   global patches
299   dx = diff(x(2:3));
300   Ut = nan(size(U));  ht = Ut;
```

Compute the PDE derivatives at interior points of the patches.

```
306   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.

```
318   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.

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

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

```
335   Ut(patches.hPts) = ht(patches.hPts);
336 end
```

4.6.3 `waterWavePDE()`: water wave PDE

This function codes the staggered lattice equation inside the patches for the nonlinear wave-like PDE system (4.3). Also, regularise the absolute value appearing the the PDEs via the one-line function `rabs()`.

```
351 function Ut = waterWavePDE(t,U,x)
352   global patches
353   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.

```
361   dx = diff(x(2:3));
362   Ut = nan(size(U));  ht = Ut;
363   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.

```
371   ii = i(2:end-1);
372   V = Ut;
373   V(ii,:) = (U(ii+1,:)+U(ii-1,:))/2;
374   V(1:2,:) = 2*U(2:3,:)-V(3:4,:);
375   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.

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

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

```
390   Ut(i,:) = -0.985*(U(i+1,:)-U(i-1,:))/(2*dx) ...
391     -0.003*U(i,:).*rabs(U(i,:)./V(i,:)) ...
392     -1.045*U(i,:).*(V(i+1,:)-V(i-1,:))/(2*dx) ...
393     +0.26*rabs(V(i,:).*U(i,:)).*(V(i+1,:)-2*U(i,:)+V(i-1,:))/dx^2/2;
```

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

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

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

```
409   Ut(patches.hPts) = ht(patches.hPts);
410 end
```

Fin.

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

Subsection contents

Makes the struct `patches` for use by the patch/gap-tooth time derivative function `patchSmooth2()`. Section 4.7.1 lists an example of its use.

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

Input If invoked with no input arguments, then executes an example of simulating a nonlinear diffusion PDE relevant to the lubrication flow of a thin layer of fluid—see Section 4.7.1 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 $[\texttt{Xlim(1)}, \texttt{Xlim(2)}] \times [\texttt{Xlim(3)}, \texttt{Xlim(4)}]$: 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 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 $\texttt{ratio} = \frac{1}{2}$ means the patches abut; and $\texttt{ratio} = 1$ 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)` $\times$ `nPatch(1)` array of the regular spatial locations x_{ij} of the microscale grid points in every patch.
- `.y` is `nSubP(2)` $\times$ `nPatch(2)` array of the regular spatial locations y_{ij} of the microscale grid points in every patch.
- `.nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.

4.7.1 If no arguments, then execute an example

```
121 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

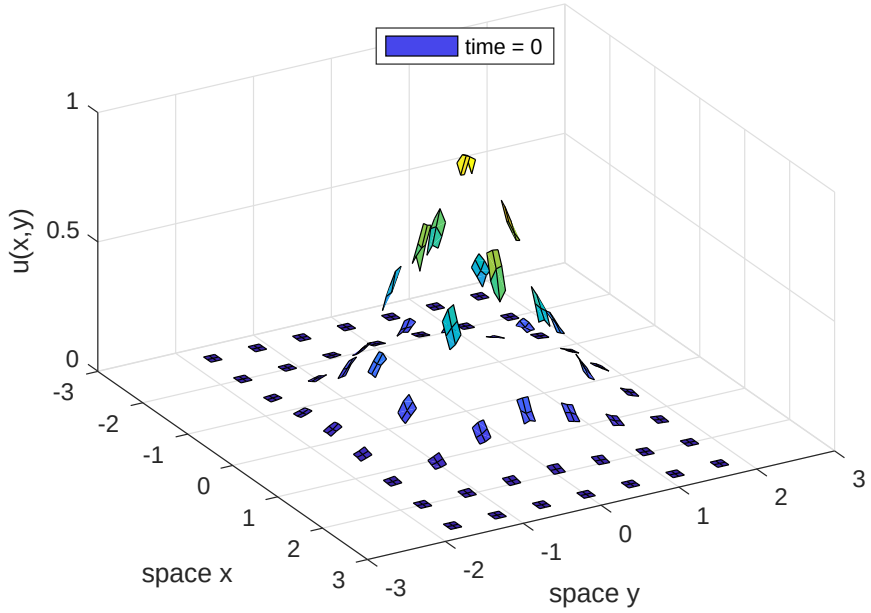
Establish global patch data struct to interface with a function coding a nonlinear 'diffusion' PDE: to be solved on 6×4-periodic domain, with 9×7 patches, spectral interpolation (0) couples the patches, each patch of half-size ratio 0.25, and with 5×5 points within each patch.

```
141 nSubP = 5;
142 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.

```
149 x = reshape(patches.x,nSubP,1,[],1);
150 y = reshape(patches.y,1,nSubP,1,[]);
151 u0 = exp(-x.^2-y.^2);
152 u0 = u0.*(0.9+0.1*rand(size(u0)));
```

Figure 4.13: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to a nonlinear 'diffusion' PDE: Figure 4.14 plots the computed field at time $t = 3$.



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.

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

Start by showing the initial conditions of Figure 4.13 while the simulation computes.

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

Save the initial condition to file for Figure 4.13.

```
set(gcf,'PaperPosition',[0 0 14 10])
print('-depsc2','configPatches2ic')
```

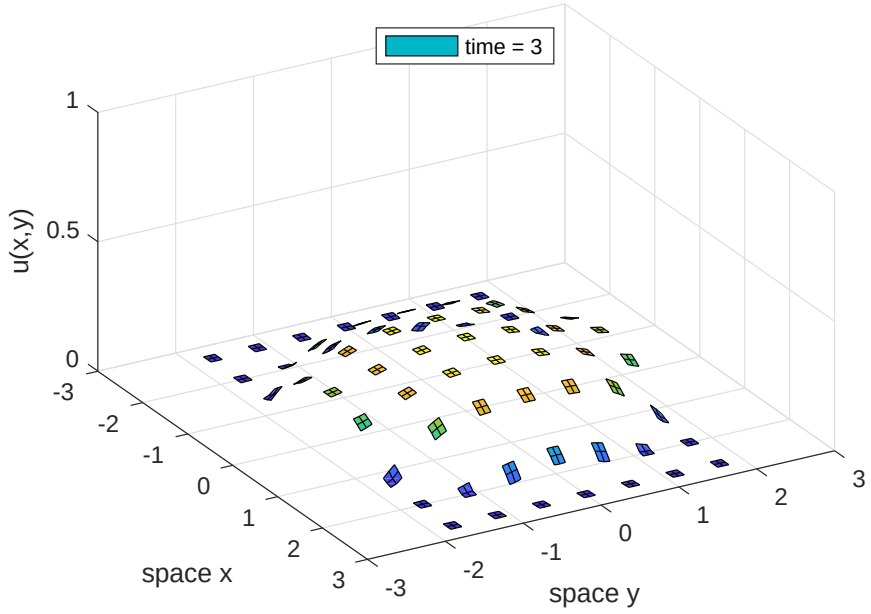
Integrate in time using standard functions.

```
disp('Wait while we simulate h_t=(h^3)_xx+(h^3)_yy')
[ts,ucts] = ode15s(@patchSmooth2,[0 3],u0(:));
```

Animate the computed simulation to end with Figure 4.14.

```
for i = 1:length(ts)
  u = patchEdgeInt2(ucts(i,:));
```

Figure 4.14: field $u(x,y,t)$ at time $t=3$ of the patch scheme applied to a nonlinear 'diffusion' PDE with initial condition in Figure 4.13.




```
205   u = reshape(permute(u,[1 3 2 4]), [numel(x) numel(y)]);
206   hsurf.ZData = u';
207   legend(['time = ' num2str(ts(i),2)])
208   pause(0.1)
209 end
210 print('-depsc2','configPatches2t3')
```

Upon finishing execution of the example, exit this function.

```
225 return
226 end%if no arguments
```

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

```
237 function ut = nonDiffPDE(t,u,x,y)
238   dx = diff(x(1:2));  dy = diff(y(1:2));  % microscale spacing
239   i = 2:size(u,1)-1;  j = 2:size(u,2)-1;  % interior points in patches
240   ut = nan(size(u));  % preallocate storage
241   ut(i,j,:,:) = diff(u(:,j,:,:).^3,2,1)/dx^2 ...
242                +diff(u(i,:,:,:).^3,2,2)/dy^2;
243 end
```

4.7.2 The code to make patches

Initially duplicate parameters as needed.

```
259 if numel(Xlim)==2, Xlim = repmat(Xlim,1,2); end
260 if numel(nPatch)==1, nPatch = repmat(nPatch,1,2); end
```

```
261 if numel(ratio)==1, ratio = repmat(ratio,1,2); end
262 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.

```
270 if nargin<8, nEdge = 1; end
271 if nEdge>1, error('multi-edge-value interp not yet implemented'), end
272 if 2*nEdge+1>nSubP, error('too many edge values requested'), end
273 patches.nEdge = nEdge;
```

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

```
282 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.

```
291 if ~ismember(ordCC,[0])
292     error('ordCC out of allowed range [0]')
293 end
```

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

```
300 patches.alt = mod(ordCC,2);
301 ordCC = ordCC+patches.alt;
302 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.)

```
318 ratio = ratio(:)'; % force to be row vector
319 if patches.alt  % eqn (7) in \cite{Cao2014a}
320   patches.Cwtsr = [1
321     ratio/2
322     (-1+ratio.^2)/8
323     (-1+ratio.^2).*ratio/48
324     (9-10*ratio.^2+ratio.^4)/384
325     (9-10*ratio.^2+ratio.^4).*ratio/3840
326     (-225+259*ratio.^2-35*ratio.^4+ratio.^6)/46080
327     (-225+259*ratio.^2-35*ratio.^4+ratio.^6).*ratio/645120 ];
328 else %
329   patches.Cwtsr = [ratio
330     ratio.^2/2
331     (-1+ratio.^2).*ratio/6
332     (-1+ratio.^2).*ratio.^2/24
333     (4-5*ratio.^2+ratio.^4).*ratio/120
334     (4-5*ratio.^2+ratio.^4).*ratio.^2/720
335     (-36+49*ratio.^2-14*ratio.^4+ratio.^6).*ratio/5040
336     (-36+49*ratio.^2-14*ratio.^4+ratio.^6).*ratio.^2/40320 ];
337 end
338 patches.Cwtsr = patches.Cwtsr(1:ordCC,:);
```

```
% should avoid this next implicit auto-replication
patches.Cwtsl = (-1).^((1:ordCC)'-patches.alt).*patches.Cwtsr;
```

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

```
X = linspace(Xlim(1),Xlim(2),nPatch(1)+1);
X = X(1:nPatch(1))+diff(X)/2;
DX = X(2)-X(1);
Y = linspace(Xlim(3),Xlim(4),nPatch(2)+1);
Y = Y(1:nPatch(2))+diff(Y)/2;
DY = Y(2)-Y(1);
```

Construct the microscale in each patch, assuming Dirichlet patch edges, and a half-patch length of `ratio(1)` · `DX` and `ratio(2)` · `DY`.

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

Fin.

4.8 `patchSmooth2()`: interface to time integrators

Subsection contents

To simulate in time with spatial patches we often need to interface a users time derivative function with time integration routines such as `ode15s` or `PIRK2`. This function provides an interface. It assumes that the sub-patch structure is *smooth* so that the patch centre-values are sensible macroscale variables, and patch edge values are determined by macroscale interpolation of the patch-centre values. Communicate patch-design variables to this function via the previously established global struct `patches`.

```
function dudt = patchSmooth2(t,u)
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 microscales.
  - `.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 microscales.

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 4.9 describes `patchEdgeInt2()`.

```
82 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.

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

Fin.

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

Subsection 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`.

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

Input

- `u` is a vector of length $\texttt{nx} \cdot \texttt{ny} \cdot \texttt{Nx} \cdot \texttt{Ny} \cdot \texttt{nVars}$ where there are `nVars` field values at each of the points in the $\texttt{nx} \times \texttt{ny} \times \texttt{Nx} \times \texttt{Ny}$ grid on the $\texttt{Nx} \times \texttt{Ny}$ array of patches.
- `patches` a struct set by `configPatches2()` which includes the following information.
  - `.x` is $\texttt{nx} \times \texttt{Nx}$ 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 microscales.
  - `.y` is similarly $\texttt{ny} \times \texttt{Ny}$ 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 microscales.
  - `.ordCC` is order of interpolation, currently only $\{0\}$.
  - `.Cwtsr` and `.Cwtsl`—not yet used

Output

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

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

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

```
91 dx = patches.x(3,1)-patches.x(2,1);
92 DX = patches.x(2,2)-patches.x(2,1);
93 rx = dx*(nx-1)/2/DX;
```

```
dy = patches.y(3,1)-patches.y(2,1);
DY = patches.y(2,2)-patches.y(2,1);
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.

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

The centre of each patch (as `nx` and `ny` are odd) is at

```
i0 = round((nx+1)/2);
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.

```
if patches.ordCC>0 % then non-spectral interpolation
error('non-spectral interpolation not yet implemented')
  dmu=nan(patches.ordCC,nPatch,nVars);
%  if patches.alt % use only odd numbered neighbours
%    dmu(1,:,:)=(u(i0,jp,:)+u(i0,jm,:))/2; % \mu
%    dmu(2,:,:)= u(i0,jp,:)-u(i0,jm,:); % \delta
%    jp=jp(jp); jm=jm(jm); % increase shifts to \pm2
%  else % standard
    dmu(1,:,:)=(u(i0,jp,:)-u(i0,jm,:))/2; % \mu\delta
    dmu(2,:,:)=(u(i0,jp,:)-2*u(i0,j,:)+u(i0,jm,:)); % \delta^2
%  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).

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

```
  u(nSubP,j,:)=u(i0,j,:)*(1-patches.alt) ...
    +sum(bsxfun(@times,patches.Cwtsr,dmu));
  u( 1,j,:)=u(i0,j,:)*(1-patches.alt) ...
    +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 `ks` $= k2\pi/N$ for 'wavenumbers' $k = (0, 1, \ldots, k_{\max}, -k_{\max}, \ldots, -1)$ for odd N, and $k = (0\,, 1\,, \ldots, k_{\max}\,, \pm(k_{\max}+1) - k_{\max}\,, \ldots, -1)$ for even N.

```
179 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.

```
189 %  if patches.alt % transform by doubling the number of fields
190 %  error('staggered grid not yet implemented')
191 %    v=nan(size(u)); % currently to restore the shape of u
192 %    u=cat(3,u(:,1:2:nPatch,:),u(:,2:2:nPatch,:));
193 %    altShift=reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
194 %    iV=[nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
195 %    r=r/2;           % ratio effectively halved
196 %    nPatch=nPatch/2; % halve the number of patches
197 %    nVars=nVars*2;   % double the number of fields
198 %  else % the values for standard spectral
199     altShift = 0;
200     iV = 1:nVars;
201 %  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, $k = (0\,, 1\,, \ldots, k_{\max}\,, +(k_{\max}+1) - k_{\max}\,, \ldots, -1)$.

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

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

```
220   if imag(u(i0,j0,:,:,:))==0, uclean = @(u) real(u);
221                             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.

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

The inverse Fourier transform gives the edge values via a shift a fraction `rx`/`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.

```
241 nFTx = 2-mod(Nx,2);
242 nFTy = 2-mod(Ny,2);
243 unj = nan(1,ny,Nx,Ny,nVars,nFTx*nFTy);
```

```
u1j = nan(1,ny,Nx,Ny,nVars,nFTx*nFTy);
uin = nan(nx,1,Nx,Ny,nVars,nFTx*nFTy);
ui1 = nan(nx,1,Nx,Ny,nVars,nFTx*nFTy);
```

Loop over the required IFFTs.

```
iFT = 0;
for iFTx = 1:nFTx
for iFTy = 1:nFTy
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??)

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

Second interpolate onto y-limits of the patches.

```
for i = 1:nx
  ks = (i-i0)*2/(nx-1)*krx; % fraction of krx along the edge
  uin(i,1,:,:,iV,iFT) = ifft2( bsxfun(@times,Ck ...
      ,exp(1i*bsxfun(@plus,ks',altShift+kry))));
  ui1(i,1,:,:,iV,iFT) = ifft2( bsxfun(@times,Ck ...
      ,exp(1i*bsxfun(@plus,ks',altShift-kry))));
end
```

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

```
if nFTy==2, kry(Ny/2+1) = -kry(Ny/2+1); end
end% iFTy-loop
if nFTx==2, krx(Nx/2+1) = -krx(Nx/2+1); end
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()`.

```
u(end,:,:,:,iV) = uclean( mean(unj,6) );
u( 1 ,:,:,:,iV) = uclean( mean(u1j,6) );
u(:,end,:,:,iV) = uclean( mean(uin,6) );
u(:, 1 ,:,:,iV) = uclean( mean(ui1,6) );
```

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

```
%if patches.alt
%  nVars=nVars/2;  nPatch=2*nPatch;
%  v(:,1:2:nPatch,:)=u(:,:,1:nVars);
%  v(:,2:2:nPatch,:)=u(:,:,nVars+1:2*nVars);
```

```
%  u=v;
%end
end% if spectral
end% function patchEdgeInt2
```

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

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

Section contents

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.

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

4.10.1 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.

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

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

```
small = 1;
jac = nan(length(i));
sizeJacobian = size(jac)
for j = 1:length(i)
  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))]
```

4.10.2 Execute a simulation

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

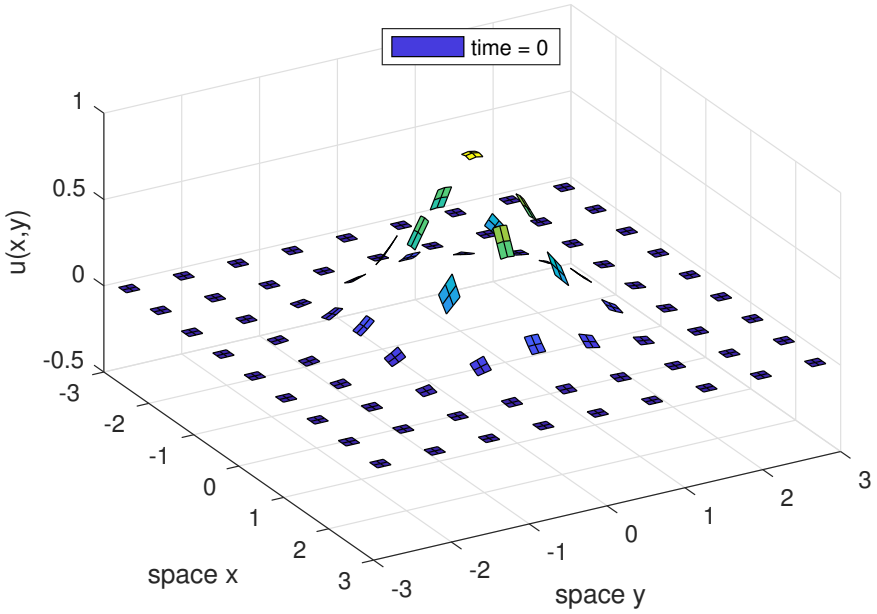
```
108 x = reshape(patches.x,nSubP,1,[],1);
109 y = reshape(patches.y,1,nSubP,1,[]);
110 u0 = exp(-x.^2-y.^2);
111 v0 = zeros(size(u0));
```

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 4.13 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,'paperposition',[0 0 14 10])
132 print('-depsc','wave2Dic')
```

Integrate in time using standard functions.

Figure 4.15: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to the simple wave PDE: Figure 4.16 plots the computed field at time $t = 2$.




```
disp('Wait while we simulate u_t=v, v_t=u_xx+u_yy')
[ts,uvs] = ode15s(@patchSmooth2,[0 2],[u0(:);v0(:)]);
```

Animate the computed simulation to end with Figure 4.16. Because of the very small time-steps, subsample to plot at most 200 times.

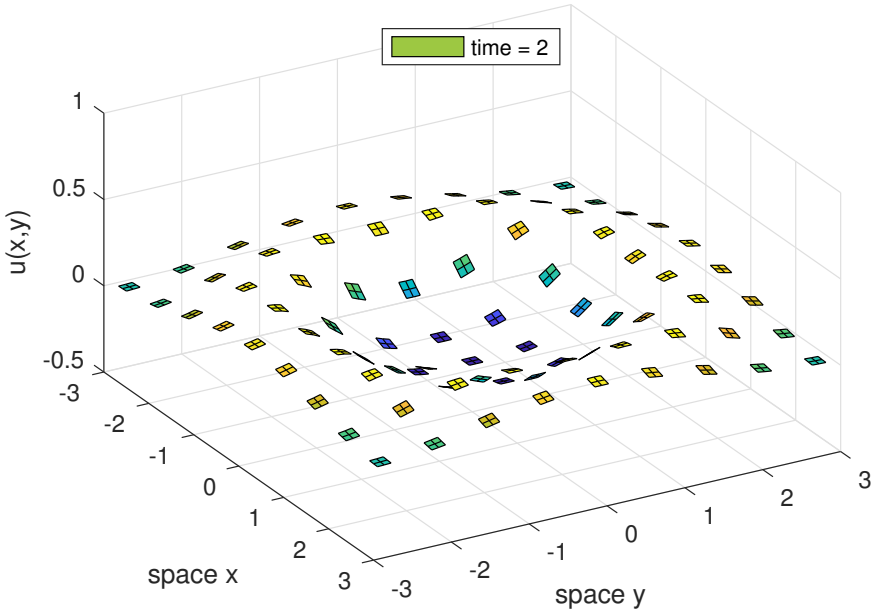
```
di = ceil(length(ts)/200);
for i = [1:di:length(ts)-1 length(ts)]
  uv = patchEdgeInt2(uvs(i,:));
  uv = reshape(permute(uv,[1 3 2 4 5]), [numel(x) numel(y) 2]);
  usurf.ZData = uv(:,:,1)';
  legend(['time = ' num2str(ts(i),2)])
  pause(0.1)
end
print('-depsc',['wave2Dt' num2str(ts(end))])
```

4.10.3 Example of simple wave PDE inside patches

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

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

Figure 4.16: field $u(x, y, t)$ at time $t = 2$ of the patch scheme applied to the simple wave PDE with initial condition in Figure 4.15.




```
191 end
```

4.11 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.

4.12 Miscellaneous tests

4.12.1 `patchEdgeInt1test`: test the spectral interpolation

Subsection contents

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

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

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

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

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

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

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

```
for k=1:nPatch/2
  u0=sin(k*patches.x*2*pi/Len);
  v0=cos(k*patches.x*2*pi/Len);
  uvi=patchEdgeInt1([u0(:);v0(:)]);
  normuError=norm(uvi(:,:,1)-u0)*norm(u0(i0,:));
  normvError=norm(uvi(:,:,2)-v0)*norm(v0(i0,:));
  if abs(normuError)+abs(normvError)>2e-13
    normuError=normuError, normvError=normvError
    error(['failed double field interpolation k=' num2str(k)])
  end
end
```

End the for-loop over various geometries.

```
end
```

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

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

Identify which microscale grid points are h or u values.

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

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

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

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

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

```
137       error(['failed double field interpolation k=' num2str(k)])
138     end
139   end
```

End for-loop over patches

```
146   end
```

Finish If no error messages, then all OK.

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

4.13 patchEdgeInt2test: tests 2D spectral interpolation

Try 99 realisations of random tests.

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

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

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

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

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

Copy and NaN the edges, then interpolate

```
52 u=u0s; u([1 end],:,:,:,:)=nan; u(:,[1 end],:,:,:)=nan;
53 u=patchEdgeInt2(u(:));
```

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

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

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 directories in the repository, say `Dir`.
- Create a LaTeX file `Dir/funs.tex`: establish as one LaTeX section that `\input{Dir/*.m}`s the files of the functions in the class, example scripts of use, and possibly test scripts, Table A.1.
- Each such `Dir/funs.tex` file is to be included from the main LaTeX file `Doc/eqnFreeDevMan.tex` so that people can most easily work on one section at a time:
  - put `\include{funs}` into `Doc/eqnFreeDevMan.tex`;
  - to include we have to use a soft link so at the command line in the directory `Doc` execute `ln -s ../Dir/funs.tex` [1]
- Each toolbox function is documented as a separate section, 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])
  print('-depsc2',filename)
  ```

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

[1] Such soft links are necessary for at least my Mac OSX and hopefully will work for other developers. Further, it has the advantage that auxiliary files are also located in the `Doc` directory.

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

```
% input *.m files for ... Author, date
%!TEX root = ../Doc/eqnFreeDevMan.tex
\chapter{...}
\label{sec:...}
\localtableofcontents
introduction...
\input{../Dir/fun.m}
\input{../Dir/funExample.m}
...
\begin{devMan}
\section{To do}
...
\section{Miscellaneous tests}
\input{../Dir/funTest.m}
...
\end{devMan}
```

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

```
% Short explanation for users typing "help fun"
% Author, date
%!TEX root = ../Doc/eqnFreeDevMan.tex
%{
\section{\texttt{...}: ...}
\label{sec:...}
\localtableofcontents
Overview LaTeX explanation.
\begin{matlab}
%}
function ...
%{
\end{matlab}
\paragraph{Input} ...
\paragraph{Output} ...
\begin{devMan}
Repeated as desired:
LaTeX between end-matlab and begin-matlab
\begin{matlab}
%}
Matlab code between %} and %{
%{
\end{matlab}
Concluding LaTeX before following final lines.
\end{devMan}
%}
```

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

Chapter contents

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

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

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

B.7 Macroscale closure

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

Gear, C. W., Kaper, T. J., Kevrekidis, I. G. & Zagaris, A. (2005), 'Projecting to a slow manifold: Singularly perturbed systems and legacy codes', *SIAM Journal on Applied Dynamical Systems* **4**(3), 711–732.

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

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

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

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

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

Gustafsson, B. (1975), 'The convergence rate for difference approximations to mixed initial boundary value problems', *Mathematics of Computation* **29**(10), 396–406.

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

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

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

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

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

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

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

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

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

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

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

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.

Zagaris, A., Vandekerckhove, C., Gear, C. W., Kaper, T. J. & Kevrekidis, I. G. (2012), 'Stability and stabilization of the constrained runs schemes for equation-free projection to a slow manifold', *DCDS-A* **32**, 2759–2803.