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

A. J. Roberts* John Maclean† J. E. Bunder‡

December 11, 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

Abstract

This 'equation-free toolbox' empowers the computer-assisted analysis of complex, multiscale systems. Its aim is to enable you to use microscopic simulators to perform system level tasks and analysis, because microscale simulations are often the best available description of a system. The methodology bypasses the derivation of macroscopic evolution equations by computing only short bursts of of the microscale simulator (Kevrekidis & Samaey 2009, Kevrekidis et al. 2004, 2003, e.g.), and often only computing on small patches of the spatial domain (Roberts et al. 2014, e.g.). This suite of functions empowers users to start implementing such methods in their own applications. Download via `https://github.com/uoa1184615/EquationFreeGit`

Contents

1 Introduction

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

Users Download via https://github.com/uoa1184615/EquationFreeGit. Place the folder of this toolbox in a path searched by MATLAB/Octave. Then read the section(s) that documents the function of interest.

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

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

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

- In 1D adapt the code at the beginning of `configPatches1.m` for Burgers' PDE (Section 3.2.2), or the staggered patches of 1D water wave equations in `waterWaveExample.m` (Section 3.9).
- in 2D adapt the code at the beginning of `configPatches2.m` for non-linear diffusion (Section 3.11.2), or the regular patches of the 2D wave equation of `wave2D.m` (Section 3.14).
- The above two are for systems that have *smooth* spatial structures on the microscale: when the microscale is 'rough' with a known period (so far only in 1D), then adapt the example of `HomogenisationExample.m` (Section 3.5).

Verification Most of these schemes have proven 'accuracy' when compared to the underlying specified microscale system. In the spatial patch schemes, we measure 'accuracy' by the order of consistency between macroscale dynamics and the specified microscale.

- Roberts & Kevrekidis (2007) and Roberts et al. (2014) proved reasonably general high-order consistency for the 1D and 2D patch schemes, respectively.
- In wave-like systems, Cao & Roberts (2016*b*) established high-order consistency for the 1D staggered patch scheme.
- A heterogeneous microscale is more difficult, but Bunder et al. (2017) showed good accuracy in a variety of circumstances, for appropriately chosen parameters.

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

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

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

Matlab appears a good choice for a first version since it is widespread, efficient, supports various parallel modes, and development costs are reasonably low. Further it is built on blas and lapack so the cache and superscalar cpu are potentially well utilised. We aim to develop functions that work for Matlab/Octave. Appendix A outlines some details for contributors.

2 Projective integration of deterministic ODEs

Chapter contents

2.1 Introduction

This section provides some good projective integration functions (Gear & Kevrekidis 2003*b*,*c*, Givon et al. 2006, Marschler et al. 2014, Maclean & Gottwald 2015, Sieber et al. 2018, e.g.). The goal is to enable computationally expensive multiscale dynamic simulations/integrations to efficiently compute over very long time scales.

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

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

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

Petersik (2019–) is also developing, in python, some projective integration functions.

Main functions

- Projective Integration by second or fourth-order Runge–Kutta is implemented by `PIRK2()` or `PIRK4()` respectively. These schemes are suitable for precise simulation of the slow dynamics, provided the time period spanned by an application of the microsolver is not too large.
- Projective Integration with a General method, `PIG()`. This function enables a Projective Integration implementation of any integration method over macroscale time-steps. It does not matter whether the method is a standard Matlab/Octave algorithm, or one supplied by the user. `PIG()` should only be used directly in very stiff systems, less stiff systems additionally require `cdmc()`.
- *Constraint-defined manifold computing*, `cdmc()`, is a helper function, based on the method introduced in Gear et al. (2005*a*), that iteratively applies the microsolver and backward projection in time. The result is to project the fast variables close to the slow manifold, without advancing the current time by the burst time of the microsolver. This function reduces errors related to the simulation length of the microsolver in the `PIG` function. In particular, it enables `PIG()` to be used on problems that are not particularly stiff.

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

Schematic for Projective Integration scheme

Set microsolver
Code a function that interfaces to your 'black-box' microsolver, including the burst time, `bT`, of the microsolver. Possible aids:

- Use the Patch functions (Figure 3.1) to simulate a large-scale PDE, lattice, etc.
- Use `cmdc()` as a wrapper for the microsolver if the slow variables may change significantly over the microsolver burst.

Set macro simulator
Set the vector of output times `tSpan/Tspan`, and set initial values `x0`.

- If using `PIRKn()`, then intervals between times are the projective time-steps.
- If using `PIG()`, then intervals between times are as needed by `macroInt`.

Do Projective Integration Invoke the appropriate Projective Integration function as, e.g., `x = PIRK2(microBurst, tSpan, x0)`, or `[t,x] = PIG(macroInt, microBurst, Tspan, x0)`. Additional optional outputs inform you of the microscale.

Set lifting/restriction If needed, set functions `restrict()` and `lift()` to convert between macro and micro problems/variables. These are optional arguments to the Projective Integration functions.

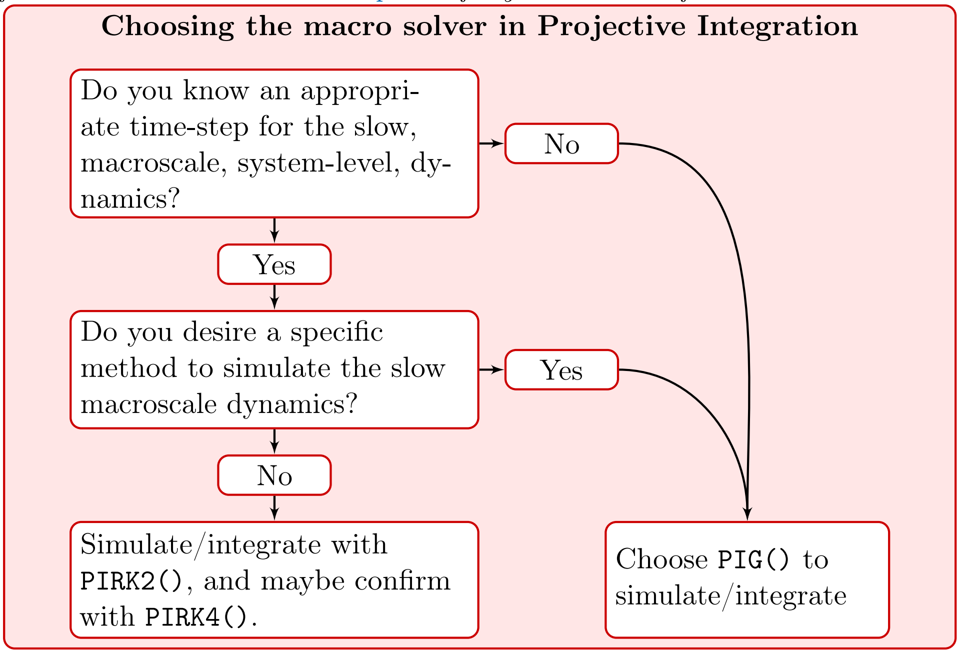
The above functions share dependence on a user-specified *microsolver* that accurately simulates some problem of interest.

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

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

Section contents

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



2.2.1 Introduction

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

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

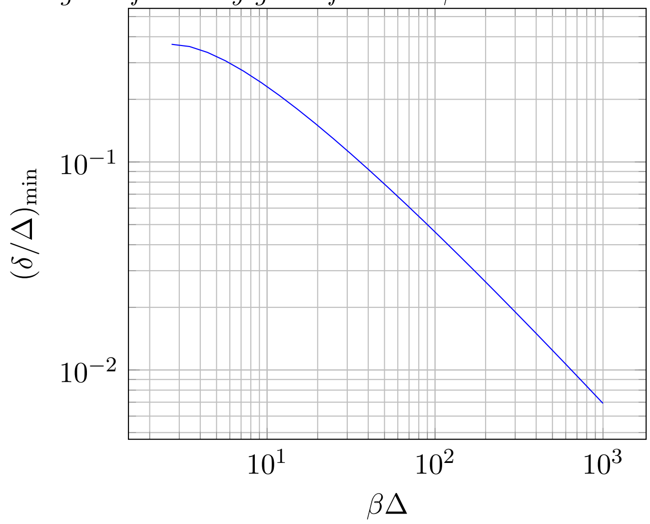
Input If there are no input arguments, then this function applies itself to the Michaelis–Menton example: see the code in Section 2.2.2 as a basic template of how to use.

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

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

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

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



- `tSpan` is an ℓ-vector of times at which the user requests output, of which the first element is always the initial time. `PIRK2()` does not use adaptive time-stepping; the macroscale time-steps are (nearly) the steps between elements of `tSpan`.
- `x0` is an n-vector of initial values at the initial time `tSpan(1)`. Elements of `x0` may be `NaN`: such `Nan`s are carried in the simulation through to the output, and often represent boundaries/edges in spatial fields.
- `bT`, *optional*, either missing, or empty (`[]`), or a scalar: if a given scalar, then it is the length of the micro-burst simulations—the minimum amount of time needed for the microscale simulation to relax to the slow manifold; else if missing or `[]`, then `microBurst()` must itself determine the length of a burst.

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

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

1. a macroscale time-step, $\Delta =$ `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 2.3.

Output If there are no output arguments specified, then a plot is drawn of the computed solution `x` versus `tSpan`.

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

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

- `tms`, optional, is an L dimensional column vector containing the microscale times within the burst simulations, each burst separated by `NaN`;
- `xms`, optional, is an $L \times n$ array of the corresponding microscale states—each rows is an accurate estimate of the state at the corresponding time `tms` and helps visualise details of the solution.
- `rm`, optional, a struct containing the 'remaining' applications of the microBurst required by the Projective Integration method during the calculation of the macrostep:
  - `rm.t` is a column vector of microscale times; and
  - `rm.x` is the array of corresponding burst states.

  The states `rm.x` do not have the same physical interpretation as those in `xms`; the `rm.x` are required in order to estimate the slow vector field during the calculation of the Runge–Kutta increments, and do *not* accurately approximate the macroscale dynamics.

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

2.2.2 If no arguments, then execute an example

```
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 $y(t)$:

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

(encoded in function `MMburst()` in the next paragraph). With initial conditions $x(0) = 1$ and $y(0) = 0$, the following code computes and plots a solution over time $0 \le t \le 6$ 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 $\Delta = 1$.

```
global MMepsilon
MMepsilon = 0.05
```

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

Upon finishing execution of the example, exit this function.

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

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

```
function [ts, xs] = MMburst(ti, xi, bT)
    global MMepsilon
    dMMdt = @(t,x) [ -x(1)+(x(1)+0.5)*x(2)
        1/MMepsilon*( x(1)-(x(1)+1)*x(2) ) ];
    if ~exist('OCTAVE_VERSION','builtin')
    [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
    else % octave version
    [ts, xs] = odeOct(dMMdt, [ti ti+bT], xi);
    end
end
```

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

2.2.3 The projective integration code

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

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

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

```
nArgOut=nargout();
saveMicro = (nArgOut>1);
```

```
saveFullMicro = (nArgOut>3);
saveSvf = (nArgOut>4);
```

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

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

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

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

```
if saveMicro
    tms = cell(nT,1);
    xms = cell(nT,1);
    tms{1} = reshape(relax_t,[],1);
    xms{1} = relax_x0;
    if saveFullMicro
        rm.t = cell(nT,1);
        rm.x = cell(nT,1);
        if saveSvf
            svf.t = nan(2*nT-2,1);
            svf.dx = nan(2*nT-2,length(x0));
        end
    end
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 macro-step.

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

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

```
    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.

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

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

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

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

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

End the main loop over all the macro-steps.

```
end
```

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

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

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

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

2.2.4 If no output specified, then plot the simulation

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

This concludes `PIRK2()`.

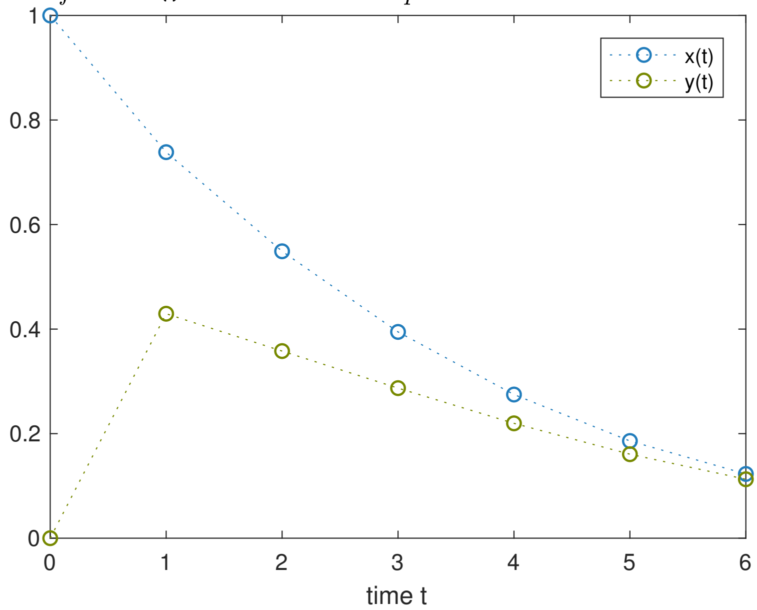
```
end
```

2.3 `egPIMM`: Example projective integration of Michaelis–Menton kinetics

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

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

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



(encoded in function `MMburst()` below). As illustrated by Figure 2.5, the slow variable $x(t)$ evolves on a time scale of one, whereas the fast variable $y(t)$ evolves on a time scale of the small parameter ϵ.

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

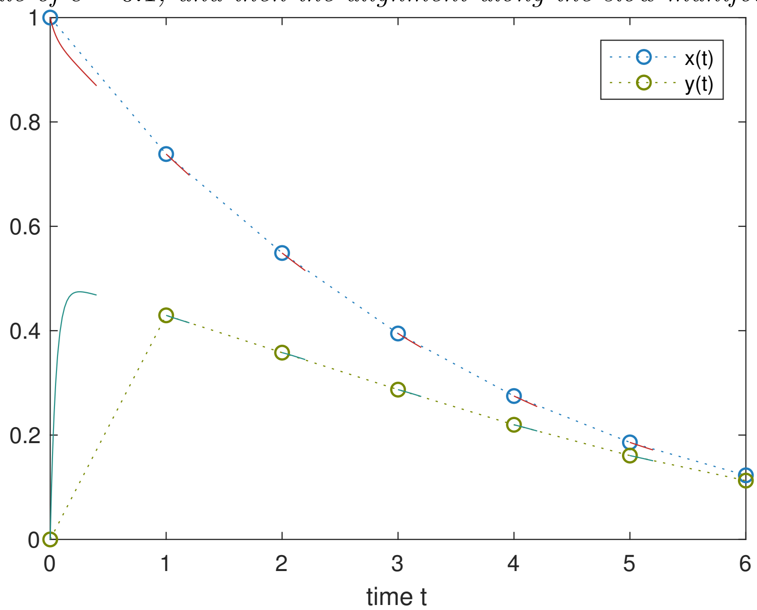
```
clear all, close all
global MMepsilon
MMepsilon = 0.1
```

First, the end of this section encodes the computation of bursts of the Michaelis–Menten system in a function `MMburst()`. Second, here set macroscale times of computation and interest into vector `ts`. Then, invoke Projective Integration with `PIRK2()` applied to the burst function, say using bursts of simulations of length 2ϵ, and starting from the initial condition for the Michaelis–Menten system, at time $t = 0$, of $(x, y) = (1, 0)$ (off the slow manifold).

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

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

Figure 2.5: Michaelis–Menten enzyme kinetics simulated with the projective integration of `PIRK2()`*: the microscale bursts show the initial transients on a time scale of $\epsilon = 0.1$, and then the alignment along the slow manifold.*



Request and plot the microscale bursts Because the initial conditions of the simulation are off the slow manifold, the initial macroscale step appears to 'jump' (Figure 2.4). In order to see the initial transient attraction to the slow manifold we plot some microscale data in Figure 2.5. Two further output variables provide this microscale burst information.

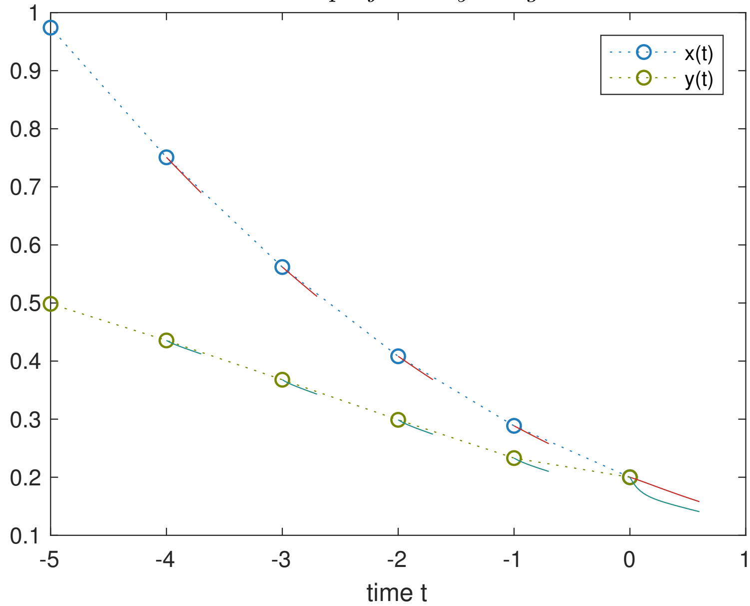
```
[xs,tMicro,xMicro] = PIRK2(@MMburst, ts, [1;0], 2*MMepsilon);
figure, plot(ts,xs,'o:',tMicro,xMicro)
xlabel('time t'), legend('x(t)','y(t)')
pause(1)
```

Figure 2.5 plots the macroscale and microscale results—also showing that the initial burst is by default twice as long. Observe the slow variable $x(t)$ is also affected by the initial transient (hence other schemes which 'freeze' slow variables are less accurate).

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

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

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



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

```
function [ts, xs] = MMburst(ti, xi, bT)
    global MMepsilon
    dMMdt = @(t,x) [ -x(1)+(x(1)+0.5)*x(2)
        1/MMepsilon*( x(1)-(x(1)+1)*x(2) ) ];
    if ~exist('OCTAVE_VERSION','builtin')
    [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
    else % octave version
    [ts, xs] = odeOct(dMMdt, [ti ti+bT], xi);
    end
end
```

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

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

Section contents

2.4.1 Introduction

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

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

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

Inputs:

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

  `[Ts,Xs] = macroInt(F,Tspan,X0)`

  where

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- `T`, an L-vector of times at which `macroInt` produced results.

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

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

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

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

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

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

2.4.2 If no arguments, then execute an example

```
if nargin==0
```

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

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

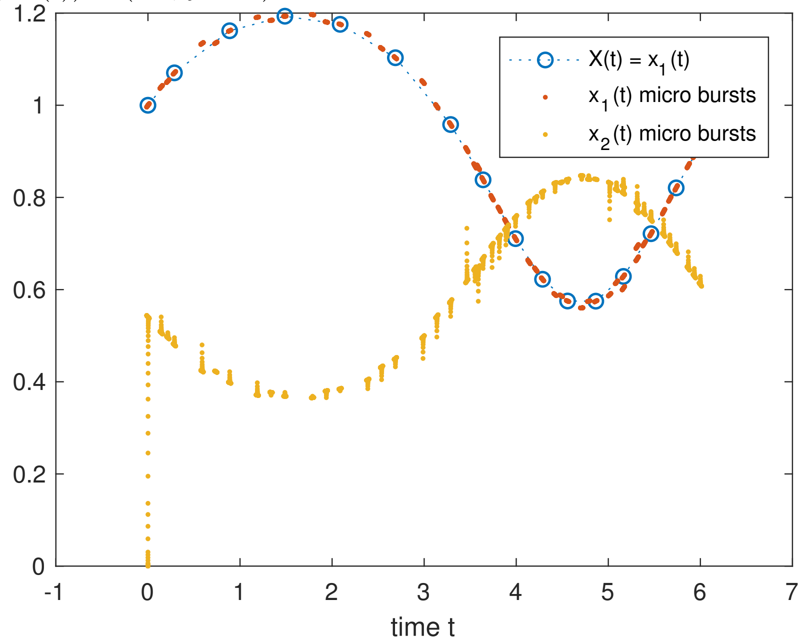
The macroscale variable is $X(t) = x_1(t)$, and the evolution dX/dt is unclear. With initial condition $X(0) = 1$, the following code computes and plots a solution of the system (2.1) over time $0 \le t \le 6$ for parameter $\epsilon = 10^{-3}$(Figure 2.7). Whenever needed by `microBurst()`, the microscale system (2.1) is initialised (‘`lifted`’) using $x_2(t) = x_2^{\text{approx}}$ (yellow dots in Figure 2.7).

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

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

Second, we code microscale bursts, here using the standard `ode45()`. We choose a burst length $2\epsilon\log(1/\epsilon)$ as the rate of decay is $\beta \approx 1/\epsilon$ but 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.

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




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

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

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

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

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

Plot output of this projective integration.

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

Upon finishing execution of the example, exit this function.

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

2.4.3 The projective integration code

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

```
if nargin < 5 || isempty(restrict)
    lift=@(X,xApprox) X;
    restrict=@(x) x;
end
```

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

```
nArgOut = nargout();
saveMicro = (nArgOut>2);
saveSvf = (nArgOut>4);
```

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

```
nT = length(Tspan)-1;
nx = length(x0);
nX = length(restrict(x0));
```

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

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

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

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

Update the initial time.

```
Tspan(1) = relaxT(end);
```

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

```
if saveMicro
    tms=cell(nT+1,1); xms=cell(nT+1,1);
    n=1;
    tms{n} = reshape(relaxT,[],1);
    xms{n} = x0MicroRelax;

    if saveSvf
        svf.T = cell(nT+1,1);
        svf.dX = cell(nT+1,1);
    else
        svf = [];
    end
else
    tms = []; xms = []; svf = [];
end
```

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

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

Run a microBurst from the given macroscale initial values.

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

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

```
  X2 = restrict(xMicroTmp(end,:));
  X1 = restrict(xMicroTmp(end-1,:));
  dt = tTmp(end)-tTmp(end-1);
  dXdt = (X2 - X1).'/dt;
```

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

```
  if saveMicro
      n=n+1;
      tms{n} = [reshape(tTmp,[],1); nan];
      xms{n} = [xMicroTmp; nan(1,nx)];
      if saveSvf
          svf.T{n-1} = t;
          svf.dX{n-1} = dXdt;
      end
```

```
  end
end% PIFun function
```

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

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

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

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

Concatenate all the additional requested outputs into arrays.

```
if saveMicro
    tms = cell2mat(tms);
    xms = cell2mat(xms);
    if saveSvf
        svf.T = cell2mat(svf.T);
        svf.dX = cell2mat(svf.dX);
    end
end
```

2.4.4 If no output specified, then plot the simulation

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

This concludes `PIG()`.

```
end
```

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

Section contents

2.5.1 Introduction

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

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

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

If no arguments, then execute an example

```
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 $\Delta = -1$.

```
global MMepsilon
MMepsilon = 0.1
ts = 0:-1:-5
bT = MMepsilon*log(abs(ts(2)-ts(1))/MMepsilon)
[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
```

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

```
function [ts, xs] = MMburst(ti, xi, bT)
    global MMepsilon
    dMMdt = @(t,x) [ -x(1)+(x(1)+0.5)*x(2)
        1/MMepsilon*( x(1)-(x(1)+1)*x(2) ) ];
    if ~exist('OCTAVE_VERSION','builtin')
    [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
```

```
    else % octave version
    [ts, xs] = odeOct(dMMdt, [ti ti+bT], xi);
    end
end
```

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

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

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

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

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

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

```
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 up to four optional outputs of the microscale bursts.

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

- `xms`, optional, is an $L \times n$ array of the corresponding microscale states—each rows is an accurate estimate of the state at the corresponding time `tms` and helps visualise details of the solution.

- `rm`, optional, a struct containing the 'remaining' applications of the microBurst required by the Projective Integration method during the calculation of the macrostep:

  - `rm.t` is a column vector of microscale times; and
  - `rm.x` is the array of corresponding burst states.

  The states `rm.x` do not have the same physical interpretation as those in `xms`; the `rm.x` are required in order to estimate the slow vector field during the calculation of the Runge–Kutta increments, and do *not* accurately approximate the macroscale dynamics.

- `svf`, optional, a struct containing the Projective Integration estimates of the slow vector field.

  - `svf.t` is a 4ℓ 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 $4\ell \times n$ array containing the estimated slow vector field.

2.5.2 The projective integration code

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

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

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

```
nArgOut = nargout();
saveMicro = (nArgOut>1);
saveFullMicro = (nArgOut>3);
saveSvf = (nArgOut>4);
```

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

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

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

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

```
if saveMicro
    tms = cell(nT,1);
    xms = cell(nT,1);
    tms{1} = reshape(relax_t,[],1);
    xms{1} = relax_x0;
    if saveFullMicro
        rm.t = cell(nT,1);
        rm.x = cell(nT,1);
        if saveSvf
            svf.t = nan(4*nT-4,1);
            svf.dx = nan(4*nT-4,length(x0));
        end
    end
end
```

Loop over the macroscale time-steps

```
for jT = 2:nT
    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.

```
    if ~isempty(bT) && 4*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));
        t3=nan; t4=nan; xm3=xm2; xm4 = xm2; dx1=xm2; dx2=xm2;
    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.

```
    [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;
```

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()`).

```
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.

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

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

    xint = xm1(end,:) + (Dt-abT)*dx3;
    [t4,xm4] = microBurst(T+Dt, xint, bT);
    del = t4(end)-t4(end-1);
    dx4 = (xm4(end,:)-xm4(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.

```
    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.

```
    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.

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

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

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

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

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

End of the main loop of all macro-steps.

```
end
```

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

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

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

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

2.5.3 If no output specified, then plot the simulation

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

This concludes `PIRK4()`.

```
end
```

2.6 `cdmc()`: constraint defined manifold computing

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

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

Input

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

Output

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

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

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

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

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

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

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

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

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

2.7 Example: PI using Runge–Kutta macrosolvers

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

Clear workspace and set a seed.

```
clear
rand('seed',1) % albeit discouraged in Matlab
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,

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

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

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

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

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

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

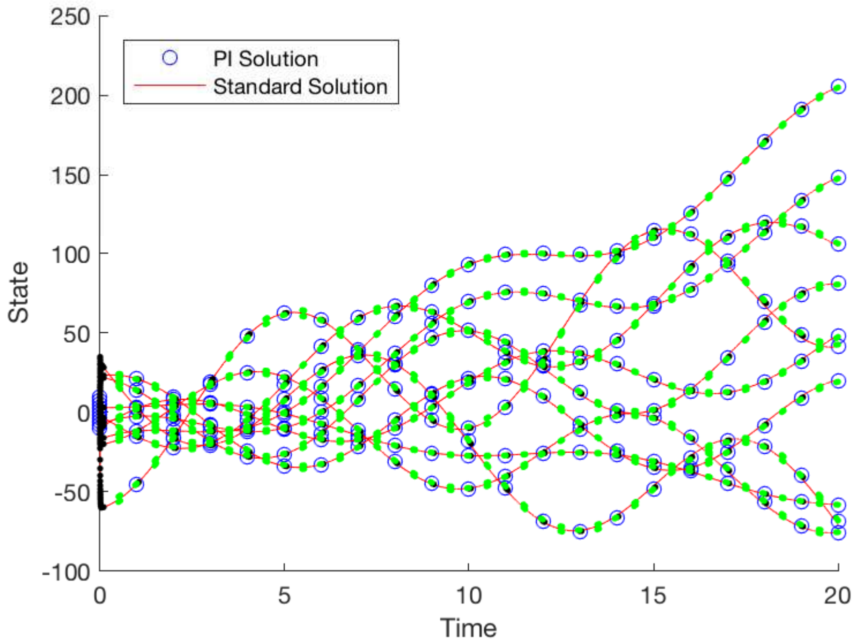
```
tSpan = 0:1:20;
x0 = linspace(-10,10,10)';
```

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).

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

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

Figure 2.8: Demonstration of PIRK4(). From initial conditions, the system rapidly transitions to an attracting invariant manifold. The PI *solution accurately tracks the evolution of the variables over time while requiring only a fraction of the computations of the standard solver.*




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

Figure 2.8 plots the output.

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

Save plot to a file.

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

A micro-burst simulation Used by `PIRKexample.m`. Code the micro-burst function using simple Euler steps. As a rule of thumb, the time-steps `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

```
function [ts, xs] = linearBurst(ti, xi, varargin)
global dxdt
dt = 0.001;
ts = ti+(0:dt:0.05)';
nts = length(ts);
xs = NaN(nts,length(xi));
xs(1,:)=xi;
for k=2:nts
    xi = xi + dt*dxdt(ts(k),xi.').';
    xs(k,:)=xi;
end
end
```

2.8 Example: Projective Integration using General macrosolvers

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

```
clear all, close all
```

Set time scale separation and the underlying ODEs:

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

```
epsilon = 1e-4;
dxdt=@(t,x) [ cos(x(1))*sin(x(2))*cos(t)
              (cos(x(1))-x(2))/epsilon ];
```

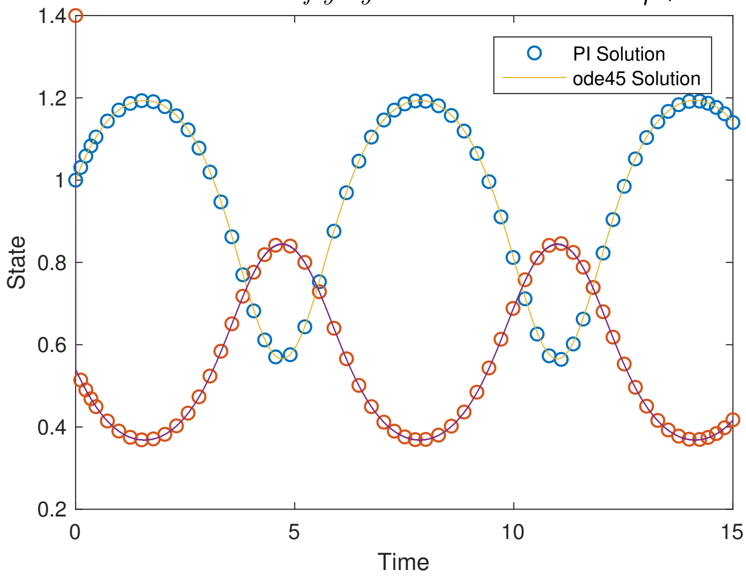
Set the 'black-box' microsolver to be an integration using a standard solver, and set the standard time of simulation for the microsolver.

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

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

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

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



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

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

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

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

figure
h = plot(ts,xs,'o', tClassic,xClassic,'-');
legend(h([1 3]),'Pro Int method','classic method')
xlabel('Time'), ylabel('State')
if ~exist('OCTAVE_VERSION','builtin')
set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
%print('-depsc2','PIGExample')
end
```

Figure 2.9 plots the output.

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

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

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

2.9 Explore: Projective Integration using constraint-defined manifold computing

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

```
clear all, close all
```

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

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

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

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

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

```
x0 = [1 0];
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.

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

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

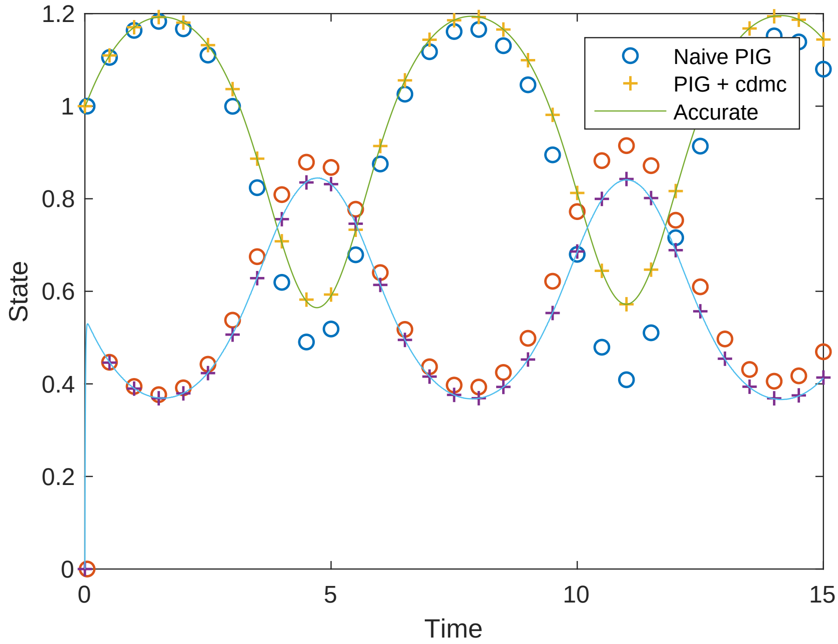


Figure 2.10 plots the output.

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

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

2.10 To do/discuss

- Implement lifting and restriction for `PIRKn()` functions.

[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.

- Could implement Projective Integration by ‘arbitrary’ Runge–Kutta scheme; that is, by having the user input a particular Butcher table—surely only specialists would be interested.
- Can maybe implement microsolvers that terminate a burst when the fast dynamics have settled using, for example, the ’Events’ function handle in ode23.
- Need projective integration of systems with fast oscillations, perhaps by DMD.
- Need projective integration for stochastic systems.

3 Patch scheme for given microscale discrete space system

Chapter contents

3.1 Introduction

Consider spatio-temporal multiscale systems where the spatial domain is so large that a given microscale code cannot be computed in a reasonable time. The *patch scheme* computes the microscale details only on small patches of the space-time domain, and produce correct macroscale predictions by craftily coupling the patches across unsimulated space (Hyman 2005, Samaey et al. 2005, 2006, Roberts & Kevrekidis 2007, Liu et al. 2015, e.g.). The resulting macroscale predictions were generally proved to be consistent with the microscale dynamics, to some specified order of accuracy, in a series of papers: 1D-space dissipative systems (Roberts & Kevrekidis 2007, Bunder et al. 2017); 2D-space dissipative systems (Roberts et al. 2014); and 1D-space wave-like systems (Cao & Roberts 2016*b*).

The microscale spatial structure is to be on a lattice such as obtained from finite difference/element/volume approximation of a PDE. The microscale is either continuous or discrete in time.

Quick start See Sections 3.2.2 and 3.11.2 which respectively list example basic code that uses the provided functions to simulate the 1D Burgers' PDE, and a 2D nonlinear 'diffusion' PDE. Then see Figure 3.1.

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

Section contents

3.2.1 Introduction

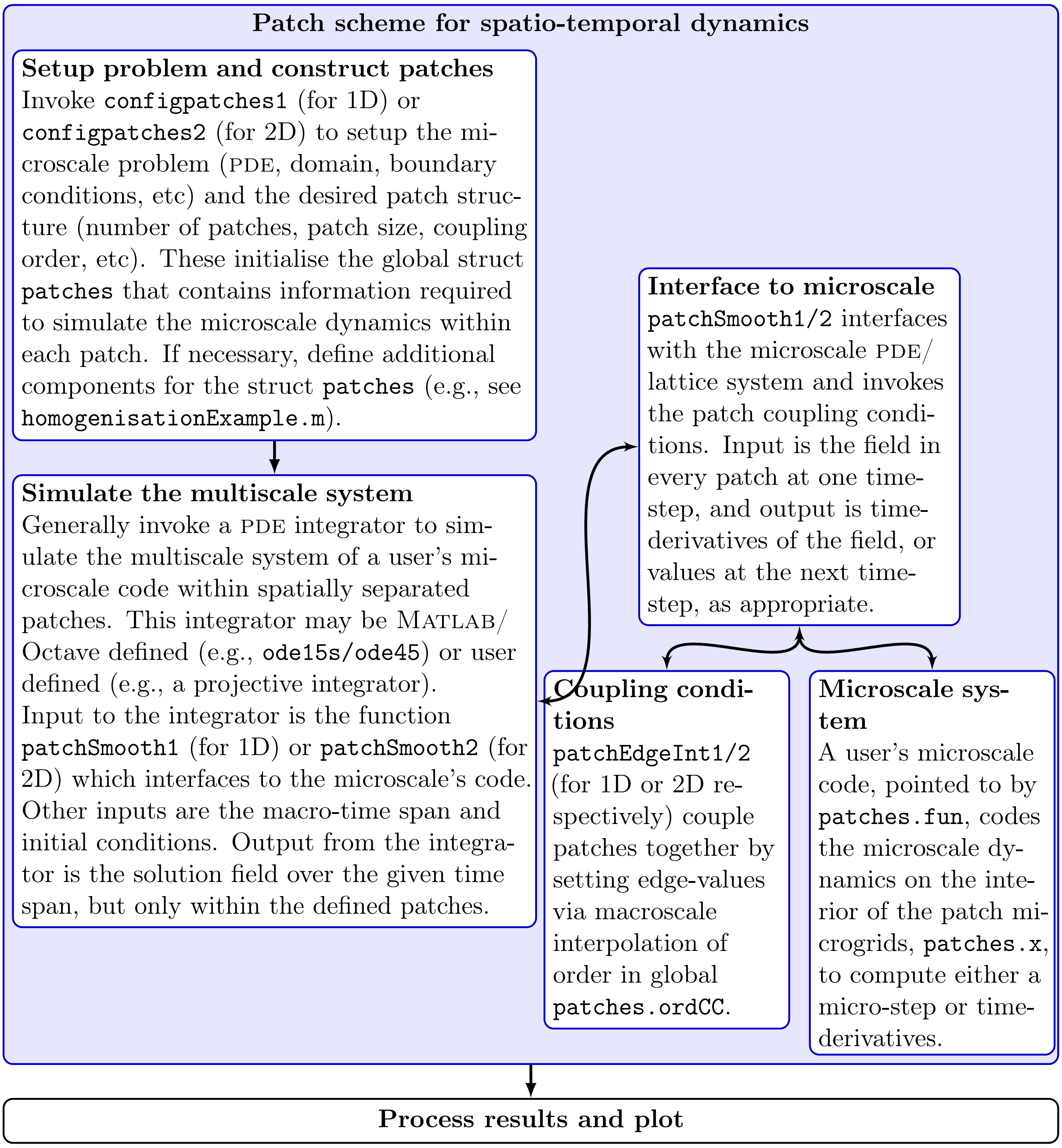
Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSmooth1()`. Section 3.2.2 lists an example of its use.

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

Input If invoked with no input arguments, then executes an example of simulating Burgers' PDE—see Section 3.2.2 for the example code.

- `fun` is the name of the user function, `fun(t,u,x)`, that computes time derivatives (or time-steps) of quantities on the patches.
- `Xlim` give the macro-space spatial 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 spatial domain.
- `nPatch` is the number of equi-spaced spaced patches.
- `ordCC`, must be ≥ -1, is the 'order' of interpolation across empty space of the macroscale mid-patch values to the edge of the patches for inter-patch coupling: where `ordCC` of 0 or -1 gives spectral interpolation; and `ordCC` being odd is for staggered spatial grids.
- `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; `ratio` $= 1$ is overlapping patches as in holistic discretisation; and small `ratio` should greatly reduce computational time.

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



- `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` (not yet implemented), *optional*, for each patch, the number of edge values set by interpolation at the edge regions of each patch. May be omitted. The default is one (suitable for microscale lattices with only nearest neighbour interactions).
- `patches.EdgyInt`, *optional*, if non-zero then interpolate to left/right edge-values from right/left next-to-edge values. So far only implemented for spectral interpolation, `ordCC` = 0.

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

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

3.2.2 If no arguments, then execute an example

```
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 PDE
3. process results

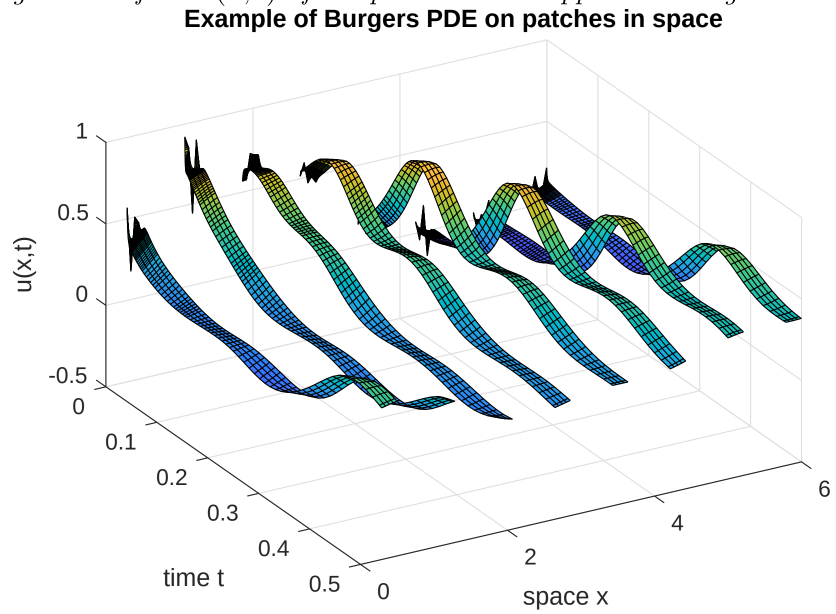
Establish global patch data struct to point to and interface with a function coding Burgers' PDE: to be solved on 2π-periodic domain, with eight patches, spectral interpolation couples the patches, each patch of half-size ratio 0.2, and with seven microscale points forming each patch.

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

Set an initial condition, with some microscale randomness.

```
u0=0.3*(1+sin(patches.x))+0.1*randn(size(patches.x));
```

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



Simulate in time using a standard stiff integrator and the interface function `patchsmooth1()` (Section 3.3).

```
if ~exist('OCTAVE_VERSION','builtin')
[ts,us] = ode15s( @patchSmooth1,[0 0.5],u0(:));
else % octave version
[ts,us] = odeOcts(@patchSmooth1,[0 0.5],u0(:));
end
```

Plot the simulation using only the microscale values interior to the patches: either set x-edges to `nan` to leave the gaps; or use `patchEdgyInt1` to re-interpolate correct patch edge values and thereby join the patches. Figure 3.2 illustrates an example simulation in time generated by the patch scheme applied to Burgers' PDE.

```
figure(1),clf
if 1, patches.x([1 end],:)=nan;  us=us.';
else us=reshape(patchEdgyInt1(us.'),[],length(ts));
end
surf(ts,patches.x(:),us), 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
```

```
function [ts,xs] = odeOcts(dxdt,tSpan,x0)
    if length(tSpan)>2, ts = tSpan;
    else ts = linspace(tSpan(1),tSpan(end),21)';
    end
    lsode_options('integration method','stiff');
    xs = lsode(@(x,t) dxdt(t,x),x0,ts);
end
```

Check and set default edgy interpolation.

```
if ~isfield(patches,'EdgyInt')
    patches.EdgyInt=0;
end
```

For compatibility, by default, do not ensemble average.

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

3.2.3 The code to make patches and interpolation

If not specified by a user, then set interpolation to compute one edge-value on each patch edge. Store in the struct `patches`.

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

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

```
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.

```
assert((ordCC>=-1) & (floor(ordCC)==ordCC), ...
    'ordCC out of allowed range integer>-2')
```

For odd `ordCC`, interpolate based upon odd neighbouring patches as is useful for staggered grids.

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

Check for staggered grid and periodic case.

```
  if patches.alt, assert(mod(nPatch,2)==0, ...
    'Require an even number of patches for staggered grid')
  end
```

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

```
patches.Cwtsr=zeros(ordCC,1);
if patches.alt  % eqn (7) in \cite{Cao2014a}
    patches.Cwtsr(1:2:ordCC)=[1 ...
      cumprod((ratio^2-(1:2:(ordCC-2)).^2)/4)./ ...
      factorial(2*(1:(ordCC/2-1)))];
    patches.Cwtsr(2:2:ordCC)=[ratio/2 ...
      cumprod((ratio^2-(1:2:(ordCC-2)).^2)/4)./ ...
      factorial(2*(1:(ordCC/2-1))+1)*ratio/2];
else %
    patches.Cwtsr(1:2:ordCC)=(cumprod(ratio^2- ...
      (((1:(ordCC/2))-1)).^2)./factorial(2*(1:(ordCC/2))-1)/ratio);
    patches.Cwtsr(2:2:ordCC)=(cumprod(ratio^2- ...
      (((1:(ordCC/2))-1)).^2)./factorial(2*(1:(ordCC/2))));
end
patches.Cwtsl=(-1).^((1:ordCC)'-patches.alt).*patches.Cwtsr;
```

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

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

Construct the microscale in each patch, assuming Dirichlet patch edges, and a half-patch length of `ratio` $\cdot$ `DX`, unless `patches.EdgyInt` is set in which case the patches are of length `ratio*DX+dx`.

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

Fin.

3.3 `patchSmooth1()`: interface to time integrators

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 mostly 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. However, we have found that microscale heterogeneous systems may be accurately simulated with this function via appropriate interpolation. Communicate patch-design variables to this function using the previously established global struct `patches` (Section 3.2).

```
function dudt=patchSmooth1(t,u)
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 3.4 describes `patchEdgeInt1()`.

```
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.

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

Fin.

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

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, or the opposite next-to-edge values (this last choice often maintains symmetry). 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). When using core averages, assumes they 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). Communicates patch-design variables via the global struct `patches`.

```
function u=patchEdgeInt1(u)
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 largely set by `configPatches1()`, and 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.
  - `.EdgyInt` in $\{0, 1\}$ is one for interpolating patch-edge values from opposite next-to-edge values (often preserves symmetry).

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.

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

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

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

If the user has not defined the patch core, then we assume it to be a single point in the middle of the patch, unless we are interpolating from next-to-edge values. For patches.nCore $\neq 1$ the half width ratio is reduced, as described by Bunder et al. (2017).

```
if ~isfield(patches,'nCore')
    patches.nCore = 1;
end
if patches.EdgyInt==0
     r = dx*(nSubP-1)/2/DX*(nSubP - patches.nCore)/(nSubP - 1);
else r = dx*(nSubP-2)/DX;
end
```

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.

```
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).

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

```
if patches.ordCC>0 % then non-spectral interpolation
  if patches.EnsAve
    uCore = sum(mean(u((i0-c):(i0+c),j,:),3),1)';
    dmu = zeros(patches.ordCC,nPatch);
  elseif patches.EdgyInt % next-to-edge values as double nVars, left first
    uCore = reshape(shiftdim(u([2 end-1],j,:),1),nPatch,[]);
    dmu = zeros(patches.ordCC,nPatch,2*nVars);
  else
    uCore = reshape(sum(u((i0-c):(i0+c),j,:),1),nPatch,nVars);
    dmu = zeros(patches.ordCC,nPatch,nVars);
  end;
  if patches.alt % use only odd numbered neighbours
    dmu(1,:,:) = (uCore(jp,:)+uCore(jm,:))/2; % \mu
    dmu(2,:,:) = (uCore(jp,:)-uCore(jm,:)); % \delta
    jp = jp(jp); jm = jm(jm); % increase shifts to \pm2
  else % standard
    dmu(1,j,:) = (uCore(jp,:)-uCore(jm,:))/2; % \mu\delta
    dmu(2,j,:) = (uCore(jp,:)-2*uCore(j,:)+uCore(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, Bunder et al. 2017), using weights computed in `configPatches1()`. Here interpolate to specified order.

In this first case of ensemble averaging, interpolate each in the ensemble.

```
if patches.EnsAve
  u(nSubP,j,:) = repmat(uCore(j)'*(1-patches.alt) ...
    +sum(bsxfun(@times,patches.Cwtsr,dmu)),[1,1,nVars]) ...
    -sum(u((nSubP-patches.nCore+1):(nSubP-1),:,:),1);
  u(1,j,:) = repmat(uCore(j)'*(1-patches.alt) ...
    +sum(bsxfun(@times,patches.Cwtsl,dmu)),[1,1,nVars]) ...
    -sum(u(2:patches.nCore,:,:),1);
```

The second case is where next-to-edge values interpolate to the opposite edge-values.

```
elseif patches.EdgyInt
  lVars=1:nVars; rVars=nVars+lVars;
  u(nSubP,j,:) = shiftdim(uCore(j,lVars),-1)*(1-patches.alt) ...
    +sum(bsxfun(@times,patches.Cwtsr,dmu(:,:,lVars)));
  u(  1  ,j,:) = shiftdim(uCore(j,rVars),-1)*(1-patches.alt) ...
    +sum(bsxfun(@times,patches.Cwtsl,dmu(:,:,rVars)));
```

Thirdly, the original, the core (one or more) of each patch interpolates to the edge action regions. When more than one in the core, the edge is set depending upon near edge values so the average near the edge is correct.

```
else
  u(nSubP,j,:) = uCore(j,:)*(1-patches.alt) ...
    + reshape(-sum(u((nSubP-patches.nCore+1):(nSubP-1),j,:),1) ...
    +sum(bsxfun(@times,patches.Cwtsr,dmu)),nPatch,nVars);
  u(1,j,:) = uCore(j,:)*(1-patches.alt) ...
    +reshape(-sum(u(2:patches.nCore,j,:),1)  ...
    +sum(bsxfun(@times,patches.Cwtsl,dmu)),nPatch,nVars);
end;
```

Case of spectral interpolation Assumes the domain is macro-periodic.

```
else% spectral interpolation
```

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' $k = (0, 1, \dots, k_{\max}, -k_{\max}, \dots, -1)$ for odd N, and $k = (0, 1, \dots, k_{\max}, (k_{\max}+1), -k_{\max}, \dots, -1)$ for even N.

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. Have not yet tested whether works for Edgy Interpolation??

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

Now set wavenumbers (when `nPatch` is even then highest wavenumber is π).

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

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

```
  if max(abs(imag(u(:))))<1e-9*max(abs(u(:)))
       uclean=@(u) real(u);
  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 if complex, treat as positive wavenumber, but if real, treat as cosine.

```
if patches.EdgyInt==0
     Cleft = fft(u(  i0   ,:,:)); Cright = Cleft;
else Cleft = fft(u(   2   ,:,:));
     Cright= fft(u(nSubP-1,:,:));
end
```

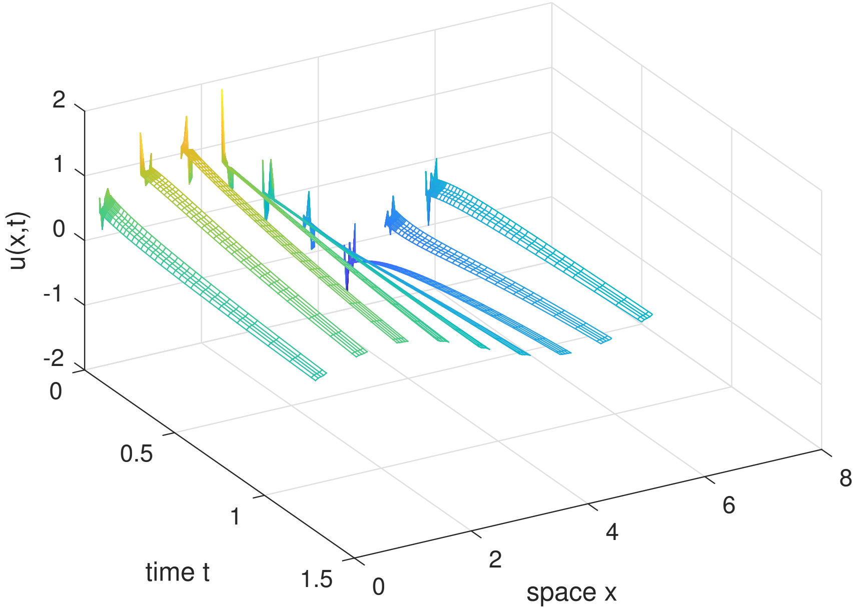
The inverse Fourier transform gives the edge values via a shift a fraction r to the next macroscale grid point. Enforce reality when appropriate.

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

Restore staggered grid when appropriate.

```
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);
```

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




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

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

3.5 `homogenisationExample`: simulate heterogeneous diffusion in 1D on patches

Section contents

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

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

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

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

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

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

3.5.1 Script to simulate via stiff or projective integration

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

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

Establish global data struct `patches` for heterogeneous diffusion on 2π-periodic domain. Use nine patches, each patch of half-size ratio 0.2. Quartic (fourth-order) interpolation `ordCC` $= 4$ provides values for the inter-patch coupling conditions.

```
global patches
nPatch = 9
ratio = 0.2
nSubP = 2*mPeriod+1
Len = 2*pi;
ordCC = 4;
configPatches1(@heteroDiff,[0 Len],nan,nPatch ...
    ,ordCC,ratio,nSubP);
```

A user may add information to `patches` in order to communicate to the time derivative function: here include the diffusivity coefficients, repeated to fill up a patch

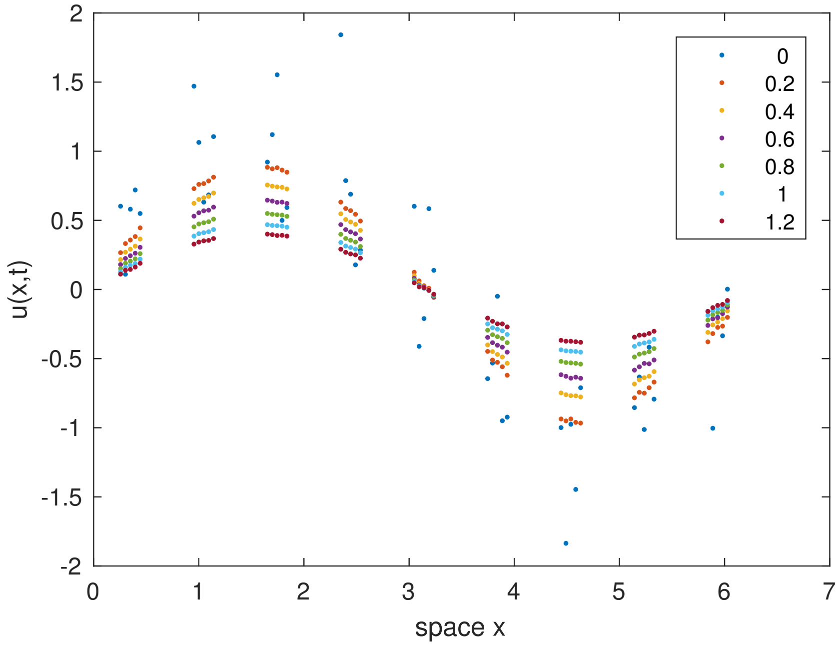
```
patches.c=repmat(cDiff,(nSubP-1)/mPeriod,1);
```

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

```
u0 = sin(patches.x)+0.4*randn(nSubP,nPatch);
if ~exist('OCTAVE_VERSION','builtin')
[ts,ucts] = ode15s(@patchSmooth1, [0 2/cHomo], u0(:));
else % octave version
[ts,ucts] = odeOcts(@patchSmooth1, [0 2/cHomo], u0(:));
end
ucts = reshape(ucts,length(ts),length(patches.x(:)),[]);
```

Plot the simulation in Figure 3.3.

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




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

The code may invoke this integration interface.

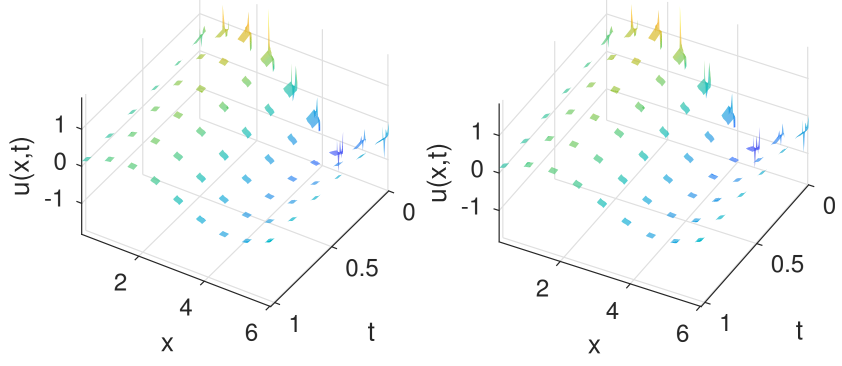
```
function [ts,xs] = odeOcts(dxdt,tSpan,x0)
    if length(tSpan)>2, ts = tSpan;
    else ts = linspace(tSpan(1),tSpan(end),21)';
    end
    lsode_options('integration method','stiff');
    xs = lsode(@(x,t) dxdt(t,x),x0,ts);
end
```

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

This second part of the script implements the following design, where the micro-integrator could be, for example, `ode45` or `rk2int`.

1. configPatches1 (done in first part)

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



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.

```
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.

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

Plot the macroscale predictions to draw Figure 3.4.

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

Also plot a surface detailing the microscale bursts as shown in the stereo Figure 3.5.

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

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

End of this example script.

3.5.2 `heteroDiff()`: heterogeneous diffusion

This function codes the lattice heterogeneous diffusion inside the patches. For 2D input arrays `u` and `x` (via edge-value interpolation of `patchSmooth1`, Section 3.3), computes the time derivative (3.1) 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`.

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

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

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

```
function [ts, ucts] = heteroBurst(ti, ui, bT)
        if ~exist('OCTAVE_VERSION','builtin')
        [ts,ucts] = ode23( @patchSmooth1,[ti ti+bT],ui(:));
        else % octave version
        [ts,ucts] = rk2Int(@patchSmooth1,[ti ti+bT],ui(:));
        end
end
```

Fin.

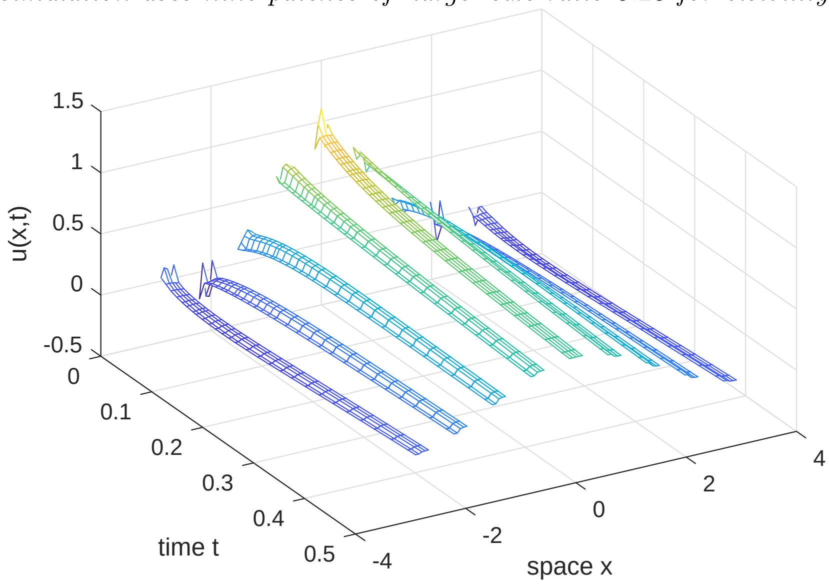
3.6 `homoDiffEdgy1`: computational homogenisation of a 1D diffusion by simulation on small patches

Figure 3.6 shows an example simulation in time generated by the patch scheme applied to macroscale diffusion propagation through a medium with microscale heterogeneity. The inter-patch coupling is realised by quartic interpolation of the patch's next-to-edge values to the patch opposite edges. Such coupling preserves symmetry in many systems, and quartic appears to be the lowest order that generally gives good accuracy.

Suppose the spatial microscale lattice is at points x_i, with constant spacing dx. With dependent variables $u_i(t)$, simulate the microscale lattice diffusion system

$$\frac{\partial u_i}{\partial t} = \frac{1}{dx^2}\delta[c_{i-1/2}\delta u_i], \tag{3.2}$$

Figure 3.6: diffusion field $u(x,t)$ of the gap-tooth scheme applied to the diffusion (3.2). The microscale random component to the initial condition, the sub-patch fluctuations, decays, leaving the emergent macroscale diffusion. This simulation uses nine patches of 'large' size ratio 0.25 *for visibility.*



in terms of the centred difference operator δ. The system has a microscale heterogeneity via the coefficients $c_{i+1/2}$ which we assume to have some given known periodicity. Figure 3.6 shows one patch simulation of this system: observe the effects of the heterogeneity within each patch.

3.6.1 Script code to simulate heterogeneous diffusion systems

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

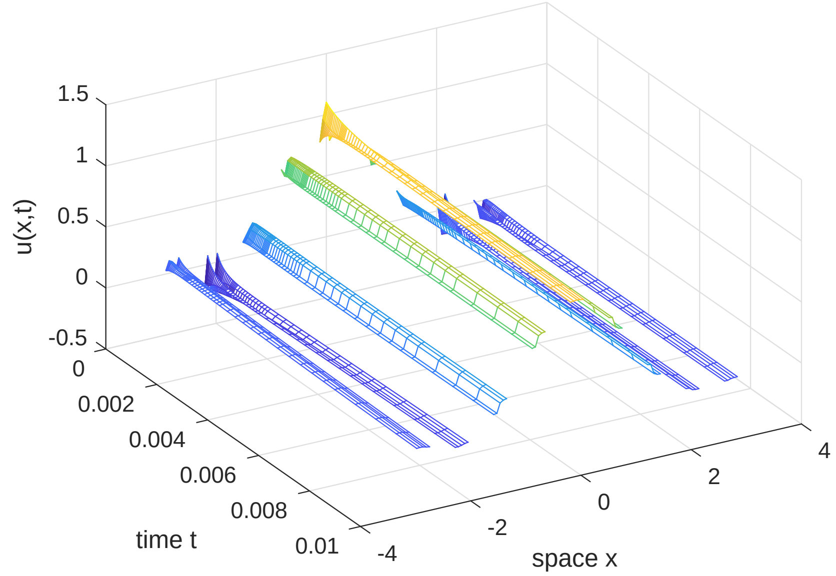
1. configPatches1, and add micro-information
2. ode15s $\leftrightarrow$ patchSmooth1 $\leftrightarrow$ heteroDiff
3. plot the simulation
4. use patchSmooth1 to explore the Jacobian

First establish the microscale heterogeneity has micro-period `mPeriod` on the lattice, and random log-normal values, albeit normalised to have harmonic mean one. This normalisation then means that macroscale diffusion on a domain of length 2π should have near integer decay rates, the squares of $0, 1, 2, \ldots$. Then the heterogeneity is repeated `nPeriodsPatch` times within each patch.

```
clear all
mPeriod = 3
cHetr = exp(1*randn(mPeriod,1));
cHetr = cHetr*mean(1./cHetr) % normalise
nPeriodsPatch=1
```

Establish the global data struct `patches` for the microscale heterogeneous

Figure 3.7: diffusion field $u(x,t)$ of the gap-tooth scheme applied to the diffusive (3.2). Over this short meso-time we see the macroscale diffusion emerging from the damped sub-patch fast quasi-equilibration.



lattice diffusion system (3.2) solved on 2π-periodic domain, with seven patches, here each patch of size ratio 0.25 from one side to the other, with five micro-grid points in each patch, and quartic interpolation (4) to provide the edge-values of the inter-patch coupling conditions. Setting `patches.EdgyInt` to one means the edge-values come from interpolating the opposite next-to-edge values of the patches (not the mid-patch values). In this case we appear to need at least fourth order (quartic) interpolation to get accurate decay rate for heterogeneous diffusion.

```
global patches
nPatch = 9
ratio = 0.25
nSubP = nPeriodsPatch*mPeriod+2
patches.EdgyInt = 1; % one to use edges for interpolation
configPatches1(@heteroDiff,[-pi pi],nan,nPatch ...
    ,4,ratio,nSubP);
```

Replicate the heterogeneous coefficients across the width of each patch.

```
patches.c=[repmat(cHetr,(nSubP-2)/mPeriod,1);cHetr(1)];
```

Simulate Set the initial conditions of a simulation to be that of a lump perturbed by significant random microscale noise, via `randn`.

```
u0 = exp(-patches.x.^2)+0.1*randn(nSubP,nPatch);
```

Integrate using standard stiff integrators.

```
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode15s(@patchSmooth1, [0 0.5], u0(:));
```

```
else % octave version
    [ts,us] = odeOcts(@patchSmooth1, [0 0.5], u0(:));
end
```

Plot space-time surface of the simulation We want to see the edge values of the patches, so we adjoin a row of `nans` in between patches. For the field values (which are rows in `us`) we need to reshape, permute, interpolate to get edge values, pad with `nans`, and reshape again.

```
xs = patches.x;  xs(end+1,:) = nan;
us = patchEdgeInt1( permute( reshape(us,length(ts) ...
     ,size(patches.x,1),size(patches.x,2)) ,[2 3 1]) );
us(end+1,:,:) = nan;
us=reshape(us,[],length(ts));
```

Now plot two space-time graphs. The first is every time step over a meso-time to see the oscillation and decay of the fast sub-patch diffusions. The second is subsampled surface over the macroscale duration of the simulation to show the propagation of the macroscale diffusion over the heterogeneous lattice.

```
for p=1:2
  switch p
  case 1, j=find(ts<0.01/nPeriodsPatch);
  case 2, [~,j]=min(abs(ts-linspace(ts(1),ts(end),50)));
  end
  figure(p),clf
  mesh(ts(j),xs(:),us(:,j)),  view(60,40)
  xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
  set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
  print('-depsc2',['homoDiffEdgyU' num2str(p)])
end
```

Compute Jacobian and its spectrum Let's explore the Jacobian dynamics for a range of orders of interpolation, all for the same patch design and heterogeneity. Here use a smaller ratio, and more patches, as we do not plot.

```
ratio=0.1
nPatch=19
leadingEvals=[];
for ord=0:2:6
ordInterp=ord
configPatches1(@heteroDiff,[-pi pi],nan,nPatch ...
    ,ord,ratio,nSubP);
```

Form the Jacobian matrix, linear operator, by numerical construction about a zero field. Use `i` to store the indices of the micro-grid points that are interior to the patches and hence are the system variables.

```
u0=0*patches.x; u0([1 end],:)=nan; u0=u0(:);
i=find(~isnan(u0));
```

Table 3.1: example parameters and list of eigenvalues (every fourth one listed is sufficient due to symmetry): `nPatch` = 19, `ratio` = 0.1, `nSubP` = 5. The columns are for various `ordCC`, in order: 0, spectral interpolation; 2, quadratic; 4, quartic; and 6, sixth order.

```
cHetr =
      6.9617
      0.4217
      2.0624
leadingEvals =
       2e-11       -2e-12        4e-12       -2e-11
     -0.9999      -1.5195      -1.0127      -1.0003
     -3.9992      -11.861      -4.7785      -4.0738
     -8.9960      -45.239      -17.164      -10.703
     -15.987      -116.27      -56.220      -30.402
     -24.969      -230.63      -151.74      -92.830
     -35.936      -378.80      -327.36      -247.37
     -48.882      -535.89      -570.87      -521.89
     -63.799      -668.21      -818.33      -855.72
     -80.678      -743.96      -976.57      -1093.4
      -29129       -29233       -29227       -29222
      -29151       -29234       -29229       -29223
```

```
nJ=length(i);
Jac=nan(nJ);
for j=1:nJ
   u0(i)=((1:nJ)==j);
   dudt=patchSmooth1(0,u0);
   Jac(:,j)=dudt(i);
end
nonSymmetric=norm(Jac-Jac')
assert(nonSymmetric<1e-10,'failed symmetry')
Jac(abs(Jac)<1e-12)=0;
```

Find the eigenvalues of the Jacobian, and list for inspection in Table 3.1: the spectral interpolation is effectively exact for the macroscale; quadratic interpolation is usually quantitatively in error; quartic interpolation appears to be the lowest order for reliable quantitative accuracy.

```
[evecs,evals]=eig(Jac);
eval=-sort(-diag(real(evals)));
leadingEvals=[leadingEvals eval(1:2:nPatch+4)]
```

End of the for-loop over orders of interpolation

```
end
```

End of the main script.

3.6.2 `heteroDiff()`: heterogeneous diffusion

This function codes the lattice heterogeneous diffusion inside the patches. For 2D input arrays `u` and `x` (via edge-value interpolation of `patchSmooth1`, Section 3.3), computes the time derivative (3.1) 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`.

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

Fin.

3.7 `BurgersExample`: simulate Burgers' PDE on patches

Section contents

Figure 3.2 shows a previous example simulation in time generated by the patch scheme applied to Burgers' PDE. The code in the example of this section similarly applies the patch scheme to a microscale space-time map (Figure 3.8), a map derived as a microscale space-time discretisation of Burgers' PDE. Then this example applies projective integration to simulate further in time.

3.7.1 Script code to simulate a microscale space-time map

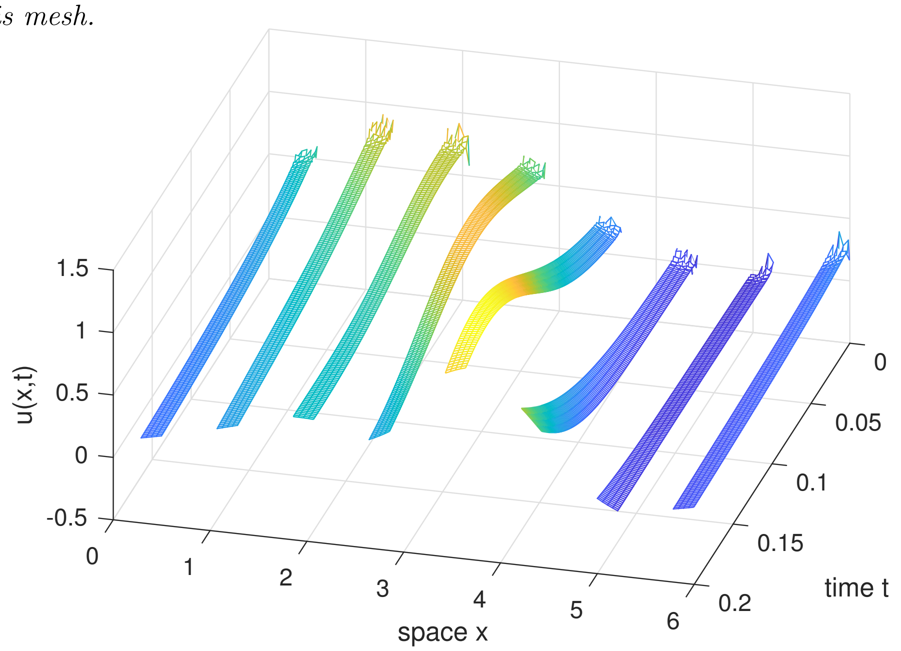
This first part of the script implements the following patch/gap-tooth scheme (left-right arrows denote function recursion).

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

Establish global data struct for the microscale Burgers' map (Section 3.7.3) 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.

```
clear all
global patches
```

Figure 3.8: a short time simulation of the Burgers' map (Section 3.7.3) on patches in space. It requires many very small time-steps only just visible in this mesh.




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

Set an initial condition, and simulate a burst of the microscale space-time map over a time 0.2 using the function `burgerBurst()` (Section 3.7.4).

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

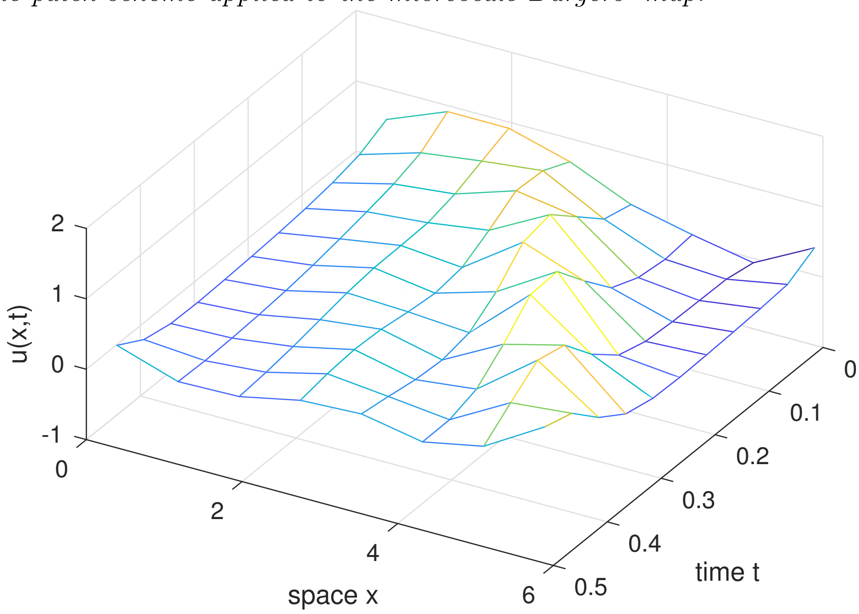
Plot the simulation. Use only the microscale values interior to the patches by setting the edges to `nan` in order to leave gaps.

```
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 3.8.

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

Figure 3.9: macroscale space-time field $u(x, t)$ in a basic projective integration of the patch scheme applied to the microscale Burgers' map.



3.7.2 Alternatively use projective integration

Around the microscale burst `burgerBurst()`, wrap the projective integration function `PIRK2()` of Section 2.2. Figure 3.9 shows the resultant macroscale prediction of the patch centre values on macroscale time-steps.

This second part of the script implements the following design.

1. configPatches1 (done in Section 3.7.1)
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.

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

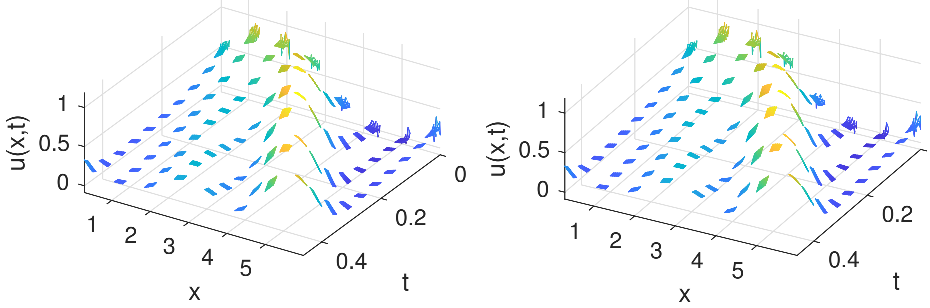
Set the desired macroscale time-steps, and microscale burst length over the time domain. Then projectively integrate in time using `PIRK2()` which is second-order accurate in the macroscale time-step.

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

Plot and save the macroscale predictions of the mid-patch values to give the macroscale mesh-surface of Figure 3.9 that shows a progressing wave solution.

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

Figure 3.10: the microscale field $u(x,t)$ during each of the microscale bursts used in the projective integration. View this stereo pair cross-eyed.




```
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
view(120,50)
set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
%print('-depsc2','BurgersU')
```

Then plot and save the microscale mesh of the microscale bursts shown in Figure 3.10 (a stereo pair). The details of the fine microscale mesh are almost invisible.

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

3.7.3 `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 are mapped (`patchSmooth1()` overrides the edge-values anyway).

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

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

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

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

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

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

Fin.

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

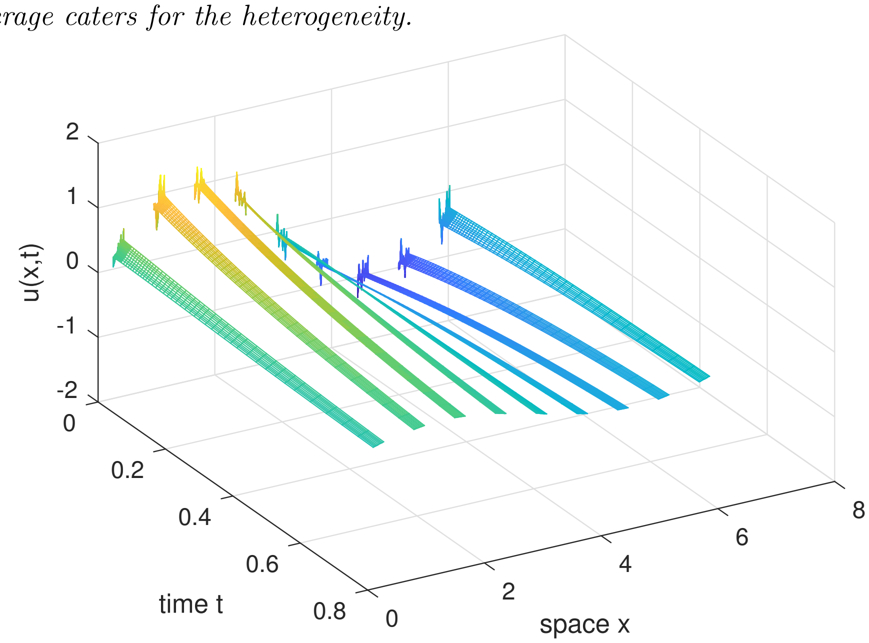
Section contents

3.8.1 Introduction

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

For a first comparison, we present a very similar example to that of Section 3.5, but whereas Section 3.5 simulates using only one diffusivity configuration, here we simulate over an ensemble. For example, Figure 3.3 is similar to Figure 3.11, but the latter is an average of an ensemble of eight different

Figure 3.11: the diffusing field $u(x,t)$ in the patch (gap-tooth) scheme applied to microscale heterogeneous diffusion with an ensemble average. The ensemble average caters for the heterogeneity.



simulations with different diffusivity configurations, whereas the former is simulated from just one diffusivity configuration. The main difference between these two is that the average over the ensemble caters for the heterogeneity in the problem.

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

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

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

$$\dot{u}_i = [c_{i-1/2}(u_{i-1} - u_i) + c_{i+1/2}(u_{i+1} - u_i)]/dx^2. \tag{3.3}$$

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

3.8.2 Script to simulate via stiff or projective integration

Say there are four different diffusivities in our diffusive medium, as defined here.

```
clear all
mPeriod = 4
rand('seed',1);
```

```
c = exp(4*rand(mPeriod,1))
cHomo = 1/mean(1./c)
```

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

```
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;
```

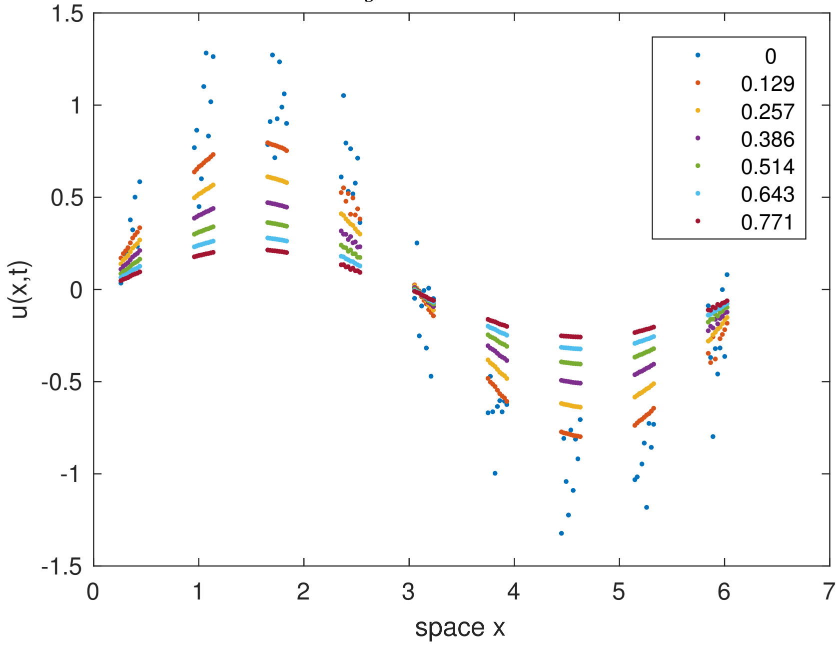
In the case of ensemble averaging, `nVars` is the size of the ensemble (for the case of no ensemble averaging `nVars` is the number of different field variables, which in this example is `nVars` = 1) and we use the ensemble described by Bunder et al. (2017) which includes all reflected and translated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

```
patches.c = c((mod(round(patches.x(1:(end-1),:) ...
  /(patches.x(2)-patches.x(1))-0.5),mPeriod)+1));
if patches.EnsAve
  nVars = mPeriod+(mPeriod>2)*mPeriod;
  patches.c = repmat(patches.c,[1,1,nVars]);
  for sx = 2:mPeriod
    patches.c(:,:,sx) = circshift( ...
      patches.c(:,:,sx-1),[sx-1,0]);
   end;
   if nVars>2
     patches.c(:,:,(mPeriod+1):end) = flipud( ...
       patches.c(:,:,1:mPeriod));
   end;
end
```

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

```
u0 = sin(patches.x)+0.2*randn(nSubP,nPatch);
if patches.EnsAve
  u0 = repmat(u0,[1,1,nVars]);
end
```

Figure 3.12: field $u(x,t)$ shows basic projective integration of patches of heterogeneous diffusion with an ensemble average: different colours correspond to the times in the legend. Once transients have decayed, this field solution is smooth due to the ensemble average.



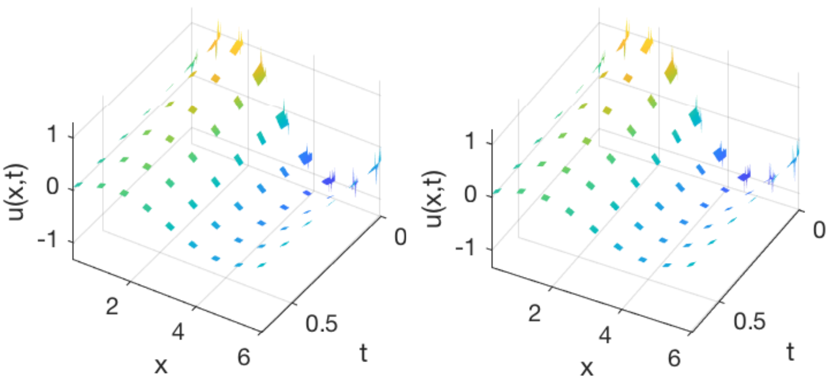

```
if ~exist('OCTAVE_VERSION','builtin')
    [ts,ucts] = ode15s( @patchSmooth1, [0 2/cHomo], u0(:));
else % octave version is slower
    [ts,ucts] = odeOcts(@patchSmooth1, [0 2/cHomo], u0(:));
end
ucts = reshape(ucts,length(ts),length(patches.x(:)),[]);
```

Plot the ensemble averaged simulation in Figure 3.11.

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

Use projective integration in time Now consider the interface, `patchSmooth1()`, to the time derivatives, and wrap around it the projective integration `PIRK2` (Section 2.2), of bursts of simulation from `heteroBurst` (Section 3.5.3), as illustrated by Figure 3.12. The rest of this code follows that of Section 3.5, but as we now evaluate an ensemble of field solutions, our final step is always

Figure 3.13: stereo pair of ensemble averaged fields $u(x,t)$ during each of the microscale bursts used in the projective integration.



an ensemble average.

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

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

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

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

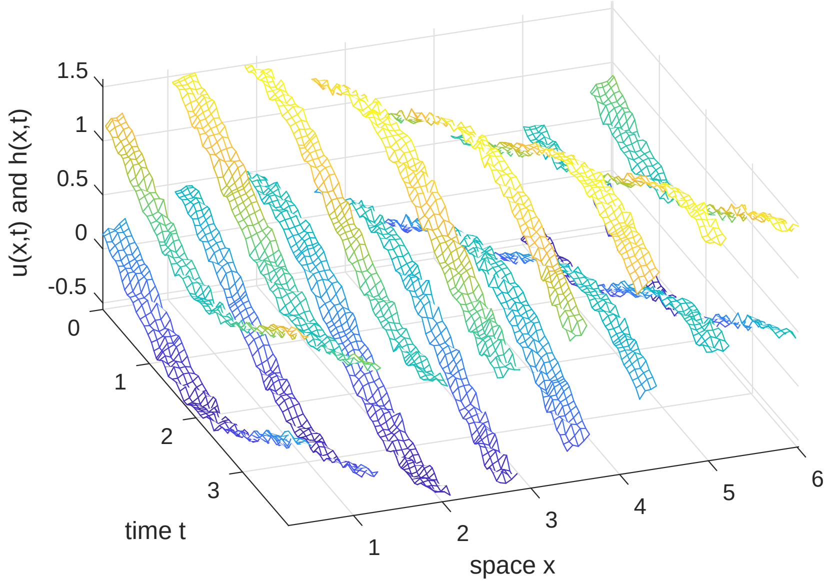
Figure 3.12 shows an average of the ensemble of macroscale predictions.

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

Also plot a surface detailing the ensemble average microscale bursts, Figure 3.13.

```
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','PaperPosition',[0 0 14 10])
%print('-depsc2','ensAveExMicro')
```

Figure 3.14: water depth $h(x,t)$ (above) and velocity field $u(x,t)$ (below) of the gap-tooth scheme applied to the ideal linear wave PDE *(3.4) with $f_1 = f_2 = 0$. The microscale random component to the initial condition persists in the simulation—but the macroscale wave still propagates.*



End of the script.

Sections 3.5.3 and 3.6.2 list the addtional functions used by this script. Fin.

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

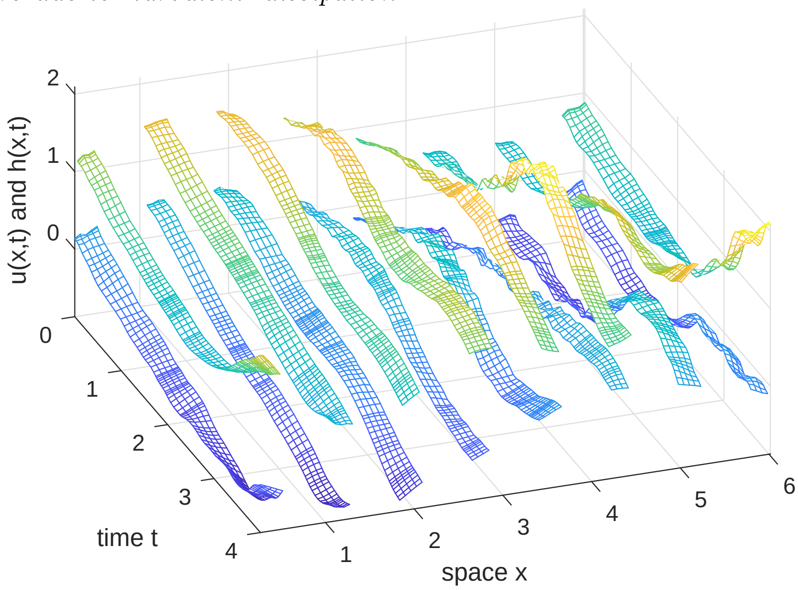
Section contents

Figure 3.14 shows an example simulation in time generated by the patch scheme applied to an ideal wave PDE (Cao & Roberts 2013). The inter-patch coupling is realised by spectral interpolation of the mid-patch values to the patch edges.

This approach, based upon the differential equations coded in Section 3.9.2, may be adapted by a user to a wide variety of 1D wave and wave-like systems. For example, the differential equations of Section 3.9.3 that describe 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

Figure 3.15: water depth $h(x,t)$ (above) and velocity field $u(x,t)$ (below) of the gap-tooth scheme applied to the Smagorinski shallow water wave PDEs *(3.5). The microscale random initial component decays where the water speed is non-zero due to 'turbulent' dissipation.*



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

$$\frac{\partial h}{\partial t} = -c_1\frac{\partial u}{\partial x} + f_1[h,u] \quad\text{and}\quad \frac{\partial u}{\partial t} = -c_2\frac{\partial h}{\partial x} + f_2[h,u], \tag{3.4}$$

where the brackets indicate that the two nonlinear functions f_1 and f_2 may involve various spatial derivatives of the fields $h(x,t)$ and $u(x,t)$. For example, Section 3.9.3 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

$$\frac{\partial h}{\partial t} = -\frac{\partial (hu)}{\partial x}\,, \tag{3.5a}$$

$$\frac{\partial u}{\partial t} = 0.985\left(\tan\theta - \frac{\partial h}{\partial x}\right) - 0.003\frac{u|u|}{h} - 1.045u\frac{\partial u}{\partial x} + 0.26h|u|\frac{\partial^2 u}{\partial x^2}\,, \tag{3.5b}$$

where $\tan\theta$ is the slope of the bed. The PDE (3.5a) represents conservation of the fluid. The momentum PDE (3.5b) represents the effects of turbulent bed drag $u|u|/h$, self-advection $u\partial u/\partial x$, nonlinear turbulent dispersion $h|u|\partial^2 u/\partial x^2$, and gravitational hydrostatic forcing $(\tan\theta - \partial h/\partial x)$. Figure 3.15 shows one simulation of this system—for the same initial condition as Figure 3.14.

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

3.9.1 Script code to simulate wave systems

This example script implements the following patch/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 `patches` for the PDEs (3.4) (linearised) solved on 2π-periodic domain, with eight patches, each patch of half-size ratio 0.2, with eleven micro-grid points within each patch, and spectral interpolation (-1) of 'staggered' macroscale patches to provide the edge-values of the inter-patch coupling conditions.

```
clear all
global patches
nPatch = 8
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 micro-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(uPts==0);
uPts = find(uPts==1);
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 (here 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`/`lsode` we subsample the results because micro-grid scale waves do not dissipate and so the integrator takes very small time-steps for all time.

```
if ~exist('OCTAVE_VERSION','builtin')
    [ts,Ucts] = ode15s( @patchSmooth1,[0 4],U0(:));
    ts = ts(1:5:end);
    Ucts = Ucts(1:5:end,:);
else % octave version is slower
    [ts,Ucts] = odeOcts(@patchSmooth1,[0 4],U0(:));
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)
```

Optionally save the plot to file.

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

For the second time through the loop, change to the Smagorinski turbulence model (3.5) of shallow water flow, keeping other parameters and the initial condition the same.

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

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

3.9.2 `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 micro-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.

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

Compute the PDE derivatives only at interior micro-grid points of the patches.

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

Here 'wastefully' compute time derivatives for both PDEs at all grid points—for simplicity—and then merge 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.

```
  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.

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

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

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

3.9.3 `waterWavePDE()`: water wave PDE

This function codes the staggered lattice equation inside the patches for the nonlinear wave-like PDE system (3.5). Also, regularise the absolute value appearing the the PDEs via the one-line function `rabs()`.

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

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

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

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

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

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

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

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

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

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

Then overwrite the unwanted $\dot u_{ij}$ with the corresponding wanted $\dot h_{ij}$.

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

Fin.

3.10 homoWaveEdgy1: computational homogenisation of a 1D wave by simulation on small patches

Figure 3.16 shows an example simulation in time generated by the patch scheme applied to macroscale wave propagation through a medium with microscale heterogeneity. The inter-patch coupling is realised by spectral interpolation of the patch's next-to-edge values to the patch opposite edges. This coupling preserves symmetry in many systems.

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 and mean longitudinal velocity. Here suppose the spatial microscale lattice is at points x_i, with constant spacing dx. With dependent variables $u_i(t)$ and $v_i(t)$, simulate the microscale lattice, weakly damped, wave system

$$\frac{\partial u_i}{\partial t} = v_i\,, \quad \frac{\partial v_i}{\partial t} = \frac{1}{dx^2}\delta[c_{i-1/2}\delta u_i] + \frac{0.02}{dx^2}\delta^2 v_i\,, \tag{3.6}$$

in terms of the centred difference operator δ. The system has a microscale heterogeneity via the coefficients $c_{i+1/2}$ which we assume to have some given known periodicity. Figure 3.16 shows one patch simulation of this system: observe the effects of the heterogeneity within each patch.

3.10.1 Script code to simulate heterogeneous wave systems

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

1. configPatches1, and add micro-information

Figure 3.16: wave field $u(x,t)$ of the gap-tooth scheme applied to the weakly damped wave (3.6). The microscale random component to the initial condition persists in the simulation until the weak damping smooths the sub-patch fluctuations—but the macroscale wave still propagates.

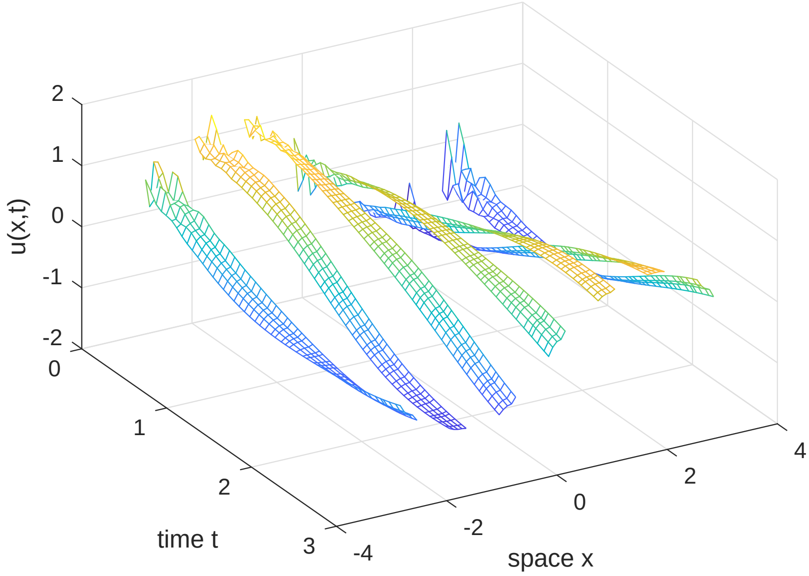
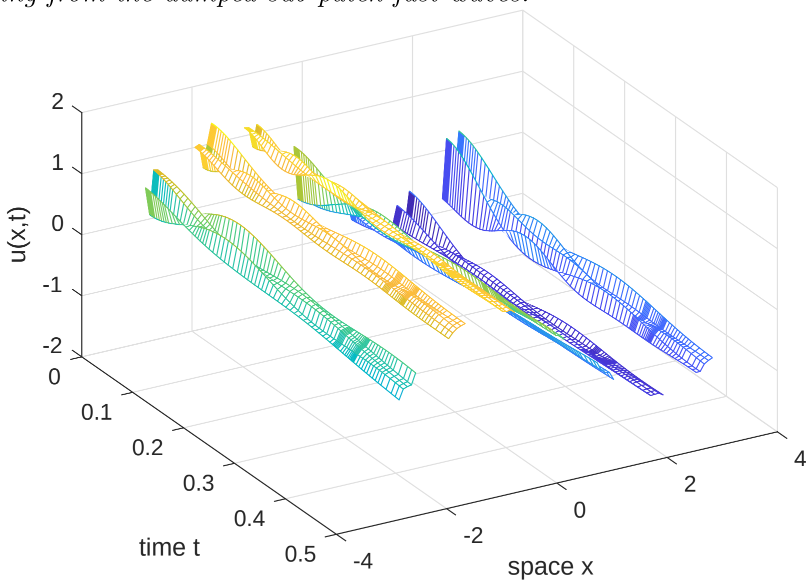


Figure 3.17: wave field $u(x,t)$ of the gap-tooth scheme applied to the weakly damped wave (3.6). Over this shorter meso-time we see the macroscale wave emerging from the damped sub-patch fast waves.



2. ode15s ↔ patchSmooth1 ↔ heteroWave
3. plot the simulation
4. use patchSmooth1 to check the Jacobian

First establish the microscale heterogeneity has micro-period `mPeriod` on the lattice, and random log-normal values, albeit normalised to have harmonic mean one. This normalisation then means that macroscale waves on a domain of length 2π have near integer frequencies, $1, 2, 3, \ldots$. Then the heterogeneity is to be repeated `nPeriodsPatch` times within each patch.

```
clear all
mPeriod = 3
cHetr = exp(1*randn(mPeriod,1));
cHetr = cHetr*mean(1./cHetr) % normalise
nPeriodsPatch=1
```

Establish the global data struct `patches` for the microscale heterogeneous lattice wave system (3.6) solved on 2π-periodic domain, with seven patches, here each patch of size ratio 0.25 from one side to the other, with five micro-grid points in each patch, and spectral interpolation (0) to provide the edge-values of the inter-patch coupling conditions. Setting `patches.EdgyInt` to one means the edge-values come from interpolating the opposite next-to-edge values of the patches (not the mid-patch values).

```
global patches
nPatch = 7
ratio = 0.25
nSubP = nPeriodsPatch*mPeriod+2
patches.EdgyInt = 1; % one to use edges for interpolation
configPatches1(@heteroWave,[-pi pi],nan,nPatch ...
    ,0,ratio,nSubP);
```

Replicate the heterogeneous coefficients across the width of each patch.

```
patches.c=[repmat(cHetr,(nSubP-2)/mPeriod,1);cHetr(1)];
```

Simulate Set the initial conditions of a simulation to be that of a macroscopic progressive wave, via sin / cos, perturbed by significant random microscale noise, via `randn`.

```
u0 = -sin(patches.x)+0.3*randn(nSubP,nPatch);
v0 = +cos(patches.x)+0.3*randn(nSubP,nPatch);
```

Integrate for about half a wave period using standard stiff integrators (which do not work efficiently until after the fast waves have decayed).

```
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode15s(@patchSmooth1, [0 3], [u0(:);v0(:)]);
else % octave version
    [ts,us] = odeOcts(@patchSmooth1, [0 3], [u0(:);v0(:)]);
end
```

Plot space-time surface of the simulation We want to see the edge values of the patches, so we adjoin a row of `nans` in between patches. For the field values (which are rows in `us`) we need to reshape, permute, interpolate to get edge values, pad with `nans`, and reshape again.

```
xs = patches.x;  xs(end+1,:) = nan;
us = patchEdgeInt1( permute( reshape(us,length(ts) ...
     ,size(patches.x,1),size(patches.x,2),2) ,[2 3 4 1]) );
us(end+1,:,:,:) = nan;
us = reshape(us,length(xs(:)),2,[]);
```

Now plot two space-time graphs. The first is every time step over a meso-time to see the oscillation and decay of the fast sub-patch waves. The second is subsampled surface over the macroscale duration of the simulation to show the propagation of the macroscale wave over the heterogeneous lattice.

```
for p=1:2
  switch p
  case 1, j=find(ts<0.5);
  case 2, [~,j]=min(abs(ts-linspace(ts(1),ts(end),50)));
  end
  figure(p),clf
  mesh(ts(j),xs(:),squeeze(us(:,1,j))),  view(60,40)
  xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
  set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
  print('-depsc2',['homoWaveEdgyU' num2str(p)])
end
```

Compute Jacobian and its spectrum Form the Jacobian matrix, linear operator, by numerical construction about a zero field. Use `i` to store the indices of the micro-grid points that are interior to the patches and hence are the systems variables.

```
u0=0*patches.x; u0([1 end],:)=nan; u0=[u0(:);u0(:)];
i=find(~isnan(u0));
nJ=length(i);
Jac=nan(nJ);
for j=1:nJ
   u0(i)=((1:nJ)==j);
   dudt=patchSmooth1(0,u0);
   Jac(:,j)=dudt(i);
end
Jac(abs(Jac)<1e-12)=0;
```

Find the eigenvalues of the Jacobian, and list for inspection in Table 3.2.

```
[evecs,evals]=eig(Jac);
eval=sort(diag(evals))
```

End of the main script.

Table 3.2: example parameters and list of eigenvalues (every fourth one listed is sufficient due to symmetry): `nPatch` $= 7$, `ratio` $= 0.25$, `nSubP` $= 5$. The spectrum is satisfactory for weakly damped macroscale waves, and medium-damped microscale sub-patch fast waves.

```
cHetr =
      0.58459
       1.0026
       3.4253
eval =
   2.2701e-16 + 1.4225e-07i
    -0.013349 +    0.99941i
    -0.053324 +     1.9952i
     -0.11971 +     2.9838i
      -5.1527 +     19.554i
      -5.2679 +     19.695i
      -5.3383 +     19.779i
      -5.3619 +     36.632i
      -5.3722 +     36.632i
      -5.4026 +     36.631i
      -5.4514 +      36.63i
```

3.10.2 `heteroWave()`: wave in heterogeneous media with weak viscous damping

This function codes the lattice heterogeneous wave equation, with weak viscosity, inside the patches. For 3D input array `u` (u_{ij} = `u(i,j,1)` and v_{ij} = `u(i,j,2)`) and 2D array `x` (obtained in full via edge-value interpolation of `patchSmooth1`, Section 3.3), computes the time derivatives at each point in the interior of a patch, output in `ut`:

$$\frac{\partial u_{ij}}{\partial t} = v_{ij}\,, \quad \frac{\partial v_{ij}}{\partial t} = \frac{1}{dx^2}\delta[c_{i-1/2}\delta u_{ij}] + \frac{0.02}{dx^2}\delta^2 v_{ij}\,.$$

The column vector (or possibly array) of diffusion coefficients c_i have previously been stored in struct `patches`.

```
function ut = heteroWave(t,u,x)
  global patches
  dx = diff(x(2:3)); % space step
  i = 2:size(u,1)-1; % interior points in a patch
  ut = nan(size(u)); % preallocate output array
  ut(i,:,1) = u(i,:,2); % du/dt=v then dvdt=
  ut(i,:,2) = diff(patches.c.*diff(u(:,:,1)))/dx^2 ...
        +0.02*diff(u(:,:,2),2)/dx^2;
end% function
```

Fin.

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

Section contents

3.11.1 Introduction

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSmooth2()`. Section 3.11.2 lists an example of its use.

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

Input If invoked with no input arguments, then executes an example of simulating a nonlinear diffusion PDE relevant to the lubrication flow of a thin layer of fluid—see Section 3.11.2 for the example code.

- `fun` is the name of the user function, `fun(t,u,x,y)`, that computes time-derivatives (or time-steps) of quantities on the 2D micro-grid within all the 2D patches.

- `Xlim` array/vector giving the macro-space domain of the computation: patches are distributed equi-spaced over the interior of the rectangle [`Xlim(1)`, `Xlim(2)`] $\times$ [`Xlim(3)`, `Xlim(4)`]: if `Xlim` is of length two, then the domain is the square of the same interval in both directions.

- `BCs` eventually 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)` gives the number of patches in each direction.

- `ordCC` is the 'order' of interpolation for inter-patch coupling across empty space of the macroscale mid-patch values to the edge-values of the patches: currently must be 0; where 0 gives spectral interpolation.

- `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; `ratio` $= 1$ would be overlapping patches as in holistic discretisation; and small `ratio` should greatly reduce computational time. If scalar, then use the same ratio in both directions, otherwise `ratio(1:2)` gives 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 micro-grid point in each patch.

- nEdge, (not yet implemented) *optional*, is the number of edge values set by interpolation at the edge regions of each patch. 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(t,u,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—not yet implemented.
- .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.

3.11.2 If no arguments, then execute an example

```
if nargin==0
```

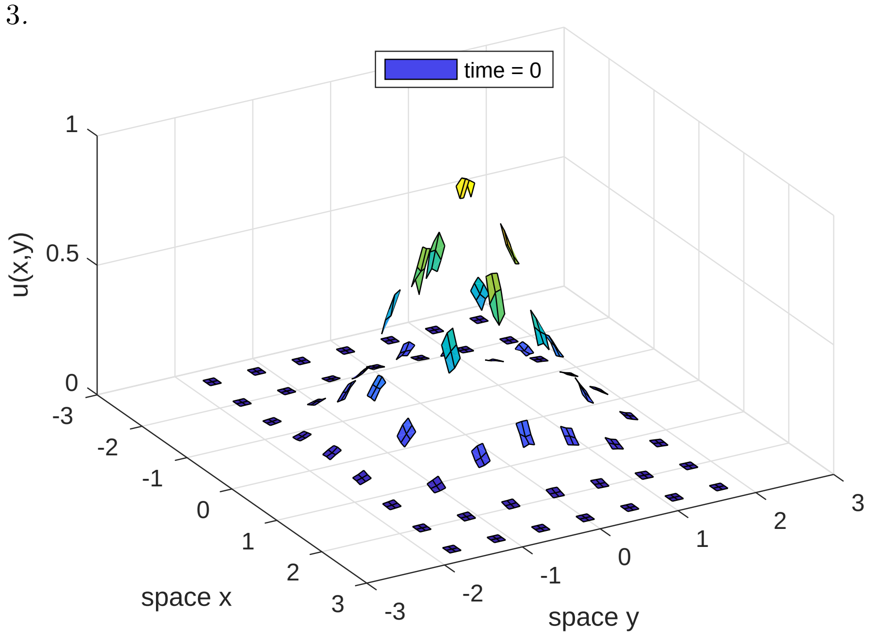
The code here shows one way to get started: a user's script may have the following three steps (arrows indicate function recursion).

1. configPatches2
2. ode15s integrator ↔ patchSmooth2 ↔ user's PDE
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 (relatively large for visualisation), and with 5×5 points within each patch. Roberts et al. (2014) established that this scheme is consistent with the PDE (as the patch spacing decreases).

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

Figure 3.18: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to a nonlinear 'diffusion' PDE: Figure 3.19 plots the computed field at time $t = 3$.



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

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

Initiate a plot of the simulation using only the microscale values interior to the patches: set x and y-edges to `nan` to leave the gaps between patches.

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

Start by showing the initial conditions of Figure 3.18 while the simulation computes.

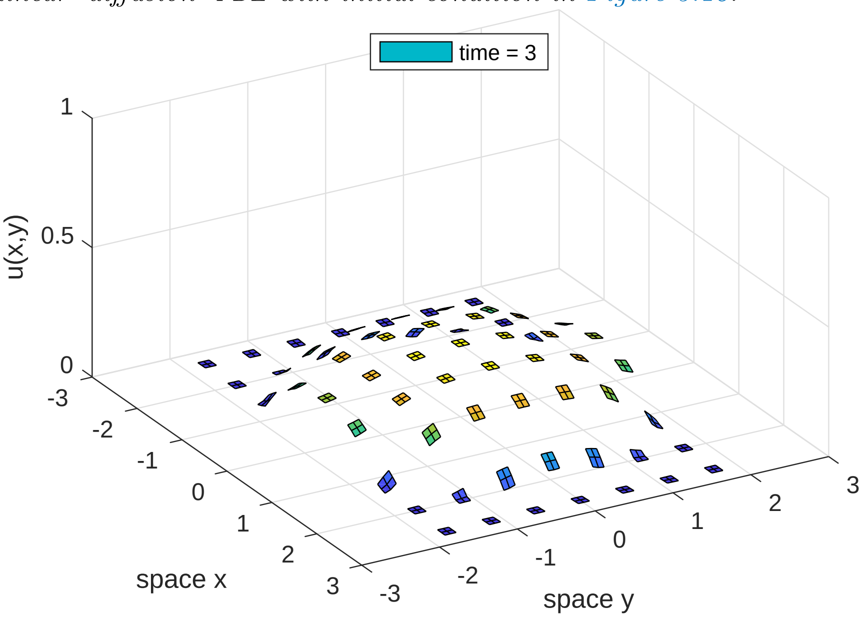
```
u = reshape(permute(u0,[1 3 2 4]), [numel(x) numel(y)]);
hsurf = surf(x(:),y(:),u');
axis([-3 3 -3 3 -0.03 1]), view(60,40)
legend('time = 0','Location','north')
xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
```

Save the initial condition to file for Figure 3.18.

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

Integrate in time using standard functions.

Figure 3.19: field $u(x,y,t)$ at time $t = 3$ of the patch scheme applied to a nonlinear 'diffusion' PDE with initial condition in Figure 3.18.




```
disp('Wait while we simulate h_t=(h^3)_xx+(h^3)_yy')
drawnow
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode15s( @patchSmooth2,[0 4],u0(:));
else % octave version is quite slow for me
    lsode_options('absolute tolerance',1e-4);
    lsode_options('relative tolerance',1e-4);
    [ts,us] = odeOcts(@patchSmooth2,[0 1],u0(:));
end
```

Animate the computed simulation to end with Figure 3.19. Use `patchEdgeInt2` to interpolate patch-edge values (even if not drawn).

```
for i = 1:length(ts)
  u = patchEdgeInt2(us(i,:));
  u = reshape(permute(u,[1 3 2 4]), [numel(x) numel(y)]);
  set(hsurf,'ZData', u');
  legend(['time = ' num2str(ts(i),2)])
  pause(0.1)
end
%print('-depsc2','configPatches2t3')
```

Upon finishing execution of the example, exit this function.

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

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

$\frac{1}{\delta y^2}(u^3_{i,j+1,k,l} - 2u^3_{i,j,k,l} + u^3_{i,j-1,k,l})$.

```
function ut = nonDiffPDE(t,u,x,y)
  dx = diff(x(1:2));  dy = diff(y(1:2));  % microgrid spacing
  i = 2:size(u,1)-1;  j = 2:size(u,2)-1;  % interior patch points
  ut = nan(size(u));  % preallocate storage
  ut(i,j,:,:) = diff(u(:,j,:,:).^3,2,1)/dx^2 ...
               +diff(u(i,:,:,:).^3,2,2)/dy^2;
end
```

3.11.3 The code to make patches

Initially duplicate parameters for both space dimensions as needed.

```
if numel(Xlim)==2, Xlim = repmat(Xlim,1,2); end
if numel(nPatch)==1, nPatch = repmat(nPatch,1,2); end
if numel(ratio)==1, ratio = repmat(ratio,1,2); end
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.

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

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

```
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.

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

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

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

Might as well precompute the weightings for the interpolation of field values for coupling—not yet used here. (Could sometime extend to coupling via derivative values.)

```
ratio = ratio(:)'; % force to be row vector
if patches.alt  % eqn (7) in \cite{Cao2014a}
  patches.Cwtsr = [1
    ratio/2
    (-1+ratio.^2)/8
    (-1+ratio.^2).*ratio/48
```

```
    (9-10*ratio.^2+ratio.^4)/384
    (9-10*ratio.^2+ratio.^4).*ratio/3840
    (-225+259*ratio.^2-35*ratio.^4+ratio.^6)/46080
    (-225+259*ratio.^2-35*ratio.^4+ratio.^6).*ratio/645120 ];
else %
  patches.Cwtsr = [ratio
    ratio.^2/2
    (-1+ratio.^2).*ratio/6
    (-1+ratio.^2).*ratio.^2/24
    (4-5*ratio.^2+ratio.^4).*ratio/120
    (4-5*ratio.^2+ratio.^4).*ratio.^2/720
    (-36+49*ratio.^2-14*ratio.^4+ratio.^6).*ratio/5040
    (-36+49*ratio.^2-14*ratio.^4+ratio.^6).*ratio.^2/40320 ];
end
patches.Cwtsr = patches.Cwtsr(1:ordCC,:);
% maybe 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)` $\cdot$ `DX` and `ratio(2)` $\cdot$ `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.

3.12 `patchSmooth2()`: interface to time integrators

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

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

Input

- `u` is a vector of length `prod(nSubP)` $\cdot$ `prod(nPatch)` $\cdot$ `nVars` where there are `nVars` field values at each of the points in the `nSubP(1)` $\times$ `nSubP(2)` $\times$ `nPatch(1)` $\times$ `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)` $\times$ `nSubP(2)` $\times$ `nPatch(1)` $\times$ `nPatch(2)` $\times$ `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)` $\times$ `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)` $\times$ `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)` $\cdot$ `prod(nPatch)` $\cdot$ `nVars` vector of time derivatives, but with patch edge-values set to zero.

Reshape the fields `u` as a 4/5D-array, and sets the edge values from macroscale interpolation of centre-patch values. Section 3.13 describes `patchEdgeInt2()`.

```
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.

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

Fin.

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

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`.

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

Input

- `u` is a vector of length `nx` · `ny` · `Nx` · `Ny` · `nVars` where there are `nVars` field values at each of the points in the `nx` × `ny` × `Nx` × `Ny` grid on the `Nx` × `Ny` array of patches.
- `patches` a struct set by `configPatches2()` which includes the following information.
  - `.x` is `nx` × `Nx` array of the spatial locations x_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.y` is similarly `ny` × `Ny` array of the spatial locations y_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.ordCC` is order of interpolation, currently only $\{0\}$.
  - `.Cwtsr` and `.Cwtsl`—not yet used

Output

- `u` is `nx` × `ny` × `Nx` × `Ny` × `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.

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

```
dx = patches.x(3,1)-patches.x(2,1);
DX = patches.x(2,2)-patches.x(2,1);
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—currently not needed.

```
%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 —not yet implemented 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 $\texttt{ks} = k2\pi/N$ 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.

```
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.

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

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

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

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

```
  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.

```
nFTx = 2-mod(Nx,2);
nFTy = 2-mod(Ny,2);
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()`.

```
if numel(size(unj))>5
        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) );
else
        u(end,:,:,:,iV) = uclean( unj );
```

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

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.

3.14 `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 ↔ patchSmooth2 ↔ wavePDE
3. process results

Establish the global data struct `patches` 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 (big enough for visualisation), and with a 5×5 micro-grid within each patch.

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

3.14.1 Check on the linear stability of the wave PDE

Construct the systems Jacobian via numerical differentiation. 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 this is a *linear* wave PDE, use large perturbations.

```
small = 1;
jac = nan(length(i));
sizeJacobian = size(jac)
for j = 1:length(i)
  uv = uv0(:);
  uv(i(j)) = uv(i(j))+small;
  tmp = patchSmooth2(0,uv)/small;
  jac(:,j) = tmp(i);
end
```

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

```
evals = eig(jac);
nEvals = length(evals)
[~,k] = sort(-abs(real(evals)));
evalsWithBiggestRealPart = evals(k(1:10))
if abs(real(evals(k(1))))>1e-4
    warning('eigenvalue failure: real-part > 1e-4')
    return, end
```

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

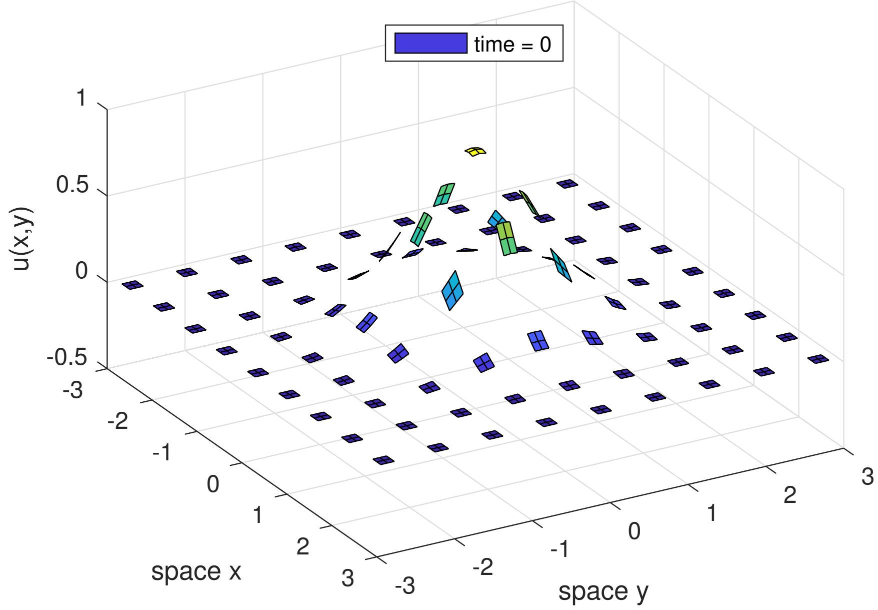
```
kwave = 0:(nPatch-1)/2;
freq = sort(reshape(sqrt(kwave'.^2+kwave.^2),1,[]));
freq = freq(diff([-1 freq])>1e-9);
freqerr = [freq; min(abs(imag(evals)-freq))]
```

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

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

Figure 3.20: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to the simple wave PDE*: Figure 3.21 plots the computed field at time $t = 2$.*



Initiate a plot of the simulation using only the microscale values interior to the patches: set x and y-edges to `nan` to leave the gaps. Start by showing the initial conditions of Figure 3.18 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 $n_x \cdot N_x \times n_y \cdot N_y$.

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

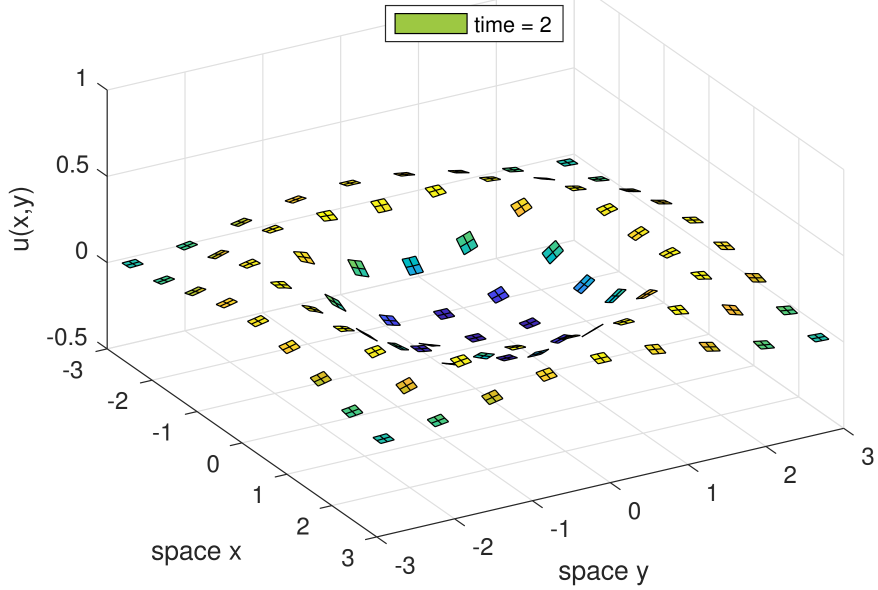
Integrate in time using standard functions.

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

Animate the computed simulation to end with Figure 3.21. Because of the very small time-steps, subsample to plot at most 100 times.

```
di = ceil(length(ts)/100);
```

Figure 3.21: field $u(x,y,t)$ at time $t = 2$ of the patch scheme applied to the simple wave PDE *with initial condition in Figure 3.20.*




```
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]);
  set(usurf,'ZData', uv(:,:,1)');
  legend(['time = ' num2str(ts(i),2)])
  pause(0.1)
end
%print('-depsc',['wave2Dt' num2str(ts(end))])
```

3.14.3 `wavePDE()`: Example of simple wave PDE inside patches

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

```
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;
end
```

```
function [ts,xs] = odeOcts(dxdt,tSpan,x0)
    if length(tSpan)>2, ts = tSpan;
    else ts = linspace(tSpan(1),tSpan(end),21)';
    end
    lsode_options('integration method','stiff');
```

```
    xs = lsode(@(x,t) dxdt(t,x),x0,ts);
end
```

3.15 To do

- Some users will have microscale that has a fixed microscale lattice spacing, in which case we should code the scale ratio r to follow from the choice of the number of lattice points in a patch.
- More than two space dimensions?
- Heterogeneous microscale via averaging regions—but I suspect should be separated from simple homogenisation
- Parallel processing versions.
- Adapt to maps in micro-time? Surely easy, just an example.

3.16 Miscellaneous tests

3.16.1 `patchEdgeInt1check`: check the finite stencil interpolation

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

```
clear all
global patches
nSubP=6
```

Check standard finite stencil interpolation Check over various types and orders of interpolation, numbers of patches, random domain lengths and random ratios.

```
for edgyInt=rem(nSubP+1,2):1
for ordCC=2:2:8
for nPatch=ordCC+(2:4)
edgyInt=edgyInt
ordCC=ordCC
nPatch=nPatch
Domain=5*[-rand rand]
ratio=0.5*rand
patches.EdgyInt=edgyInt;
configPatches1(@sin,Domain,nan,nPatch,ordCC,ratio,nSubP);
```

Check multiple fields simultaneously Set profiles to be various powers of x, and evaluate the interpolation.

```
ps=1:ordCC
cs=randn(size(ps))
u0=patches.x.^shiftdim(ps,-1).*shiftdim(cs,-1)+randn;
ui=patchEdgeInt1(u0(:));
```

The interior patches should have zero error.

```
j=ordCC/2+1:nPatch-ordCC/2;
iError=ui(:,j,:)-u0(:,j,:);
normError=norm(iError(:))
assert(normError<5e-12,'failed finite stencil interpolation')
```

End the for-loops over various parameters.

```
end,end,end
return%%%%%%%%%%%%%%%%%%%
```

3.16.2 `patchEdgeInt1test`: test the spectral interpolation

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

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

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

```
for nPatch=5:10
nPatch=nPatch
Len=10*rand
ratio=0.5*rand
patches.EdgyInt=1 % optional test
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
    error(['failed double field interpolation k=' num2str(k)])
  end
end
```

End for-loop over patches

```
end
```

Finish If no error messages, then all OK.

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

3.16.3 `patchEdgeInt2test`: tests 2D spectral interpolation

Try 99 realisations of random tests.

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

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

```
Lx=1+3*rand, Ly=1+3*rand
nSubP=1+2*randi(3,1,2)
ratios=rand(1,2)/2
nPatch=2+randi(4,1,2)
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.

```
nV=randi(3)
[nx,Nx]=size(patches.x);
[ny,Ny]=size(patches.y);
u0s=nan(nx,ny,Nx,Ny,nV);
for iV=1:nV
  kx=randi([0 ceil((nPatch(1)-1)/2)])
  ky=randi([0 ceil((nPatch(2)-1)/2)])
```

```
  phix=pi*rand*(2*kx~=nPatch(1))
  phiy=pi*rand*(2*ky~=nPatch(2))
  % generate 2D array via auto-replication
  u0=sin(2*pi*kx*patches.x(:) /Lx+phix) ...
   .*sin(2*pi*ky*patches.y(:)'/Ly+phiy);
  % reshape into 4D array
  u0=reshape(u0,[nx Nx ny Ny]);
  u0=permute(u0,[1 3 2 4]);
  % store into 5D array
  u0s(:,:,:,:,iV)=u0;
end
```

Copy and NaN the edges, then interpolate

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

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

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

Appendix A Create, document and test algorithms

- Upon 'finalising' a version of the toolbox:
  1. pdflatex and bibtex `Doc/eqnFreeDevMan.tex` to ensure all is documented properly;
  2. execute `bibexport eqnFreeDevMan` to update the local bibliographic data-file;
  3. pdflatex `Doc/eqnFreeUserMan.tex`, several times, to get a shorter and more user friendly version;
  4. replace the root file `eqnFreeUserMan-newest.pdf` by a renamed copy of the new `Doc/eqnFreeUserMan.pdf`
- To create and document the various functions, we adapt an idea due to Neil D. Lawrence of the University of Sheffield in order to interleave Matlab/Octave code, and its documentation in LaTeX (Table A.2).
- Each class of toolbox functions is located in separate folders in the repository, say `Dir`.
- Create a LaTeX file `Dir/funs.tex`: establish as one LaTeX chapter that `\input{../Dir/*.m}`s the files of the functions in the class, example scripts of use, and possibly test scripts, Table A.1.
- Each such `Dir/funs.tex` file is to be included from the main LaTeX file `Doc/docBody.tex` so that people can most easily work on one chapter at a time:
  - create a 'link' file `Doc/funs.tex` whose only active content is the command `\input{../Dir/funs.tex}` ;
  - put `\include{funs}` into `Doc/docBody.tex` ;
  - in `Doc/docBody.tex` modify the `\graphicspath` command to include `{../Dir/Figs}` .
- Each toolbox function is documented as a separate section, within its chapter, with tests and examples as separate sections.
- Each function-section and test-section is to be created as a Matlab/ Octave `Dir/*.m` file, say `Dir/fun1.m`, so that users simply invoke the function in Matlab/Octave as usual by `fun1(...)`.

  Some editors may need to be told that `fun1.m` is a LaTeX file. For example, TexShop on the Mac requires one to execute (once) 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,'PaperUnits','centimeters','PaperPosition',[0 0 14 10]);% cm
  print('-depsc2','filename')
  ```

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

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

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

  When appropriate, if a function is invoked with no output arguments (`if nargout==0`), then draw some reasonable graph of the results.
- In all MATLAB/Octave code, prefer camal case for variable names (not underscores).
- When a function is 'finalised', wrap (most) of the lines to be no more than 60 characters so that readers looking at the source can read the plain text reasonably.
- In the documentation (e.g., Higham 1998, Ch. 4): write actively, not passively (e.g., avoid "–tion" words, and avoid "is/are verbed" phrases); avoid wishy-washy "can"; use the present tense; cross-reference precisely; avoid useless padding such as "note that"; and so on.

Table A.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{ch:...}
\localtableofcontents
\section{Introduction}
introduction...
\input{../Dir/fun.m} % prefix associated files with 'fun'
\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
\subsection{Introduction}
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/user 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

Have coded several schemes.

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, perhaps via DMD.

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 integration.

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 (e.g., `cdmc()`). 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/user will not code in a restriction 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 implied by a restriction, 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/Octave 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 many more CPUs than patches, then maybe simply duplicate all patch simulations; if many fewer 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, may connect to LAMMPS (Plimpton et al. 2016). Second, will evaluate performance, issues, and then consider what other established packages are most promising.

Bibliography

Bunder, J. E., Roberts, A. J. & Kevrekidis, I. G. (2017), 'Good coupling for the multiscale patch scheme on systems with microscale heterogeneity', *J. Computational Physics* **337**, 154–174.

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

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

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

Cao, M. & Roberts, A. J. (2013), Multiscale modelling couples patches of wave-like simulations, *in* S. McCue, T. Moroney, D. Mallet & J. Bunder, eds, 'Proceedings of the 16th Biennial Computational Techniques and Applications Conference, CTAC-2012', Vol. 54 of *ANZIAM J.*, pp. C153–C170.

Cao, M. & Roberts, A. J. (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., Kaper, T. J., Kevrekidis, I. G. & Zagaris, A. (2005*a*), 'Projecting to a slow manifold: singularly perturbed systems and legacy codes', *SIAM J. Applied Dynamical Systems* **4**(3), 711–732.
http://www.siam.org/journals/siads/4-3/60829.html

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

Gear, C. W. & Kevrekidis, I. G. (2003*a*), 'Computing in the past with forward integration', *Phys. Lett. A* **321**, 335–343.

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

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

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

Higham, N. J. (1998), *Handbook of writing for the mathematical sciences*, 2nd edition edn, SIAM.

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

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

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

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

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

Maclean, J. & Gottwald, G. A. (2015), 'On convergence of higher order schemes for the projective integration method for stiff ordinary differential equations', *Journal of Computational and Applied Mathematics* **288**, 44–69.
http://www.sciencedirect.com/science/article/pii/S0377042715002149

Marschler, C., Sieber, J., Berkemer, R., Kawamoto, A. & Starke, J. (2014), 'Implicit methods for equation-free analysis: Convergence results and analysis of emergent waves in microscopic traffic models', *SIAM J. Appl. Dyn. Syst.* **13**(2), 1202–1238.

Petersik, P. (2019–), Equation-free modeling, Technical report, [https://github.com/pjpetersik/eqnfree].

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

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

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

Roberts, A. J., MacKenzie, T. & Bunder, J. (2014), 'A dynamical systems approach to simulating macroscale spatial dynamics in multiple dimensions', *J. Engineering Mathematics* **86**(1), 175–207.
http://arxiv.org/abs/1103.1187

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

Samaey, G., 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.