Equation-Free function toolbox for Matlab/Octave:

Full Developers Manual

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

December 14, 2020

* 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 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 domain 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 space adapt the code at the beginning of `configPatches1.m` for Burgers' PDE (Section 3.1.1), or the staggered patches of 1D water wave equations in `waterWaveExample.m` (Section 3.7).
- In 2D space adapt the code at the beginning of `configPatches2.m` for nonlinear diffusion (Section 3.10.1), or the regular patches of the 2D wave PDE of `wave2D.m` (Section 3.13).
- In 3D space adapt the code at the beginning of `configPatches3.m` for wave propagation through a heterogeneous medium (Section 3.15.1), or the patches of the 3D heterogeneous diffusion of `homoDiffEdgy3.m` (Section 3.18).
- Other provided examples include cases of macroscale *computational homogenisation* of microscale heterogeneity.

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. Further, Bunder et al. (2020) developed a new 'edgy' inter-patch interpolation that is proven to be good for simulating the macroscale homogenised dynamics of microscale heterogeneous systems—now coded in the toolbox.

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 number p of field variables, and evolving in 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.

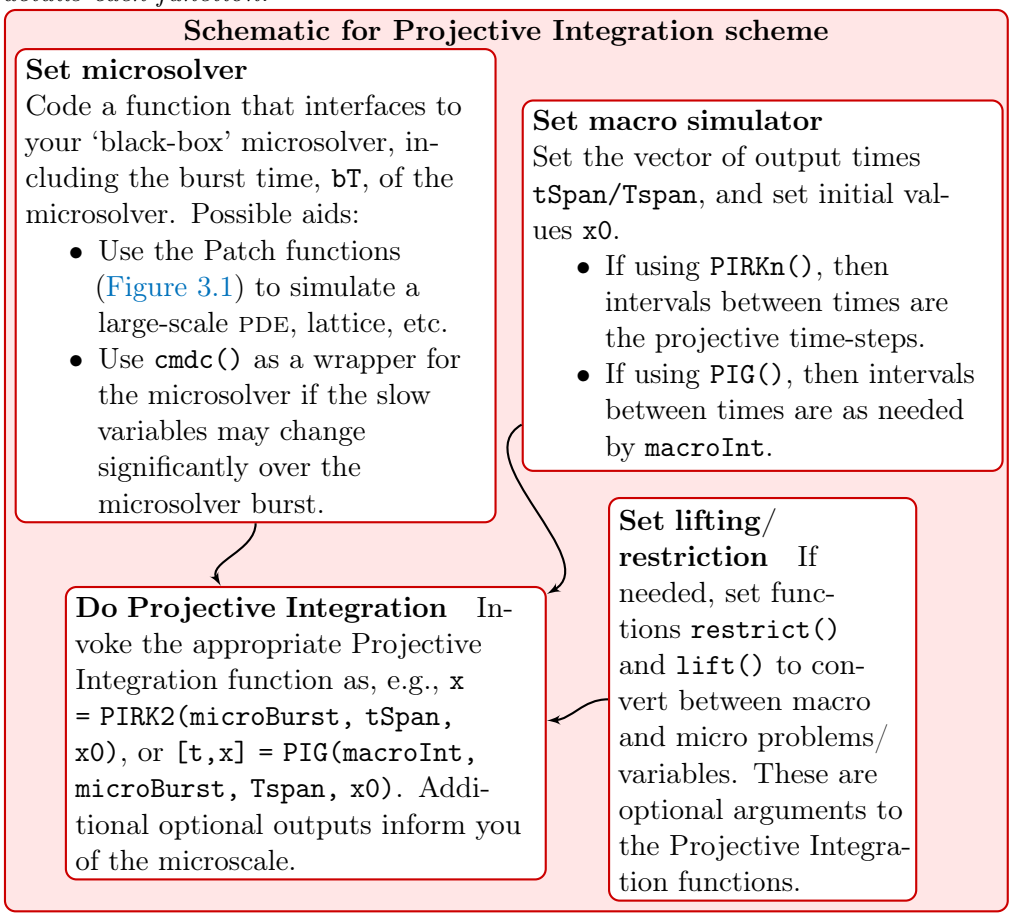
The Projective Integration section of this toolbox consists of several functions. Each function implements over a long-time scale a variant of a standard numerical method to simulate/integrate the emergent dynamics of the complex system. Each function has standardised inputs and outputs.

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.

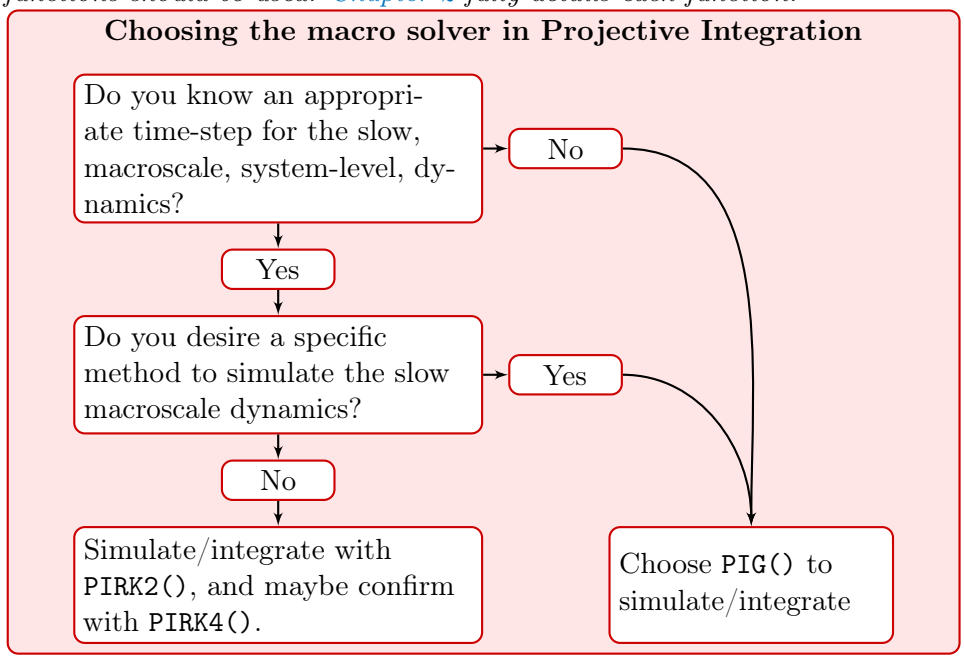
Figure 2.1: The Projective Integration method greatly accelerates simulation/ integration of a system exhibiting multiple time scales. The Projective Integration Chapter 2 presents several separate functions, as well as several optional wrapper functions that may be invoked. This chart overviews constructing 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.



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()`.

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 PIRK2(): projective integration of second-order accuracy

Section contents

2.2.1 Introduction

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

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

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

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

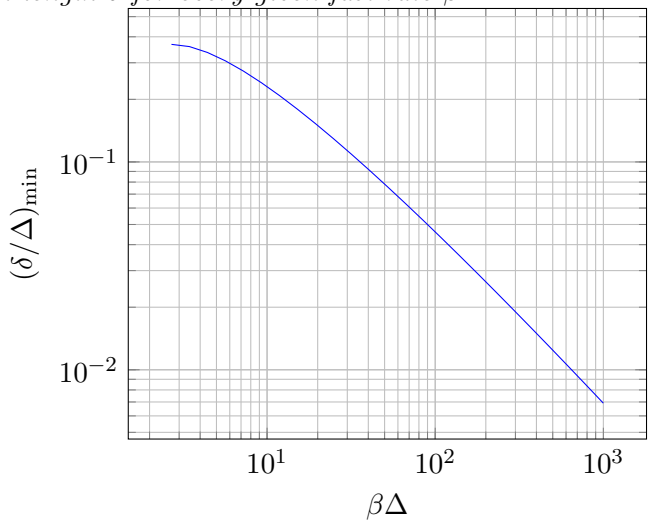
  ```
  [tOut, xOut] = microBurst(tStart, xStart, bT)
  ```

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

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

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

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.



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

```
175 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 $\Delta = 1$.

```
196 global MMepsilon
197 MMepsilon = 0.05
198 ts = 0:6
199 bT = MMepsilon*log((ts(2)-ts(1))/MMepsilon)
200 [x,tms,xms] = PIRK2(@MMburst, ts, [1;0], bT);
201 figure, plot(ts,x,'o:',tms,xms)
202 title('Projective integration of Michaelis--Menten enzyme kinetics')
203 xlabel('time t'), legend('x(t)','y(t)')
```

Upon finishing execution of the example, exit this function.

```
209 return
210 end%if no arguments
```

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

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

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

2.2.3 The projective integration code

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

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

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

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

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

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

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

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

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

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

Loop over the macroscale time-steps

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

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

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

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

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

Check for round-off error.

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

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

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

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

```
    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)`.

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

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

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

2.2.4 If no output specified, then plot the simulation

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

This concludes `PIRK2()`.

```
463 end
```

2.3 egPIMM: Example projective integration of Michaelis–Menton kinetics

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

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

(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.

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).

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

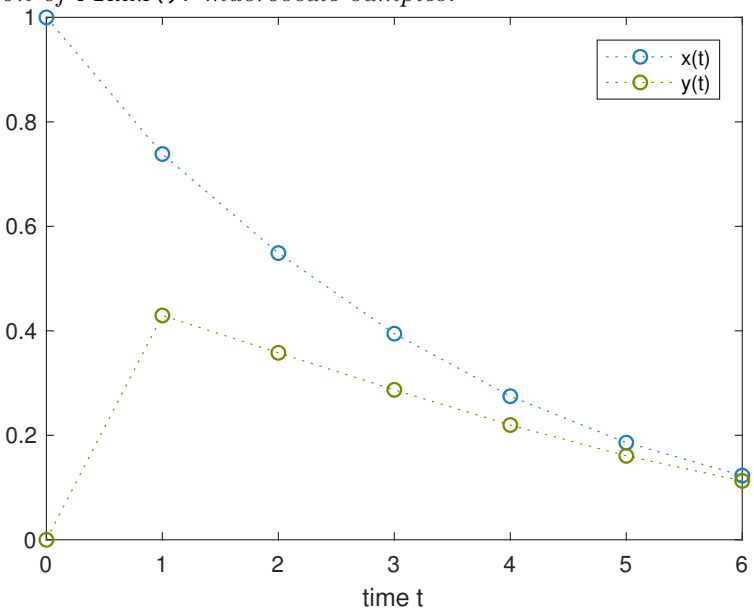


Figure 2.5: Michaelis–Menten enzyme kinetics simulated with the projective integration of `PIRK2()`*: the microscale bursts show the initial transients on a time scale of* $\epsilon = 0.1$*, and then the alignment along the slow manifold.*

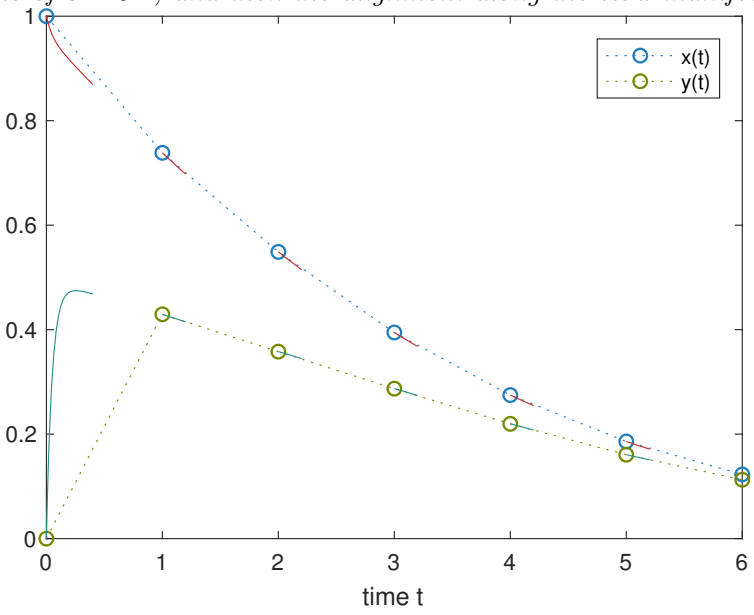
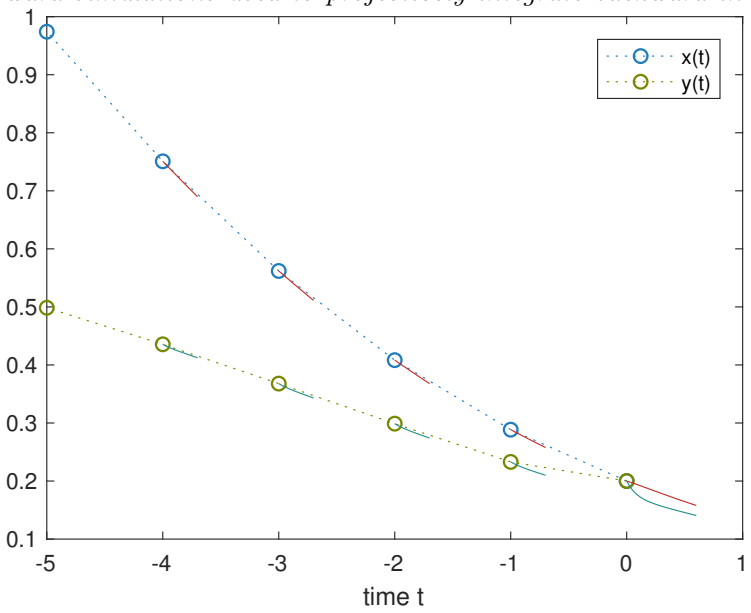


Figure 2.6: Michaelis–Menten enzyme kinetics at $\epsilon = 0.1$ simulated backward with the projective integration of `PIRK2()`: the microscale bursts show the short forward simulations used to projectively integrate backward in time.



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*, Frewen et al. 2009). 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)')
```

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
```

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

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

Section contents

2.4.1 Introduction

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.

```
30 function [T,X,tms,xms,svf] = PIG(macroInt,microBurst,Tspan,x0 ...
31                                  ,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 ith *row* of `Xs`, `Xs(i,:)`, is to be the vector $\vec{X}(t)$ at time $t =$ `Ts(i)`. Remember that in `PIG()` the function `F(T,X)` is to be estimated by Projective Integration.

- `microBurst()` is a function that produces output from the user-specified code for a burst of microscale simulation. The function must 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

```
180 if nargin==0
```

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

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

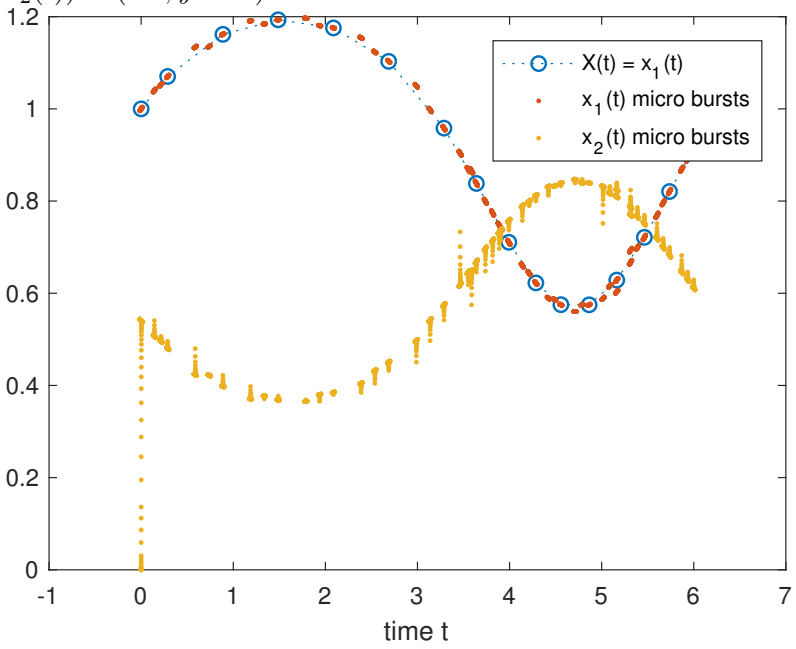
The macroscale variable is $X(t) = x_1(t)$, and the evolution dX/dt is unclear. With initial condition $X(0) = 1$, the following code computes and plots a solution of the system (2.1) over time $0 \le t \le 6$ for parameter $\epsilon = 10^{-3}$(Figure 2.7). Whenever needed by `microBurst()`, the microscale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

```
214 epsilon = 1e-3;
215 dxdt=@(t,x) [ cos(x(1))*sin(x(2))*cos(t)
216              ( 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*, Frewen et al. 2009).

```
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.

```
269 return
270 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.

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

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

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

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

```
308 nT = length(Tspan)-1;
309 nx = length(x0);
310 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.

```
320 if nargin<7
321     microBurst = @(t,x) cdmc(microBurst,t,x);
322 else
323 warning(['A ' class(cdmcFlag) ' seventh input to PIG().'...
324  ' PIG will not use constraint-defined manifold computing.'])
325 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.

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

Update the initial time.

```
346 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.

```
358 if saveMicro
359     tms=cell(nT+1,1); xms=cell(nT+1,1);
360     n=1;
361     tms{n} = reshape(relaxT,[],1);
362     xms{n} = x0MicroRelax;
363
364     if saveSvf
365         svf.T = cell(nT+1,1);
366         svf.dX = cell(nT+1,1);
367     else
368         svf = [];
369     end
370 else
371     tms = []; xms = []; svf = [];
372 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

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

Run a microBurst from the given macroscale initial values.

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

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

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

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

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

```
418   end
419 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.

```
432 PIF = @(t,x) PIFun(t,x);
433 [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.

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

Concatenate all the additional requested outputs into arrays.

```
450 if saveMicro
451     tms = cell2mat(tms);
452     xms = cell2mat(xms);
453     if saveSvf
454         svf.T = cell2mat(svf.T);
455         svf.dX = cell2mat(svf.dX);
456     end
457 end
```

2.4.4 If no output specified, then plot the simulation

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

This concludes `PIG()`.

```
476 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.

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

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

If no arguments, then execute an example

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

Example of Michaelis–Menton backwards in time The Michaelis–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 $t = -5$ (Frewen et al. 2009, e.g.). 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 $\Delta = -1$.

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

Upon finishing execution of the example, exit this function.

```
63 return
64 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 `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.

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

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

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

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

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

```
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,:);
```

```
273         t2=nan; xm2=nan(1,size(xm1,2));
274         t3=nan; t4=nan; xm3=xm2; xm4 = xm2; dx1=xm2; dx2=xm2;
275     else
```

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

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

Check for round-off error.

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

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

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

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

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

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

```
327     xint = xm1(end,:) + (Dt/2-abT)*dx1;
328     [t2,xm2] = microBurst(T+Dt/2, xint, bT);
329     del = t2(end)-t2(end-1);
330     dx2 = (xm2(end,:)-xm2(end-1,:))/del;
331
332     xint = xm1(end,:) + (Dt/2-abT)*dx2;
333     [t3,xm3] = microBurst(T+Dt/2, xint, bT);
334     del = t3(end)-t3(end-1);
335     dx3 = (xm3(end,:)-xm3(end-1,:))/del;
336
337     xint = xm1(end,:) + (Dt-abT)*dx3;
338     [t4,xm4] = microBurst(T+Dt, xint, bT);
339     del = t4(end)-t4(end-1);
340     dx4 = (xm4(end,:)-xm4(end-1,:))/del;
```

Check for round-off error.

```
346     xt = [reshape(t2(end-1:end),[],1) xm2(end-1:end,:)];
347     if norm(diff(xt))/norm(xt,'fro') < roundingTol
```

```
348     warning(['significant round-off error in 2nd projection at T=' num2str(T)
349     end
```

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

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

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

```
365     end
```

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

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

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

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

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

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

End of the main loop of all macro-steps.

```
411 end
```

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

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

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

```
428 if saveMicro
429     tms = cell2mat(tms);
430     xms = cell2mat(xms);
431     if saveFullMicro
```

```
432             rm.t = cell2mat(rm.t);
433             rm.x = cell2mat(rm.x);
434         end
435     end
```

2.5.3 If no output specified, then plot the simulation

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

This concludes `PIRK4()`.

```
453     end
```

2.6 cdmc(): constraint defined manifold computing

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

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

Input

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

Output

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

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

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

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

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

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

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

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

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

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

2.7 Example: PI using Runge–Kutta macrosolvers

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

Clear workspace and set a seed.

```
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);
```

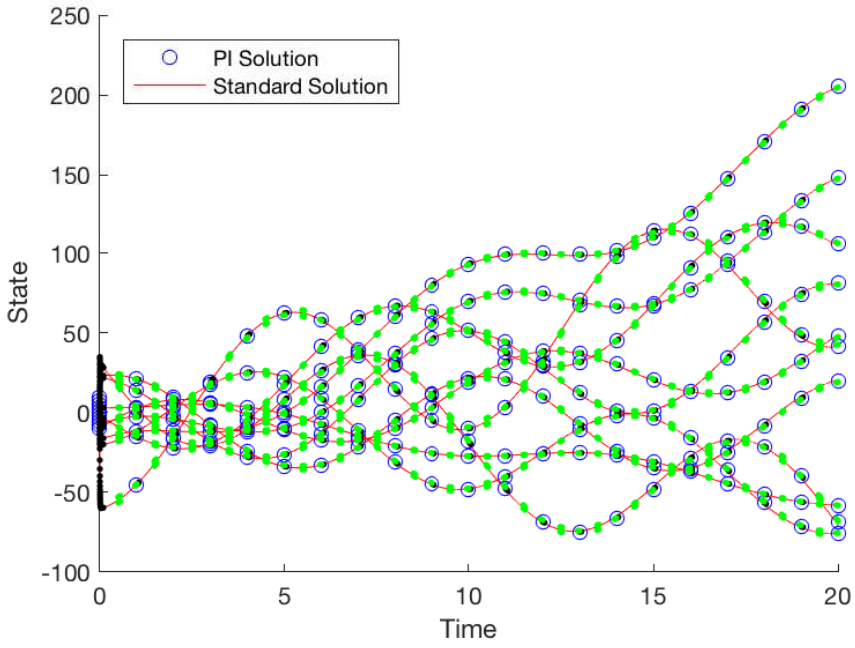
To verify, we also compute the trajectories using a standard integrator.

```
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');
```

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.*




```
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 $\texttt{dt} \leq 1/|\texttt{fastband}(1)|$ and the time to simulate with each application of the microsolver, `bT`, should be larger than or equal to $1/|\texttt{fastband}(2)|$. 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));
```

```
23  xs(1,:)=xi;
24  for k=2:nts
25      xi = xi + dt*dxdt(ts(k),xi.').';
26      xs(k,:)=xi;
27  end
28  end
```

2.8 Example: Projective Integration using General macrosolvers

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

```
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];
```

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).

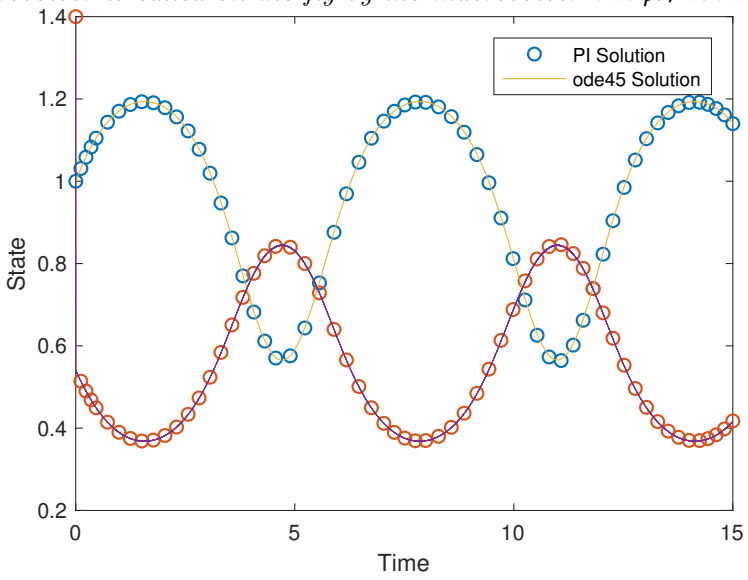
```
if ~exist('OCTAVE_VERSION','builtin')
    macInt='ode45'; else macInt='odeOct'; end
tic
[ts,xs,tms,xms] = PIG(macInt,microBurst,tSpan,x0);
secsPIGusingODEasMacro = toc
tic
[tClassic,xClassic] = feval(macInt,dxdt,tSpan,x0);
secsODEalone = toc
```

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,'-');
```

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




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

Figure 2.9 plots the output.

- The problem may be made more stiff or less stiff by changing the time-scale separation parameter $\epsilon =$ `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.

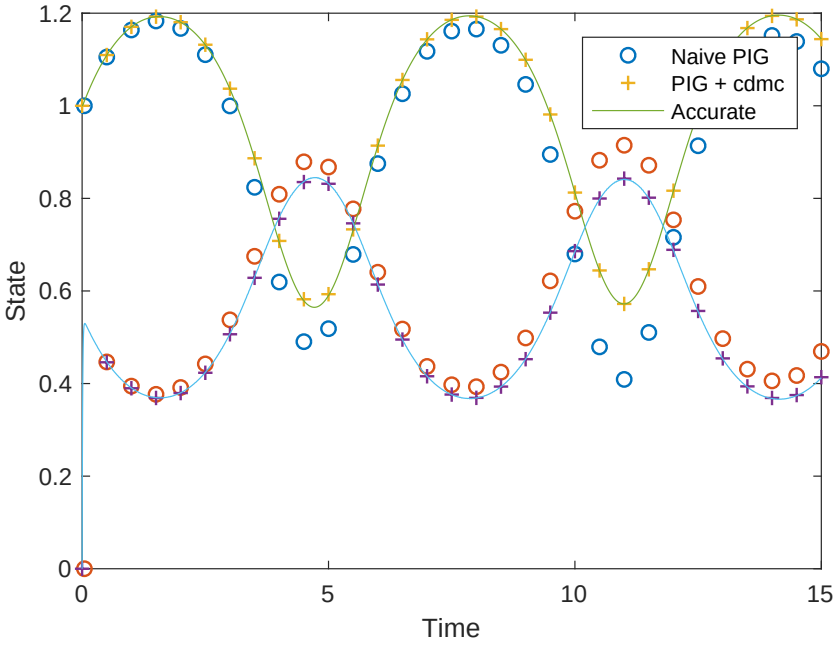
```
if ~exist('OCTAVE_VERSION','builtin')
    macInt='ode45'; else macInt='odeOct'; end
[nt,nx] = PIG(macInt,microBurst,tSpan,x0,[],[],'no cdmc');
[ct,cx] = PIG(macInt,microBurst,tSpan,x0);
[tClassic,xClassic] = feval(macInt,dxdt,tSpan,x0);
```

Figure 2.10 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`

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()`.



or `50*epsilon`[1] to worsen the error in both schemes. This example reflects a general principle: most Projective Integration schemes incur a global error term proportional to the burst time of the microsolver and independent of the order of the microsolver. The `PIRKn()` schemes are written to eliminate this error, but `PIG()` works with any user-defined macrosolver and cannot reduce this error, except by using the function `cdmc()`, its default.

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

2.10 To do/discuss

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

3 Patch scheme for given microscale discrete space system

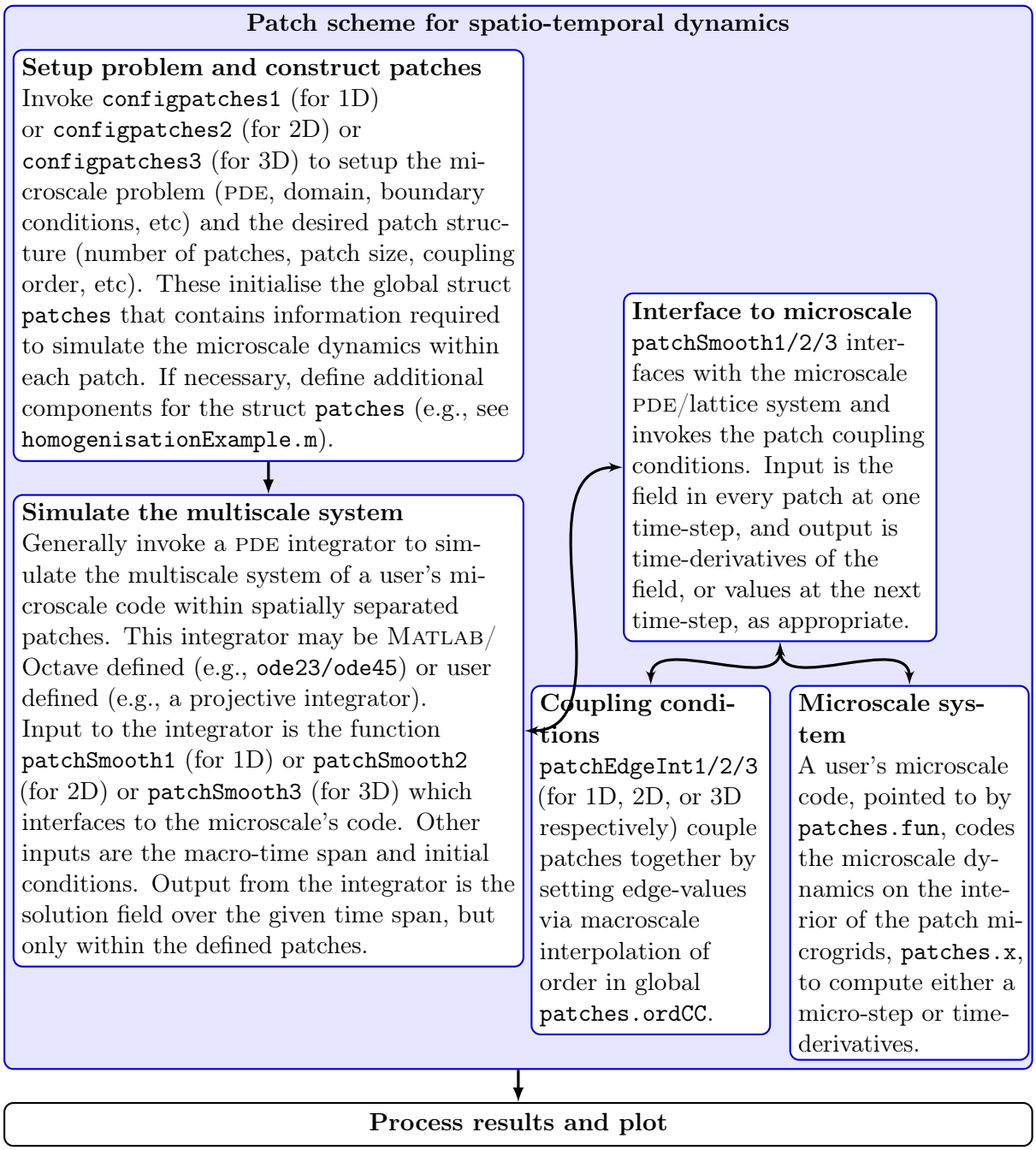
Chapter contents

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, Bunder et al. 2020); 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 3.1.1 and 3.10.1 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.

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 PDE*s, 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.*



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

Section contents

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

```
19 function patches = configPatches1(fun,Xlim,BCs ...
20     ,nPatch,ordCC,ratio,nSubP,varargin)
```

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

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)`, that computes time derivatives (or time-steps) of quantities on the 1D micro-grid within all the 1D 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 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 specifies staggered spatial grids.
- `ratio` (real) is the ratio of (depending upon `EdgyInt`) either the half-width or full-width of a patch to the spacing of the patch mid-points. So either `ratio` $= \frac{1}{2}$ means the patches abut and `ratio` $= 1$ is overlapping patches as in holistic discretisation, or `ratio` $= 1$ means the patches abut. Small `ratio` should greatly reduce computational time.
- `nSubP` is the number of equi-spaced microscale lattice points in each patch. If not using `EdgyInt`, then must be odd so that there is a centre-patch lattice point.
- `nEdge` (not yet implemented), *optional*, default=1, for each patch, 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 neighbour interactions).

- `EdgyInt`, true/false, *optional*, default=false. If true, then interpolate to left/right edge-values from right/left next-to-edge values. If false or omitted, then interpolate from centre-patch values.
- `nEnsem`, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.
- `hetCoeffs`, *optional*, default empty. Supply a 1/2D array of microscale heterogeneous coefficients to be used by the given microscale `fun` in each patch. Say the given array `cs` is of size $m_x \times n_c$, where n_c is the number of different sets of coefficients. The coefficients are to be the same for each and every patch; however, macroscale variations are catered for by the n_c coefficients being n_c parameters in some macroscale formula.
  - If `nEnsem` $= 1$, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the first point in each patch.
  - If `nEnsem` > 1 (value immaterial), then reset `nEnsem` $:= m_x$ and construct an ensemble of all m_x phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When `EdgyInt` is true, and when the coefficients are diffusivities/elasticities, then this coupling cunningly preserves symmetry .
- `nCore`, *optional-experimental*, default one, but if more, and only for non-EdgyInt, then interpolates from an average over the core of a patch, a core of size ??. Then edge values are set according to interpolation of the averages?? or so that average at edges is the interpolant??
- `'parallel'`, true/false, *optional*, default=false. If false, then all patch computations are on the user's main CPU—although a user may well separately invoke, say, a GPU to accelerate sub-patch computations.

  If true, and it requires that you have MATLAB's Parallel Computing Toolbox, then it will distribute the patches over multiple CPUs/cores. In MATLAB, only one array dimension can be split in the distribution, so it chooses the one space dimension x. A user may correspondingly distribute arrays with property `patches.codist`, or simply use formulas invoking the preset distributed arrays `patches.x`. If a user has not yet established a parallel pool, then a 'local' pool is started.

Output The struct `patches` is created and set with the following components. If no output variable is provided for `patches`, then make the struct available as a global variable.[1]

```
144 if nargout==0, global patches, end
```

- `.fun` is the name of the user's function `fun(t,u,patches)` or `fun(t,u)`, that computes the time derivatives (or steps) on the patchy lattice.

[1] When using `spmd` parallel computing, it is generally best to avoid global variables, and so instead prefer using an explicit output variable.

- `.ordCC` is the specified order of inter-patch coupling.
- `.stag` 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` (4D) is `nSubP`$\times 1 \times 1 \times$`nPatch` array of the regular spatial locations x_{iI} of the ith microscale grid point in the Ith patch.
- `.ratio` is the size ratio of every patch.
- `.nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.
- `.le`, `.ri` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.
- `.cs` either
  - `[]` 0D, or
  - if `nEnsem` $= 1$, (`nSubP(1)` $- 1) \times n_c$ 2D array of microscale heterogeneous coefficients, or
  - if `nEnsem` > 1, (`nSubP(1)` $- 1) \times n_c \times m_x$ 3D array of m_x ensemble of phase-shifts of the microscale heterogeneous coefficients.
- `.parallel`, logical: true if patches are distributed over multiple CPUs/ cores for the Parallel Computing Toolbox, otherwise false (the default is to activate the *local* pool).
- `.codist`, *optional*, describes the particular parallel distribution of arrays over the active parallel pool.

3.1.1 If no arguments, then execute an example

```
209 if nargin==0
```

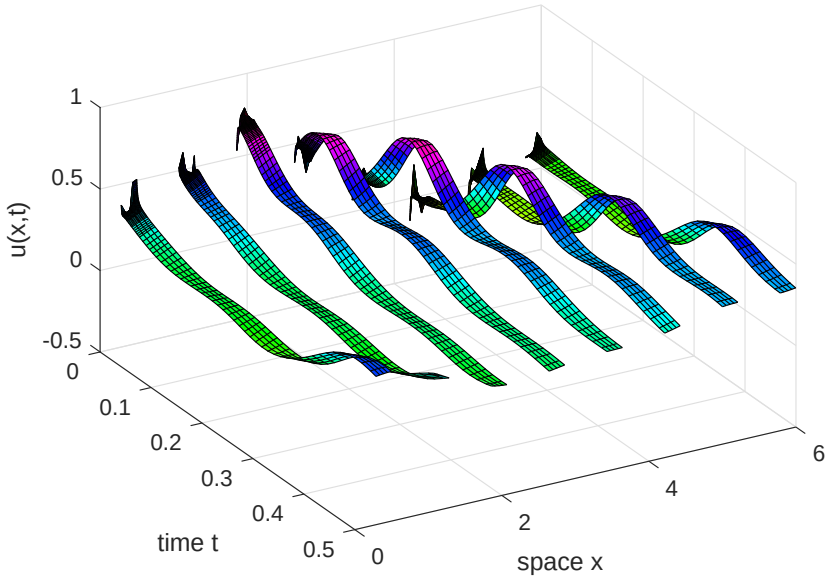
The code here shows one way to get started: a user's script may have the following three steps (left-right arrows denote function recursion).

1. configPatches1
2. ode15s integrator $\leftrightarrow$ patchSmooth1 $\leftrightarrow$ user's 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.

```
230 global patches
231 patches = configPatches1(@BurgersPDE,[0 2*pi], nan, 8, 0, 0.2, 7);
```

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



Set some initial condition, with some microscale randomness.

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

Simulate in time using a standard stiff integrator and the interface function `patchsmooth1()` (Section 3.2).

```
245 if ~exist('OCTAVE_VERSION','builtin')
246 [ts,us] = ode15s( @patchSmooth1,[0 0.5],u0(:));
247 else % octave version
248 [ts,us] = odeOcts(@patchSmooth1,[0 0.5],u0(:));
249 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.

```
261 figure(1),clf
262 if 1, patches.x([1 end],:,:,:)=nan;  us=us.';
263 else us=reshape(patchEdgyInt1(us.'),[],length(ts));
264 end
265 surf(ts,patches.x(:),us)
266 view(60,40), colormap(hsv)
267 title('Burgers PDE: patches in space, continuous time')
268 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
269 ifOurCf2eps(mfilename)
```

Upon finishing execution of the example, exit this function.

```
280 return
```

```
281 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 $\dot{u}_{ij} = \frac{1}{\delta x^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) - 30u_{ij}\frac{1}{2\delta x}(u_{i+1,j} - u_{i-1,j})$. Here there is only one field variable, and one in the ensemble, so for simpler coding of the PDE we squeeze them out (with no need to reshape when via patchSmooth1()).

```
15 function ut=BurgersPDE(t,u,patches)
16   u=squeeze(u);     % omit singleton dimensions
17   dx=diff(patches.x(1:2));  % microscale spacing
18   i=2:size(u,1)-1;  % interior points in patches
19   ut=nan+u;         % preallocate output array
20   ut(i,:)=diff(u,2)/dx^2 ...
21     -30*u(i,:).*(u(i+1,:)-u(i-1,:))/(2*dx);
22 end
```

```
10 function [ts,xs] = odeOcts(dxdt,tSpan,x0)
11     if length(tSpan)>2, ts = tSpan;
12     else ts = linspace(tSpan(1),tSpan(end),21)';
13     end
14     lsode_options('integration method','non-stiff');
15     xs = lsode(@(x,t) dxdt(t,x),x0,ts);
16 end
```

3.1.2 Parse input arguments and defaults

```
297 p = inputParser;
298 fnValidation = @(f) isa(f, 'function_handle'); %test for fn name
299 addRequired(p,'fun',fnValidation);
300 addRequired(p,'Xlim',@isnumeric);
301 addRequired(p,'BCs'); % nothing yet decided
302 addRequired(p,'nPatch',@isnumeric);
303 addRequired(p,'ordCC',@isnumeric);
304 addRequired(p,'ratio',@isnumeric);
305 addRequired(p,'nSubP',@isnumeric);
306 addParameter(p,'nEdge',1,@isnumeric);
307 addParameter(p,'EdgyInt',false,@islogical);
308 addParameter(p,'nEnsem',1,@isnumeric);
309 addParameter(p,'hetCoeffs',[],@isnumeric);
310 addParameter(p,'parallel',false,@islogical);
311 addParameter(p,'nCore',1,@isnumeric);
312 parse(p,fun,Xlim,BCs,nPatch,ordCC,ratio,nSubP,varargin{:});
```

Set the optional parameters.

```
318 patches.nEdge = p.Results.nEdge;
319 patches.EdgyInt = p.Results.EdgyInt;
320 patches.nEnsem = p.Results.nEnsem;
321 cs = p.Results.hetCoeffs;
```

```
322 patches.parallel = p.Results.parallel;
323 patches.nCore = p.Results.nCore;
```

Check parameters.

```
330 assert(patches.nEdge==1 ...
331       ,'multi-edge-value interp not yet implemented')
332 assert(2*patches.nEdge+1<=nSubP ...
333       ,'too many edge values requested')
334 if patches.nCore>1
335     warning('nCore>1 not yet tested in this version')
336     end
```

3.1.3 The code to make patches and interpolation

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

```
348 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.

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

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

```
365 patches.stag=mod(ordCC,2);
366 ordCC=ordCC+patches.stag;
367 patches.ordCC=ordCC;
```

Check for staggered grid and periodic case.

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

Might as well precompute the weightings to interpolate field values for coupling. (Could sometime extend to coupling via derivative values.) Store the size ratio in `patches`.

```
384 patches.ratio=ratio;
385 if ordCC>0
386     [Cwtsr,Cwtsl] = patchCwts(ratio,ordCC,patches.stag);
387     patches.Cwtsr = Cwtsr;  patches.Cwtsl = Cwtsl;
388 end
```

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

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

Construct the microscale grid in each patch, assuming Dirichlet patch edges, and a half-patch length of `ratio` · `DX`, unless `patches.EdgyInt` is true in which case the patches are of length `ratio*DX+dx`. Reshape the grid to be 4D to suit dimensions (micro,Vars,Ens,macro).

```
409 assert(patches.EdgyInt | mod(nSubP,2)==1, ...
410     'configPatches1: nSubP must be odd')
411 i0=(nSubP+1)/2;
412 if ~patches.EdgyInt, dx = ratio*DX/(i0-1);
413 else                 dx = ratio*DX/(nSubP-2);
414 end
415 patches.x = dx*(-i0+1:i0-1)'+X; % micro-grid
416 patches.x = reshape(patches.x,nSubP,1,1,nPatch);
```

3.1.4 Set ensemble inter-patch communication

For `EdgyInt` or centre interpolation respectively,

- the right-edge/centre realisations `1:nEnsem` are to interpolate to left-edge `le`, and
- the left-edge/centre realisations `1:nEnsem` are to interpolate to `re`.

`re` and `li` are 'transposes' of each other as `re(li)=le(ri)` are both `1:nEnsem`. Alternatively, one may use the statement

```
c=hankel(c(1:nSubP-1),c([nSubP 1:nSubP-2]));
```

to *correspondingly* generates all phase shifted copies of microscale heterogeneity (see `homoDiffEdgy1` of Section 3.5).

The default is nothing shifty. This setting reduces the number of if-statements in function `patchEdgeInt1()`.

```
446 nE = patches.nEnsem;
447 patches.le = 1:nE;
448 patches.ri = 1:nE;
```

However, if heterogeneous coefficients are supplied via `hetCoeffs`, then do some non-trivial replications. First, get microscale periods, patch size, and replicate many times in order to subsequently sub-sample: `nSubP` times should be enough. If `cs` is more then 2D, then the higher-dimensions are reshaped into the 2nd dimension.

```
460 if ~isempty(cs)
461   [mx,nc] = size(cs);
462   nx = nSubP(1);
463   cs = repmat(cs,nSubP,1);
```

If only one member of the ensemble is required, then sub-sample to patch size, and store coefficients in `patches` as is.

```
471   if nE==1, patches.cs = cs(1:nx-1,:); else
```

But for `nEnsem` > 1 an ensemble of m_x phase-shifts of the coefficients is constructed from the over-supply. Here code phase-shifts over the periods—the phase shifts are like Hankel-matrices.

```
480     patches.nEnsem = mx;
481     patches.cs = nan(nx-1,nc,mx);
482     for i = 1:mx
483         is = (i:i+nx-2);
484         patches.cs(:,:,i) = cs(is,:);
485     end
486     patches.cs = reshape(patches.cs,nx-1,nc,[]);
```

Further, set a cunning left/right realisation of inter-patch coupling. The aim is to preserve symmetry in the system when also invoking `EdgyInt`. What this coupling does without `EdgyInt` is unknown. Use auto-replication.

```
496     patches.le = mod((0:mx-1)'+mod(nx-2,mx),mx)+1;
497     patches.ri = mod((0:mx-1)'-mod(nx-2,mx),mx)+1;
```

Issue warning if the ensemble is likely to be affected by lack of scale separation. Need to justify this and the arbitrary threshold more carefully??

```
505 if ratio*patches.nEnsem>0.9, warning( ...
506 'Probably poor scale separation in ensemble of coupled phase-shifts')
507 scaleSeparationParameter = ratio*patches.nEnsem
508 end
```

End the two if-statements.

```
514   end%if-else nEnsem>1
515 end%if not-empty(cs)
```

If parallel code then first assume this is not within an `spmd`-environment, and so we invoke `spmd...end` (which starts a parallel pool if not already started). At this point, the global `patches` is copied for each worker processor and so it becomes *composite* when we distribute any one of the fields. Hereafter, *all fields in the global variable* `patches` *must only be referenced within an* `spmd`*-environment.*[2]

```
534 if patches.parallel
535 %  theparpool=gcp()
536   spmd
```

Second, choose to slice parallel workers in the spatial direction.

```
543   pari = 1;
544   patches.codist=codistributor1d(3+pari);
```

`patches.codist.Dimension` is the index that is split among workers. Then distribute the coordinate direction among the workers: the function must be invoked inside an `spmd`-group in order for this to work—so we do not need `parallel` in argument list.

[2] If subsequently outside spmd, then one must use functions like `getfield(patches{1},'a')`.

```
554   switch pari
555     case 1, patches.x=codistributed(patches.x,patches.codist);
556   otherwise
557     error('should never have bad index for parallel distribution')
558   end%switch
559   end%spmd
```

If not parallel, then clean out `patches.codist` if it exists. May not need, but safer.

```
567 else% not parallel
568   if isfield(patches,'codist'), rmfield(patches,'codist'); end
569 end%if-parallel
```

Fin

```
578 end% function
```

3.2 patchSmooth1(): interface 1D space to time integrators

To simulate in time with 1D 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 enough* so that the patch centre-values are sensible macroscale variables, and patch edge values are determined by macroscale interpolation of the patch-centre or edge values. Nonetheless, microscale heterogeneous systems may be accurately simulated with this function via appropriate interpolation. Communicate patch-design variables (Section 3.1) either via the global struct `patches` or via an optional third argument (except that this last is required for parallel computing of `spmd`).

```
28 function dudt=patchSmooth1(t,u,patches)
29 if nargin<3, global patches, end
```

Input

- `u` is a vector/array of length `nSubP`$\cdot$`nVars`$\cdot$`nEnsem`$\cdot$`nPatch` where there are `nVars`$\cdot$`nEnsem` field values at each of the points in the `nSubP`$\times$`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,patches)` that computes the time derivatives on the patchy lattice. The array `u` has size `nSubP` $\times$ `nVars` $\times$ $\times$`nEnsemnPatch`. Time derivatives must be computed into the same sized array, although herein the patch edge values are overwritten by zeros.
  - `.x` is `nSubP` $\times 1 \times 1 \times$ `nPatch` array of the spatial locations x_i of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.

Output

- `dudt` is a vector/array of of time derivatives, but with patch edge-values set to zero. It is of total length `nSubP`$\cdot$`nVars`$\cdot$`nEnsem`$\cdot$`nPatch` and the same dimensions as `u`.

Reshape the fields `u` as a 4D-array, and sets the edge values from macroscale interpolation of centre-patch values. Section 3.3 describes `patchEdgeInt1()`.

```
84 sizeu = size(u);
85 u = patchEdgeInt1(u,patches);
```

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 the user/ integrator as same sized array as input.

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

Fin.

3.3 `patchEdgeInt1()`: sets patch-edge values from interpolation over the 1D macroscale

Couples 1D patches across 1D space by computing their edge values from macroscale interpolation of either the mid-patch value (Roberts 2003, Roberts & Kevrekidis 2007), or the patch-core average (Bunder et al. 2017), or the opposite next-to-edge values—this last alternative often maintains symmetry. This function is primarily used by `patchSmooth1()` but is also useful for user graphics. 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).

Communicate patch-design variables via a second argument (optional, except required for parallel computing of `spmd`) or otherwise via the global struct `patches`.

```
30 function u=patchEdgeInt1(u,patches)
31 if nargin<2, global patches, end
```

Input

- `u` is a vector/array of length $\texttt{nSubP}\cdot\texttt{nVars}\cdot\texttt{nEnsem}\cdot\texttt{nPatch}$ where there are $\texttt{nVars}\cdot\texttt{nEnsem}$ field values at each of the points in the $\texttt{nSubP}\times\texttt{nPatch}$ grid.
- `patches` a struct largely set by `configPatches1()`, and which includes the following.
  - `.x` is $\texttt{nSubP}\times 1\times 1\times\texttt{nPatch}$ array of the spatial locations x_{iI} 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.
  - `.stag` in $\{0,1\}$ is one for staggered grid (alternating) interpolation.
  - `.Cwtsr` and `.Cwtsl` define the coupling coefficients for finite width interpolation.
  - `.EdgyInt` true/false is true for interpolating patch-edge values from opposite next-to-edge values (often preserves symmetry).
  - `.nEnsem` the number of realisations in the ensemble.
  - `.parallel` whether serial or parallel.

Output

- `u` is 4D array, $\texttt{nSubP}\times\texttt{nVars}\times\texttt{nEnsem}\times\texttt{nPatch}$, of the fields with edge values set by interpolation.

Test for reality of the field values, and define a function accordingly. Could be problematic if some variables are real and some are complex, or if variables are of quite different sizes.

```
98    if max(abs(imag(u(:))))<1e-9*max(abs(u(:)))
99         uclean=@(u) real(u);
100   else uclean=@(u) u;
101   end
```

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

```
109 [nx,~,~,Nx] = size(patches.x);
110 nEnsem = patches.nEnsem;
111 nVars = round(numel(u)/numel(patches.x)/nEnsem);
112 assert(numel(u) == nx*nVars*nEnsem*Nx ...
113   ,'patchEdgeInt1: input u has wrong size for parameters')
114 u = reshape(u,nx,nVars,nEnsem,Nx);
```

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. Get the size ratios of the patches.

```
123 r = patches.ratio(1);
```

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

```
134 I = 1:Nx; Ip = mod(I,Nx)+1; Im = mod(I-2,Nx)+1;
```

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

```
141 i0 = round((nx+1)/2);
142 c = round((patches.nCore-1)/2);
```

Lagrange interpolation gives patch-edge values Consequently, compute centred differences of the patch core/edge averages/values for the macro-interpolation of all fields. Assumes the domain is macro-periodic. Should revise to use 5D arrays in order to be consistent with 2D and 3D code?? and to work with parallel??

```
162 if patches.ordCC>0 % then finite-width polynomial interpolation
```

```
167   if patches.EdgyInt % next-to-edge values
168     Ux = u([2 nx-1],:,:,I);
169   else % interpolate mid-patch values/sums
170     Ux = sum( u((i0-c):(i0+c),:,:,I) ,1);
171   end;
```

Just in case any last array dimension(s) are one, we have to force a padding of the sizes, then adjoin the extra dimension for the subsequent array of differences.

```
179 szUx0=size(Ux);
180 szUx0=[szUx0 ones(1,4-length(szUx0)) patches.ordCC];
```

Use finite difference formulas for the interpolation, so store finite differences in these arrays. When parallel, in order to preserve the distributed array structure we use an index at the end for the differences.

```
189   if patches.parallel
190     dmu = zeros(szUx0,patches.codist); % 5D
191   else
192     dmu = zeros(szUx0); % 5D
193   end
```

First compute differences, either μ and δ, or $\mu\delta$ and δ^2 in space.

```
200   if patches.stag % use only odd numbered neighbours
201     dmu(:,:,:,I,1) = (Ux(:,:,:,Ip)+Ux(:,:,:,Im))/2; % \mu
202     dmu(:,:,:,I,2) = (Ux(:,:,:,Ip)-Ux(:,:,:,Im)); % \delta
203     Ip = Ip(Ip); Im = Im(Im); % increase shifts to \pm2
204   else % standard
205     dmu(:,:,:,I,1) = (Ux(:,:,:,Ip)-Ux(:,:,:,Im))/2; % \mu\delta
206     dmu(:,:,:,I,2) = (Ux(:,:,:,Ip)-2*Ux(:,:,:,I) ...
207                       +Ux(:,:,:,Im)); % \delta^2
208   end% if odd/even
```

Recursively take δ^2 of these to form higher order centred differences in space.

```
215   for k = 3:patches.ordCC
216     dmu(:,:,:,:,k) =       dmu(:,:,:,Ip,k-2) ...
217       -2*dmu(:,:,:,I,k-2) +dmu(:,:,:,Im,k-2);
218   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.

For the case where single-point values interpolate to patch-edge values: when we have an ensemble of configurations, different realisations are coupled to each other as specified by `patches.le` and `patches.ri`.

```
233   if patches.nCore==1
234     k=1+patches.EdgyInt; % use centre/core or two edges
235     u(nx,:,patches.ri,I) = Ux(1,:,:,:)*(1-patches.stag) ...
236       +sum( shiftdim(patches.Cwtsr,-4).*dmu(1,:,:,:,:) ,5);
237     u(1 ,:,patches.le,I) = Ux(k,:,:,:)*(1-patches.stag) ...
238       +sum( shiftdim(patches.Cwtsl,-4).*dmu(k,:,:,:,:) ,5);
```

For a non-trivial core then more needs doing: 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.

```
248   else error('not yet considered, july--dec 2020 ??')
249     u(nx,:,:,I) = Ux(:,:,I)*(1-patches.stag) ...
250       + reshape(-sum(u((nx-patches.nCore+1):(nx-1),:,:,I),1) ...
251       +sum( patches.Cwtsr.*dmu ),Nx,nVars);
252     u(1,:,:,I) = Ux(:,:,I)*(1-patches.stag) ...
```

```
253         +reshape(-sum(u(2:patches.nCore,:,:,I),1)  ...
254         +sum( patches.Cwtsl.*dmu ),Nx,nVars);
255     end;
```

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

```
265 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 `Nx` patches we resolve 'wavenumbers' $|k| <$ `Nx`$/2$, so set row vector `ks` $= k2\pi/N$ for 'wavenumbers' $k = (0\,, 1\,, \ldots, k_{\max}\,, -k_{\max}\,, \ldots, -1)$ for odd N, and $k = (0\,, 1\,, \ldots, k_{\max}\,, (k_{\max}+1)\,, -k_{\max}\,, \ldots, -1)$ 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??

```
289   if patches.stag % transform by doubling the number of fields
290     v = nan(size(u)); % currently to restore the shape of u
291     u = [u(:,:,:,1:2:Nx) u(:,:,:,2:2:Nx)];
292     stagShift = 0.5*[ones(1,nVars) -ones(1,nVars)];
293     iV = [nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
294     r = r/2;           % ratio effectively halved
295     Nx = Nx/2; % halve the number of patches
296     nVars = nVars*2;   % double the number of fields
297   else % the values for standard spectral
298     stagShift = 0;
299     iV = 1:nVars;
300   end
```

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

```
307   kMax = floor((Nx-1)/2);
308   ks = shiftdim( ...
309       2*pi/Nx*(mod((0:Nx-1)+kMax,Nx)-kMax) ...
310       ,-2);
```

Compute the Fourier transform across patches of the patch centre or next-to-edge 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. When using an ensemble of configurations, different configurations might be coupled to each other, as specified by `patches.le` and `patches.ri`.

```
323 if ~patches.EdgyInt
324     Cleft = fft(u(i0  ,:,:,:),[],4);
325     Cright = Cleft;
326 else
327     Cleft = fft(u(2   ,:,:,:),[],4);
```

```
328      Cright= fft(u(nx-1,:,:,:),[],4);
329  end
```

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

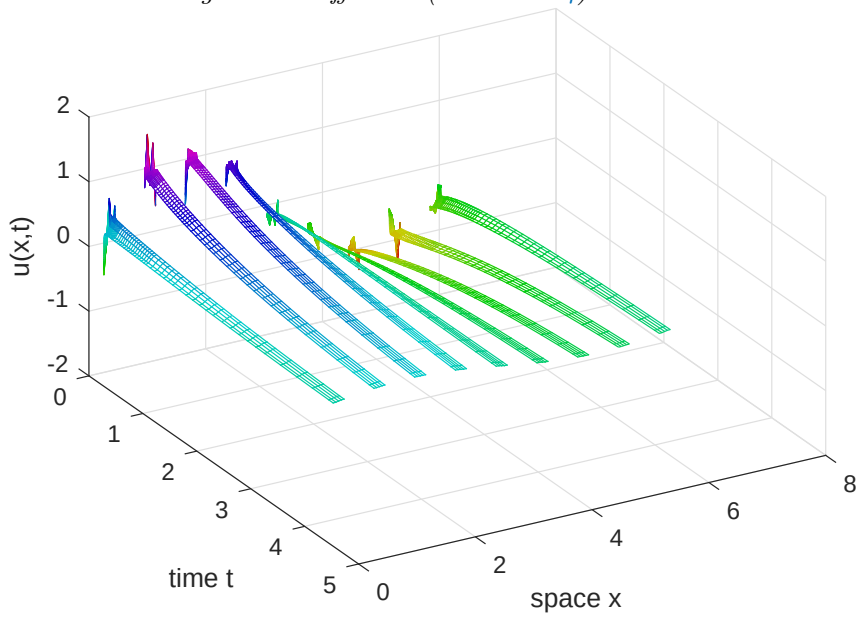
```
336    u(nx,iV,patches.ri,:) = uclean( ifft( ...
337        Cleft.*exp(1i*ks.*(stagShift+r)) ,[],4));
338    u(1 ,iV,patches.le,:) = uclean( ifft( ...
339        Cright.*exp(1i*ks.*(stagShift-r)) ,[],4));
```

Restore staggered grid when appropriate. This dimensional shifting appears to work. Is there a better way to do this??

```
347  if patches.stag
348    nVars = nVars/2;
349    u=reshape(u,nx,nVars,2,nEnsem,Nx);
350    Nx = 2*Nx;
351    v(:,:,:,1:2:Nx) = u(:,:,1,:,:);
352    v(:,:,:,2:2:Nx) = u(:,:,2,:,:);
353    u = v;
354  end
355  end% if spectral
```

Fin, returning the 4D array of field values.

Figure 3.3: the diffusing field $u(x,t)$ in the patch (gap-tooth) scheme applied to microscale heterogeneous diffusion (Section 3.4).



3.4 `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 ↔ patchSmooth1 ↔ heteroDiff
3. process results

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

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

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

3.4.1 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).

```
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. Here include the diffusivity coefficients, repeated to fill up a patch.

```
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,'hetCoeffs',cDiff);
```

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 3.2) to the microscale differential equations.

```
u0 = sin(patches.x)+0.3*randn(nSubP,1,1,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(:)),[]);
```

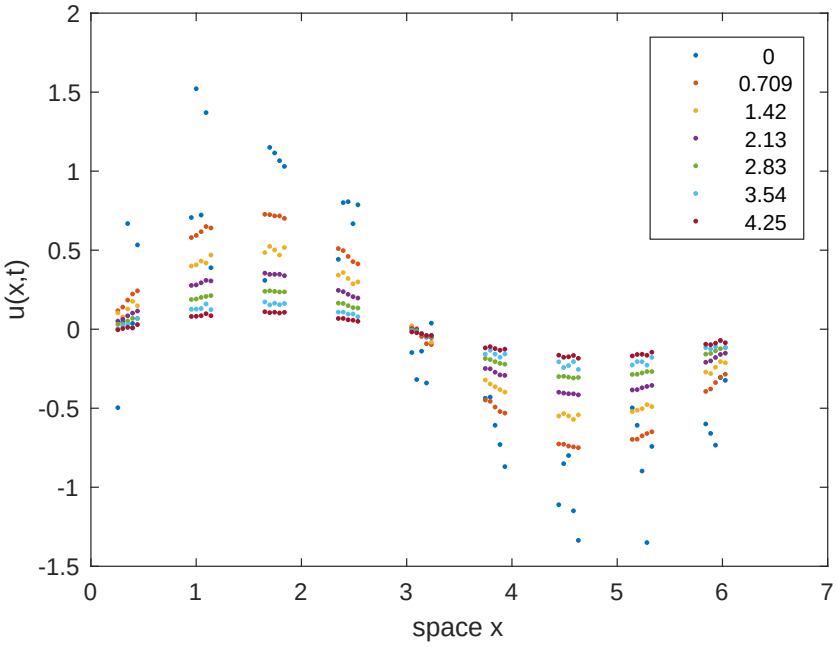
Plot the simulation in Figure 3.3.

```
figure(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)')
ifOurCf2eps([mfilename 'CtsU'])
```

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','non-stiff');
```

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.




```
15      xs = lsode(@(x,t) dxdt(t,x),x0,ts);
16  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 3.4.3), as illustrated by Figure 3.4.

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

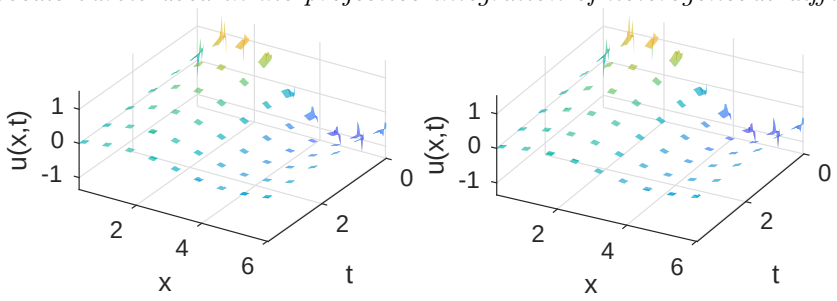
1. configPatches1 (done in first part)
2. PIRK2 ↔ heteroBurst ↔ micro-integrator ↔ patchSmooth1 ↔ heteroDiff
3. process results

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

```
141 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.

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.




```
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))
ifOurCf2eps([mfilename 'U'])
```

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

```
figure(3),clf
for k = 1:2, subplot(2,2,k)
  surf(tss,xs(:),uss',  'EdgeColor','none')
  ylabel('x'), xlabel('t'), zlabel('u(x,t)')
  axis tight, view(126-4*k,45)
end
ifOurCf2eps([mfilename 'Micro'])
```

End of this example script.

3.4.2 `heteroDiff()`: heterogeneous diffusion

This function codes the lattice heterogeneous diffusion inside the patches. For 2D input arrays `u` and `x` (via edge-value interpolation of `patchSmooth1`, Section 3.2), computes the time derivative (3.1) at each point in the interior of a patch, output in `ut`. The column vector of diffusivities c_i, and possibly Burgers' advection coefficients b_i, have previously been stored in struct `patches.cs`.

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

```
26    % possibly include heterogeneous Burgers' advection
27    if size(patches.cs,2)>1 % check for advection coeffs
28        buu = patches.cs(:,2,:).*u.^2;
29        ut(i,:) = ut(i,:)-(buu(i+1,:)-buu(i-1,:))/(dx*2);
30    end
31  end% function
```

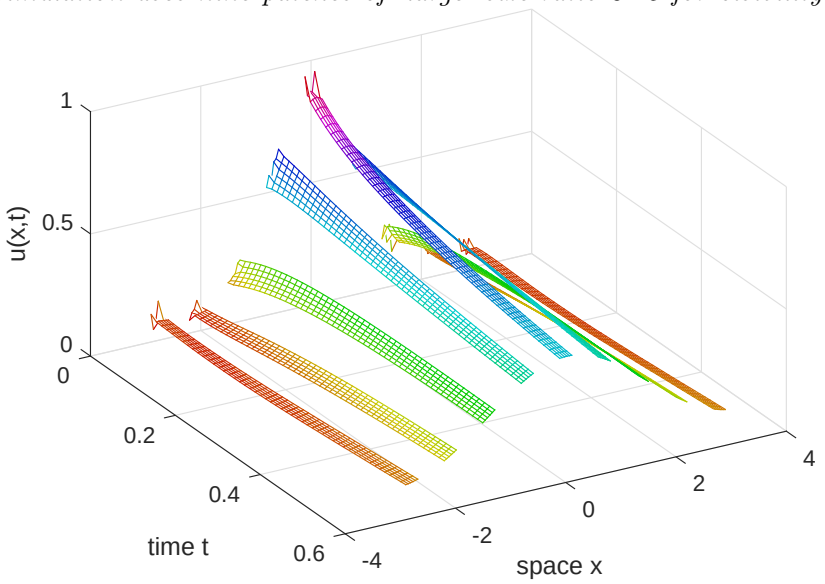
3.4.3 `heteroBurst()`: a burst of heterogeneous diffusion

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

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

Fin.

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.*



3.5 `homoDiffEdgy1`: computational homogenisation of a 1D heterogeneous 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

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

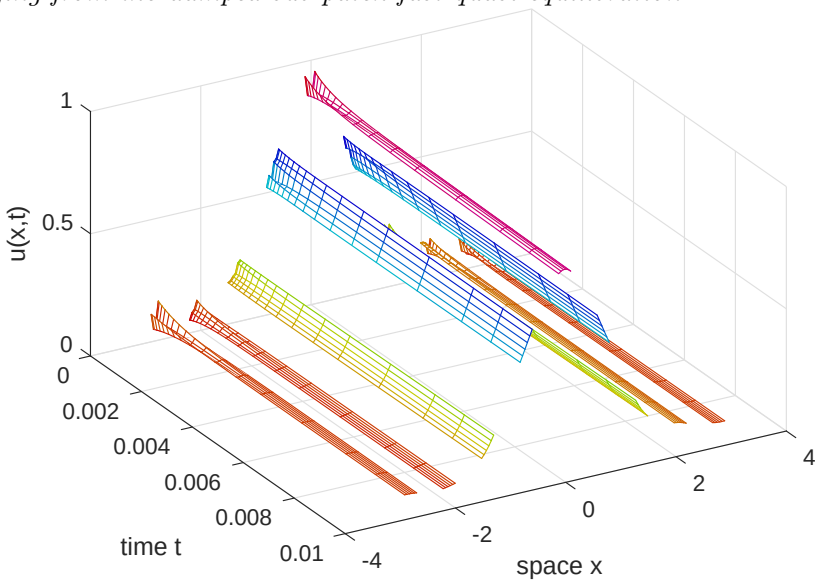
in terms of the centred difference operator δ. The system has a microscale heterogeneity via the coefficients $c_{i+1/2}$ which we assume to have some given known periodicity. Figure 3.6 shows one patch simulation of this system: observe the effects of the heterogeneity within each patch.

3.5.1 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
2. ode15s ↔ patchSmooth1 ↔ heteroDiff
3. plot the simulation

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.



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 to fill each patch, and phase-shifted for an ensemble.

```
mPeriod = 3%randi([2 5])
% set random diffusion coefficients
cHetr=exp(0.3*randn(mPeriod,1));
%cHetr = [3.966;2.531;0.838;0.331;7.276];
cHetr = cHetr*mean(1./cHetr) % normalise
```

Establish the global data struct `patches` for the microscale heterogeneous 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 reasonable decay rate for heterogeneous diffusion. When simulating an ensemble of configurations, `nSubP` (the number of points in a patch) need not be dependent on the period of the heterogeneous diffusion.

```
global patches
nPatch = 9
ratio = 0.25;
```

```
nSubP = mPeriod+1 %randi([mPeriod+1 2*mPeriod+2])
nEnsem = mPeriod % number realisations in ensemble
if mod(nSubP,mPeriod)==2, nEnsem=1, end
configPatches1(@heteroDiff,[-pi pi],nan,nPatch ...
    ,4,ratio,nSubP,'EdgyInt',true,'nEnsem',nEnsem ...
    ,'hetCoeffs',cHetr);
```

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

```
u0 = 0.8*exp(-patches.x.^2)+0.2*rand(nSubP,1,nEnsem,nPatch);
du0dt = patchSmooth1(0,u0(:));
```

Integrate using standard integrators.

```
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@patchSmooth1, [0 0.6], u0(:));
else % octave version
    [ts,us] = odeOcts(@patchSmooth1, 0.6*linspace(0,1).^2, 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 `nan`s in between patches. For the field values (which are rows in `us`) we need to reshape, permute, interpolate to get edge values, pad with `nan`s, and reshape again. In the case of an ensemble of phase-shifts, we plot the mean over the ensemble.

```
xs = squeeze(patches.x);
us = patchEdgeInt1( permute( reshape(us ...
    ,length(ts),nSubP,nEnsem,nPatch) ,[2 1 3 4]) );
ustd = squeeze(std(us,0,3));
us = squeeze(mean(us,3));
if 0, % omit interpolated edges
    us([1 end],:,:) = nan;
    ustd([1 end],:,:) = nan;
else % insert nans between patches
    xs(end+1,:) = nan;
    us(end+1,:,:) = nan;
    ustd(end+1,:,:) = nan;
end
us=reshape(permute(us,[1 3 2]),[],length(ts));
ustd=reshape(permute(ustd,[1 3 2]),[],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);
```

```
  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), colormap(0.8*hsv)
  xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
  ifOurCf2eps([mfilename 'U' num2str(p)])
end
pause(3)
```

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.

```
nPatch = 13
ratio = 0.01;

leadingEvals=[];
for ord=0:2:8
  ordInterp=ord
  configPatches1(@heteroDiff,[-pi pi],nan,nPatch ...
      ,ord,ratio,nSubP,'EdgyInt',true,'nEnsem',nEnsem ...
    ,'hetCoeffs',cHetr);
```

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 = zeros(nSubP,1,nEnsem,nPatch);
  u0([1 end],:,:,:)=nan; 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
  nonSymmetric=norm(Jac-Jac')
  assert(nonSymmetric<5e-9,'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.

The number of zero eigenvalues, `nZeroEv`, indicates the number of decoupled systems in this patch configuration.

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
```

```
280   [evecs,evals]=eig(Jac);
281   eval=-sort(-diag(real(evals)));
282   nZeroEv=sum(eval(:)>-1e-5)
283   leadingEvals=[leadingEvals eval(1:3*nPatch)];
284 %  leadingEvals=[leadingEvals eval([1, (nZeroEv+1):2:(nZeroEv*nPatch+4)])];
```

End of the for-loop over orders of interpolation, and output the tables of eigenvalues.

```
291 end
292 disp('     spectral    quadratic      quartic  sixth-order  ...')
293 leadingEvals=leadingEvals
```

End of the main script.

3.5.2 `heteroDiff()`: heterogeneous diffusion

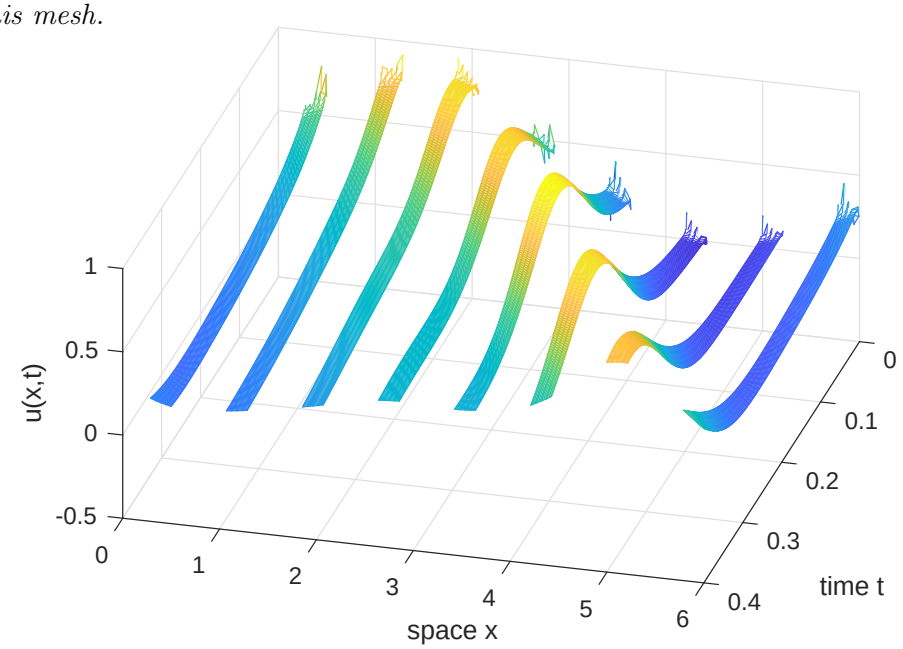
This function codes the lattice heterogeneous diffusion inside the patches. For 2D input arrays `u` and `x` (via edge-value interpolation of `patchSmooth1`, Section 3.2), computes the time derivative (3.1) at each point in the interior of a patch, output in `ut`. The column vector of diffusivities c_i, and possibly Burgers' advection coefficients b_i, have previously been stored in struct `patches.cs`.

```
21 function ut = heteroDiff(t,u,patches)
22   dx = diff(patches.x(2:3));   % space step
23   i = 2:size(u,1)-1;   % interior points in a patch
```

```
24    ut = nan+u;          % preallocate output array
25    ut(i,:,:,:) = diff(patches.cs(:,1,:).*diff(u))/dx^2;
26    % possibly include heterogeneous Burgers' advection
27    if size(patches.cs,2)>1 % check for advection coeffs
28        buu = patches.cs(:,2,:).*u.^2;
29        ut(i,:) = ut(i,:)-(buu(i+1,:)-buu(i-1,:))/(dx*2);
30    end
31  end% function
```

Fin.

Figure 3.8: a short time simulation of the Burgers' map (Section 3.6.3) on patches in space. It requires many very small time-steps only just visible in this mesh.



3.6 `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.

3.6.1 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 ↔ patchSmooth1 ↔ burgersMap
3. process results

Establish global data struct for the microscale Burgers' map (Section 3.6.3) 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.

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

Set an initial condition, and simulate a burst of the microscale space-time map over a time 0.2 using the function `burgerBurst()` (Section 3.6.4).

```
64 u0 = 0.4*(1+sin(patches.x))+0.1*randn(size(patches.x));
65 [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.

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

Save the plot to file to form Figure 3.8.

```
83 ifOurCf2eps([mfilename 'MapU'])
```

3.6.2 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 3.6.1)
2. PIRK2 $\leftrightarrow$ burgerBurst $\leftrightarrow$ patchSmooth1 $\leftrightarrow$ burgersMap
3. process results

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

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

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

```
124 ts = linspace(0,0.5,11);
125 bT = 3*(ratio*Len/nPatch/(nSubP/2-1))^2
```

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.

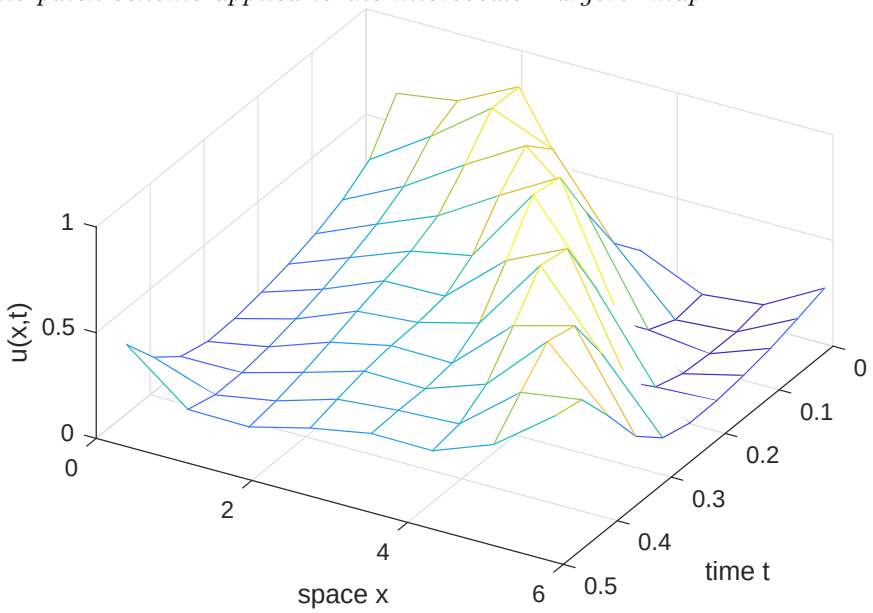
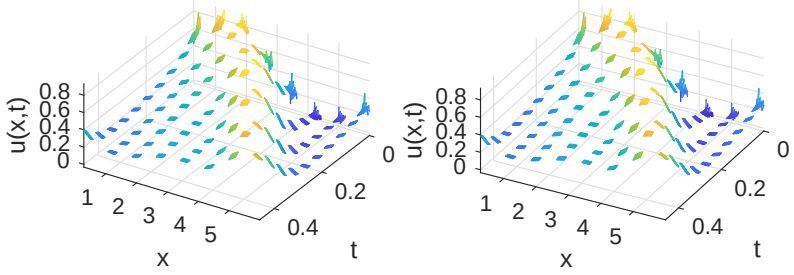


Figure 3.10: the microscale field $u(x,t)$ during each of the microscale bursts used in the projective integration. View this stereo pair cross-eyed.




```
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)')
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
view(120,50)
ifOurCf2eps([mfilename 'U'])
```

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)
```

```
157   mesh(tss,xs(:),uss')
158   ylabel('space x'),xlabel('time t'),zlabel('u(x,t)')
159   axis tight, view(126-4*k,50)
160 end
161 ifOurCf2eps([mfilename 'Micro'])
```

3.6.3 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).

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

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

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

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

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

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

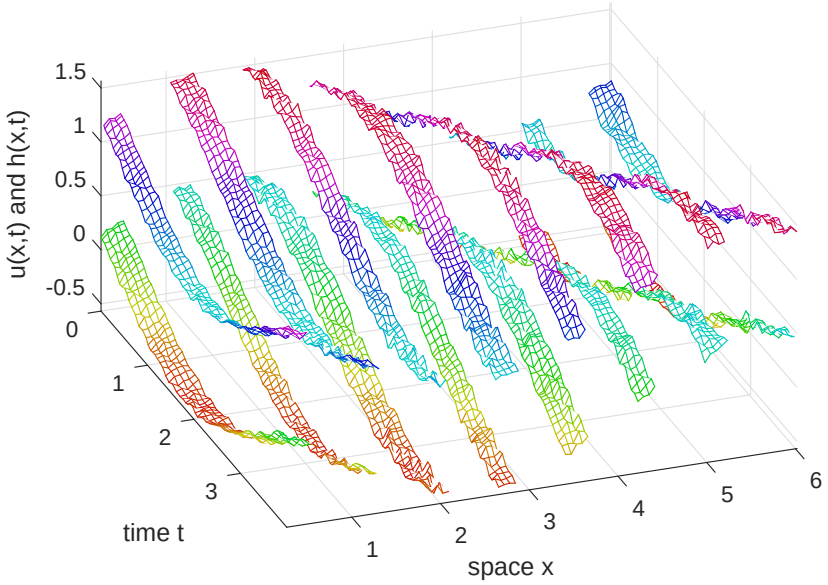
```
29   us = nan(ndt+1,numel(ui));
30   us(1,:) = reshape(ui,1,[]);
31   for j = 1:ndt
32     ui = patchSmooth1(ts(j),ui);
33     us(j+1,:) = reshape(ui,1,[]);
34   end
```

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

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

Fin.

Figure 3.11: 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 (3.3) *with $f_1 = f_2 = 0$. The microscale random component to the initial condition persists in the simulation—but the macroscale wave still propagates.*



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

Section contents

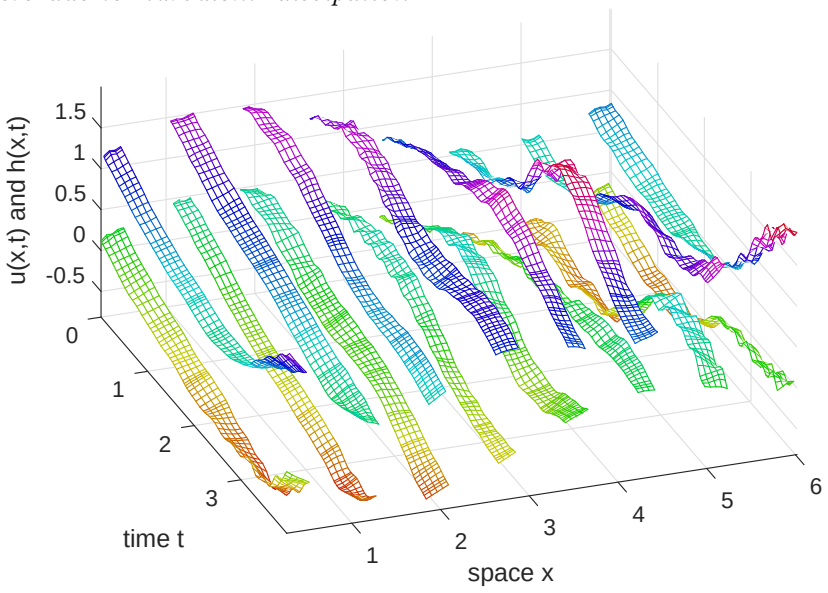
Figure 3.11 shows an example simulation in time generated by the patch scheme applied to an ideal wave PDE (Cao & Roberts 2013). The inter-patch coupling is realised by spectral interpolation of the mid-patch values to the patch edges.

This approach, based upon the differential equations coded in Section 3.7.2, may be adapted by a user to a wide variety of 1D wave and wave-like systems. For example, the differential equations of Section 3.7.3 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 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

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

Figure 3.12: water depth $h(x,t)$ (above) and velocity field $u(x,t)$ (below) of the gap-tooth scheme applied to the Smagorinski shallow water wave PDEs *(3.4). The microscale random initial component decays where the water speed is non-zero due to 'turbulent' dissipation.*



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 3.7.3 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

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

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

where $\tan\theta$ is the slope of the bed. The PDE (3.4a) represents conservation of the fluid. The momentum PDE (3.4b) 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 3.12 shows one simulation of this system—for the same initial condition as Figure 3.11.

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.

3.7.1 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 (3.3) (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.

```
119 global patches
120 nPatch = 8
121 ratio = 0.2
122 nSubP = 11 %of the form 4*n-1
123 Len = 2*pi;
124 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.

```
134 uPts = mod( (1:nSubP)'+(1:nPatch) ,2);
135 hPts = find(uPts==0);
136 uPts = find(uPts==1);
137 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).

```
148 U0 = nan(nSubP,nPatch);
149 U0(hPts) = 1+0.5*sin(patches.x(hPts));
150 U0(uPts) = 0+0.5*sin(patches.x(uPts));
151 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.

```
161 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.

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

```
174       [ts,Ucts] = odeOcts(@patchSmooth1,[0 4],U0(:));
175   end
```

Plot the simulation.

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

Optionally save the plot to file.

```
192   ifOurCf2eps([mfilename num2str(k) 'CtsUH'])
```

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

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

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

3.7.2 `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.

```
24  function Ut = idealWavePDE(t,U,patches)
25    dx = diff(patches.x(2:3));
26    U = squeeze(U);
27    Ut = nan(size(U));  ht = Ut;
```

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

```
34    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.

```
46    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.

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

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

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

3.7.3 `waterWavePDE()`: water wave PDE

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

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

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

```
25    dx = diff(patches.x(2:3));
26    U = squeeze(U);
27    Ut = nan(size(U));  ht = Ut;
28    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.

```
36    ii = i(2:end-1);
37    V = Ut;
38    V(ii,:) = (U(ii+1,:)+U(ii-1,:))/2;
39    V(1:2,:) = 2*U(2:3,:)-V(3:4,:);
40    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.

```
47    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 $\texttt{U}_i$ and $\texttt{V}_{i\pm1}$; and h-values in $\texttt{V}_i$ and $\texttt{U}_{i\pm1}$.

```
55    Ut(i,:) = -0.985*(U(i+1,:)-U(i-1,:))/(2*dx) ...
56      -0.003*U(i,:).*rabs(U(i,:)./V(i,:)) ...
57      -1.045*U(i,:).*(V(i+1,:)-V(i-1,:))/(2*dx) ...
58      +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:

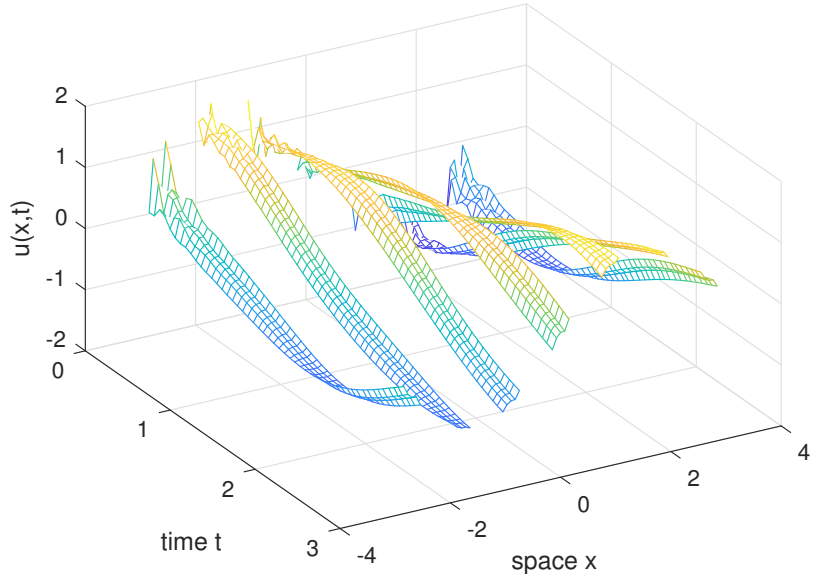
$$\begin{aligned} u_{xx} &\approx \frac{1}{4\delta^2}(u_{i-2} - 2u_i + u_{i+2}) \\ &= \frac{1}{4\delta^2}(u_{i-2} + u_i - 4u_i + u_i + u_{i+2}) \\ &= \frac{1}{2\delta^2}\left(\frac{u_{i-2} + u_i}{2} - 2u_i + \frac{u_i + u_{i+2}}{2}\right) \\ &= \frac{1}{2\delta^2}\left(\bar{u}_{i-1} - 2u_i + \bar{u}_{i+1}\right).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

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

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

Fin.

Figure 3.13: wave field $u(x,t)$ of the gap-tooth scheme applied to the weakly damped wave (3.5). 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.



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

Figure 3.13 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

$$\frac{\partial u_i}{\partial t} = v_i\,, \quad \frac{\partial v_i}{\partial t} = \frac{1}{dx^2}\delta[c_{i-1/2}\delta u_i] + \frac{0.02}{dx^2}\delta^2 v_i\,, \tag{3.5}$$

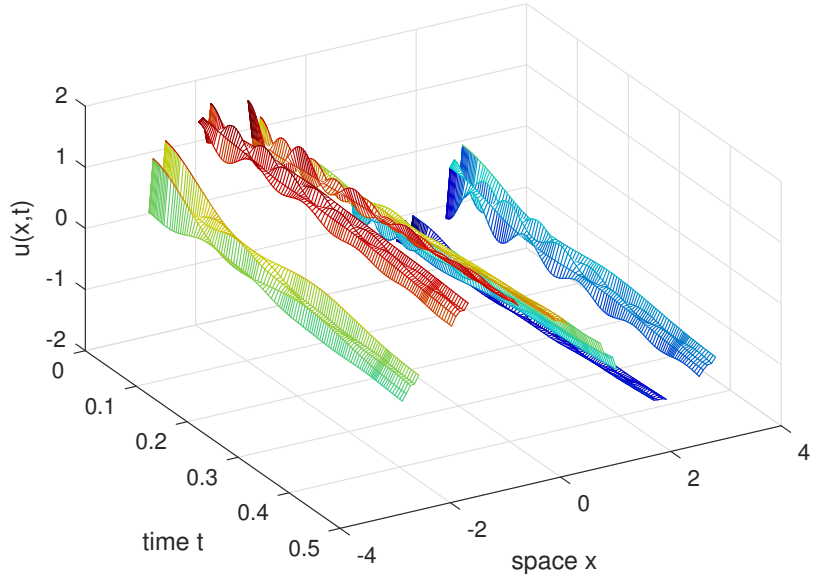
in terms of the centred difference operator δ. The system has a microscale heterogeneity via the coefficients $c_{i+1/2}$ which we assume to have some given known periodicity. Figure 3.13 shows one patch simulation of this system: observe the effects of the heterogeneity within each patch.

3.8.1 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 3.14: wave field $u(x,t)$ of the gap-tooth scheme applied to the weakly damped wave (3.5). 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.

```
91  mPeriod = 3
92  cHetr = exp(1*randn(mPeriod,1));
93  cHetr = cHetr*mean(1./cHetr) % normalise
94  nPeriodsPatch=1
```

Establish the global data struct `patches` for the microscale heterogeneous lattice wave system (3.5) 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).

```
111 global patches
112 nPatch = 7
113 ratio = 0.25
114 nSubP = nPeriodsPatch*mPeriod+2
115 configPatches1(@heteroWave,[-pi pi],nan,nPatch ...
116     ,0,ratio,nSubP,'EdgyInt',true,'hetCoeffs',cHetr);
```

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`.

```
uv0(:,1,1,:) = -sin(patches.x)+0.3*randn(nSubP,1,1,nPatch);
uv0(:,2,1,:) = +cos(patches.x)+0.3*randn(nSubP,1,1,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], uv0(:));
else % octave version
    [ts,us] = odeOcts(@patchSmooth1, [0 3], uv0(:));
end
```

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

```
xs = squeeze(patches.x);
us = patchEdgeInt1( permute( reshape(us,length(ts) ...
     ,nSubP,2,nPatch) ,[2 3 1 4]) );
xs(end+1,:) = nan; us(end+1,:,:,:) = nan;
us = reshape(permute(us,[1 4 2 3]),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)')
  ifOurCf2eps([mfilename 'U' 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=repmat(0*patches.x,1,2); u0([1 end],:)=nan; u0=u0(:);
i=find(~isnan(u0));
nJ=length(i);
Jac=nan(nJ);
```

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
```

```
194 for j=1:nJ
195    u0(i)=((1:nJ)==j);
196    dudt=patchSmooth1(0,u0);
197    Jac(:,j)=dudt(i);
198 end
199 Jac(abs(Jac)<1e-12)=0;
```

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

```
231 [evecs,evals]=eig(Jac);
232 eval=sort(diag(evals));
233 slowestEvals=eval(2:4:4*nPatch)
```

End of the main script.

3.8.2 `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,1,j)` and v_{ij} = `u(i,2,j)`) and 2D array `x` (obtained in full via edge-value interpolation of `patchSmooth1`, Section 3.2), computes the time derivatives at each point in the interior of a patch, output in `ut`:

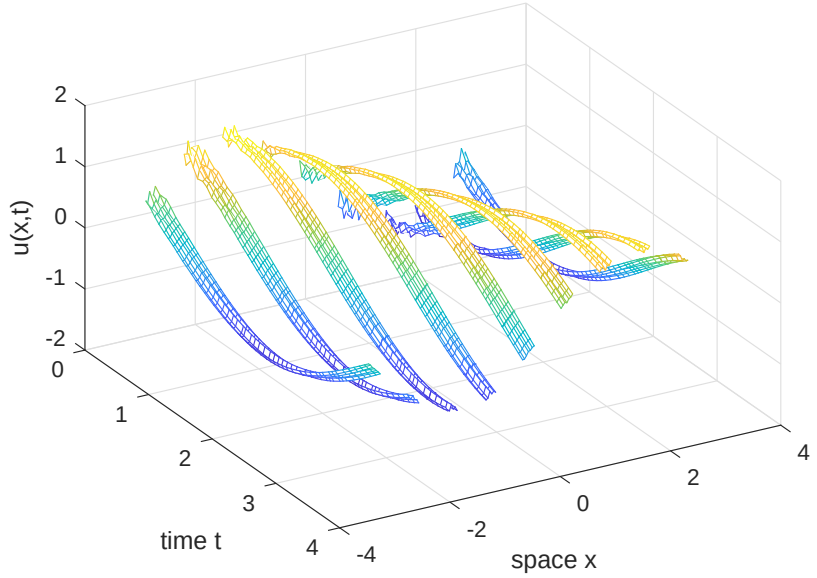
$$\frac{\partial u_{ij}}{\partial t} = v_{ij}\,, \quad \frac{\partial v_{ij}}{\partial t} = \frac{1}{dx^2}\delta[c_{i-1/2}\delta u_{ij}] + \frac{0.02}{dx^2}\delta^2 v_{ij}\,.$$

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

```
function ut = heteroWave(t,u,patches)
  u = squeeze(u);
  dx = diff(patches.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.cs.*diff(u(:,1,:)))/dx^2 ...
        +0.02*diff(u(:,2,:),2)/dx^2;
end% function
```

Fin.

Figure 3.15: wave field $u(x,t)$ of the gap-tooth scheme applied to the wave (3.6). The microscale random component to the initial condition, the sub-patch fluctuations, decays, leaving the emergent macroscale wave in the heterogeneous media. This simulation uses nine patches of 'large' size ratio 0.25 for visibility.



3.9 `waveEdgy1`: simulate a 1D, first-order, wave PDE on small patches

Section contents

Figure 3.15 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 spectral interpolation of the patch's next-to-edge values to the patch opposite edges. Such coupling preserves symmetry in many systems, and in this first-order wave PDE preserves skew-symmetry.

The first-order wave-like PDE is $u_t = -\frac{1}{2}(cu)_x - \frac{1}{2}cu_x$, which when c is constant becomes the canonical first-order wave PDE $u_t = -cu_x$. The differential operator on the right-hand side is skew-symmetric: letting $\mathcal{D} = -\frac{1}{2}(c\cdot)_x - \frac{1}{2}c\partial_x$ then $\int v\mathcal{D}u\,dx = \int -v(cu)_x - v\frac{1}{2}cu_x\,dx = -\int \frac{1}{2}v_xcu + \frac{1}{2}(vc)_xu\,dx = -\int u\mathcal{D}v\,dx$.

To discretise in space, suppose the spatial microscale lattice is at points x_i, with constant spacing d. With dependent variables $u_i(t)$, simulate the microscale lattice, in terms of the centred difference δ and mean μ, wave system

$$\frac{du_i}{dt} = -\frac{1}{2d}\left[\delta(c_i\mu u_i) + \mu(c_i\delta u_i)\right] = -\frac{1}{2d}\left[c_{i+\frac{1}{2}}u_{i+1} - c_{i-\frac{1}{2}}u_{i-1}\right]. \qquad (3.6)$$

Figure 3.15 shows one patch simulation of this space-time system, except it also includes a $\nu = 0.001$ small 'viscous' dissipation, $\nu\delta^2 u_i/d^2$, to weakly damp the microscale, sub-patch, fast waves.

3.9.1 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
2. ode15s ↔ patchSmooth1 ↔ waveFirst
3. plot the simulation
4. use patchSmooth1 to explore the Jacobian

First establish the microscale heterogeneity has (odd-valued) micro-period `mPeriod` on the lattice, and random log-normal values, normalised. This normalisation means that macroscale wave on a domain of length 2π should have nearly integer frequencies, $0, 1, 2, \ldots$—except that the normalisation is exact only for periods 3 and 5. Then the heterogeneity is repeated `nPeriodsPatch` times within each patch.

```
mPeriod = 5 % needs to be odd for a wave
cHetr = exp(0.1*randn(mPeriod,1)); % 0.3 appears max reasonable
if mPeriod==3,
    cHetr=cHetr*mean(cHetr.^2)/prod(cHetr) % normalise
elseif mPeriod==5,
    cHetr=cHetr*mean(cHetr.^2.*cHetr([3 4 5 1 2]).^2)/prod(cHetr)
else cHetr=cHetr*mean(1./cHetr) % roughly normalise
end
nPeriodsPatch=1 % also needs to be odd
```

Establish the global data struct `patches` for the microscale heterogeneous lattice wave system (3.6) solved on 2π-periodic domain, with nine 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 via the inter-patch coupling conditions. Setting `EdgyInt` to

- true means the edge-values come from interpolating the opposite next-to-edge values of the patches (not the mid-patch values); whereas
- false means the time integration appears OK, but the Jacobian is, correctly, not skew-symmetric for this case of interpolating mid-patch values.

```
global patches
nPatch = 9
ratio = 0.25
EdgyInt=true
nPeriodsPatch = (2-EdgyInt)*nPeriodsPatch;
nSubP = nPeriodsPatch*mPeriod+1+EdgyInt
configPatches1(@waveFirst,[-pi pi],nan,nPatch,4 ...
    ,ratio,nSubP,'EdgyInt',EdgyInt,'hetCoeffs',cHetr);
```

Specify the weak damping of the sub-patch, fast, microscale waves.

```
patches.nu=0.003;
```

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

```
xs=squeeze(patches.x);
u0 = -sin(xs)+0.1*randn(nSubP,nPatch);
```

Integrate using standard stiff integrators.

```
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@patchSmooth1, [0 3.5], u0(:));
else % octave version
    [ts,us] = odeOcts(@patchSmooth1, [0 0.5], u0(:));
end
```

Plot space-time surface of the simulation Let's see the edge values of the patches. For the field values (which are rows in `us`) we need to reshape, permute, interpolate with `patchEdgeInt1` to get edge values, pad with `nan`s, and reshape again.

```
xs(end+1,:) = nan;
us = patchEdgeInt1( permute( reshape(us ...
     ,length(ts),nSubP,nPatch) ,[2 1 3]) );
us(end+1,:,:,:) = nan;
us=reshape(permute(squeeze(us),[1 3 2]),[],length(ts));
```

Now plot a space-time graph. Subsample the data over the macroscale duration of the simulation to show the propagation of the macroscale wave over the heterogeneous lattice.

```
[~,j]=min(abs(ts-linspace(ts(1),ts(end),50)));
figure(1),clf
mesh(ts(j),xs(:),us(:,j)),  view(60,40)
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
ifOurCf2eps([mfilename 'U' num2str(2)])
```

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. Set the weak damping to zero so we explore the ideal case of the wave system (3.6).

```
ratio=0.01
nPatch=19
leadingFreqs=[];
for ord=0:2:8
    ordInterp=ord
    configPatches1(@waveFirst,[-pi pi],nan,nPatch,ord ...
```

Table 3.3: example parameters and list of eigenvalues (every second one listed is sufficient due to symmetry): `nPatch` = 19, `ratio` = 0.03, `nSubP` = 7. The columns are for various `ordCC`, in order: 0, spectral interpolation; 2, quadratic; 4, quartic; and 6, sixth order. Rows are ordered in the effective wavenumber of the corresponding eigenvector (the number of zero crossings).

```
cHetr =
    0.6614
    1.5758
    1.8645
    1.4600
    0.8486
leadingFreqs =
         0         0         0         0
    1.0000    0.9819    0.9996    1.0000
    2.0000    1.8574    1.9879    1.9989
    2.9999    2.5318    2.9138    2.9830
    3.9997    2.9320    3.6688    3.8910
    4.9995    3.0146    4.1015    4.5720
    5.9991    2.7705    4.0640    4.7890
    6.9985    2.2261    3.4699    4.3042
    7.9978    1.4402    2.3421    3.0200
    8.9969    0.4981    0.8278    1.0897
  698.0262  698.0852  698.0370  698.0283
  698.1548  698.1728  698.1563  698.1549
```

```
204         ,ratio,nSubP,'EdgyInt',EdgyInt,'hetCoeffs',cHetr);
205     patches.nu=0;
```

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.

```
215     u0=0*patches.x; u0([1 end],:)=nan; u0=u0(:);
216     i=find(~isnan(u0));
217     nJ=length(i);
218     Jac=nan(nJ);
219     for j=1:nJ
220        u0(i)=((1:nJ)==j);
221        dudt=patchSmooth1(0,u0);
222        Jac(:,j)=dudt(i);
223     end
224     nonSkewSymmetric=norm(Jac+Jac')
225     assert(nonSkewSymmetric<1e-10,'failed skew-symmetry')
226     Jac(abs(Jac)<1e-12)=0;
```

Find the eigenvalues of the Jacobian, and list for inspection in Table 3.3: the spectral interpolation is effectively exact for the macroscale; quadratic interpolation is usually qualitatively good; quartic interpolation appears to

be the lowest order for quantitative accuracy.

```
268     [evecs,evals]=eig(Jac);
269     maxRealPartEvals=max(abs(real(diag(evals))))
270     assert(maxRealPartEvals<1e-10,'failed real-part zero')
271     freqs=imag(diag(evals));
```

Use a count of zero crossings in the corresponding eigenvector in order to try to sort on the spatial wavenumber.

```
279     [~,j]=sort(sum(abs(diff(sign(real(evecs))))));
280     leadingFreqs=[leadingFreqs -freqs(j(1:2:nPatch+4))];
```

End of the for-loop over orders of interpolation, and display the spectra.

```
287 end
288 disp('     spectral    quadratic      quartic  sixth-order ...')
289 leadingFreqs = leadingFreqs
```

End of the main script.

3.9.2 `waveFirst()`: first-order wave PDE

This function codes a lattice, first-order, heterogeneous, wave PDE inside patches. Optionally adds some viscous dissipation. For 2D input arrays `u` and `x` (via edge-value interpolation of `patchSmooth1`, Section 3.2), computes the time derivative (3.6) at each point in the interior of a patch, output in `ut`.

```
17 function ut = waveFirst(t,u,patches)
18   u=squeeze(u);
19   dx = diff(patches.x(2:3)); % space step
20   i = 2:size(u,1)-1; % interior points in a patch
21   ut = nan+u;        % preallocate output array
22   ut(i,:) = -(patches.cs(i).*u(i+1,:) ...
23               -patches.cs(i-1).*u(i-1,:))/(2*dx) ...
24              +patches.nu*diff(u,2)/dx^2;
25 end% function
```

Fin.

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

Section contents

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSmooth2()`. Section 3.10.1 lists an example of its use.

```
19 function patches = configPatches2(fun,Xlim,BCs ...
20     ,nPatch,ordCC,ratio,nSubP,varargin)
```

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 3.10.1 for the example code.

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)`, 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 $[\texttt{Xlim(1)}, \texttt{Xlim(2)}] \times [\texttt{Xlim(3)}, \texttt{Xlim(4)}]$: if `Xlim` is of length two, then the domain is the square domain of the same interval in both directions.
- `BCs` eventually and somehow will define the macroscale boundary conditions. Currently, `BCs` is ignored and the system is assumed macro-periodic in the specified rectangular domain.
- `nPatch` sets 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 (≥ 1) in each direction.
- `ordCC` is the 'order' of interpolation for inter-patch coupling across empty space of the macroscale patch values to the edge-values of the patches: currently must be $0, 2, 4, \ldots$; where 0 gives spectral interpolation.
- `ratio` (real) is the ratio of (depending upon `EdgyInt`) either the half-width or full-width of a patch to the spacing of the patch mid-points. So either `ratio` $= \frac{1}{2}$ means the patches abut and `ratio` $= 1$ is overlapping patches as in holistic discretisation, or `ratio` $= 1$ means the patches abut. 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 of the two directions.
- `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. If not using `EdgyInt`, then must be odd so that there is/are centre-patch micro-grid point/lines in each patch.

- `nEdge` (not yet implemented), *optional*, default=1, for each patch, 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 neighbour interactions).
- `EdgyInt`, true/false, *optional*, default=false. If true, then interpolate to left/right/top/bottom edge-values from right/left/bottom/top next-to-edge values. If false or omitted, then interpolate from centre-patch lines.
- `nEnsem`, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.
- `hetCoeffs`, *optional*, default empty. Supply a 2/3D array of microscale heterogeneous coefficients to be used by the given microscale `fun` in each patch. Say the given array `cs` is of size $m_x \times m_y \times n_c$, where n_c is the number of different sets of coefficients. For example, in heterogeneous diffusion, $n_c = 2$ for the diffusivities in the *two* different spatial directions. The coefficients are to be the same for each and every patch; however, macroscale variations are catered for by the n_c coefficients being n_c parameters in some macroscale formula.
  - If `nEnsem` $= 1$, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the $(1, 1)$-point in each patch.
  - If `nEnsem` > 1 (value immaterial), then reset `nEnsem` $:= m_x \cdot m_y$ and construct an ensemble of all $m_x \cdot m_y$ phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When `EdgyInt` is true, and when the coefficients are diffusivities/elasticities in x and y directions, respectively, then this coupling cunningly preserves symmetry.
- `'parallel'`, true/false, *optional*, default=false. If false, then all patch computations are on the user's main CPU—although a user may well separately invoke, say, a GPU to accelerate sub-patch computations.

  If true, and it requires that you have MATLAB's Parallel Computing Toolbox, then it will distribute the patches over multiple CPUs/cores. In MATLAB, only one array dimension can be split in the distribution, so it chooses the one space dimension x, y corresponding to the highest `\nPatch` (if a tie, then chooses the rightmost of x, y). A user may correspondingly distribute arrays with property `patches.codist`, or simply use formulas invoking the preset distributed arrays `patches.x`, and `patches.y`. If a user has not yet established a parallel pool, then a 'local' pool is started.

Output The struct `patches` is created and set with the following components. If no output variable is provided for `patches`, then make the struct available as a global variable.[3]

```
154 if nargout==0, global patches, end
```

- `.fun` is the name of the user's function `fun(t,u,patches)` or `fun(t,u)`, that computes the time derivatives (or steps) on the patchy lattice.
- `.ordCC` is the specified order of inter-patch coupling.
- `.stag` 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`$\times$2-array of weights for the inter-patch interpolation onto the right/top and left/bottom edges (respectively) with patch:macroscale ratio as specified.
- `.x` (6D) is `nSubP(1)` $\times 1 \times 1 \times 1 \times$ `nPatch(1)` $\times 1$ array of the regular spatial locations x_{iI} of the microscale grid points in every patch.
- `.y` (6D) is $1 \times$ `nSubP(2)` $\times 1 \times 1 \times 1 \times$ `nPatch(2)` array of the regular spatial locations y_{jJ} of the microscale grid points in every patch.
- `.ratio` 1×2, are the size ratios of every patch.
- `.nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.
- `.le`, `.ri`, `.bo`, `.to` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.
- `.cs` either
  - `[]` 0D, or
  - if `nEnsem` $= 1$, (`nSubP(1)` $- 1$) $\times$ (`nSubP(2)` $- 1$) $\times n_c$ 3D array of microscale heterogeneous coefficients, or
  - if `nEnsem` > 1, (`nSubP(1)` $- 1$) $\times$ (`nSubP(2)` $- 1$) $\times n_c \times m_x m_y$ 4D array of $m_x m_y$ ensemble of phase-shifts of the microscale heterogeneous coefficients.
- `.parallel`, logical: true if patches are distributed over multiple CPUs/cores for the Parallel Computing Toolbox, otherwise false (the default is to activate the *local* pool).
- `.codist`, *optional*, describes the particular parallel distribution of arrays over the active parallel pool.

3.10.1 If no arguments, then execute an example

```
230 if nargin==0
```

[3] When using `spmd` parallel computing, it is generally best to avoid global variables, and so instead prefer using an explicit output variable.

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. ode23 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 0.4 (relatively large for visualisation), and with 5×5 points forming each patch. Roberts et al. (2014) established that this scheme is consistent with the PDE (as the patch spacing decreases).

```
global patches
patches = configPatches2(@nonDiffPDE,[-3 3 -2 2], nan ...
    , [9 7], 0, 0.4, 5 ,'EdgyInt',false);
```

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

```
u0 = exp(-patches.x.^2-patches.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: optionally set x and y-edges to `nan` to leave the gaps between patches.

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

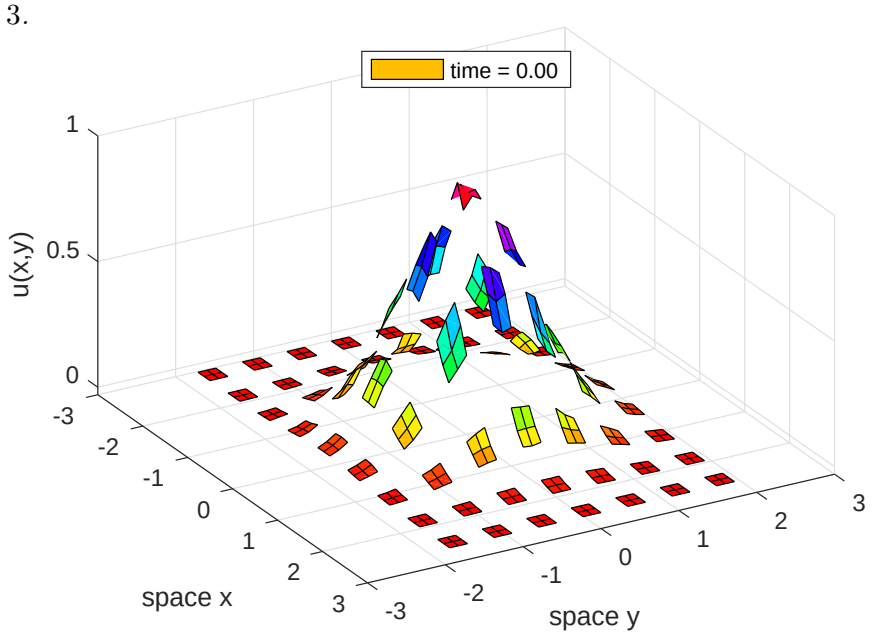
Start by showing the initial conditions of Figure 3.16 while the simulation computes.

```
u = reshape(permute(squeeze(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 = 0.00','Location','north')
xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
colormap(hsv)
ifOurCf2eps([mfilename 'ic'])
```

Integrate in time to $t = 2$ using standard functions. In MATLAB `ode15s` would be natural as the patch scheme is naturally stiff, but `ode23` is quicker (Maclean et al. 2020, Fig. 4). Ask for output at non-uniform times because the diffusion slows.

```
disp('Wait to simulate nonlinear diffusion h_t=(h^3)_xx+(h^3)_yy')
drawnow
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@patchSmooth2,linspace(0,2).^2,u0(:));
else % octave version is quite slow for me
```

Figure 3.16: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to a nonlinear 'diffusion' PDE: Figure 3.17 plots the computed field at time $t = 3$.




```
309     lsode_options('absolute tolerance',1e-4);
310     lsode_options('relative tolerance',1e-4);
311     [ts,us] = odeOcts(@patchSmooth2,[0 1],u0(:));
312 end
```

Animate the computed simulation to end with Figure 3.17. Use `patchEdgeInt2` to interpolate patch-edge values (but not corner values, and even if not drawn).

```
321 for i = 1:length(ts)
322   u = patchEdgeInt2(us(i,:));
323   u = reshape(permute(squeeze(u) ...
324       ,[1 3 2 4]), [numel(x) numel(y)]);
325   set(hsurf,'ZData', u');
326   legend(['time = ' num2str(ts(i),'%4.2f')])
327   pause(0.1)
328 end
329 ifOurCf2eps([mfilename 't3'])
```

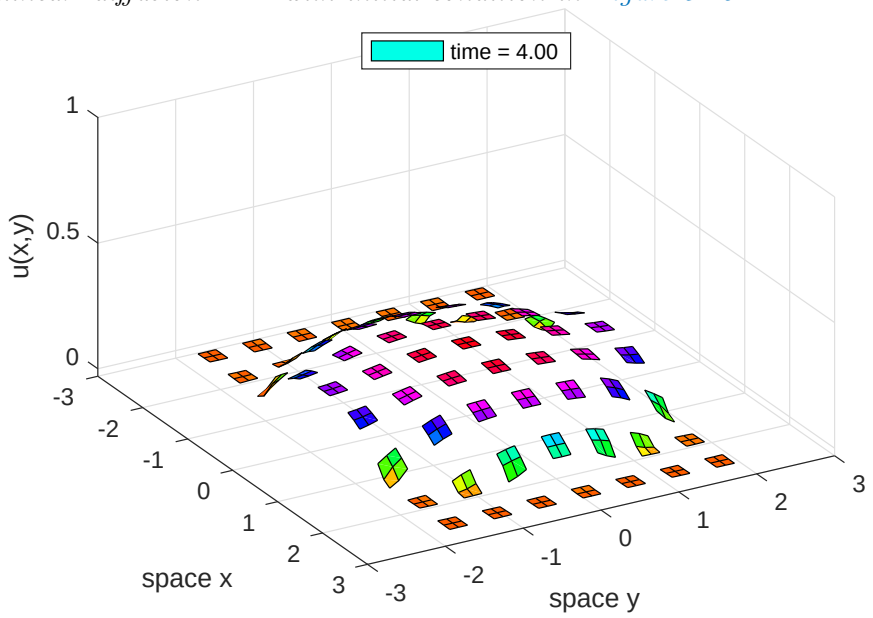
Upon finishing execution of the example, exit this function.

```
344 return
345 end%if no arguments
```

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

```
13 function ut = nonDiffPDE(t,u,patches)
14   if nargin<3, global patches, end
```

Figure 3.17: field $u(x,y,t)$ at time $t = 3$ of the patch scheme applied to a nonlinear 'diffusion' PDE with initial condition in Figure 3.16.




```
15    u = squeeze(u); % reduce to 4D
16    dx = diff(patches.x(1:2));  % microgrid spacing
17    dy = diff(patches.y(1:2));
18    i = 2:size(u,1)-1;  j = 2:size(u,2)-1;  % interior patch points
19    ut = nan+u;  % preallocate output array
20    ut(i,j,:,:) = diff(u(:,j,:,:).^3,2,1)/dx^2 ...
21                  +diff(u(i,:,:,:).^3,2,2)/dy^2;
22  end
```

3.10.2 Parse input arguments and defaults

```
359  p = inputParser;
360  fnValidation = @(f) isa(f, 'function_handle');%test for fn name
361  addRequired(p,'fun',fnValidation);
362  addRequired(p,'Xlim',@isnumeric);
363  addRequired(p,'BCs'); % nothing yet decided
364  addRequired(p,'nPatch',@isnumeric);
365  addRequired(p,'ordCC',@isnumeric);
366  addRequired(p,'ratio',@isnumeric);
367  addRequired(p,'nSubP',@isnumeric);
368  addParameter(p,'nEdge',1,@isnumeric);
369  addParameter(p,'EdgyInt',false,@islogical);
370  addParameter(p,'nEnsem',1,@isnumeric);
371  addParameter(p,'hetCoeffs',[],@isnumeric);
372  addParameter(p,'parallel',false,@islogical);
373  %addParameter(p,'nCore',1,@isnumeric); % not yet implemented
374  parse(p,fun,Xlim,BCs,nPatch,ordCC,ratio,nSubP,varargin{:});
```

Set the optional parameters.

```
380 patches.nEdge = p.Results.nEdge;
381 patches.EdgyInt = p.Results.EdgyInt;
382 patches.nEnsem = p.Results.nEnsem;
383 cs = p.Results.hetCoeffs;
384 patches.parallel = p.Results.parallel;
385 %patches.nCore = p.Results.nCore;
```

Initially duplicate parameters for both space dimensions as needed.

```
393 if numel(Xlim)==2, Xlim = repmat(Xlim,1,2); end
394 if numel(nPatch)==1, nPatch = repmat(nPatch,1,2); end
395 if numel(ratio)==1, ratio = repmat(ratio,1,2); end
396 if numel(nSubP)==1, nSubP = repmat(nSubP,1,2); end
```

Check parameters.

```
403 assert(patches.nEdge==1 ...
404       ,'multi-edge-value interp not yet implemented')
405 assert(all(2*patches.nEdge<nSubP) ...
406       ,'too many edge values requested')
407 %if patches.nCore>1
408 %    warning('nCore>1 not yet tested in this version')
409 %    end
```

3.10.3 The code to make patches

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

```
423 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 (not yet??) -1.

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

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

```
440 patches.stag = mod(ordCC,2);
441 assert(patches.stag==0,'staggered not yet implemented??')
442 ordCC = ordCC+patches.stag;
443 patches.ordCC = ordCC;
```

Check for staggered grid and periodic case.

```
449 if patches.stag, assert(all(mod(nPatch,2)==0), ...
450     'Require an even number of patches for staggered grid')
451 end
```

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

Store the size ratio in `patches`.

```
ratio = reshape(ratio,1,2); % force to be row vector
patches.ratio=ratio;
if ordCC>0
    [Cwtsr,Cwtsl] = patchCwts(ratio,ordCC,patches.stag);
    patches.Cwtsr = Cwtsr;  patches.Cwtsl = Cwtsl;
end
```

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

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

Construct the microscale in each patch, assuming Dirichlet patch edges, and half-patch widths of `ratio(1)`·`DX` and `ratio(2)`·`DY`, unless `patches.EdgyInt` is true in which case every patch is of widths `ratio(1)*DX+dx` and `ratio(2)*DY+dy`.

```
nSubP = reshape(nSubP,1,2); % force to be row vector
assert(patches.EdgyInt | all(mod(nSubP,2)==1), ...
    'configPatches2: nSubP must be odd')
i0 = (nSubP(1)+1)/2;
if ~patches.EdgyInt, dx = ratio(1)*DX/(i0-1);
else                 dx = ratio(1)*DX/(nSubP(1)-2);
end
patches.x = dx*(-i0+1:i0-1)'+X;  % micro-grid
patches.x = reshape(patches.x,nSubP(1),1,1,1,nPatch(1),1);
i0 = (nSubP(2)+1)/2;
if ~patches.EdgyInt, dy = ratio(2)*DY/(i0-1);
else                 dy = ratio(2)*DY/(nSubP(2)-2);
end
patches.y = dy*(-i0+1:i0-1)'+Y;  % micro-grid
patches.y = reshape(patches.y,1,nSubP(2),1,1,1,nPatch(2));
```

3.10.4 Set ensemble inter-patch communication

For `EdgyInt` or centre interpolation respectively,

- the right-edge/centre realisations `1:nEnsem` are to interpolate to left-edge `le`, and
- the left-edge/centre realisations `1:nEnsem` are to interpolate to `re`.

`re` and `li` are 'transposes' of each other as `re(li)=le(ri)` are both `1:nEnsem`. Similarly for bottom-edge/centre interpolation to top-edge via `to`, and top-edge/centre interpolation to bottom-edge via `bo`.

The default is nothing shifty. This setting reduces the number of if-statements in function `patchEdgeInt2()`.

```
nE = patches.nEnsem;
patches.le = 1:nE;  patches.ri = 1:nE;
patches.bo = 1:nE;  patches.to = 1:nE;
```

However, if heterogeneous coefficients are supplied via `hetCoeffs`, then do some non-trivial replications. First, get microscale periods, patch size, and replicate many times in order to subsequently sub-sample: `nSubP` times should be enough. If `cs` is more then 3D, then the higher-dimensions are reshaped into the 3rd dimension.

```
if ~isempty(cs)
  [mx,my,nc] = size(cs);
  nx = nSubP(1); ny = nSubP(2);
  cs = repmat(cs,nSubP);
```

If only one member of the ensemble is required, then sub-sample to patch size, and store coefficients in `patches` as is.

```
  if nE==1, patches.cs = cs(1:nx-1,1:ny-1,:); else
```

But for `nEnsem` > 1 an ensemble of $m_x m_y$ phase-shifts of the coefficients is constructed from the over-supply. Here code phase-shifts over the periods—the phase shifts are like Hankel-matrices.

```
    patches.nEnsem = mx*my;
    patches.cs = nan(nx-1,ny-1,nc,mx,my);
    for j = 1:my
        js = (j:j+ny-2);
        for i = 1:mx
            is = (i:i+nx-2);
            patches.cs(:,:,:,i,j) = cs(is,js,:);
        end
    end
    patches.cs = reshape(patches.cs,nx-1,ny-1,nc,[]);
```

Further, set a cunning left/right/bottom/top realisation of inter-patch coupling. The aim is to preserve symmetry in the system when also invoking `EdgyInt`. What this coupling does without `EdgyInt` is unknown. Use auto-replication.

```
    le = mod((0:mx-1)+mod(nx-2,mx),mx)+1;
    patches.le = reshape(  le'+mx*(0:my-1)  ,[],1);
    ri = mod((0:mx-1)-mod(nx-2,mx),mx)+1;
    patches.ri = reshape(  ri'+mx*(0:my-1)  ,[],1);
    bo = mod((0:my-1)+mod(ny-2,my),my)+1;
    patches.bo = reshape( (1:mx)'+mx*(bo-1) ,[],1);
    to = mod((0:my-1)-mod(ny-2,my),my)+1;
    patches.to = reshape( (1:mx)'+mx*(to-1) ,[],1);
```

Issue warning if the ensemble is likely to be affected by lack of scale separation. Need to justify this and the arbitrary threshold more carefully??

```
598 if prod(ratio)*patches.nEnsem>0.9, warning( ...
599 'Probably poor scale separation in ensemble of coupled phase-shifts')
600 scaleSeparationParameter = ratio*patches.nEnsem
601 end
```

End the two if-statements.

```
607   end%if-else nEnsem>1
608 end%if not-empty(cs)
```

If parallel code then first assume this is not within an `spmd`-environment, and so we invoke `spmd...end` (which starts a parallel pool if not already started). At this point, the global `patches` is copied for each worker processor and so it becomes *composite* when we distribute any one of the fields. Hereafter, *all fields in the global variable* `patches` *must only be referenced within an* `spmd`*-environment.*[4]

```
627 if patches.parallel
628 %  theparpool=gcp()
629   spmd
```

Second, decide which dimension is to be sliced among parallel workers (for the moment, do not consider slicing the ensemble). Choose the direction of most patches, biased towards the last.

```
638   [~,pari]=max(nPatch+0.01*(1:2));
639   patches.codist=codistributor1d(4+pari);
```

`patches.codist.Dimension` is the index that is split among workers. Then distribute the appropriate coordinate direction among the workers: the function must be invoked inside an `spmd`-group in order for this to work—so we do not need `parallel` in argument list.

```
649   switch pari
650     case 1, patches.x=codistributed(patches.x,patches.codist);
651     case 2, patches.y=codistributed(patches.y,patches.codist);
652   otherwise
653     error('should never have bad index for parallel distribution')
654   end%switch
655   end%spmd
```

If not parallel, then clean out `patches.codist` if it exists. May not need, but safer.

```
663 else% not parallel
664   if isfield(patches,'codist'), rmfield(patches,'codist'); end
665 end%if-parallel
```

Fin

```
674 end% function
```

[4] If subsequently outside spmd, then one must use functions like `getfield(patches{1},'a')`.

3.11 patchSmooth2(): interface 2D space to time integrators

To simulate in time with 2D 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 enough* so that the patch centre-values are sensible macroscale variables, and patch edge-values are determined by macroscale interpolation of the patch-centre or edge values. Nonetheless, microscale heterogeneous systems may be accurately simulated with this function via appropriate interpolation. Communicate patch-design variables (Section 3.10) either via the global struct `patches` or via an optional third argument (except that this last is required for parallel computing of `spmd`).

```
28 function dudt = patchSmooth2(t,u,patches)
29 if nargin<3, global patches, end
```

Input

- `u` is a vector/array of length `prod(nSubP)·nVars·nEnsem·prod(nPatch)` where there are `nVars` · `nEnsem` field values at each of the points in the `nSubP(1)` × `nSubP(2)` × `nPatch(1)` × `nPatch(2)` grid.
- `t` is the current time to be passed to the user's time derivative function.
- `patches` a struct set by `configPatches2()` with the following information used here.
  - `.fun` is the name of the user's function `fun(t,u,patches)` that computes the time derivatives on the patchy lattice. The array `u` has size `nSubP(1)` × `nSubP(2)` × `nVars` × `nEsem` × `nPatch(1)` × `nPatch(2)`. Time derivatives must be computed into the same sized array, although herein the patch edge-values are overwritten by zeros.
  - `.x` is `nSubP(1)` × 1 × 1 × 1`nPatch(1)` × 1 array of the spatial locations x_i of the microscale (i, j)-grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and micro-scales.
  - `.y` is similarly 1 × `nSubP(2)` × 1 × 1 × 1 × `nPatch(2)` array of the spatial locations y_j of the microscale (i, j)-grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and micro-scales.

Output

- `dudt` is a vector/array of of time derivatives, but with patch edge-values set to zero. It is of total length `prod(nSubP)` · `nVars` · `nEnsem` · `prod(nPatch)` and the same dimensions as `u`.

Reshape the fields `u` as a 6D-array, and sets the edge values from macroscale interpolation of centre-patch values. Section 3.12 describes `patchEdgeInt2()`.

```
94  sizeu = size(u);
95  u = patchEdgeInt2(u,patches);
```

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 the user/ integrator as same sized array as input.

```
105 dudt = patches.fun(t,u,patches);
106 dudt([1 end],:,:,:,:,:) = 0;
107 dudt(:,[1 end],:,:,:,:) = 0;
108 dudt = reshape(dudt,sizeu);
```

Fin.

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

Couples 2D patches across 2D space by computing their edge values via macroscale interpolation. Research (Roberts et al. 2014, Bunder et al. 2019) indicates the patch centre-values are sensible macroscale variables, and macroscale interpolation of these determine patch-edge values. However, for computational homogenisation in multi-D, interpolating patch next-to-edge values appears better (Bunder et al. 2020). This function is primarily used by `patchSmooth2()` but is also useful for user graphics. [5]

Communicate patch-design variables via a second argument (optional, except required for parallel computing of `spmd`) or otherwise via the global struct `patches`.

```
29 function u = patchEdgeInt2(u,patches)
30 if nargin<2, global patches, end
```

Input

- `u` is a vector/array of length `prod(nSubP)·nVars·nEnsem·prod(nPatch)` where there are `nVars · nEnsem` field values at each of the points in the `nSubP1·nSubP2·nPatch1·nPatch2` grid on the `nPatch1·nPatch2` array of patches.
- `patches` a struct set by `configPatches2()` which includes the following information.
  - `.x` is `nSubP1`×1×1×1×`nPatch1`×1 array of the spatial locations x_{iI} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.y` is similarly $1 \times$ `nSubP2` $\times 1 \times 1 \times 1 \times$ `nPatch2` array of the spatial locations y_{jJ} 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 $\{0, 2, 4, \ldots\}$
  - `.stag` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation.
  - `.Cwtsr` and `.Cwtsl` define the coupling coefficients for finite width interpolation in both the x, y-directions.
  - `.EdgyInt` true/false is true for interpolating patch-edge values from opposite next-to-edge values (often preserves symmetry).
  - `.nEnsem` the number of realisations in the ensemble.
  - `.parallel` whether serial or parallel.

[5] Script `patchEdgeInt2test.m` verifies this code.

Output

- `u` is 6D array, `nSubP1` · `nSubP2` · `nVars` · `nEnsem` · `nPatch1` · `nPatch2`, of the fields with edge values set by interpolation (and corner vales set to `NaN`).

Test for reality of the field values, and define a function accordingly. Could be problematic if some variables are real and some are complex, or if variables are of quite different sizes.

```
109   if max(abs(imag(u(:))))<1e-9*max(abs(u(:)))
110        uclean=@(u) real(u);
111   else uclean=@(u) u;
112   end
```

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

```
120 [~,ny,~,~,~,Ny] = size(patches.y);
121 [nx,~,~,~,Nx,~] = size(patches.x);
122 nEnsem = patches.nEnsem;
123 nVars = round(numel(u)/numel(patches.x)/numel(patches.y)/nEnsem);
124 assert(numel(u) == nx*ny*Nx*Ny*nVars*nEnsem ...
125   ,'patchEdgeInt2: input u has wrong size for parameters')
126 u = reshape(u,[nx ny nVars nEnsem Nx Ny ]);
```

Get the size ratios of the patches in each direction.

```
132 rx = patches.ratio(1);
133 ry = patches.ratio(2);
```

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 four immediate neighbours.

```
145 I=1:Nx; Ip=mod(I,Nx)+1; Im=mod(I-2,Nx)+1;
146 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 for centre-patch interpolation) is at indices

```
153 i0 = round((nx+1)/2);
154 j0 = round((ny+1)/2);
```

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

```
166 ordCC=patches.ordCC;
167 if ordCC>0 % then finite-width polynomial interpolation
```

The patch-edge values are either interpolated from the next-to-edge values, or from the centre-cross values (not the patch-centre value itself as that seems to have worse properties in general). Have not yet implemented core averages??

```
177   if patches.EdgyInt % next-to-edge values
178     Ux = u([2 nx-1],2:(ny-1),:,:,I,J);
179     Uy = u(2:(nx-1),[2 ny-1],:,:,I,J);
180   else
181     Ux = u(i0,2:(ny-1),:,:,I,J);
182     Uy = u(2:(nx-1),j0,:,:,I,J);
183   end;
```

Just in case any last array dimension(s) are one, we have to force a padding of the sizes, then adjoin the extra dimension for the subsequent array of differences.

```
191 szUx0=size(Ux); szUx0=[szUx0 ones(1,6-length(szUx0)) ordCC];
192 szUy0=size(Uy); szUy0=[szUy0 ones(1,6-length(szUy0)) ordCC];
```

Use finite difference formulas for the interpolation, so store finite differences $(\mu\delta, \delta^2, \mu\delta^3, \delta^4, \ldots)$ in these arrays. When parallel, in order to preserve the distributed array structure we use an index at the end for the differences.

```
202   if patches.parallel
203     dmux = zeros(szUx0,patches.codist); % 7D
204     dmuy = zeros(szUy0,patches.codist); % 7D
205   else
206     dmux = zeros(szUx0); % 7D
207     dmuy = zeros(szUy0); % 7D
208   end
```

First compute differences $\mu\delta$ and δ^2 in both space directions.

```
215   if patches.stag % use only odd numbered neighbours
216     error('polynomial interpolation not yet for staggered patch coupling')
217   else %disp('starting standard interpolation')
218     dmux(:,:,:,:,I,:,1) = (Ux(:,:,:,:,Ip,:) ...
219                            -Ux(:,:,:,:,Im,:))/2; %\mu\delta
220     dmux(:,:,:,:,I,:,2) =  Ux(:,:,:,:,Ip,:) ...
221        -2*Ux(:,:,:,:,I,:) +Ux(:,:,:,:,Im,:);    % \delta^2
222     dmuy(:,:,:,:,:,J,1) = (Uy(:,:,:,:,:,Jp) ...
223                            -Uy(:,:,:,:,:,Jm))/2; %\mu\delta
224     dmuy(:,:,:,:,:,J,2) =  Uy(:,:,:,:,:,Jp) ...
225        -2*Uy(:,:,:,:,:,J) +Uy(:,:,:,:,:,Jm);    % \delta^2
226   end% if odd/even
```

Recursively take δ^2 of these to form higher order centred differences in both space directions.

```
233    for k = 3:ordCC
234     dmux(:,:,:,:,I,:,k)   =     dmux(:,:,:,:,Ip,:,k-2) ...
235       -2*dmux(:,:,:,:,I,:,k-2) +dmux(:,:,:,:,Im,:,k-2);
236     dmuy(:,:,:,:,:,J,k)   =     dmuy(:,:,:,:,:,Jp,k-2) ...
237       -2*dmuy(:,:,:,:,:,J,k-2) +dmuy(:,:,:,:,:,Jm,k-2);
238   end
```

Interpolate macro-values to be Dirichlet edge values for each patch (Roberts &

Kevrekidis 2007, Bunder et al. 2017), using weights computed in `configPatches2()`. Here interpolate to specified order.

For the case where next-to-edge values interpolate to the opposite edge-values: when we have an ensemble of configurations, different configurations might be coupled to each other, as specified by `patches.le`, `patches.ri`, `patches.to` and `patches.bo`.

```
k=1+patches.EdgyInt; % use centre or two edges
u(nx,2:(ny-1),:,patches.ri,I,:) ...
  = Ux(1,:,:,:,:,:)*(1-patches.stag) ...
    +sum( shiftdim(patches.Cwtsr(:,1),-6).*dmux(1,:,:,:,:,:,:) ,7);
u(1 ,2:(ny-1),:,patches.le,I,:) ...
  = Ux(k,:,:,:,:,:)*(1-patches.stag) ...
    +sum( shiftdim(patches.Cwtsl(:,1),-6).*dmux(k,:,:,:,:,:,:) ,7);
u(2:(nx-1),ny,:,patches.to,:,J) ...
  = Uy(:,1,:,:,:,:)*(1-patches.stag) ...
    +sum( shiftdim(patches.Cwtsr(:,2),-6).*dmuy(:,1,:,:,:,:,:) ,7);
u(2:(nx-1),1 ,:,patches.bo,:,J) ...
  = Uy(:,k,:,:,:,:)*(1-patches.stag) ...
    +sum( shiftdim(patches.Cwtsl(:,2),-6).*dmuy(:,k,:,:,:,:,:) ,7);
u([1 nx],[1 ny],:,:,:,:)=nan; % remove corner values
```

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

```
else% spectral interpolation
```

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

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.stag % 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,:));
   stagShift=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
    stagShift = 0;
```

```
310      iV = 1:nVars;
311   end
```

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

```
321   kMax = floor((Nx-1)/2);
322   krx = shiftdim( rx*2*pi/Nx*(mod((0:Nx-1)+kMax,Nx)-kMax) ,-3);
323   kMay = floor((Ny-1)/2);
324   kry = shiftdim( ry*2*pi/Ny*(mod((0:Ny-1)+kMay,Ny)-kMay) ,-4);
```

Compute the Fourier transform of the centre-cross values. Unless doing patch-edgy interpolation when FT the next-to-edge values. If there are an even number of points, then if complex, treat as positive wavenumber, but if real, treat as cosine. When using an ensemble of configurations, different configurations might be coupled to each other, as specified by `patches.le`, `patches.ri`, `patches.to` and `patches.bo`.

```
338 ix=(2:nx-1)';  iy=2:ny-1; % indices of interior
339 if ~patches.EdgyInt
340      % here try central cross interpolation
341      Cle = fft(fft(u(i0,iy,:,:,:,:),[],5),[],6);
342      Cbo = fft(fft(u(ix,j0,:,:,:,:),[],5),[],6);
343      Cri=Cle; Cto=Cbo;
344 else % edgyInt uses next-to-edge values
345      Cle = fft(fft(u(   2,iy ,:,patches.le,:,:),[],5),[],6);
346      Cri = fft(fft(u(nx-1,iy ,:,patches.ri,:,:),[],5),[],6);
347      Cbo = fft(fft(u(ix,2    ,:,patches.bo,:,:),[],5),[],6);
348      Cto = fft(fft(u(ix,ny-1 ,:,patches.to,:,:),[],5),[],6);
349 end
```

Now invert the double Fourier transforms to complete interpolation. (Should stagShift be multiplied by rx/ry??) Enforce reality when appropriate.

```
357 u(nx,iy,:,:,:,:) = uclean( ifft(ifft( ...
358     Cle.*exp(1i*(stagShift+krx))  ,[],5),[],6) );
359 u( 1,iy,:,:,:,:) = uclean( ifft(ifft( ...
360     Cri.*exp(1i*(stagShift-krx))  ,[],5),[],6) );
361 u(ix,ny,:,:,:,:) = uclean( ifft(ifft( ...
362     Cbo.*exp(1i*(stagShift+kry))  ,[],5),[],6) );
363 u(ix, 1,:,:,:,:) = uclean( ifft(ifft( ...
364     Cto.*exp(1i*(stagShift-kry))  ,[],5),[],6) );
365 end% if spectral
```

Nan the corner values of every 2D patch.

```
371 u([1 nx],[1 ny],:,:,:,:)=nan;
372 end% function patchEdgeInt2
```

Fin, returning the 6D array of field values with interpolated edges.

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

Section contents

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

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

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

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

Establish 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.

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

3.13.1 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.

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

Now construct the Jacobian. Since this is a *linear* wave PDE, use large perturbations.

```
62 small = 1;
63 jac = nan(length(i));
64 sizeJacobian = size(jac)
65 for j = 1:length(i)
66   uv = uv0(:);
67   uv(i(j)) = uv(i(j))+small;
```

```
68    tmp = patchSmooth2(0,uv)/small;
69    jac(:,j) = tmp(i);
70  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.

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

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

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

3.13.2 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: $n_x \times n_y \times 1 \times 1 \times N_x \times N_y$.

```
109 u0 = exp(-patches.x.^2-patches.y.^2);
110 v0 = zeros(size(u0));
```

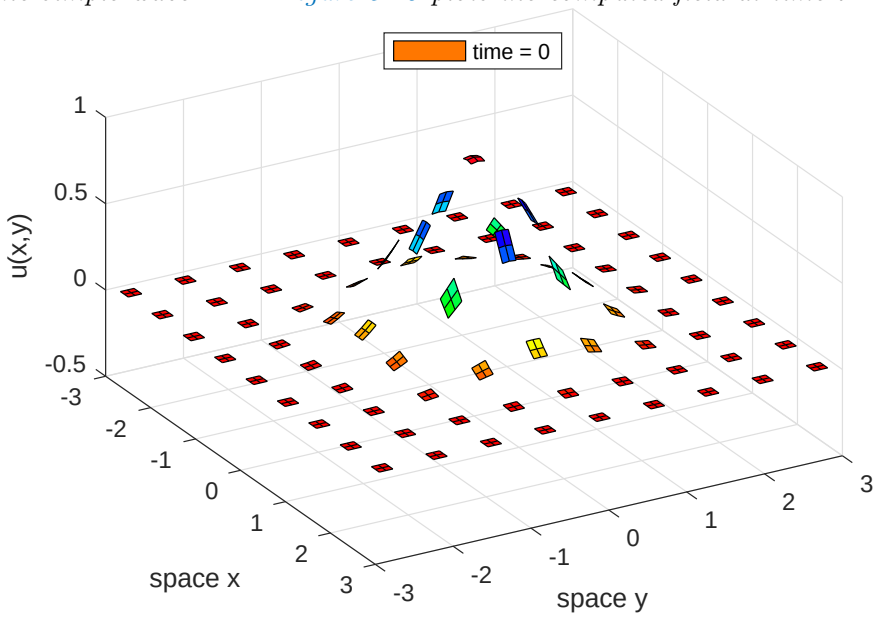
Initiate a plot of the simulation using only the microscale values interior to the patches: set x and y-edges to nan to leave the gaps. Start by showing the initial conditions of Figure 3.16 while the simulation computes. To mesh/surf plot we need to ?? 'transpose' to size $n_x \times N_x \times n_y \times N_y$, then reshape to size $n_x \cdot N_x \times n_y \cdot N_y$.

```
122 x = squeeze(patches.x); y = squeeze(patches.y);
123 x([1 end],:) = nan; y([1 end],:) = nan;
124 u = reshape(permute(squeeze(u0),[1 3 2 4]), [numel(x) numel(y)]);
125 usurf = surf(x(:),y(:),u');
126 axis([-3 3 -3 3 -0.5 1]), view(60,40)
127 xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
128 legend('time = 0','Location','north')
129 colormap(hsv)
130 drawnow
131 ifOurCf2eps([mfilename 'ic'])
```

Integrate in time using standard functions.

```
144 disp('Wait while we simulate u_t=v, v_t=u_xx+u_yy')
145 uv0 = cat(3,u0,v0);
146 if ~exist('OCTAVE_VERSION','builtin')
```

Figure 3.18: initial field $u(x,y,t)$ at time $t=0$ of the patch scheme applied to the simple wave PDE*: Figure 3.19 plots the computed field at time $t=2$.*




```
147 [ts,uvs] = ode23( @patchSmooth2,[0 6],uv0(:));
148 else % octave version is slower
149 [ts,uvs] = odeOcts(@patchSmooth2,linspace(0,6),uv0(:));
150 end
```

Animate the computed simulation to end with Figure 3.19. Because of the very small time-steps, subsample to plot at most 100 times.

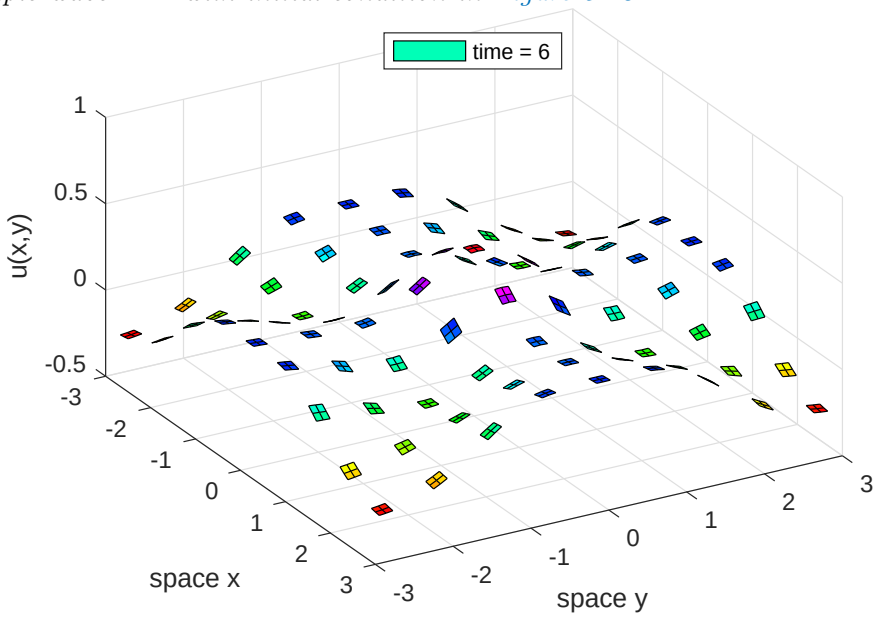
```
158 di = ceil(length(ts)/100);
159 for i = [1:di:length(ts)-1 length(ts)]
160   uv = patchEdgeInt2(uvs(i,:));
161   uv = reshape(permute(uv,[1 5 2 6 3 4]), [numel(x) numel(y) 2]);
162   set(usurf,'ZData', uv(:,:,1)');
163   legend(['time = ' num2str(ts(i),2)])
164   pause(0.1)
165 end
166 ifOurCf2eps([mfilename 't' num2str(ts(end))])
```

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

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

```
14 function uvt = wavePDE(t,uv,patches)
15   dx = diff(patches.x(1:2));
16   dy = diff(patches.y(1:2));  % microscale spacing
17   i = 2:size(uv,1)-1;
18   j = 2:size(uv,2)-1; % interior patch-points
19   uvt = nan+uv;  % preallocate storage
20   uvt(i,j,1,:) = uv(i,j,2,:);
```

Figure 3.19: field $u(x,y,t)$ at time $t=6$ of the patch scheme applied to the simple wave PDE *with initial condition in Figure 3.18.*




```
21    uvt(i,j,2,:) = diff(uv(:,j,1,:),2,1)/dx^2 ...
22                   +diff(uv(i,:,1,:),2,2)/dy^2;
23  end
```

```
10  function [ts,xs] = odeOcts(dxdt,tSpan,x0)
11      if length(tSpan)>2, ts = tSpan;
12      else ts = linspace(tSpan(1),tSpan(end),21)';
13      end
14      lsode_options('integration method','non-stiff');
15      xs = lsode(@(x,t) dxdt(t,x),x0,ts);
16  end
```

3.14 homoDiffEdgy2: computational homogenisation of a 2D diffusion via simulation on small patches

This section extends to 2D the 1D code discussed in Section 3.5. First set random heterogeneous diffusivities of random period in each of the two directions. Crudely normalise by the harmonic mean so the decay time scale is roughly one.

```
mPeriod = randi([2 3],1,2)
cHetr = exp(1*randn([mPeriod 2]));
cHetr = cHetr*mean(1./cHetr(:))
```

Configure the patch scheme with some arbitrary choices of domain, patches, size ratios. Use spectral interpolation as we test other orders subsequently. In 2D we appear to get only real eigenvalues by using edgy interpolation. What happens for non-edgy interpolation is unknown.

```
edgyInt = true;
nEnsem = 1 %prod(mPeriod) % or just set one
if nEnsem==1% use more patches
    nPatch = [9 9]
    nSubP = (2-edgyInt)*mPeriod+1+edgyInt
else % when nEnsem>1 use fewer patches
    nPatch = [5 5]
    nSubP = mPeriod+randi([1 4],1,2) % +2 is decoupled
end
ratio = 0.2+0.2*rand(1,2)
configPatches2(@heteroDiff2,[-pi pi -pi pi],nan,nPatch ...
    ,0,ratio,nSubP ,'EdgyInt',edgyInt ,'nEnsem',nEnsem ...
    ,'hetCoeffs',cHetr );
```

Simulate Set initial conditions of a simulation, replicated for each in the ensemble.

```
global patches
u0 = 0.8*cos(patches.x).*sin(patches.y) ...
     +0.1*randn([nSubP,1,1,nPatch]);
u0 = repmat(u0,1,1,1,nEnsem,1,1);
```

Integrate using standard integrators, unevenly spaced in time to better display transients.

```
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@patchSmooth2, 0.3*linspace(0,1).^2, u0(:));
else % octave version
    [ts,us] = odeOcts(@patchSmooth2, 0.3*linspace(0,1).^2, u0(:));
end
```

Plot the solution as an animation over time.

```
if ts(end)>0.099, disp('plot animation of solution field')
figure(1), clf, colormap(hsv)
```

Get spatial coordinates and pad them with NaNs to separate patches.

```
x = squeeze(patches.x); y = squeeze(patches.y);
x(end+1,:)=nan;  y(end+1,:)=nan; % pad with nans
```

For every time step draw the surface and pause for a short display.

```
for i = 1:length(ts)
```

Get the row vector of data, form into the 6D array via the interpolation to the edges, then pad with Nans between patches, and reshape to suit the surf function.

```
  u = squeeze( mean( patchEdgeInt2(us(i,:)) ,4));
  u(end+1,:,:,:)=nan; u(:,end+1,:,:)=nan;
  u = reshape(permute(u,[1 3 2 4]), [numel(x) numel(y)]);
```

If the initial time then draw the surface with labels, otherwise just update the surface data.

```
  if i==1
       hsurf = surf(x(:),y(:),u'); view(60,40)
       axis([-pi pi -pi pi -1 1]),   caxis([-1 1])
       xlabel('x'), ylabel('y'), zlabel('u(x,y)')
  else set(hsurf,'ZData', u');
  end
  legend(['time = ' num2str(ts(i),2)],'Location','north')
  pause(0.05)
```

finish the animation loop and if-plot.

```
end%for over time
end%if-plot
```

3.14.1 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. Except here use a small ratio as we do not plot.

```
ratio = [0.1 0.1]
nLeadEvals=prod(nPatch)+max(nPatch);
leadingEvals=[];
```

Evaluate eigenvalues for spectral as the base case for polynomial interpolation of order $2, 4, \ldots$.

```
maxords=10;
for ord=0:2:maxords
    ord=ord
```

Configure with same parameters, then because they are reset by this configuration, restore coupling.

```
161     configPatches2(@heteroDiff2,[-pi pi -pi pi],nan,nPatch ...
162         ,ord,ratio,nSubP,'EdgyInt',edgyInt,'nEnsem',nEnsem ...
163         ,'hetCoeffs',cHetr);
```

Find which elements of the 6D array are interior micro-grid points and hence correspond to dynamical variables.

```
170     u0 = zeros([nSubP,1,nEnsem,nPatch]);
171     u0([1 end],:,:) = nan;
172     u0(:,[1 end],:) = nan;
173     i = find(~isnan(u0));
```

Construct the Jacobian of the scheme as the matrix of the linear transformation, obtained by transforming the standard unit vectors.

```
181     jac = nan(length(i));
182     sizeJacobian = size(jac)
183     for j = 1:length(i)
184       u = u0(:)+(i(j)==(1:numel(u0))');
185       tmp = patchSmooth2(0,u);
186       jac(:,j) = tmp(i);
187     end
```

Test for symmetry, with error if we know it should be symmetric.

```
194     notSymmetric=norm(jac-jac')
195     if edgyInt, assert(notSymmetric<1e-7,'failed symmetry')
196     elseif notSymmetric>1e-7, disp('failed symmetry')
197     end
```

Find all the eigenvalues (as `eigs` is unreliable).

```
203     if edgyInt, [evecs,evals] = eig((jac+jac')/2,'vector');
204     else evals = eig(jac);
205     end
206     biggestImag=max(abs(imag(evals)));
207     if biggestImag>0, biggestImag=biggestImag, end
```

Sort eigenvalues on their real-part with most positive first, and most negative last. Store the leading eigenvalues in `egs`, and write out when computed all orders. The number of zero eigenvalues, `nZeroEv`, gives the number of decoupled systems in this patch configuration.

```
217     [~,k] = sort(-real(evals));
218     evals=evals(k); evecs=evecs(:,k);
219     if ord==0, nZeroEv=sum(abs(evals(:))<1e-5), end
220     if ord==0, evec0=evecs(:,1:nZeroEv*nLeadEvals);
221     else % find evec closest to that of each leading spectral
222         [~,k]=max(abs(evecs'*evec0));
223         evals=evals(k); % sort in corresponding order
224     end
225     leadingEvals=[leadingEvals evals(nZeroEv*(1:nLeadEvals))];
226 end
```

```
227 disp('     spectral    quadratic      quartic  sixth-order ...')
228 leadingEvals=leadingEvals
```

Plot the errors in the eigenvalues using the spectral ones as accurate. Only plot every second, `iEv`, as all are repeated eigenvalues.

```
237 if maxords>2
238     iEv=2:2:12;
239     figure(2);
240     err=abs(leadingEvals-leadingEvals(:,1)) ...
241         ./(1e-7+abs(leadingEvals(:,1)));
242     semilogy(2:2:maxords,err(iEv,2:end)','o:')
243     xlabel('coupling order')
244     ylabel('eigenvalue relative error')
245     leg=legend( ...
246         strcat('$',num2str(real(leadingEvals(iEv,1)),'%.4f'),'$') ...
247         ,'Location','northeastoutside');
248     if ~exist('OCTAVE_VERSION','builtin')
249         title(leg,'eigenvalues'), end
250     legend boxoff
251 end%if-plot
```

3.14.2 `heteroDiff2()`: heterogeneous diffusion

This function codes the lattice heterogeneous diffusion inside the patches. For 6D input arrays `u`, `x`, and `y` (via edge-value interpolation of `patchSmooth2`, Section 3.11), computes the time derivative (3.1) at each point in the interior of a patch, output in `ut`. The two 2D array of diffusivities, c^x_{ij} and c^y_{ij}, have previously been stored in `patches.cs` (3D).

```
19 function ut = heteroDiff2(t,u,patches)
20   dx = diff(patches.x(2:3));  % x space step
21   dy = diff(patches.y(2:3));  % y space step
22   ix = 2:size(u,1)-1; % x interior points in a patch
23   iy = 2:size(u,2)-1; % y interior points in a patch
24   ut = nan+u;         % preallocate output array
25   ut(ix,iy,:,:,:,:) ...
26   = diff(patches.cs(:,iy,1,:).*diff(u(:,iy,:,:,:,:),1),1)/dx^2 ...
27    +diff(patches.cs(ix,:,2,:).*diff(u(ix,:,:,:,:,:),1,2),1,2)/dy^2;
28 end% function
```

Fin.

3.15 `configPatches3()`: configures spatial patches in 3D

Section contents

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSmooth3()`, and possibly other patch functions. Sections 3.15.1 and 3.18 list examples of its use.

```
20 function patches = configPatches3(fun,Xlim,BCs ...
21     ,nPatch,ordCC,ratio,nSubP,varargin)
```

Input If invoked with no input arguments, then executes an example of simulating a heterogeneous wave PDE—see Section 3.15.1 for the example code.

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)`, that computes time-derivatives (or time-steps) of quantities on the 3D micro-grid within all the 3D patches.
- `Xlim` array/vector giving the macro-space domain of the computation: patches are distributed equi-spaced over the interior of the rectangular cuboid $[\texttt{Xlim(1)},\texttt{Xlim(2)}] \times [\texttt{Xlim(3)},\texttt{Xlim(4)} \times [\texttt{Xlim(5)},\texttt{Xlim(6)}]$: if `Xlim` is of length two, then the domain is the cubic domain of the same interval in all three directions.
- `BCs` eventually and somehow will define the macroscale boundary conditions. Currently, `BCs` is ignored and the system is assumed macro-periodic in the specified rectangular domain.
- `nPatch` sets the number of equi-spaced spaced patches: if scalar, then use the same number of patches in all three directions, otherwise `nPatch(1:3)` gives the number (≥ 1) of patches in each direction.
- `ordCC` is the 'order' of interpolation for inter-patch coupling across empty space of the macroscale patch values to the edge-values of the patches: currently must be $0, 2, 4, \ldots$; where 0 gives spectral interpolation.
- `ratio` (real) is the ratio of (depending upon `EdgyInt`) either the half-width or full-width of a patch to the spacing of the patch mid-points. So either `ratio` $= \frac{1}{2}$ means the patches abut and `ratio` $= 1$ is overlapping patches as in holistic discretisation, or `ratio` $= 1$ means the patches abut. Small `ratio` should greatly reduce computational time. If scalar, then use the same ratio in all three directions, otherwise `ratio(1:3)` gives the ratio in each of the three directions.

- `nSubP` is the number of equi-spaced microscale lattice points in each patch: if scalar, then use the same number in all three directions, otherwise `nSubP(1:3)` sets the number in each direction. If not using `EdgyInt`, then must be odd so that there is/are centre-patch micro-grid point/planes in each patch.
- `'nEdge'` (not yet implemented), *optional*, default=1, for each patch, 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 neighbour interactions).
- `'EdgyInt'`, true/false, *optional*, default=false. If true, then interpolate to left/right/top/bottom/front/back face-values from right/left/bottom/top/back/front next-to-face values. If false or omitted, then interpolate from centre-patch planes.
- `'nEnsem'`, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.
- `'hetCoeffs'`, *optional*, default empty. Supply a 3/4D array of microscale heterogeneous coefficients to be used by the given microscale `fun` in each patch. Say the given array `cs` is of size $m_x \times m_y \times m_z \times n_c$, where n_c is the number of different arrays of coefficients. For example, in heterogeneous diffusion, $n_c = 3$ for the diffusivities in the *three* different spatial directions. The coefficients are to be the same for each and every patch. However, macroscale variations are catered for by the n_c coefficients being n_c parameters in some macroscale formula.
  - If `nEnsem` $= 1$, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the $(1, 1, 1)$-point in each patch.
  - If `nEnsem` > 1 (value immaterial), then reset `nEnsem` $:= m_x \cdot m_y \cdot m_z$ and construct an ensemble of all $m_x \cdot m_y \cdot m_z$ phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When `EdgyInt` is true, and when the coefficients are diffusivities/elasticities in x, y, z-directions, respectively, then this coupling cunningly preserves symmetry.
- `'parallel'`, true/false, *optional*, default=false. If false, then all patch computations are on the user's main CPU—although a user may well separately invoke, say, a GPU to accelerate sub-patch computations.

  If true, and it requires that you have MATLAB's Parallel Computing Toolbox, then it will distribute the patches over multiple CPUs/cores. In MATLAB, only one array dimension can be split in the distribution, so it chooses the one space dimension x, y, z corresponding to the highest `\nPatch` (if a tie, then chooses the rightmost of x, y, z). A user may correspondingly distribute arrays with property `patches.codist`, or simply use formulas invoking the preset distributed arrays `patches.x`, `patches.y`, and `patches.z`. If a user has not yet established a parallel pool, then a 'local' pool is started.

Output The struct `patches` is created and set with the following components. If no output variable is provided for `patches`, then make the struct available as a global variable.[6]

```
160 if nargout==0, global patches, end
```

- `.fun` is the name of the user's function `fun(t,u,patches)` or `fun(t,u)` that computes the time derivatives (or steps) on the patchy lattice.
- `.ordCC` is the specified order of inter-patch coupling.
- `.stag` 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` $\times$ 3-array of weights for the inter-patch interpolation onto the right/top/front and left/bottom/back faces (respectively) with patch:macroscale ratio as specified.
- `.x` (8D) is `nSubP(1)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(1)` $\times 1 \times 1$ array of the regular spatial locations x_{iI} of the microscale grid points in every patch.
- `.y` (8D) is $1 \times$ `nSubP(2)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(2)` $\times 1$ array of the regular spatial locations y_{jJ} of the microscale grid points in every patch.
- `.z` (8D) is $1 \times 1 \times$ `nSubP(3)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(3)` array of the regular spatial locations z_{kK} of the microscale grid points in every patch.
- `.ratio` 1×3, are the size ratios of every patch.
- `.nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.
- `.le`, `.ri`, `.bo`, `.to`, `.ba`, `.fr` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.
- `.cs` either
  - `[]` 0D, or
  - if `nEnsem` $= 1$, (`nSubP(1)`-1)$\times$(`nSubP(2)`-1)$\times$(`nSubP(3)`-1)$\times n_c$ 4D array of microscale heterogeneous coefficients, or
  - if `nEnsem` > 1, (`nSubP(1)` $- 1$) $\times$ (`nSubP(2)` $- 1$) $\times$ (`nSubP(3)` $- 1$) $\times n_c \times m_x m_y m_z$ 5D array of $m_x m_y m_z$ ensemble of phase-shifts of the microscale heterogeneous coefficients.
- `.parallel`, logical: true if patches are distributed over multiple CPUs/cores for the Parallel Computing Toolbox, otherwise false (the default is to activate the *local* pool).
- `.codist`, *optional*, describes the particular parallel distribution of arrays over the active parallel pool.

[6] When using `spmd` parallel computing, it is generally best to avoid global variables, and so instead prefer using an explicit output variable.

3.15.1 If no arguments, then execute an example

```
242 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. configPatches3
2. ode23 integrator ↔ patchSmooth3 ↔ user's PDE
3. process results

Set random heterogeneous coefficients of period two in each of the three directions. Crudely normalise by the harmonic mean so the decay time scale is roughly one.

```
260 mPeriod = [2 2 2];
261 cHetr = exp(0.3*randn([mPeriod 3]));
262 cHetr = cHetr*mean(1./cHetr(:))
```

Establish global patch data struct to interface with a function coding a nonlinear 'diffusion' PDE: to be solved on $[-\pi,\pi]^3$-periodic domain, with 5^3 patches, spectral interpolation (0) couples the patches, each patch of half-size ratio 0.4 (relatively large for visualisation), and with 4^3 points forming each patch.

```
275 global patches
276 patches = configPatches3(@heteroWave3,[-pi pi], nan ...
277     , 5, 0, 0.35, mPeriod+2 ,'EdgyInt',true ...
278     ,'hetCoeffs',cHetr);
```

Set a wave initial state using auto-replication of the spatial grid, and as Figure 3.20 shows. This wave propagates diagonally across space.

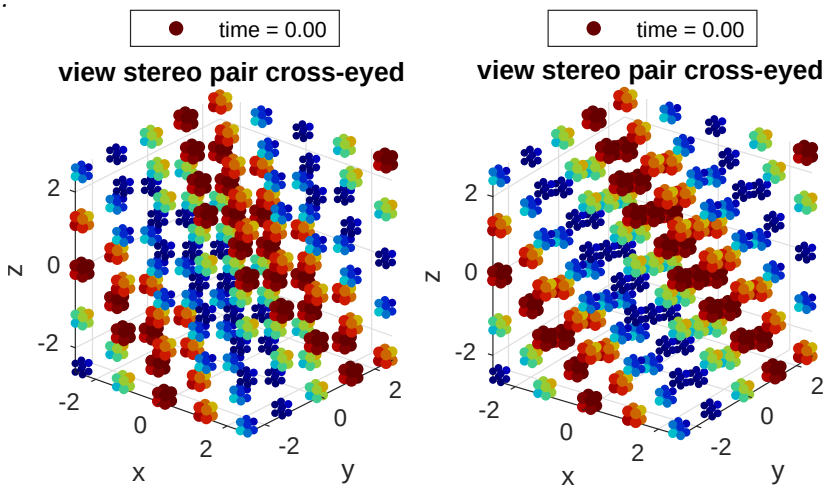
```
286 u0 = 0.5+0.5*sin(patches.x+patches.y+patches.z);
287 v0 =    -0.5*cos(patches.x+patches.y+patches.z)*sqrt(3);
288 uv0 = cat(4,u0,v0);
```

Integrate in time to $t = 6$ using standard functions. In Matlab `ode15s` would be natural as the patch scheme is naturally stiff, but `ode23` is much quicker (Maclean et al. 2020, Fig. 4).

```
305 disp('Simulate heterogeneous wave u_tt=div[C*grad(u)]')
306 if ~exist('OCTAVE_VERSION','builtin')
307     [ts,us] = ode23(@patchSmooth3,linspace(0,6),uv0(:));
308 else %disp('octave version is very slow for me')
309     lsode_options('absolute tolerance',1e-4);
310     lsode_options('relative tolerance',1e-4);
311     [ts,us] = odeOcts(@patchSmooth3,[0 1 2],uv0(:));
312 end
```

Animate the computed simulation to end with Figure 3.21. Use `patchEdgeInt3` to obtain patch-face values (but not edge nor corner values, and even if not drawn) in order to most easily reconstruct the array data structure.

Figure 3.20: initial field $u(x, y, z, t)$ at time $t = 0$ of the patch scheme applied to a heterogeneous wave PDE*: Figure 3.21 plots the computed field at time $t = 6$.*



Replicate x, y, and z arrays to get individual spatial coordinates of every data point. Then, optionally, set faces to `nan` so the plot just shows patch-interior data.

```
figure(1), clf, colormap(0.8*jet)
xs = patches.x+0*patches.y+0*patches.z;
ys = patches.y+0*patches.x+0*patches.z;
zs = patches.z+0*patches.y+0*patches.x;
if 1, xs([1 end],:,:,:)=nan;
      xs(:,[1 end],:,:)=nan;
      xs(:,:,[1 end],:)=nan;
end;%option
j=find(~isnan(xs));
```

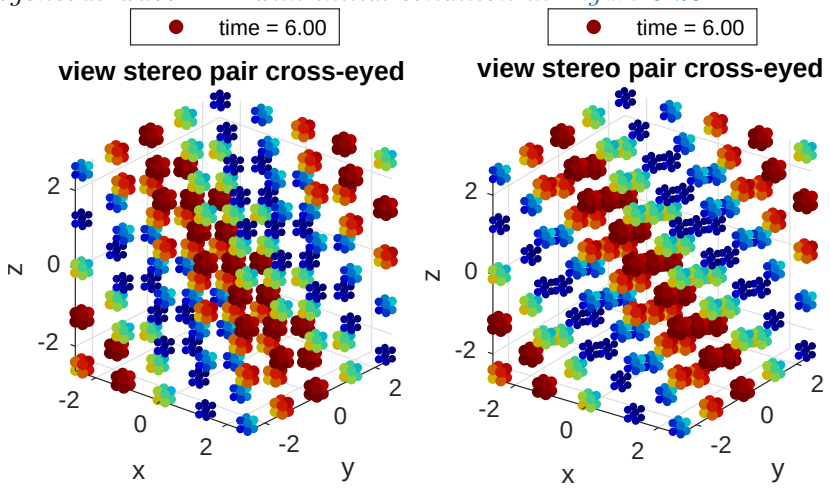
In the scatter plot, these functions `pix()` and `col()` map the u-data values to the size of the dots and to the colour of the dots, respectively.

```
pix = @(u) 15*abs(u)+7;
col = @(u) sign(u).*abs(u);
```

Loop to plot at each and every time step.

```
for i = 1:length(ts)
  uv = patchEdgeInt3(us(i,:));
  u = uv(:,:,:,1,:);
  for p=1:2
    subplot(1,2,p)
    if (i==1)| exist('OCTAVE_VERSION','builtin')
      scat(p) = scatter3(xs(j),ys(j),zs(j),'filled');
      axis equal, caxis(col([0 1])), view(45-5*p,25)
      xlabel('x'), ylabel('y'), zlabel('z')
      title('view stereo pair cross-eyed')
    end % in matlab just update values
```

Figure 3.21: field $u(x, y, z, t)$ at time $t = 6$ of the patch scheme applied to the heterogeneous wave PDE *with initial condition in Figure 3.20.*




```
361       set(scat(p),'CData',col(u(j)) ...
362          ,'SizeData',pix((8+xs(j)-ys(j)+zs(j))/6+0*u(j)));
363       legend(['time = ' num2str(ts(i),'%4.2f')],'Location','north')
364     end
```

Optionally save the initial condition to graphic file for Figure 3.16, and optionally save the last plot.

```
372     if i==1,
373       ifOurCf2eps([mfilename 'ic'])
374       disp('Type space character to animate simulation')
375       pause
376     else pause(0.05)
377     end
378   end% i-loop over all times
379   ifOurCf2eps([mfilename 'fin'])
```

Upon finishing execution of the example, exit this function.

```
394   return
395   end%if no arguments
```

3.15.2 `heteroWave3()`: heterogeneous Waves

This function codes the lattice heterogeneous waves inside the patches. The wave PDE is

$$u_t = v, \quad v_t = \vec{\nabla}(C\vec{\nabla} \cdot u)$$

for diagonal matrix C which has microscale variations. For 8D input arrays `u`, `x`, `y`, and `z` (via edge-value interpolation of `patchSmooth3`, Section 3.16), computes the time derivative at each point in the interior of a patch, output in `ut`. The three 3D array of heterogeneous coefficients, c^x_{ijk}, c^y_{ijk} and c^z_{ijk}, have previously been stored in `patches.cs` (4D).

Supply patch information as a third argument (required by parallel computation), or otherwise by a global variable.

```
function ut = heteroWave3(t,u,patches)
  if nargin<3, global patches, end
```

Microscale space-steps, and interior point indices.

```
  dx = diff(patches.x(2:3));  % x micro-scale step
  dy = diff(patches.y(2:3));  % y micro-scale step
  dz = diff(patches.z(2:3));  % z micro-scale step
  i = 2:size(u,1)-1; % x interior points in a patch
  j = 2:size(u,2)-1; % y interior points in a patch
  k = 2:size(u,3)-1; % z interior points in a patch
```

Reserve storage and then assign interior patch values to the heterogeneous diffusion time derivatives. Using `nan+u` appears quicker than `nan(size(u),patches.codist)`

```
  ut = nan+u;  % preallocate output array
  ut(i,j,k,1,:) = u(i,j,k,2,:);
  ut(i,j,k,2,:) ...
  =diff(patches.cs(:,j,k,1,:).*diff(u(:,j,k,1,:),1),1)/dx^2 ...
  +diff(patches.cs(i,:,k,2,:).*diff(u(i,:,k,1,:),1,2),1,2)/dy^2 ...
  +diff(patches.cs(i,j,:,3,:).*diff(u(i,j,:,1,:),1,3),1,3)/dz^2;
end% function
```

3.15.3 Parse input arguments and defaults

```
p = inputParser;
fnValidation = @(f) isa(f, 'function_handle'); %test for fn name
addRequired(p,'fun',fnValidation);
addRequired(p,'Xlim',@isnumeric);
addRequired(p,'BCs'); % nothing yet decided
addRequired(p,'nPatch',@isnumeric);
addRequired(p,'ordCC',@isnumeric);
addRequired(p,'ratio',@isnumeric);
addRequired(p,'nSubP',@isnumeric);
addParameter(p,'nEdge',1,@isnumeric);
addParameter(p,'EdgyInt',false,@islogical);
addParameter(p,'nEnsem',1,@isnumeric);
addParameter(p,'hetCoeffs',[],@isnumeric);
addParameter(p,'parallel',false,@islogical);
%addParameter(p,'nCore',1,@isnumeric); % not yet implemented
parse(p,fun,Xlim,BCs,nPatch,ordCC,ratio,nSubP,varargin{:});
```

Set the optional parameters.

```
patches.nEdge = p.Results.nEdge;
patches.EdgyInt = p.Results.EdgyInt;
patches.nEnsem = p.Results.nEnsem;
cs = p.Results.hetCoeffs;
```

```
437 patches.parallel = p.Results.parallel;
438 %patches.nCore = p.Results.nCore;
```

Initially duplicate parameters for three space dimensions as needed.

```
446 if numel(Xlim)==2, Xlim = repmat(Xlim,1,3); end
447 if numel(nPatch)==1, nPatch = repmat(nPatch,1,3); end
448 if numel(ratio)==1, ratio = repmat(ratio,1,3); end
449 if numel(nSubP)==1, nSubP = repmat(nSubP,1,3); end
```

Check parameters.

```
456 assert(patches.nEdge==1 ...
457       ,'multi-edge-value interp not yet implemented')
458 assert(all(2*patches.nEdge<nSubP) ...
459       ,'too many edge values requested')
460 %if patches.nCore>1
461 %    warning('nCore>1 not yet tested in this version')
462 %    end
```

3.15.4 The code to make patches

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

```
476 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 (not yet??) -1.

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

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

```
493 patches.stag = mod(ordCC,2);
494 assert(patches.stag==0,'staggered not yet implemented??')
495 ordCC = ordCC+patches.stag;
496 patches.ordCC = ordCC;
```

Check for staggered grid and periodic case.

```
502 if patches.stag, assert(all(mod(nPatch,2)==0), ...
503     'Require an even number of patches for staggered grid')
504 end
```

Might as well precompute the weightings for the interpolation of field values for coupling. (Could sometime extend to coupling via derivative values.) Store the size ratio in `patches`.

```
513 ratio = reshape(ratio,1,3); % force to be row vector
514 patches.ratio = ratio;
515 if ordCC>0
516     [Cwtsr,Cwtsl] = patchCwts(ratio,ordCC,patches.stag);
```

```
517     patches.Cwtsr = Cwtsr;  patches.Cwtsl = Cwtsl;
518 end
```

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

```
526 X = linspace(Xlim(1),Xlim(2),nPatch(1)+1);
527 DX = X(2)-X(1);
528 X = X(1:nPatch(1))+diff(X)/2;
529 Y = linspace(Xlim(3),Xlim(4),nPatch(2)+1);
530 DY = Y(2)-Y(1);
531 Y = Y(1:nPatch(2))+diff(Y)/2;
532 Z = linspace(Xlim(5),Xlim(6),nPatch(3)+1);
533 DZ = Z(2)-Z(1);
534 Z = Z(1:nPatch(3))+diff(Z)/2;
```

Construct the microscale in each patch, assuming Dirichlet patch edges, and half-patch widths of `ratio(1)` · `DX`, `ratio(2)` · `DY` and `ratio(3)` · `DZ`, unless `patches.EdgyInt` is true in which case every patch is of widths `ratio*DX+dx`, `ratio*DY+dy` and `ratio*DZ+dz`.

```
546 nSubP = reshape(nSubP,1,3); % force to be row vector
547 assert(patches.EdgyInt | all(mod(nSubP,2)==1), ...
548     'configPatches3: nSubP must be odd')
549 i0 = (nSubP(1)+1)/2;
550 if ~patches.EdgyInt, dx = ratio(1)*DX/(i0-1);
551 else                 dx = ratio(1)*DX/(nSubP(1)-2);
552 end
553 patches.x = dx*(-i0+1:i0-1)'+X;  % micro-grid
554 patches.x = reshape(patches.x,nSubP(1),1,1,1,1,nPatch(1),1,1);
555 i0 = (nSubP(2)+1)/2;
556 if ~patches.EdgyInt, dy = ratio(2)*DY/(i0-1);
557 else                 dy = ratio(2)*DY/(nSubP(2)-2);
558 end
559 patches.y = dy*(-i0+1:i0-1)'+Y;  % micro-grid
560 patches.y = reshape(patches.y,1,nSubP(2),1,1,1,1,nPatch(2),1);
561 i0 = (nSubP(3)+1)/2;
562 if ~patches.EdgyInt, dz = ratio(3)*DZ/(i0-1);
563 else                 dz = ratio(3)*DZ/(nSubP(3)-2);
564 end
565 patches.z = dz*(-i0+1:i0-1)'+Z;  % micro-grid
566 patches.z = reshape(patches.z,1,1,nSubP(3),1,1,1,1,nPatch(3));
```

3.15.5 Set ensemble inter-patch communication

For `EdgyInt` or centre interpolation respectively,

- the right-face/centre realisations `1:nEnsem` are to interpolate to left-face `le`, and
- the left-face/centre realisations `1:nEnsem` are to interpolate to `re`.

`re` and `li` are 'transposes' of each other as `re(li)=le(ri)` are both `1:nEnsem`. Similarly for bottom-face/centre interpolation to top-face via `to`, top-face/centre interpolation to bottom-face via `bo`, back-face/centre interpolation to front-face via `fr`, and front-face/centre interpolation to back-face via `ba`.

The default is nothing shifty. This setting reduces the number of if-statements in function `patchEdgeInt3()`.

```
nE = patches.nEnsem;
patches.le = 1:nE;  patches.ri = 1:nE;
patches.bo = 1:nE;  patches.to = 1:nE;
patches.ba = 1:nE;  patches.fr = 1:nE;
```

However, if heterogeneous coefficients are supplied via `hetCoeffs`, then do some non-trivial replications. First, get microscale periods, patch size, and replicate many times in order to subsequently sub-sample: `nSubP` times should be enough. If `cs` is more then 4D, then the higher-dimensions are reshaped into the 4th dimension.

```
if ~isempty(cs)
  [mx,my,mz,nc] = size(cs);
  nx = nSubP(1); ny = nSubP(2); nz = nSubP(3);
  cs = repmat(cs,nSubP);
```

If only one member of the ensemble is required, then sub-sample to patch size, and store coefficients in `patches` as is.

```
  if nE==1, patches.cs = cs(1:nx-1,1:ny-1,1:nz-1,:); else
```

But for `nEnsem` > 1 an ensemble of $m_x m_y m_z$ phase-shifts of the coefficients is constructed from the over-supply. Here code phase-shifts over the periods—the phase shifts are like Hankel-matrices.

```
    patches.nEnsem = mx*my*mz;
    patches.cs = nan(nx-1,ny-1,nz-1,nc,mx,my,mz);
    for k = 1:mz
      ks = (k:k+nz-2);
      for j = 1:my
        js = (j:j+ny-2);
        for i = 1:mx
          is = (i:i+nx-2);
          patches.cs(:,:,:,:,i,j,k) = cs(is,js,ks,:);
        end
      end
    end
    patches.cs = reshape(patches.cs,nx-1,ny-1,nz-1,nc,[]);
```

Further, set a cunning left/right/bottom/top/front/back realisation of inter-patch coupling. The aim is to preserve symmetry in the system when also invoking `EdgyInt`. What this coupling does without `EdgyInt` is unknown. Use auto-replication.

```
    mmx=(0:mx-1)'; mmy=0:my-1; mmz=shiftdim(0:mz-1,-1);
    le = mod(mmx+mod(nx-2,mx),mx)+1;
```

```
657       patches.le = reshape(  le+mx*(mmy+my*mmz)  ,[],1);
658       ri = mod(mmx-mod(nx-2,mx),mx)+1;
659       patches.ri = reshape(  ri+mx*(mmy+my*mmz)  ,[],1);
660       bo = mod(mmy+mod(ny-2,my),my)+1;
661       patches.bo = reshape( 1+mmx+mx*(bo-1+my*mmz) ,[],1);
662       to = mod(mmy-mod(ny-2,my),my)+1;
663       patches.to = reshape( 1+mmx+mx*(to-1+my*mmz) ,[],1);
664       ba = mod(mmz+mod(nz-2,mz),mz)+1;
665       patches.ba = reshape( 1+mmx+mx*(mmy+my*(ba-1)) ,[],1);
666       fr = mod(mmz-mod(nz-2,mz),mz)+1;
667       patches.fr = reshape( 1+mmx+mx*(mmy+my*(fr-1)) ,[],1);
```

Issue warning if the ensemble is likely to be affected by lack of scale separation. Need to justify this and the arbitrary threshold more carefully??

```
675   if prod(ratio)*patches.nEnsem>0.9, warning( ...
676   'Probably poor scale separation in ensemble of coupled phase-shifts')
677   scaleSeparationParameter = ratio*patches.nEnsem
678   end
```

End the two if-statements.

```
684     end%if-else nEnsem>1
685   end%if not-empty(cs)
```

If parallel code then first assume this is not within an `spmd`-environment, and so we invoke `spmd...end` (which starts a parallel pool if not already started). At this point, the global `patches` is copied for each worker processor and so it becomes *composite* when we distribute any one of the fields. Hereafter, *all fields in the global variable* `patches` *must only be referenced within an* `spmd`*-environment.*[7]

```
704   if patches.parallel
705     spmd
```

Second, decide which dimension is to be sliced among parallel workers (for the moment, do not consider slicing the ensemble). Choose the direction of most patches, biased towards the last.

```
714     [~,pari]=max(nPatch+0.01*(1:3));
715     patches.codist=codistributor1d(5+pari);
```

`patches.codist.Dimension` is the index that is split among workers. Then distribute the appropriate coordinate direction among the workers: the function must be invoked inside an `spmd`-group in order for this to work—so we do not need `parallel` in argument list.

```
725     switch pari
726       case 1, patches.x=codistributed(patches.x,patches.codist);
727       case 2, patches.y=codistributed(patches.y,patches.codist);
728       case 3, patches.z=codistributed(patches.z,patches.codist);
```

[7] If subsequently outside spmd, then one must use functions like `getfield(patches{1},'a')`.

```
729   otherwise
730     error('should never have bad index for parallel distribution')
731   end%switch
732   end%spmd
```

If not parallel, then clean out `patches.codist` if it exists. May not need, but safer.

```
740 else% not parallel
741   if isfield(patches,'codist'), rmfield(patches,'codist'); end
742 end%if-parallel
```

Fin

```
752 end% function
```

3.16 patchSmooth3(): interface 3D space to time integrators

To simulate in time with 3D 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 enough* so that the patch centre-values are sensible macroscale variables, and patch edge-values are determined by macroscale interpolation of the patch-centre or edge values. Nonetheless, microscale heterogeneous systems may be accurately simulated with this function via appropriate interpolation. Communicate patch-design variables (Section 3.15) either via the global struct `patches` or via an optional third argument (except that this last is required for parallel computing of `spmd`).

```
28 function dudt = patchSmooth3(t,u,patches)
29 if nargin<3, global patches, end
```

Input

- `u` is a vector/array of length `prod(nSubP)·nVars·nEnsem·prod(nPatch)` where there are `nVars` · `nEnsem` field values at each of the points in the `nSubP(1)`×`nSubP(2)`×`nSubP(3)`×`nPatch(1)`×`nPatch(2)`×`nPatch(3)` spatial grid.
- `t` is the current time to be passed to the user's time derivative function.
- `patches` a struct set by `configPatches3()` with the following information used here.
  - `.fun` is the name of the user's function `fun(t,u,patches)` that computes the time derivatives on the patchy lattice. The array `u` has size `nSubP(1)` × `nSubP(2)` × `nSubP(3)` × `nVars` × `nEsem` × `nPatch(1)` × `nPatch(2)` × `nPatch(3)`. Time derivatives must be computed into the same sized array, although herein the patch edge-values are overwritten by zeros.
  - `.x` is `nSubP(1)` × 1 × 1 × 1 × 1`nPatch(1)` × 1 × 1 array of the spatial locations x_i of the microscale (i, j, k)-grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.y` is similarly 1 × `nSubP(2)` × 1 × 1 × 1 × 1 × `nPatch(2)` × 1 array of the spatial locations y_j of the microscale (i, j, k)-grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.z` is similarly 1 × 1 × `nSubP(3)` × 1 × 1 × 1 × 1 × `nPatch(3)` array of the spatial locations z_k of the microscale (i, j, k)-grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.

Output

- `dudt` is a vector/array of of time derivatives, but with patch edge-values set to zero. It is of total length `prod(nSubP)` · `nVars` · `nEnsem` · `prod(nPatch)` and the same dimensions as `u`.

Sets the edge-face values from macroscale interpolation of centre-patch values, and if necessary, reshapes the fields `u` as a 8D-array. Section 3.17 describes `patchEdgeInt3()`.

```
105 %warning('patchSmooth3 starting interpolation')
106 %assert(iscodistributed(u)|~patches.parallel,'u not codist zero')
107 sizeu = size(u);
108 %warning('patchSmooth3 starting patchEdgeInt')
109 u = patchEdgeInt3(u,patches);
```

Ask the user function for the time derivatives computed in the array, overwrite its edge/face values with the dummy value of zero, then return to the user/integrator as same sized array as input.

```
119 %warning('patchSmooth3 checking on u')
120 %assert(iscodistributed(u)|~patches.parallel,'u not codist zero')
121 %warning('patchSmooth3 starting dudt function')
122 dudt = patches.fun(t,u,patches);
123 %assert(iscodistributed(dudt)|~patches.parallel,'dudt not codist one')
124 dudt([1 end],:,:,:,:,:,:) = 0;
125 dudt(:,[1 end],:,:,:,:,:) = 0;
126 dudt(:,:,[1 end],:,:,:,:) = 0;
127 %assert(iscodistributed(dudt)|~patches.parallel,'dudt not codist two')
128 dudt = reshape(dudt,sizeu);
129 %assert(iscodistributed(dudt)|~patches.parallel,'dudt not codist three')
130 %warning('patchSmooth3 finished dudt')
```

Fin.

3.17 patchEdgeInt3(): sets 3D patch face values from 3D macroscale interpolation

Couples 3D patches across 3D space by computing their face values via macroscale interpolation. Assumes that the patch centre-values are sensible macroscale variables, and patch face values are determined by macroscale interpolation of the patch centre-plane values (Roberts et al. 2014, Bunder et al. 2019), or patch next-to-face values which appears better (Bunder et al. 2020). This function is primarily used by `patchSmooth3()` but is also useful for user graphics. [8]

Communicate patch-design variables via a second argument (optional, except required for parallel computing of `spmd`) or otherwise via the global struct `patches`.

```
27 function u = patchEdgeInt3(u,patches)
28 if nargin<2, global patches, end
```

Input

- `u` is a vector/array of length `prod(nSubP)·nVars·nEnsem·prod(nPatch)` where there are `nVars` · `nEnsem` field values at each of the points in the `nSubP1` · `nSubP2` · `nSubP3` · `nPatch1` · `nPatch2` · `nPatch3` grid on the `nPatch1` · `nPatch2` · `nPatch3` array of patches.
- `patches` a struct set by `configPatches3()` which includes the following information.
  - `.x` is `nSubP1` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch1` $\times 1 \times 1$ array of the spatial locations x_{iI} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and micro-scales.
  - `.y` is similarly $1 \times$ `nSubP2` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch2` $\times 1$ array of the spatial locations y_{jJ} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and micro-scales.
  - `.z` is similarly $1 \times 1 \times$ `nSubP3` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch3` array of the spatial locations z_{kK} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and micro-scales.
  - `.ordCC` is order of interpolation, currently only $\{0, 2, 4, \ldots\}$
  - `.stag` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation.
  - `.Cwtsr` and `.Cwtsl` define the coupling coefficients for finite width interpolation in each of the x, y, z-directions.
  - `.EdgyInt` true/false is true for interpolating patch-face values from opposite next-to-face values (often preserves symmetry).
  - `.nEnsem` the number of realisations in the ensemble.

[8] Script `patchEdgeInt3test.m` verifies this code.

– `.parallel` whether serial or parallel.

Output

- `u` is 8D array, `nSubP1` · `nSubP2` · `nSubP3` · `nVars` · `nEnsem` · `nPatch1` · `nPatch2` · `nPatch3`, of the fields with face values set by interpolation (edge and corner vales set to `NaN`).

Test for reality of the field values, and define a function accordingly. Could be problematic if some variables are real and some are complex, or if variables are of quite different sizes.

```
%warning('patchedgeint3---one')
  if max(abs(imag(u(:))))<1e-9*max(abs(u(:)))
       uclean=@(u) real(u);
  else uclean=@(u) u;
  end
```

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

```
%warning('patchedgeint3---two')
[~,~,nz,~,~,~,~,Nz] = size(patches.z);
[~,ny,~,~,~,~,Ny,~] = size(patches.y);
[nx,~,~,~,~,Nx,~,~] = size(patches.x);
nEnsem = patches.nEnsem;
nVars = round( numel(u)/numel(patches.x) ...
    /numel(patches.y)/numel(patches.z)/nEnsem );
assert(numel(u) == nx*ny*nz*Nx*Ny*Nz*nVars*nEnsem ...
  ,'patchEdgeInt3: input u has wrong size for parameters')
u = reshape(u,[nx ny nz nVars nEnsem Nx Ny Nz]);
```

Get the size ratios of the patches in each direction.

```
%warning('patchedgeint3---three')
rx = patches.ratio(1);
ry = patches.ratio(2);
rz = patches.ratio(3);
```

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 six immediate neighbours.

```
I=1:Nx; Ip=mod(I,Nx)+1; Im=mod(I-2,Nx)+1;
J=1:Ny; Jp=mod(J,Ny)+1; Jm=mod(J-2,Ny)+1;
K=1:Nz; Kp=mod(K,Nz)+1; Km=mod(K-2,Nz)+1;
```

The centre of each patch (as `nx`, `ny` and `nz` are odd for centre-patch interpolation) is at indices

```
i0 = round((nx+1)/2);
j0 = round((ny+1)/2);
k0 = round((nz+1)/2);
```

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

```
178 %warning('patchedgeint3---four')
179 ordCC=patches.ordCC;
180 if ordCC>0 % then finite-width polynomial interpolation
```

The patch-edge values are either interpolated from the next-to-edge-face values, or from the centre-cross-plane values (not the patch-centre value itself as that seems to have worse properties in general). Have not yet implemented core averages??

```
190   if patches.EdgyInt % next-to-face values
191     Ux = u([2 nx-1],2:(ny-1),2:(nz-1),:,:,I,J,K);
192     Uy = u(2:(nx-1),[2 ny-1],2:(nz-1),:,:,I,J,K);
193     Uz = u(2:(nx-1),2:(ny-1),[2 nz-1],:,:,I,J,K);
194   else
195     Ux = u(i0,2:(ny-1),2:(nz-1),:,:,I,J,K);
196     Uy = u(2:(nx-1),j0,2:(nz-1),:,:,I,J,K);
197     Uz = u(2:(nx-1),2:(ny-1),k0,:,:,I,J,K);
198   end;
```

Just in case the last array dimension(s) are one, we have to force a padding of the sizes, then adjoin the extra dimension for the subsequent array of differences.

```
206 szUx0=size(Ux); szUx0=[szUx0 ones(1,8-length(szUx0)) ordCC];
207 szUy0=size(Uy); szUy0=[szUy0 ones(1,8-length(szUy0)) ordCC];
208 szUz0=size(Uz); szUz0=[szUz0 ones(1,8-length(szUz0)) ordCC];
```

Use finite difference formulas for the interpolation, so store finite differences $(\mu\delta, \delta^2, \mu\delta^3, \delta^4, \ldots)$ in these arrays. When parallel, in order to preserve the distributed array structure we use an index at the end for the differences.

```
218 %warning('patchedgeint3---five')
219   if patches.parallel
220     dmux = zeros(szUx0,patches.codist); % 9D
221     dmuy = zeros(szUy0,patches.codist); % 9D
222     dmuz = zeros(szUz0,patches.codist); % 9D
223   else
224     dmux = zeros(szUx0); % 9D
225     dmuy = zeros(szUy0); % 9D
226     dmuz = zeros(szUz0); % 9D
227   end
```

First compute differences $\mu\delta$ and δ^2 in both space directions.

```
234 %warning('patchedgeint3---six')
235   if patches.stag % use only odd numbered neighbours
236     error('polynomial interpolation not yet for staggered patch coupling')
237   else %warning('starting standard interpolation')
238     dmux(:,:,:,:,:,I,:,:,1) = (Ux(:,:,:,:,:,Ip,:,:) ...
```

```
                                -Ux(:,:,:,:,:,Im,:,:))/2; %\mu\delta
    dmux(:,:,:,:,:,I,:,:,2) = (Ux(:,:,:,:,:,Ip,:,:) ...
       -2*Ux(:,:,:,:,:,:,I,:,:) +Ux(:,:,:,:,:,:,Im,:,:));   %\delta^2
    dmuy(:,:,:,:,:,:,J,:,1) = (Uy(:,:,:,:,:,:,:,Jp,:) ...
                                -Uy(:,:,:,:,:,:,:,Jm,:))/2; %\mu\delta
    dmuy(:,:,:,:,:,:,J,:,2) = (Uy(:,:,:,:,:,:,:,Jp,:) ...
       -2*Uy(:,:,:,:,:,:,:,J,:) +Uy(:,:,:,:,:,:,:,Jm,:));   %\delta^2
    dmuz(:,:,:,:,:,:,:,K,1) = (Uz(:,:,:,:,:,:,:,:,Kp) ...
                                -Uz(:,:,:,:,:,:,:,:,Km))/2; %\mu\delta
    dmuz(:,:,:,:,:,:,:,K,2) = (Uz(:,:,:,:,:,:,:,:,Kp) ...
       -2*Uz(:,:,:,:,:,:,:,:,K) +Uz(:,:,:,:,:,:,:,:,Km));   %\delta^2
  end% if odd/even
```

Recursively take δ^2 of these to form higher order centred differences in all three space directions.

```
%warning('patchedgeint3---seven')
   for k = 3:ordCC
    dmux(:,:,:,:,:,I,:,:,k) =     dmux(:,:,:,:,:,Ip,:,:,k-2) ...
    -2*dmux(:,:,:,:,:,I,:,:,k-2) +dmux(:,:,:,:,:,Im,:,:,k-2);
    dmuy(:,:,:,:,:,:,J,:,k) =     dmuy(:,:,:,:,:,:,Jp,:,k-2) ...
    -2*dmuy(:,:,:,:,:,:,J,:,k-2) +dmuy(:,:,:,:,:,:,Jm,:,k-2);
    dmuz(:,:,:,:,:,:,:,K,k) =     dmuz(:,:,:,:,:,:,:,Kp,k-2) ...
    -2*dmuz(:,:,:,:,:,:,:,K,k-2) +dmuz(:,:,:,:,:,:,:,Km,k-2);
  end
```

Interpolate macro-values to be Dirichlet face values for each patch (Roberts & Kevrekidis 2007, Bunder et al. 2017), using the weights pre-computed by `configPatches3()`. Here interpolate to specified order.

For the case where next-to-face values interpolate to the opposite face-values: when we have an ensemble of configurations, different configurations might be coupled to each other, as specified by `patches.le`, `patches.ri`, `patches.to`, `patches.bo`, `patches.fr` and `patches.ba`.

```
%warning('patchedgeint3---eight')
k=1+patches.EdgyInt; % use centre or two faces
u(nx,2:(ny-1),2:(nz-1),:,patches.ri,I,:,:) ...
  = Ux(1,:,:,:,:,:,:,:)*(1-patches.stag) ...
  +sum( shiftdim(patches.Cwtsr(:,1),-8).*dmux(1,:,:,:,:,:,:,:,:) ,9);
u(1 ,2:(ny-1),2:(nz-1),:,patches.le,I,:,:) ...
  = Ux(k,:,:,:,:,:,:,:)*(1-patches.stag) ...
  +sum( shiftdim(patches.Cwtsl(:,1),-8).*dmux(k,:,:,:,:,:,:,:,:) ,9);
u(2:(nx-1),ny,2:(nz-1),:,patches.to,:,J,:) ...
  = Uy(:,1,:,:,:,:,:,:)*(1-patches.stag) ...
  +sum( shiftdim(patches.Cwtsr(:,2),-8).*dmuy(:,1,:,:,:,:,:,:,:) ,9);
u(2:(nx-1),1 ,2:(nz-1),:,patches.bo,:,J,:) ...
  = Uy(:,k,:,:,:,:,:,:)*(1-patches.stag) ...
  +sum( shiftdim(patches.Cwtsl(:,2),-8).*dmuy(:,k,:,:,:,:,:,:,:) ,9);
u(2:(nx-1),2:(ny-1),nz,:,patches.fr,:,:,K) ...
  = Uz(:,:,1,:,:,:,:,:)*(1-patches.stag) ...
```

```
297   +sum( shiftdim(patches.Cwtsr(:,3),-8).*dmuz(:,:,1,:,:,:,:,:,:) ,9);
298 u(2:(nx-1),2:(ny-1),1 ,:,patches.ba,:,:,K) ...
299   = Uz(:,:,k,:,:,:,:,:)*(1-patches.stag) ...
300   +sum( shiftdim(patches.Cwtsl(:,3),-8).*dmuz(:,:,k,:,:,:,:,:,:) ,9);
```

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

```
310 else% spectral interpolation
```

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 face-patch values $U_{j\pm r} = \sum_k C_k e^{ik2\pi/N(j\pm r)} = \sum_k C'_k e^{ik2\pi j/N}$ where $C'_k = C_k e^{ikr2\pi/N}$. For N patches we resolve 'wavenumbers' $|k| < N/2$, so set row vector `ks` $= k2\pi/N$ for 'wavenumbers' $k = (0, 1, \ldots, k_{\max}, -k_{\max}, \ldots, -1)$ for odd N, and $k = (0\,, 1\,, \ldots, k_{\max}\,, \pm(k_{\max}+1) - k_{\max}\,, \ldots, -1)$ for even N.

Deal with staggered grid by doubling the number of fields and halving the number of patches (`configPatches3` tests there are an even number of patches). Then the patch-ratio is effectively halved. The patch faces are near the middle of the gaps and swapped.

```
335  if patches.stag % transform by doubling the number of fields
336  error('staggered grid not yet implemented??')
337    v=nan(size(u)); % currently to restore the shape of u
338    u=cat(3,u(:,1:2:nPatch,:),u(:,2:2:nPatch,:));
339    stagShift=reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
340    iV=[nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
341    r=r/2;           % ratio effectively halved
342    nPatch=nPatch/2; % halve the number of patches
343    nVars=nVars*2;   % double the number of fields
344  else % the values for standard spectral
345     stagShift = 0;
346     iV = 1:nVars;
347  end
```

Now set wavenumbers in the three directions into three vectors at the correct dimension. In the case of even N these compute the $+$-case for the highest wavenumber zig-zag mode, $k = (0\,, 1\,, \ldots, k_{\max}\,, +(k_{\max}+1) - k_{\max}\,, \ldots, -1)$.

```
357   kMax = floor((Nx-1)/2);
358   krx = shiftdim( rx*2*pi/Nx*(mod((0:Nx-1)+kMax,Nx)-kMax) ,-4);
359   kMay = floor((Ny-1)/2);
360   kry = shiftdim( ry*2*pi/Ny*(mod((0:Ny-1)+kMay,Ny)-kMay) ,-5);
361   kMaz = floor((Nz-1)/2);
362   krz = shiftdim( rz*2*pi/Nz*(mod((0:Nz-1)+kMaz,Nz)-kMaz) ,-6);
```

Compute the Fourier transform of the patch values on the centre-planes for all the fields. Unless doing patch-edgy interpolation when FT the next-to-face values. If there are an even number of points, then if complex, treat as positive wavenumber, but if real, treat as cosine. When using an ensemble of configurations, different configurations might be coupled to each other, as

specified by `patches.le`, `patches.ri`, `patches.to`, `patches.bo`, `patches.fr` and `patches.ba`.

```
% indices of interior
ix=(2:nx-1)';  iy=2:ny-1;  iz=shiftdim(2:nz-1,-1);
if ~patches.EdgyInt
     Cle = fft(fft(fft( u(i0,iy,iz,:,:,:,:,:) ...
         ,[],6),[],7),[],8);
     Cbo = fft(fft(fft( u(ix,j0,iz,:,:,:,:,:) ...
         ,[],6),[],7),[],8);
     Cba = fft(fft(fft( u(ix,iy,k0,:,:,:,:,:) ...
         ,[],6),[],7),[],8);
     Cri = Cle;    Cto = Cbo;    Cfr = Cba;
else
     Cle = fft(fft(fft( u(   2,iy,iz ,:,patches.le,:,:,:) ...
         ,[],6),[],7),[],8);
     Cri = fft(fft(fft( u(nx-1,iy,iz ,:,patches.ri,:,:,:) ...
         ,[],6),[],7),[],8);
     Cbo = fft(fft(fft( u(ix,2   ,iz ,:,patches.bo,:,:,:) ...
         ,[],6),[],7),[],8);
     Cto = fft(fft(fft( u(ix,ny-1,iz ,:,patches.to,:,:,:) ...
         ,[],6),[],7),[],8);
     Cba = fft(fft(fft( u(ix,iy,2    ,:,patches.ba,:,:,:) ...
         ,[],6),[],7),[],8);
     Cfr = fft(fft(fft( u(ix,iy,nz-1 ,:,patches.fr,:,:,:) ...
         ,[],6),[],7),[],8);
end
```

Now invert the triple Fourier transforms to complete interpolation. (Should stagShift be multiplied by rx/ry/rz??) Enforce reality when appropriate.

```
u(nx,iy,iz,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
    Cle.*exp(1i*(stagShift+krx))  ,[],6),[],7),[],8) );
u( 1,iy,iz,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
    Cri.*exp(1i*(stagShift-krx))  ,[],6),[],7),[],8) );
u(ix,ny,iz,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
    Cbo.*exp(1i*(stagShift+kry))  ,[],6),[],7),[],8) );
u(ix, 1,iz,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
    Cto.*exp(1i*(stagShift-kry))  ,[],6),[],7),[],8) );
u(ix,iy,nz,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
    Cba.*exp(1i*(stagShift+krz))  ,[],6),[],7),[],8) );
u(ix,iy, 1,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
    Cfr.*exp(1i*(stagShift-krz))  ,[],6),[],7),[],8) );
end% if spectral
```

Nan the values in corners and edges, of every 3D patch.

```
%warning('patchedgeint3---nine')
u(:,[1 ny],[1 nz],:,:,:,:,:)=nan;
u([1 nx],:,[1 nz],:,:,:,:,:)=nan;
u([1 nx],[1 ny],:,:,:,:,:,:)=nan;
```

```
449 end% function patchEdgeInt3
```

Fin, returning the 8D array of field values with interpolated faces.

3.18 homoDiffEdgy3: computational homogenisation of a 3D diffusion via simulation on small patches

Simulate heterogeneous diffusion in 3D space on 3D patches as an example application. Then compute macroscale eigenvalues of the patch scheme applied to this heterogeneous diffusion to validate and to compare various orders of inter-patch interpolation.

This code extends to 3D the 2D code discussed in Section 3.14. First set random heterogeneous diffusivities of random (small) period in each of the three directions. Crudely normalise by the harmonic mean so the decay time scale is roughly one.

```
mPeriod = randi([2 3],1,3)
cHetr = exp(0.3*randn([mPeriod 3]));
cHetr = cHetr*mean(1./cHetr(:))
```

Configure the patch scheme with some arbitrary choices of domain, patches, size ratios. Use spectral interpolation as we test other orders subsequently. In 3D we appear to get only real eigenvalues by using edgy interpolation. What happens for non-edgy interpolation is unknown.

```
nSubP=mPeriod+2;
nPatch=[5 5 5];
configPatches3(@heteroDiff3, [-pi pi], nan, nPatch ...
    ,0, 0.3, nSubP, 'EdgyInt',true  ...
    ,'hetCoeffs',cHetr );
```

3.18.1 Simulate heterogeneous diffusion

Set initial conditions of a simulation as shown in Figure 3.22.

```
global patches
u0 = exp(-patches.x.^2/4-patches.y.^2/2-patches.z.^2);
u0 = u0.*(1+0.3*rand(size(u0)));
```

Integrate using standard integrators, unevenly spaced in time to better display transients.

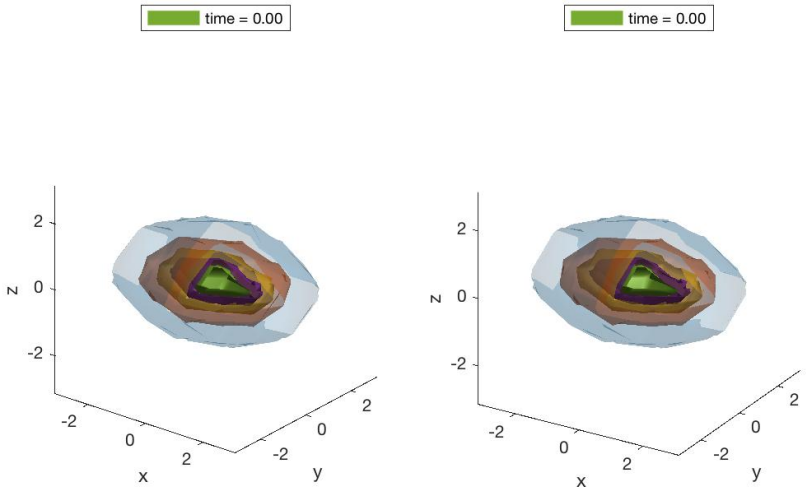
```
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@patchSmooth3, 0.3*linspace(0,1,50).^2, u0(:));
else % octave version
    [ts,us] = odeOcts(@patchSmooth3, 0.3*linspace(0,1).^2, u0(:));
end
```

Plot the solution as an animation over time.

```
figure(1), clf
rgb=get(gca,'defaultAxesColorOrder');
colormap(0.8*hsv)
```

Get spatial coordinates of patch interiors.

Figure 3.22: initial field $u(x,y,z,0)$ of the patch scheme applied to a heterogeneous diffusion PDE. *Plotted are the isosurfaces at field values $u = 0.1, 0.3, \ldots, 0.9$, with the front quadrant omitted so you can see inside. Figure 3.23 plots the isosurfaces of the computed field at time $t = 0.3$.*




```
96  x = reshape( patches.x([2:end-1],:,:,:) ,[],1);
97  y = reshape( patches.y(:,[2:end-1],:,:) ,[],1);
98  z = reshape( patches.z(:,:,[2:end-1],:) ,[],1);
```

For every time step draw the surface and pause for a short display.

```
105 for i = 1:length(ts)
```

Get the row vector of data, form into a 6D array, then omit patch faces, and reshape to suit the isosurface function. We do not use interpolation to get face values as the interpolation omits the corner edges and so breaks up the isosurfaces.

```
115   u = reshape( us(i,:) ,[nSubP nPatch]);
116   u = u([2:end-1],[2:end-1],[2:end-1],:,:,:);
117   u = reshape( permute(u,[1 4 2 5 3 6]) ...
118        , [numel(x) numel(y) numel(z)]);
```

Optionally cut-out the front corner so we can see inside.

```
124   u( (x>0) & (y'<0) & (shiftdim(z,-2)>0) ) = nan;
```

The `isosurface` function requires us to transpose x and y.

```
131 v = permute(u,[2 1 3]);
```

Draw cross-eyed stereo view of some isosurfaces.

```
137     clf;
138     for p=1:2
139       subplot(1,2,p)
140       for iso=5:-1:1
141          isov=(iso-0.5)/5;
142          hsurf(iso) = patch(isosurface(x,y,z,v,isov));
143          isonormals(x,y,z,v,hsurf(iso))
144          set(hsurf(iso) ,'FaceColor',rgb(iso,:) ...
145              ,'EdgeColor','none' ...
146              ,'FaceAlpha',iso/5);
147          hold on
148       end
149       axis equal, view(45-7*p,25)
150       axis(pi*[-1 1 -1 1 -1 1])
151       xlabel('x'), ylabel('y'), zlabel('z')
152       legend(['time = ' num2str(ts(i),'%4.2f')],'Location','north')
153       camlight, lighting gouraud
154       hold off
155     end% each p
156     if i==1 % pause for the viewer
157          makeJpeg=false;
158          if makeJpeg, print(['Figs/' mfilename 't0'],'-djpeg'), end
159          disp('Press any key to start animation of isosurfaces')
160          pause
161     else pause(0.05)
162     end
```

Finish the animation loop, and optionally output the isosurfaces of the final field, Figure 3.23.

```
178 end%for over time
179 if makeJpeg, print(['Figs/' mfilename 'tFin'],'-djpeg'), end
```

3.18.2 Compute Jacobian and its spectrum

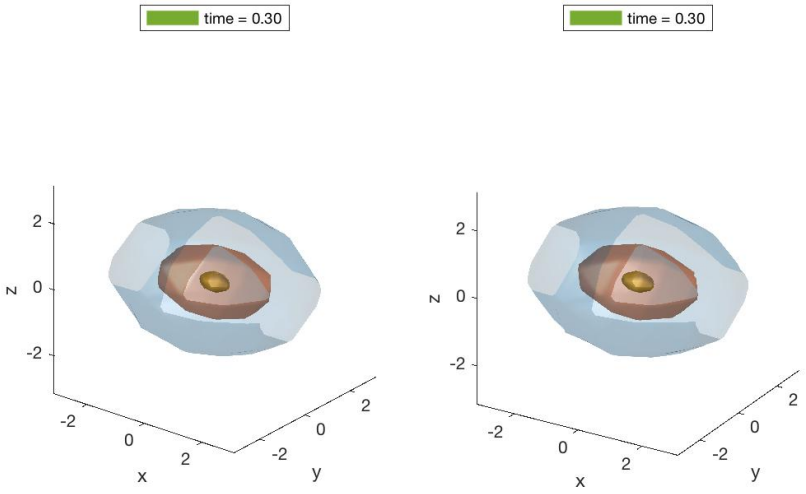
Let's explore the Jacobian dynamics for a range of orders of interpolation, all for the same random patch design and heterogeneity. Except here use a small ratio as we do not plot and then the scale separation is clearest.

```
195 ratio = 0.025*(1+rand(1,3))
196 nSubP=randi([3 5],1,3)
197 nPatch=[3 3 3]
198 nEnsem = prod(mPeriod) % or just set one
```

Find which elements of the 8D array are interior micro-grid points and hence correspond to dynamical variables.

```
205       u0 = zeros([nSubP,1,nEnsem,nPatch]);
206       u0([1 end],:,:,:) = nan;
207       u0(:,[1 end],:,:) = nan;
208       u0(:,:,[1 end],:) = nan;
```

Figure 3.23: final field $u(x, y, z, 0.3)$ of the patch scheme applied to a heterogeneous diffusion PDE. *Plotted are the isosurfaces at field values $u = 0.1, 0.3, \ldots, 0.9$, with the front quadrant omitted so you can see inside.*




```
209     i = find(~isnan(u0));
210     sizeJacobian = length(i)
211     assert(sizeJacobian<4000 ...
212         ,'Jacobian is too big to quickly generate and analyse')
```

Store this many eigenvalues in array across different orders of interpolation.

```
219 nLeadEvals=prod(nPatch)+max(nPatch);
220 leadingEvals=[];
```

Evaluate eigenvalues for spectral as the base case for polynomial interpolation of order $2, 4, \ldots$.

```
228 maxords=6;
229 for ord=0:2:maxords
230     ord=ord
```

Configure with same heterogeneity.

```
236     configPatches3(@heteroDiff3,[-pi pi],nan,nPatch ...
237         ,ord,ratio,nSubP,'EdgyInt',true,'nEnsem',nEnsem ...
238         ,'hetCoeffs',cHetr);
```

Construct the Jacobian of the scheme as the matrix of the linear transformation, obtained by transforming the standard unit vectors.

```
246     jac = nan(length(i));
247     for j = 1:length(i)
```

```
248         u = u0(:)+(i(j)==(1:numel(u0))');
249         tmp = patchSmooth3(0,u);
250         jac(:,j) = tmp(i);
251       end
```

Test for symmetry, with error if we know it should be symmetric.

```
258       notSymmetric=norm(jac-jac')
259   %    if notSymmetric>1e-7, spy(abs(jac-jac')>1e-7), end%??
260       assert(notSymmetric<1e-7,'failed symmetry')
```

Find all the eigenvalues (as `eigs` is unreliable), and put eigenvalues in a vector.

```
267       [evecs,evals] = eig((jac+jac')/2,'vector');
268       biggestImag=max(abs(imag(evals)));
269       if biggestImag>0, biggestImag=biggestImag, end
```

Sort eigenvalues on their real-part with most positive first, and most negative last. Store the leading eigenvalues in `egs`, and write out when computed all orders. The number of zero eigenvalues, `nZeroEv`, gives the number of decoupled systems in this patch configuration.

```
279       [~,k] = sort(-real(evals));
280       evals=evals(k); evecs=evecs(:,k);
281       if ord==0, nZeroEv=sum(abs(evals(:))<1e-5), end
282       if ord==0, evec0=evecs(:,1:nZeroEv*nLeadEvals);
283       else % find evec closest to that of each leading spectral
284           [~,k]=max(abs(evecs'*evec0));
285           evals=evals(k); % re-sort in corresponding order
286       end
287       leadingEvals=[leadingEvals evals(nZeroEv*(1:nLeadEvals))];
288   end
289   disp('     spectral    quadratic      quartic  sixth-order ...')
290   leadingEvals=leadingEvals
```

3.18.3 `heteroDiff3()`: heterogeneous diffusion

This function codes the lattice heterogeneous diffusion inside the patches. For 8D input array `u` (via edge-value interpolation of `patchEdgeInt3`, such as by `patchSmooth3`, Section 3.16), computes the time derivative (3.1) at each point in the interior of a patch, output in `ut`. The three 3D array of diffusivities, c^x_{ijk}, c^y_{ijk} and c^z_{ijk}, have previously been stored in `patches.cs` (4+D).

Supply patch information as a third argument (required by parallel computation), or otherwise by a global variable.

```
23  function ut = heteroDiff3(t,u,patches)
24    if nargin<3, global patches, end
```

Microscale space-steps. Q: is using `i,j,k` slower than `2:end-1`??

```
31    dx = diff(patches.x(2:3));  % x micro-scale step
32    dy = diff(patches.y(2:3));  % y micro-scale step
33    dz = diff(patches.z(2:3));  % z micro-scale step
34    i = 2:size(u,1)-1; % x interior points in a patch
35    j = 2:size(u,2)-1; % y interior points in a patch
36    k = 2:size(u,3)-1; % y interior points in a patch
```

Reserve storage and then assign interior patch values to the heterogeneous diffusion time derivatives. Using `nan+u` appears quicker than `nan(size(u),patches.codist)`

```
44    ut = nan+u; % reserve storage
45    ut(i,j,k,:,:,:,:,:) ...
46    = diff(patches.cs(:,j,k,1,:).*diff(u(:,j,k,:,:,:,:,:),1),1)/dx^2 ...
47     +diff(patches.cs(i,:,k,2,:).*diff(u(i,:,k,:,:,:,:,:),1,2),1,2)/dy^2 ...
48     +diff(patches.cs(i,j,:,3,:).*diff(u(i,j,:,:,:,:,:,:),1,3),1,3)/dz^2;
49  end% function
```

Fin.

3.19 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.
- Core averages code.
- Adapt to maps in micro-time? Surely easy, just an example.

3.20 Miscellaneous tests

3.20.1 `patchEdgeInt1test`: test the 1D patch coupling

A script to test the spectral and finite-order polynomial interpolation of function `patchEdgeInt1()`. Tests one or several variables, normal and staggered grids, and also tests centre and edge interpolation. But does not yet test core averaging, etc.

Start by establishing global data struct for the range of various cases.

```
clear all
global patches
nSubP=5
```

Test standard spectral interpolation

Test over various numbers of patches, random domain lengths and random ratios. Say do three realisations for each number of patches.

```
nReal=3
for nPatch=repmat(5:10,1,nReal)
nPatch=nPatch
Len=10*rand
ratio=0.5*rand
configPatches1(@sin,[0,Len],nan,nPatch,0,ratio,nSubP ...
    ,'EdgyInt',rand<0.5); % random Edgy or not
if mod(nPatch,2)==0, disp('Avoiding highest wavenumber'), end
kMax=floor((nPatch-1)/2);
if patches.EdgyInt, i0=[2 nSubP-1], else i0=(nSubP+1)/2, end
```

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=rms(ui(:)-u0(:));
  if abs(normError)>5e-14
    normError=normError, k=k
    error(['failed single var interpolation k=' num2str(k)])
  end
end
```

Test multiple fields Use this to measure some of the errors in order to omit singleton dimensions, and also squish any errors if the second argument is essential zero (to cater for cosine aliasing errors).

```
  normDiff=@(u,v) ...
  norm(squeeze(u)-squeeze(v))*norm(squeeze(v(i0,:,:,:)));
```

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 by multiplying by result norm. Not yet working for edgy interpolation.

```
for k=1:(nPatch-1)/2 % not checking the highest wavenumber
  u0=sin(k*patches.x*2*pi/Len);
  v0=cos(k*patches.x*2*pi/Len);
  uvi=patchEdgeInt1( reshape([u0 v0],[],1) );
  normuError=normDiff(uvi(:,1,:,:),u0);
  normvError=normDiff(uvi(:,2,:,:),v0);
  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
disp('Passed standard spectral interpolation tests')
```

Now test spectral interpolation on staggered grid

Must have even number of patches for a staggered grid.

```
disp('*** spectral interpolation on staggered grid')
for nPatch=repmat(6:2:20,1,nReal)
nPatch=nPatch
ratio=0.5*rand
nSubP=7; % of form 4*N-1
Len=10*rand
configPatches1(@simpleWavepde,[0,Len],nan,nPatch,-1,ratio,nSubP ...
    ,'EdgyInt',rand<0.5);
if mod(nPatch,4)==0, disp('Avoiding highest wavenumber'), end
kMax=floor((nPatch/2-1)/2)
if patches.EdgyInt, i0=[2 nSubP-1], else i0=(nSubP+1)/2, end%??
```

Identify which microscale grid points are h or u values.

```
uPts=mod( (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('Staggered: 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(:));
```

```
155   if abs(normError)>5e-14
156     normError=normError
157     patches=patches
158     error(['staggered: failed single sys interpolation k=' num2str(k)])
159   end
160 end
```

Test multiple fields Use this to measure some of the errors in order to omit singleton dimensions, and also squish any errors if the third argument is essential zero (to cater for cosine aliasing errors).

```
171   normDiff=@(u,v,w) ...
172   norm(squeeze(u)-squeeze(v))*norm(squeeze(w(i0,:,:,:)));
```

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.

```
182 fprintf('Staggered: Two field-pairs test.\n')
183 x0=patches.x((nSubP+1)/2,1);
184 patches.x=patches.x-x0;
185 oddP=1:2:nPatch; evnP=2:2:nPatch;
186 for k=1:kMax
187   U0=nan(nSubP,1,1,nPatch); V0=U0;
188   U0(hPts)=rand*sin(k*patches.x(hPts)*2*pi/Len);
189   U0(uPts)=rand*sin(k*patches.x(uPts)*2*pi/Len);
190   V0(hPts)=rand*cos(k*patches.x(hPts)*2*pi/Len);
191   V0(uPts)=rand*cos(k*patches.x(uPts)*2*pi/Len);
192   UVi=patchEdgeInt1([U0 V0]);
193 %  U0i0odd=squeeze(U0(i0,:,:,oddP)), U0i0evn=squeeze(U0(i0,:,:,evnP))
194 %  V0i0odd=squeeze(V0(i0,:,:,oddP)), V0i0evn=squeeze(V0(i0,:,:,evnP))
195   normuError=[normDiff(UVi(:,1,:,oddP),U0(:,:,:,oddP),U0(:,:,:,evnP))
196       normDiff(UVi(:,1,:,evnP),U0(:,:,:,evnP),U0(:,:,:,oddP))]';
197   normvError=[normDiff(UVi(:,2,:,oddP),V0(:,:,:,oddP),V0(:,:,:,evnP))
198       normDiff(UVi(:,2,:,evnP),V0(:,:,:,evnP),V0(:,:,:,oddP))]';
199   if norm(normuError)+norm(normvError)>2e-13
200     normuError=normuError, normvError=normvError
201 %    [U0 UVi(:,1,:,:) V0 UVi(:,2,:,:)]
202     patches=patches
203     error(['staggered: failed double field interpolation k=' num2str(k)])
204   end
205 end
```

End for-loop over patches

```
212 end
```

Check standard finite width interpolation

Check over various types and orders of interpolation, numbers of patches, random domain lengths and random ratios. (The `@sin` is a dummy.)

```
for edgyInt=[mod(nSubP,2)==0 true]
for ordCC=2:2:8
for nPatch=ordCC+(2:4)
    edgyInt=edgyInt
    ordCC=ordCC
    nPatch=nPatch
    Domain=5*[-rand rand]
    ratio=0.5*rand
    configPatches1(@sin,Domain,nan,nPatch,ordCC,ratio,nSubP ...
        ,'EdgyInt',edgyInt);
```

Check multiple fields simultaneously Set profiles to be various powers of x, `ps`, and store as different 'variables' at each point.

```
    ps=1:ordCC
    cs=randn(size(ps))
    u0=patches.x.^ps.*cs+randn;
```

Then evaluate the interpolation and squeeze the singleton dimension of an 'ensemble'.

```
    ui=patchEdgeInt1(u0(:));
    ui=squeeze(ui);
```

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
disp('Passed all standard polynomial interpolation')
```

3.20.2 `patchEdgeInt2test`: tests 2D patch coupling

A script to test the spectral and finite-order polynomial interpolation of function `patchEdgeInt2()`. Tests one or several variables, normal grids, and also tests centre and edge interpolation. But does not yet test staggered grids, core averaging, etc.

Start by establishing global data struct for the range of various cases.

```
clear all, close all
global patches
```

Test standard spectral interpolation

Test over various numbers of patches, random domain lengths and random ratios. Try 99 realisations of random tests.

```
for realisation=1:99
```

Choose and configure random sized domains, random sub-patch resolution, random size-ratios, random number of periodic-patches, randomly edge or mid-patch interpolation.

```
Lx = 1+3*rand, Ly = 1+3*rand
nSubP = 1+2*randi(3,1,2)
ratios = rand(1,2)/2
nPatch = randi([3 6],1,2)
edgyInt = (rand>0.5)
configPatches2(@sin,[0 Lx 0 Ly],nan,nPatch,0,ratios,nSubP ...
    ,'EdgyInt',edgyInt)
```

Choose a random number of fields, then generate trigonometric shape with random wavenumber and random phase shift. But if an even number of patches in either direction, then do not test the highest wavenumber because of aliasing problem.

```
nV=randi(3)
[nx,Nx]=size(squeeze(patches.x));
[ny,Ny]=size(squeeze(patches.y));
u0=nan(nx,ny,nV,1,Nx,Ny);
for iV=1:nV
  kx=randi([0 floor((nPatch(1)-1)/2)])
  ky=randi([0 floor((nPatch(2)-1)/2)])
  phix=pi*rand*(2*kx~=nPatch(1))
  phiy=pi*rand*(2*ky~=nPatch(2))
  % generate 6D array via auto-replication
  u0(:,:,iV,1,:,:)=sin(2*pi*kx*patches.x/Lx+phix) ...
                  .*sin(2*pi*ky*patches.y/Ly+phiy);
end
```

Copy and NaN the edges, then interpolate

```
u=u0; u([1 end],:,:)=nan; u(:,[1 end],:)=nan;
u=patchEdgeInt2(u(:));
```

Compute difference, ignoring the nans which should only be in the corners. If there is an error in the interpolation, then abort the script for checking: please record parameter values and inform us.

```
err=u-u0;
normerr=norm(err(~isnan(err)))
assert(normerr<1e-12, '2D interpolation failed')
```

End the for-loop over realisations

```
92 disp('***** This test passed')
93 end
```

Check standard finite width interpolation

Check over various types and orders of interpolation, numbers of patches, random domain lengths and random ratios. (The `@sin` is a dummy.)

```
111 for ordCC=2:2:8
112 for realisations=1:9
113     ordCC=ordCC
114     nPatch=ordCC+randi([2 4],1,2)
115     edgyInt = (rand>0.5)
116     nSubP = 1+2*randi(3,1,2)
117     Domain=5*[-rand rand -rand rand]
118     ratios=0.5*rand(1,2)
119     configPatches2(@sin,Domain,nan,nPatch,ordCC,ratios,nSubP ...
120         ,'EdgyInt',edgyInt);
```

Check multiple fields simultaneously Set profiles to be various powers of x, `ps`, and store as different 'variables' at each point.

```
129     [ps,qs]=meshgrid(0:ordCC);
130     ps=reshape(ps,1,1,[]); qs=reshape(qs,1,1,[]);
131     u0=ones(size(ps)).*patches.x.^ps.*patches.y.^qs;
```

Then evaluate the interpolation.

```
137     ui=patchEdgeInt2(u0(:));
```

The interior patches should have zero error. Appear to need error tolerance of 10^{-8} because of the size of the domain and the high order of interpolation.

```
145     I=ordCC/2+1:nPatch(1)-ordCC/2;
146     J=ordCC/2+1:nPatch(2)-ordCC/2;
147     iError=ui(:,:,:,:,I,J)-u0(:,:,:,:,I,J);
148     normError=norm(iError(~isnan(iError)))
149     assert(normError<5e-8 ...
150     ,'failed finite stencil interpolation')
```

End the for-loops over various parameters.

```
157 end,end %order and realisations
158 disp('Passed all standard polynomial interpolation')
```

Finished

If no error messages, then all OK.

```
172 disp('**** If you read this, then all the tests were successful')
```

3.20.3 patchEdgeInt3test: tests 3D patch coupling

A script to test the spectral and finite-order polynomial interpolation of function `patchEdgeInt3()`. Tests one or several variables, normal grids, and also tests centre and edge interpolation. But does not yet test staggered grids, core averaging, etc.

Start by establishing global data struct for the range of various cases.

```
clear all, close all
global patches
```

Test standard spectral interpolation

Test over various numbers of patches, random domain lengths and random ratios. Try 99 realisations of random tests.

```
for realisation=1:99
```

Choose and configure random sized domains, random sub-patch resolution, random size-ratios, random number of periodic-patches, randomly edge or mid-patch interpolation.

```
Lx = 1+3*rand, Ly = 1+3*rand, Lz = 1+3*rand
nSubP = 1+2*randi(3,1,3)
ratios = 0.5*rand(1,3)
nPatch = randi([3 6],1,3)
edgyInt = (rand>0.5)
configPatches3(@sin,[0 Lx 0 Ly 0 Lz],nan,nPatch,0,ratios,nSubP ...
    ,'EdgyInt',edgyInt)
```

Choose a random number of fields, then generate trigonometric shape with random wavenumber and random phase shift. But if an even number of patches in either direction, then do not test the highest wavenumber because of aliasing problem.

```
nV=randi(3)
[nx,Nx]=size(squeeze(patches.x));
[ny,Ny]=size(squeeze(patches.y));
[nz,Nz]=size(squeeze(patches.z));
u0=nan(nx,ny,nz,nV,1,Nx,Ny,Nz);
for iV=1:nV
  ks=floor( rand(1,3).*floor(([Nx Ny Nz]+1)/2) )
  phis = pi*rand(1,3).*(2*ks~=[Nx Ny Nz])
  % generate 8D array via auto-replication
  u0(:,:,:,iV,1,:,:,:)=sin(2*pi*ks(1)*patches.x/Lx+phis(1)) ...
                      .*sin(2*pi*ks(2)*patches.y/Ly+phis(2)) ...
                      .*sin(2*pi*ks(3)*patches.z/Lz+phis(3));
end
```

Copy and NaN the edges, then interpolate

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

```
75  u=patchEdgeInt3(u(:));
```

Compute difference, ignoring the nans which should only be in the corners. If there is an error in the interpolation, then abort the script for checking: please record parameter values and inform us.

```
84  err=u-u0;
85  normerr=norm(err(~isnan(err)))
86  assert(normerr<1e-12, '3D interpolation failed')
```

End the for-loop over realisations

```
93  disp('***** This test passed')
94  end
95  disp('If you read this, then all spectral tests passed')
```

Check polynomial finite width interpolation

Check over various types and orders of interpolation, numbers of patches, random domain lengths and random ratios. (The `@sin` is a dummy.)

```
113 for ordCC=2:2:6
114 for realisations=1:9
115     ordCC=ordCC
116     nPatch=ordCC+randi([1 4],1,3)
117     edgyInt = (rand>0.5)
118     nSubP = 1+2*randi(3,1,3)
119     Domain=5*[-rand rand -rand rand -rand rand]
120     ratios=0.5*rand(1,3)
121     configPatches3(@sin,Domain,nan,nPatch,ordCC,ratios,nSubP ...
122         ,'EdgyInt',edgyInt);
```

Check multiple fields simultaneously Set profiles to be various powers of x, y, z, namely `ps`, `qs`, `rs`, and store as different 'variables' at each point.

```
132     [ps,qs,rs]=meshgrid(0:ordCC);
133     ps=reshape(ps,1,1,1,[]);
134     qs=reshape(qs,1,1,1,[]);
135     rs=reshape(rs,1,1,1,[]);
136     u0=ones(size(ps)).*patches.x.^ps ...
137         .*patches.y.^qs.*patches.z.^rs;
```

Then evaluate the interpolation.

```
143     ui=patchEdgeInt3(u0(:));
```

The interior patches should have zero error. Appear to need error tolerance of 10^{-8} because of the size of the domain and the high order of interpolation.

```
151     I=ordCC/2+1:nPatch(1)-ordCC/2;
152     J=ordCC/2+1:nPatch(2)-ordCC/2;
153     K=ordCC/2+1:nPatch(3)-ordCC/2;
154     iError=ui(:,:,:,:,:,I,J,K)-u0(:,:,:,:,:,I,J,K);
155     normError=norm(iError(~isnan(iError)))
```

```
156     assert(normError<5e-8 ...
157     ,'failed finite stencil polynomial interpolation')
```

End the for-loops over various parameters.

```
164 end,end %order and realisations
165 disp('Passed all polynomial interpolation tests')
```

Finished

If no error messages, then all OK.

```
179 disp('**** If you read this, then all the tests were successful')
```

4 Matlab parallel computation of the patch scheme

Chapter contents

Get familiar with the patch scheme of Chapter 3.

For large-scale simulations, we here assume you have a compute cluster with many independent computer processors linked by a high-speed network. The functions we provide here aims to distribute computations in parallel across the cluster. MATLAB's Parallel Computing Toolbox empowers a reasonably straightforward way to implement this.

The examples listed herein are all *Proof of Principle*: as coded they are all small enough that non-parallel execution is here actually much quicker that

the parallel execution.[1] One needs significantly larger and/or more detailed problems than these examples before parallel execution is effective.

The patch scheme has a clear domain decomposition of assigning relatively few patches to each processor.

As in all parallel cluster computing, interprocessor communication time all to often dominates. It is important to reduce communication as much as possible compared to computation. Consequently, parallel computing is only effective when there is a very large amount of microscale computation done on each processor per communication.

To minimise communication in time-dependent problems we have drafted a special integrator `RK2mesoPatch`, Section 4.3, that communicates between patches only on a meso-time.

[1] Although I have so far only tested with two parallel workers??

4.1 `chanDispSpmd`: simulation of a 1D shear dispersion via simulation on small patches across a channel

Section contents

Simulate shear dispersion along 1D channel emergent from micro-scale dynamics in 2D space, via 1D patches as a Proof of Principle example of parallel computing with `spmd`. In this shear dispersion, although the micro-scale diffusivities are one-ish, the shear causes an effective longitudinal 'diffusivity' of the order of Pe^2 —which is typically much larger than the micro-scale diffusivity (Taylor 1953, e.g.).

The spatial domain is the channel (large) L-periodic in x and $|y| < 1$. Seek to predict a concentration field $c(x, y, t)$ satisfying the linear advection-diffusion PDE

$$\frac{\partial c}{\partial t} = -\operatorname{Pe} u(y)\frac{\partial c}{\partial x} + \frac{\partial}{\partial x}\Big[\kappa_x(y)\frac{\partial c}{\partial x}\Big] + \frac{\partial}{\partial y}\Big[\kappa_y(y)\frac{\partial c}{\partial y}\Big]. \tag{4.1}$$

where Pe denotes a Peclet number, parabolic advection velocity $u(y) = \frac{3}{2}(1-y^2)$ with noise, and parabolic cross-channel diffusivity $\kappa(y) = (1-y^2)$ with noise. The noise is to be multiplicative and log-normal to ensure advection and diffusion are all positive, and to be periodic in x.

For a microscale computation we discretise in space with x-spacing δx, and n_y points over $|y| < 1$ with spacing $\delta y := 2/n_y$ at $y_j := -1 + (j - \frac{1}{2})\delta y$, $j = 1 : n_y$. A microscale discretisation of PDE (4.1) is then

$$\frac{\partial c_{ij}}{\partial t} = -\operatorname{Pe} u(y_j)\frac{c_{i+1,j} - c_{i-1,j}}{2\delta x} + \frac{d_{i,j+1/2} - d_{i,j-1/2}}{\delta y} + \frac{D_{i+1/2,j} - D_{i-1/2,j}}{\delta x},$$
$$d_{ij} := \kappa_y(y_j)\frac{c_{i,j+1/2} - c_{i,j-1/2}}{\delta y}, \quad D_{ij} := \kappa_x(y_j)\frac{c_{i+1/2,j} - c_{i-1/2,j}}{\delta x}. \tag{4.2}$$

These are coded in Section 4.1.4 for the computation.

Choose one of four cases:

- `theCase=1` is corresponding code without parallelisation (in this toy problem it is much the quickest because there is no expensive communication);
- `theCase=2` illustrates that `RK2mesoPatch` invokes `spmd` computation if parallel has been configured.
- `theCase=3` shows how users explicitly invoke `spmd`-blocks around the time integration.

- `theCase=4` invokes projective integration for long-time simulation via short bursts of the micro-computation, bursts done within `spmd`-blocks for parallel computing.

First, clear all to remove any existing globals, old composites, etc—although a parallel pool persists. Then choose the case.

```
clear all
theCase = 2
```

The micro-scale PDE is evaluated at positions y_j across the channel, $|y| < 1$. The even indexed points are the collocation points for the PDE, whereas the even indexed points are the half-grid points for specification of y-diffusivities.

```
ny = 7
y = linspace(-1,1,2*ny+1);
yj = y(2:2:end);
```

Set micro-scale advection (array 1) and diffusivity (array 2) with (roughly) parabolic shape (Watt & Roberts 1995, MacKenzie & Roberts 2003, e.g.). Here modify the parabola by a heterogeneous log-normal factor with specified period along the channel: modify the strength of the heterogeneity by the coefficient of `randn` from zero to perhaps one: coefficient 0.3 appears a good moderate value. Remember that `configPatches1` reshapes `cHetr` to 2D.

```
mPeriod = 4
cHetr = shiftdim([3/2 1],-1).*(1-y.^2) ...
        .*exp(0.3*randn([mPeriod 2*ny+1 2]));
```

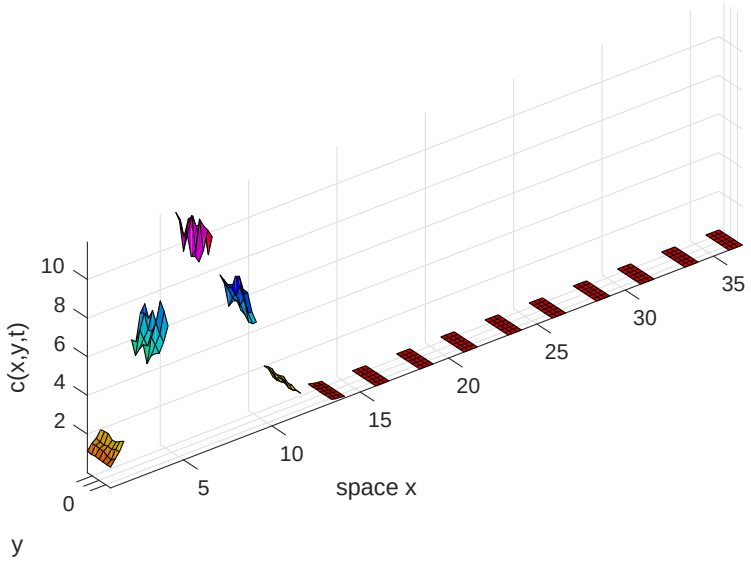
Configure the patch scheme with some arbitrary choices of domain, patches, size ratios—here each patch is a unit cube in space. Set `patches` information to be global so the info can be used for Cases 1–2 without being explicitly passed as arguments. Choose the parallel option if not Case 1, which invokes `spmd`-block internally, so that field variables become *distributed* across cpus.

```
if theCase<=2, global patches, end
nPatch=15
nSubP=2+mPeriod
ratio=0.2+0.2*(theCase<4)
Len=nPatch/ratio
disp('**** Setting configPatches1')
patches = configPatches1(@chanDispMicro, [0 Len], nan ...
    , nPatch, 0, ratio, nSubP, 'EdgyInt',true  ...
    ,'hetCoeffs',cHetr ,'parallel',(theCase>1) );
```

When using parallel then additional parameters to `patches` should be set within a `spmd` block (because `patches` is a co-distributed structure).

```
Peclet = 10
if theCase==1, patches.Pe = Peclet;
else     spmd, patches.Pe = Peclet; end
end
```

Figure 4.1: initial field $u(x, y, 0)$ of the patch scheme applied to a heterogeneous advection-diffusion PDE. *Figure 4.2 plots the roughly smooth field values at time $t = 4$. In this example the patches are relatively large, ratio 0.4, for visibility.*



4.1.1 Simulate heterogeneous advection-diffusion

Set initial conditions of a simulation as shown in Figure 4.1.

```
147 disp('**** Set initial condition and test dc0dt =')
148 if theCase==1
```

Without parallel processing, invoke the usual operations.

```
154     c0 = 10*exp(-(ratio*patches.x-2.5).^2/2) +0*yj;
155     c0 = c0.*(1+0.2*rand(size(c0)));
156     dc0dt = patchSmooth1(0,c0);
```

With parallel, we must use an `spmd`-block for computations: there is no difference in cases 2–4 here. Also, we must sometimes explicitly code how to distribute some new arrays over the cpus. Now `patchSmooth1` does not invoke `spmd` so higher level code must, as here. Even if `patches` is global, inside `spmd`-block we *must* pass it explicitly as a parameter to `patchSmooth1`.

```
169 else, spmd
170     c0 = 10*exp(-(ratio*patches.x-2.5).^2/2) +0*yj;
171     c0 = c0.*(1+0.2*rand(size(c0),patches.codist));
172     dc0dt = patchSmooth1(0,c0,patches)
173     end%spmd
174 end%if theCase
```

Integrate in time, either via the automatic `ode23` or via `RK2mesoPatch` which reduces communication between patches. By default, `RK2mesoPatch` does ten micro-steps for each specified meso-step in `ts`. For stability: with noise up to 0.3, need micro-steps less than 0.005; with noise 1, need micro-steps less

than 0.0015.

```
196 warning('Integrating system in time, wait patiently')
197 ts=4*linspace(0,1);
```

Go to the selected case.

```
203 switch theCase
```

1. For non-parallel, we could use `RK2mesoPatch` as indicated below, but instead choose to use standard `ode23` as here `patchSmooth1` accesses patch information via global `patches`. For post-processing, reshape each and every row of the computed solution to the correct array size—namely that of the initial condition.

```
215 case 1
216 %    [cs,uerrs] = RK2mesoPatch(ts,c0);
217     [ts,cs] = ode23(@patchSmooth1,ts,c0(:));
218     cs=reshape(cs,[length(ts) size(c0)]);
```

2. In the second case, `RK2mesoPatch` detects a parallel patch code has been requested, but has only one cpu worker, so it auto-initiates an `spmd`-block for the integration. Both this and the next case return *composite* results, so just keep one version of the results.

```
230 case 2
231     cs = RK2mesoPatch(ts,c0);
232     cs = cs{1};
```

3. In this third case, a user could merge this explicit `spmd`-block with the previous one that sets the initial conditions.

```
241 case 3,spmd
242     cs = RK2mesoPatch(ts,c0,[],patches);
243     end%spmd
244     cs = cs{1};
```

4. In this fourth case, use Projective Integration (PI) over long times (`PIRK4` also works). Currently the PI is done serially, with parallel `spmd`-blocks only invoked inside function `aBurst()` (Section 4.2.3) to compute each burst of the micro-scale simulation. For a Peclet number of ten, the macro-scale time-step needs to be less than about 0.5 (which here is very little projection)—presumably the mean advection in a macro-step needs to be less than about the patch spacing. The function `microBurst()` here interfaces to `aBurst()` (Section 4.1.3) in order to provide shaped initial states, and to provide the patch information.

```
262 case 4
263     microBurst = @(tb0,xb0,bT) ...
264         aBurst(tb0 ,reshape(xb0,size(c0)) ,patches);
265     ts = 0:0.7:5
266     cs = PIRK2(microBurst,ts,gather(c0(:)));
267     cs = reshape(cs,[length(ts) size(c0)]);
```

End the four cases.

```
274 end%switch theCase
```

4.1.2 Plot the solution

Optionally set to save some plots to file.

```
285 if 0, global OurCf2eps, OurCf2eps=true, end
```

Animate the computed solution field over time

```
291 figure(1), clf, colormap(0.8*hsv)
```

First get the x-coordinates and omit the patch-edge values from the plot (because they are not here interpolated).

```
299 if theCase==1, x = patches.x;
300 else, spmd
301     x = gather( patches.x );
302     end%spmd
303     x = x{1};
304 end
305 x([1 end],:,:,:) = nan;
```

For every time step draw the field values as dots and pause for a short display.

```
312 nTimes = length(ts)
313 for l = 1:nTimes
```

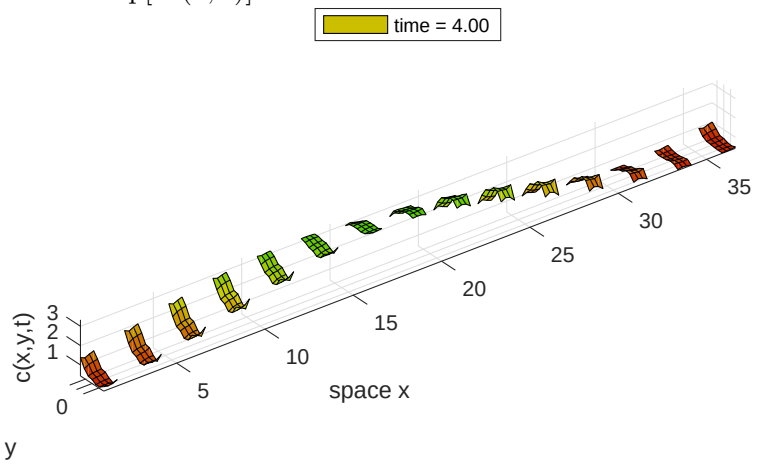
At each time, squeeze sub-patch data into a 3D array, permute to get all the x-variation in the first two dimensions, and reshape into x-variation for each and every (y).

```
322   c = reshape( permute( squeeze( ...
323       cs(l,:,:,:,:) ) ,[1 3 2]) ,numel(x),ny);
```

Draw surface of each patch, to show both micro-scale and macro-scale variation in space.

```
330   if l==1
331     hp = surf(x(:),yj,c');
332     axis([0 Len -1 1 0 max(c(:))])
333     axis equal
334     xlabel('space x'), ylabel('y'); zlabel('c(x,y,t)')
335     ifOurCf2eps([mfilename 't0'])
336     legend(['time = ' num2str(ts(l),'%4.2f')] ...
337         ,'Location','north')
338     disp('**** pausing, press blank to animate')
339     pause
340  else
341     hp.ZData = c';
342     legend(['time = ' num2str(ts(l),'%4.2f')])
```

Figure 4.2: final field $c(x, y, 4)$ of the patch scheme applied to a heterogeneous advection-diffusion PDE (4.1) *with heterogeneous factor log-normal, here distributed* $\exp[\mathcal{N}(0, 1)]$.




```
343      pause(0.1)
344    end
```

Finish the animation loop, and optionally save the final plot to file, Figure 4.2.

```
360  end%for over time
361  ifOurCf2eps([mfilename 'tFin'])
```

Macro-scale view Plot a macro-scale mesh of the predictions: at each of a selection of times, for every patch, plot the patch-mean value at the mean-x.

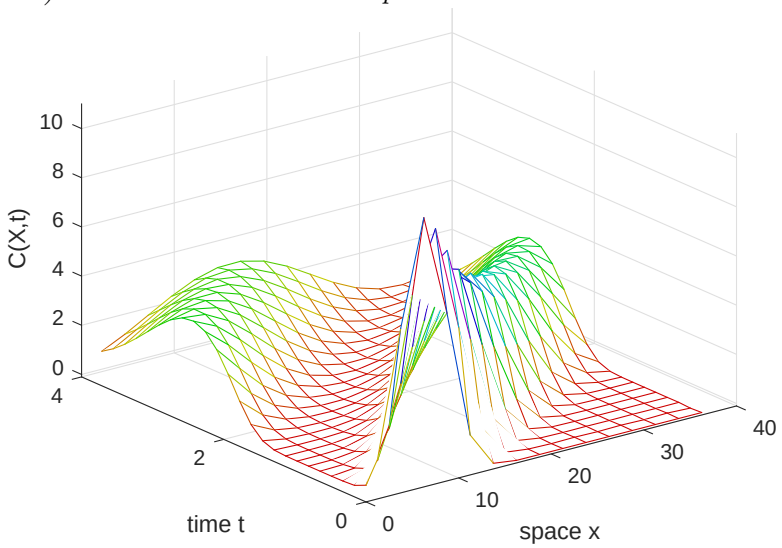
```
371  figure(2), clf, colormap(0.8*hsv)
372  X = squeeze(mean(x(2:end-1,:,:,:)));
373  C = squeeze(mean(mean(cs(:,2:end-1,:,:,:),2),3));
374  j = 1:ceil(nTimes/30):nTimes;
375  mesh(X,ts(j),C(j,:));
376  xlabel('space x'),ylabel('time t'),zlabel('C(X,t)')
377  zlim([-0.1 11])
378  ifOurCf2eps([mfilename 'Macro'])
```

4.1.3 `microBurst` function for Projective Integration

Projective Integration stability appears to require bursts longer than 0.2. Each burst is done in parallel processing. Here use `RK2mesoPatch` to take take meso-steps, each with default ten micro-steps so the micro-scale step is 0.0033. With macro-step 0.5, these parameters usually give stable projective integration.

```
401  function [tbs,xbs] = aBurst(tb0,xb0,patches)
402      normx=max(abs(xb0(:)));
403      disp(['* aBurst t=' num2str(tb0) '  |x|=' num2str(normx)])
404      assert(normx<20,'solution exploding')
405      tbs = tb0+(0:0.033:0.2);
```

Figure 4.3: macro-scale view of heterogeneous advection-diffusion PDE *along a (periodic) channel obtained via the patch scheme.*




```
406     spmd
407       xb0 = codistributed(xb0,patches.codist);
408       xbs = RK2mesoPatch(tbs,xb0,[],patches);
409     end%spmd
410     xbs=reshape(xbs{1},length(tbs),[]);
411 end%function
```

Fin.

4.1.4 `chanDispMicro()`: heterogeneous 2D advection-diffusion along a 1D channel

This function codes the lattice heterogeneous diffusion inside the patches. For 4D input arrays `c` and `x` (via edge-value interpolation of `patchSmooth1`, Section 3.2), computes the time derivative (4.2) at each point in the interior of a patch, output in `ct`. The heterogeneous advections and diffusivities, $u_i(y_j)$ and $\kappa_i(y_{j+1/2})$, have previously been merged and stored in the one array `patches.cs` (2D).

```
21 function ct = chanDispMicro(t,c,p)
22   [nx,ny,~,~]=size(c); % micro-grid points in patches
23   ix = 2:nx-1;         % x interior points in a patch
24   dx = diff(p.x(2:3)); % x space step
25   dy = 2/ny;           % y space step
26   ct = nan+c;          % preallocate output array
27   pcs = reshape(p.cs,nx-1,[],2);
```

Compute the flux using 'ghost' nodes at ends, so that the flux is zero at $y = \pm 1$ either because the end values are replicated so the differences are zero, or because the diffusivities in `cs` are zero at the extremities.

```
36   ydif = pcs(ix,1:2:end,2) ...
37          .*(c(ix,[1:end end],:,:)-c(ix,[1 1:end],:,:))/dy;
```

Now evaluate advection-diffusion time derivative (4.2). Could use upwind advection and no longitudinal diffusion, or, as here, centred advection and diffusion.

```
46    ct(ix,:,:,:) = (ydif(:,2:end,:,:)-ydif(:,1:end-1,:,:))/dy ...
47    + diff(pcs(:,2:2:end,2).*diff(c))/dx^2 ...
48    - p.Pe*pcs(ix,2:2:end,1).*(c(ix+1,:,:,:)-c(ix-1,:,:,:))/(2*dx);
49  end% function
```

4.2 `spmdHomoDiff31`: computational homogenisation of a 1D dispersion via parallel simulation on small 3D patches of diffusion

Section contents

Simulate heterogeneous dispersion along 1D space on 3D patches as a Proof of Principle example of parallel computing with `spmd`. The discussion here only addresses issues with `spmd` parallel computing. For discussion on the 3D patch scheme with heterogeneous diffusion, see code and documentation for `homoDiffEdgy3` in Section 3.18.

Choose one of four cases:

- `theCase=1` is corresponding code without parallelisation (in this toy problem it is much the quickest because there is no expensive communication);
- `theCase=2` for minimising coding by a user of `spmd`-blocks;
- `theCase=3` is for users happier to explicitly invoke `spmd`-blocks.
- `theCase=4` invokes projective integration for long-time simulation via short bursts of the micro-computation, bursts done within `spmd`-blocks for parallel computing.

First, clear all to remove any existing globals, old composites, etc—although a parallel pool persists. Then choose the case.

```
45 clear all
46 theCase = 2
```

Set micro-scale heterogeneity with various periods.

```
53 mPeriod = [4 3 2] %1+randperm(3)
54 cHetr = exp(0.3*randn([mPeriod 3]));
55 cHetr = cHetr*mean(1./cHetr(:))
```

Configure the patch scheme with some arbitrary choices of domain, patches, size ratios—here each patch is a unit cube in space. Choose some random order of interpolation. Set `patches` information to be global so the info can be used for Case 1 without being explicitly passed as arguments. Choose the parallel option if not Case 1, which invokes `spmd`-block internally, so that field variables become *distributed* across cpus.

```
69 if any(theCase==[1 2]), global patches, end
70 nSubP=mPeriod+2
71 nPatch=[9 1 1]
72 ratio=0.3
```

```
xLim=[0 nPatch(1)/ratio 0 1 0 1]
ordCC=2*randi([0 3])
disp('**** Setting configPatches3')
patches = configPatches3(@heteroDiff3, xLim, nan ...
    , nPatch, ordCC, [ratio 1 1], nSubP, 'EdgyInt',true  ...
    ,'hetCoeffs',cHetr ,'parallel',(theCase>1) );
```

4.2.1 Simulate heterogeneous diffusion

Set initial conditions of a simulation as shown in Figure 4.4.

```
disp('**** Set initial condition and testing du0dt =')
if theCase==1
```

Without parallel processing, invoke the usual operations.

```
    u0 = exp( -(patches.x-xLim(2)/2).^2/xLim(2) ...
              -patches.y.^2/2-patches.z.^2 );
    u0 = u0.*(1+0.2*rand(size(u0)));
    du0dt = patchSmooth3(0,u0);
```

With parallel, must use an `spmd`-block for computations: there is no difference in cases 2–4 here. Also, we must sometimes explicitly code how to distribute some new arrays over the cpus. Now `patchSmooth3` does not invoke `spmd` so higher level code must, as here. Even if `patches` is global, inside `spmd`-block we must pass it explicitly as a parameter to `patchSmooth3`.

```
else, spmd
    u0 = exp( -(patches.x-xLim(2)/2).^2/xLim(2) ...
              -patches.y.^2/2-patches.z.^2/4 );
    u0 = u0.*(1+0.2*rand(size(u0),patches.codist));
    du0dt = patchSmooth3(0,u0,patches);
    end%spmd
end%if theCase
```

Integrate in time. Use non-uniform time-steps for fun, and to show more of the initial rapid transients.

Alternatively, use `RK2mesoPatch` which reduces communication between patches, recalling that, by default, `RK2mesoPatch` does ten micro-steps for each specified step in `ts`. For unit cube patches, need micro-steps less than about 0.004 for stability.

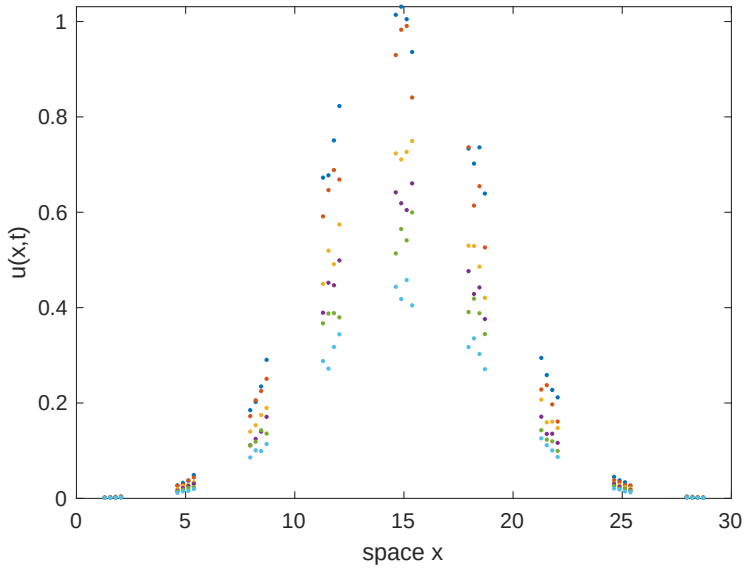
```
warning('Integrating system in time, wait patiently')
ts=0.4*linspace(0,1,21).^2;
```

Go to the selected case.

```
switch theCase
```

1. For non-parallel, we could use `RK2mesoPatch` as indicated below, but instead choose to use standard `ode23` as here `patchSmooth3` accesses patch information via global `patches`. For post-processing, reshape each

Figure 4.4: initial field $u(x, y, z, 0)$ of the patch scheme applied to a heterogeneous diffusion PDE. *The vertical spread indicates the extent of the structure in u in the cross-section variables y, z. Figure 4.5 plots the nearly smooth field values at time $t = 0.4$.*



and every row of the computed solution to the correct array size—that of the initial condition.

```
159 case 1
160 %    [us,uerrs] = RK2mesoPatch(ts,u0);
161      [ts,us] = ode23(@patchSmooth3,ts,u0(:));
162      us=reshape(us,[length(ts) size(u0)]);
```

2. In the second case, `RK2mesoPatch` detects a parallel patch code has been requested, but has only one cpu worker, so it auto-initiates an `spmd`-block for the integration. Both this and the next case return *composite* results, so just keep one version of the results.

```
174 case 2
175      us = RK2mesoPatch(ts,u0);
176      us = us{1};
```

3. In this third case, a user could merge this explicit `spmd`-block with the previous one that sets the initial conditions.

```
185 case 3,spmd
186      us = RK2mesoPatch(ts,u0,[],patches);
187      end%spmd
188      us = us{1};
```

4. In this fourth case, use Projective Integration (PI) over long times (`PIRK4` also works). Currently the PI is done serially, with parallel `spmd`-blocks only invoked inside function `aBurst()` (Section 4.2.3) to compute each burst of the micro-scale simulation. A macro-scale time-step of about 3 seems good to resolve the decay of the macro-scale

'homogenised' diffusion. [2] The function `microBurst()` here interfaces to `aBurst()` (Section 4.2.3) in order to provide shaped initial states, and to provide the patch information.

```
206 case 4
207     microBurst = @(tb0,xb0,bT) ...
208         aBurst(tb0 ,reshape(xb0,size(u0)) ,patches);
209     ts = 0:3:51
210     us = PIRK2(microBurst,ts,gather(u0(:)));
211     us = reshape(us,[length(ts) size(u0)]);
```

End the four cases.

```
218 end%switch theCase
```

4.2.2 Plot the solution

Optionally save some plots to file.

```
229 if 0, global OurCf2eps, OurCf2eps=true, end
```

Animate the solution field over time. Since the spatial domain is long in x and thin in y, z, just plot field values as a function of x.

```
237 figure(1), clf
238 if theCase==1
239     x = reshape( patches.x(2:end-1,:,:,:) ,[],1);
240 else, spmd
241     x = reshape(gather( patches.x(2:end-1,:,:,:) ),[],1);
242     end%spmd
243     x = x{1};
244 end
```

For every time step draw the field values as dots and pause for a short display.

```
251 nTimes = length(ts)
252 for l = 1:length(ts)
```

At each time, squeeze interior point data into a 4D array, permute to get all the x-variation in the first two dimensions, and reshape into x-variation for each and every (y, z).

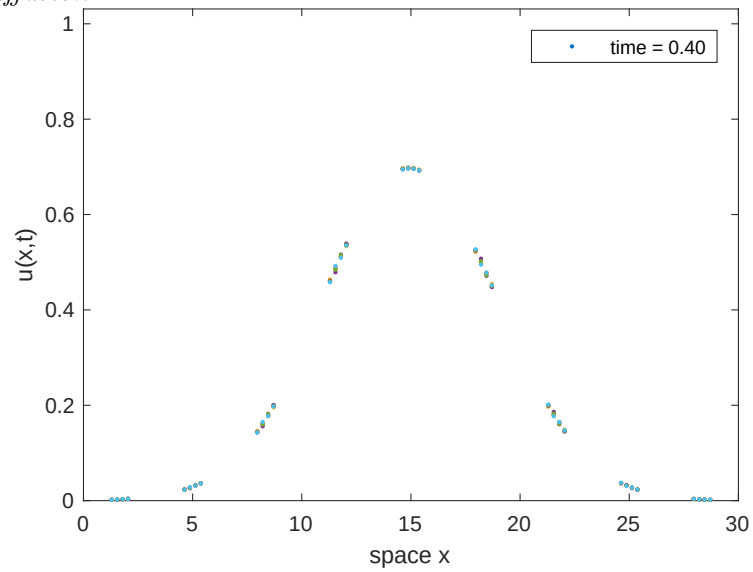
```
261   u = reshape( permute( squeeze( ...
262       us(l,2:end-1,2:end-1,2:end-1,:) ) ,[1 4 2 3]) ,numel(x),[]);
```

Draw point data to show spread at each cross-section, as well as macro-scale variation in the long space direction.

```
269   if l==1
270     hp = plot(x,u,'.');
271     axis([xLim(1:2) 0 max(u(:))])
272     xlabel('space x'), ylabel('u(x,y,z,t)')
273     ifOurCf2eps([mfilename 't0'])
```

[2] Curiously, `PIG()` appears to suffer unrecoverable instabilities with its variable step size!

Figure 4.5: final field $u(x, y, z, 0.4)$ of the patch scheme applied to a heterogeneous diffusion PDE.




```
274      legend(['time = ' num2str(ts(1),'%4.2f')])
275      disp('**** pausing, press blank to animate')
276      pause
277    else
278      for p=1:size(u,2), hp(p).YData=u(:,p); end
279      legend(['time = ' num2str(ts(1),'%4.2f')])
280      pause(0.1)
281    end
```

Finish the animation loop, and optionally output the final plot, Figure 4.5.

```
294  end%for over time
295  ifOurCf2eps([mfilename 'tFin'])
```

4.2.3 `microBurst` function for Projective Integration

Projective Integration stability seems to need bursts longer than 0.2. Here take ten meso-steps, each with default ten micro-steps so the micro-scale step is 0.002. With macro-step 3, these parameters usually give stable projective integration (but not always).

```
311  function [tbs,xbs] = aBurst(tb0,xb0,patches)
312      normx=max(abs(xb0(:)));
313      disp(['aBurst t = ' num2str(tb0) '  |x| = ' num2str(normx)])
314      assert(normx<10,'solution exploding')
315      tbs = tb0+(0:0.02:0.2);
316      spmd
317        xb0 = codistributed(xb0,patches.codist);
318        xbs = RK2mesoPatch(tbs,xb0,[],patches);
319      end%spmd
320      xbs=reshape(xbs{1},length(tbs),[]);
```

```
321 end%function
```

Fin.

4.3 RK2mesoPatch()

This is a Runge–Kutta, 2nd order, integration of a given deterministic system of ODEs on patches. It invokes meso-time updates of the patch-edge values in order to reduce interpolation costs, and uses a linear variation in edge-values over the meso-time-step (Bunder et al. 2016, case $Q = 2$). This function is aimed primarily for large problems executed on a computer cluster to markedly reduce expensive communication between computers.

If using within projective integration, it appears quite tricky to get all the time-steps chosen appropriately. One has to choose times for: the micro-scale time-step, the meso-time interval between communications, the longer meso-time burst length, and the macro-scale integration time-step.

```
27 function [xs,errs] = RK2mesoPatch(ts,x0,nMicro,patches)
28 if nargin<4, global patches, end
```

Input

- `patches.fun()` is a function such as `dxdt=fun(t,x,patches)` that computes the right-hand side of the ODE $d\vec{x}/dt = \vec{f}(t, \vec{x})$ where $\vec{x}$ is a vector/array, t is a scalar, and the result $\vec{f}$ is a correspondingly sized vector/array.
- `x0` is an vector/array of initial values at the time `ts(1)`.
- `ts` is a vector of meso-scale times to compute the approximate solution, say in $\mathbb{R}^\ell$ for $\ell \geq 2$.
- `nMicro`, optional, default 10, is the number of micro-time-steps taken for each meso-scale time-step.
- `patches` struct set by `configPatches`n and provided as either as parameter, or as a global variable.

Output

- `xs`, 5/7/9D (depending upon `nD`) array of length $\ell \times \cdots$ of approximate solution vector/array at the specified times. But, if using parallel computing via `spmd`, then `xs` is a *composite* 5/7/9D array, so outside of an `spmd`-block access a single copy of the array via `xs{1}`. Similarly for `errs`.
- `errs`, column vector in $\mathbb{R}^\ell$ of local error estimate for the step from t_{k-1} to t_k.

Code of RK2 integration Set default number of micro-scale time-steps in each requested meso-scale step of `ts`. Cannot use `nargin` inside explicit `spmd`, but can use it if the `spmd` is already active from the code that invokes this function.

```
79 if nargin<3|isempty(nMicro), nMicro=10; end
```

If patches are set to be in parallel (there must be a parallel pool), but only one worker available (i.e., not already inside `spmd`), then invoke function recursively inside `spmd`. Q: is `numlabs` defined without the parallel computing toolbox??

```
%warning('checking spmd status in RK2mesoPatch')
%disp(['class-patches = ',class(patches)])
if isequal(class(patches),'Composite') && numlabs==1
    spmd, %warning('recursing into RK2mesoPatch')
      [xs,errs] = RK2mesoPatch(ts,x0,nMicro,patches);
      %warning('finished recursion into RK2mesoPatch')
    end% spmd
    assert(isequal(class(xs)  ,'Composite'),' xs  not composite')
    assert(isequal(class(errs),'Composite'),'errs not composite')
    return
end
%warning('bypassed start spmd in RK2mesoPatch')
```

Set the number of space dimensions from the number stored patch-size ratios.

```
nD = length(patches.ratio);
```

Set the micro-time-steps and create storage for outputs.

```
dt = diff(ts)/nMicro;
xs = nan([numel(ts) size(x0)]);
errs = nan(numel(ts),1);
```

Initialise first result to the given initial condition, and evaluate the initial time derivative into `f1`. Use inter-patch interpolation to ensure edge values of the initial condition are defined and are reasonable. [3]

```
%warning('RK2mesoPatch: x0 = patchEdgeInt3(x0)')
switch nD
  case 1, x0 = patchEdgeInt1(x0,patches);
          xs(1,:,:,:,:) = gather(x0);
  case 2, x0 = patchEdgeInt2(x0,patches);
          xs(1,:,:,:,:,:,:) = gather(x0);
  case 3, x0 = patchEdgeInt3(x0,patches);
          xs(1,:,:,:,:,:,:,:,:) = gather(x0);
  end;%switch nD
%assert(iscodistributed(x0),'x0 not codist one')
errs(1) = 0;
%warning('RK2mesoPatch: patches.fun one')
f1 = patches.fun(ts(1),x0,patches);
```

Compute the meso-time-steps from t_k to t_{k+1}, copying the derivative `f1` at the end of the last micro-time-step to be the derivative at the start of this one.

[3] These `gather()` functions cause all-to-all interprocessor communication once every meso-step. Maybe better to use distributed array instead, (although need to then need to put time index last instead of first??), but we need to do some inter-cpu communication in order to estimate errors.

```
152 for k = 1:numel(dt)
```

Perform meso-time burst with the new interpolation for edge values, and an interpolation of the time derivatives to get derivative estimates of the edge-values.

```
160   switch nD
161     case 1, dx0 = patchEdgeInt1(f1,patches);
162     case 2, dx0 = patchEdgeInt2(f1,patches);
163     case 3, dx0 = patchEdgeInt3(f1,patches);
164     end;%switch nD
```

Perform the micro-time steps.

```
170   for m=1:nMicro
171     f0 = f1;
172 %    assert(iscodistributed(f0),'f0 not codist')
```

For all micro-time derivative evaluations, include that the edge values are varying according to the estimate made at the start of the meso-time-step.

```
180     switch nD
181       case 1, f0([1 end],:,:,:)=dx0([1 end],:,:,:);
182       case 2, f0([1 end],:,:,:,:,:)=dx0([1 end],:,:,:,:,:);
183               f0(:,[1 end],:,:,:,:)=dx0(:,[1 end],:,:,:,:);
184       case 3
185         f0([1 end],:,:,:,:,:,:)=dx0([1 end],:,:,:,:,:,:);
186         f0(:,[1 end],:,:,:,:,:)=dx0(:,[1 end],:,:,:,:,:);
187         f0(:,:,[1 end],:,:,:,:)=dx0(:,:,[1 end],:,:,:,:);
188     end;%switch nD
189 %    assert(iscodistributed(f0),'f0 not codist two')
```

Simple second-order accurate Runge–Kutta micro-scale time-step.

```
196     xh = x0+f0*dt(k)/2;
197 %    assert(iscodistributed(xh),'xh not codist')
198     fh = patches.fun(ts(k)+dt(k)*(m-0.5),xh,patches);
199 %    assert(iscodistributed(fh),'fh not codist one')
200     switch nD
201       case 1, fh([1 end],:,:,:)=dx0([1 end],:,:,:);
202       case 2, fh([1 end],:,:,:,:,:)=dx0([1 end],:,:,:,:,:);
203               fh(:,[1 end],:,:,:,:)=dx0(:,[1 end],:,:,:,:);
204       case 3
205         fh([1 end],:,:,:,:,:,:)=dx0([1 end],:,:,:,:,:,:);
206         fh(:,[1 end],:,:,:,:,:)=dx0(:,[1 end],:,:,:,:,:);
207         fh(:,:,[1 end],:,:,:,:)=dx0(:,:,[1 end],:,:,:,:);
208     end;%switch nD
209 %    assert(iscodistributed(fh),'fh not codist two')
210     x0 = x0+fh*dt(k);
211 %    assert(iscodistributed(x0),'x0 not codist two')
```

End the burst of micro-time-steps.

```
217   end
```

At the end of each meso-step burst, refresh the interpolate of the edge values, evaluate time-derivative, and temporarily fill-in edges of derivatives (to ensure error estimate is reasonable).

```
226   switch nD
227     case 1, x0 = patchEdgeInt1(x0,patches);
228             xs(k+1,:,:,:,:) = gather(x0);
229     case 2, x0 = patchEdgeInt2(x0,patches);
230             xs(k+1,:,:,:,:,:,:) = gather(x0);
231     case 3, x0 = patchEdgeInt3(x0,patches);
232             xs(k+1,:,:,:,:,:,:,:,:) = gather(x0);
233     end;%switch nD
234 %  assert(iscodistributed(x0),'x0 not codist three')
235   f1 = patches.fun(ts(k+1),x0,patches);
236   switch nD
237     case 1, f1([1 end],:,:,:)=dx0([1 end],:,:,:);
238     case 2, f1([1 end],:,:,:,:,:)=dx0([1 end],:,:,:,:,:);
239             f1(:,[1 end],:,:,:,:)=dx0(:,[1 end],:,:,:,:);
240     case 3
241       f1([1 end],:,:,:,:,:,:,:)=dx0([1 end],:,:,:,:,:,:,:);
242       f1(:,[1 end],:,:,:,:,:,:)=dx0(:,[1 end],:,:,:,:,:,:);
243       f1(:,:,[1 end],:,:,:,:,:)=dx0(:,:,[1 end],:,:,:,:,:);
244     end;%switch nD
```

Use the time derivative at t_{k+1} to estimate an error by storing the difference with what Simpson's rule would estimate over the last micro-time step performed.

```
252   f0=f0-2*fh+f1;
253 %  assert(iscodistributed(f0),'f2ndDeriv not codist')
254   errs(k+1) = sqrt(gather(mean(f0(:).^2,'omitnan')))*dt(k)/6;
255 end%for-loop
256 end%function
```

End of the function with results returned in `xs` and `errs`.

4.4 `rotFilmSpmd`: simulation of a 1D shear dispersion via simulation on small patches across a channel

Section contents

As an example application, consider the flow of a shallow layer of fluid on a solid flat rotating substrate, such as in spin coating (Wilson et al. 2000, Oron et al. 1997, §II.K, e.g.) or large-scale shallow water waves (Dellar & Salmon 2005, Hereman 2009, e.g.). Let $\vec{x} = (x, y)$ parametrise location on the rotating substrate, and let the fluid layer have thickness $h(\vec{x}, t)$ and move with depth-averaged horizontal velocity $\vec{v}(\vec{x}, t) = (u, v)$. We take as given (with its simplified physics) that the (non-dimensional) governing set of PDEs is the nonlinear system (Bunder & Roberts 2018, eq. (1))

$$\frac{\partial h}{\partial t} = -\nabla \cdot (h\vec{v}), \tag{4.3a}$$

$$\frac{\partial \vec{v}}{\partial t} = \begin{bmatrix} -b & f \\ -f & -b \end{bmatrix} \vec{v} - (\vec{v}\cdot\nabla)\vec{v} - g\nabla h + \vec{\nabla}\cdot(\nu\vec{\nabla}\vec{v})\,, \tag{4.3b}$$

where $b(\vec{x})$ represents heterogeneous 'bed' drag, f is the Coriolis coefficient, g is the acceleration due to gravity, $\nu(\vec{x})$ is a heterogeneous 'kinematic viscosity', and we neglect surface tension.

The aim is to simulate the macroscale dynamics which (for constant b) is approximately that of the nonlinear diffusion $\partial h/\partial t \approx \frac{gb}{b^2+f^2}\vec{\nabla}\cdot(h\vec{\nabla}h)$ (Bunder & Roberts 2018, eq. (2)). But there is no known algebraic closure for the macroscale in the case of heterogeneous $b(\vec{x})$ and $\nu(\vec{x})$, nonetheless the patch scheme automatically predicts a sensible macroscale for such heterogeneous dynamics (Figure 4.7).

For the microscale computation, Section 4.4.4 discretises the PDEs (4.3) in space with x, y-spacing $\delta x, \delta y$.

Choose one of four cases:

- `theCase=1` is corresponding code without parallelisation (in this toy problem it is much the quickest because there is no expensive communication);
- `theCase=2` illustrates that `RK2mesoPatch` invokes `spmd` computation if parallel has been configured.
- `theCase=3` shows how users explicitly invoke `spmd`-blocks around the time integration.

- `theCase=4` invokes projective integration for long-time simulation via short bursts of the micro-computation, bursts done within `spmd`-blocks for parallel computing.

First, clear all to remove any existing globals, old composites, etc—although a parallel pool persists. Then choose the case.

```
clear all
theCase = 1
```

Set micro-scale bed drag (array 1) and diffusivity (arrays 2–3) to be a heterogeneous log-normal factor with specified period: modify the strength of the heterogeneity by the coefficient of `randn` from zero to perhaps one: coefficient 0.3 appears a good moderate value.

```
mPeriod = 5
bnu = shiftdim([1 0.5 0.5],-1) ...
      .*exp(0.3*randn([mPeriod mPeriod 3]));
```

Configure the patch scheme with these choices of domain, patches, size ratios—here each patch is square in space. In Cases 1–2, set `patches` information to be global so the info can be used without being explicitly passed as arguments.

```
if theCase<=2, global patches, end
```

In Case 4, double the size of the domain and use more separated patches accordingly, to maintain the spatial microscale grid spacing to be 0.055. Here use fourth order edge-based coupling between patches. Choose the parallel option if not Case 1, which invokes `spmd`-block internally, so that field variables become *distributed* across cpus.

```
nSubP = 2+mPeriod
nPatch = 9
ratio = 0.2+0.2*(theCase<4)
Len = 2*pi*(1+(theCase==4))
disp('**** Setting configPatches2')
patches = configPatches2(@rotFilmMicro, [0 Len], nan ...
    , nPatch, 4, ratio, nSubP, 'EdgyInt',true  ...
    ,'hetCoeffs',bnu ,'parallel',(theCase>1) );
```

When using parallel, any additional parameters to `patches`, such as physical parameters for the microcode, must be set within a `spmd` block (because `patches` is a co-distributed structure). Here set frequency of substrate rotation, and strength of gravity.

```
f = 5,  g = 1
if theCase==1, patches.f = f; patches.g = g;
else     spmd, patches.f = f; patches.g = g; end
end
```

4.4.1 Simulate heterogeneous advection-diffusion

Set initial conditions of a simulation as shown in Figure 4.6. Here the initial condition is a (periodic) Gaussian in h and zero velocity $\vec{v}$, with additive

random perturbations.

```
142 disp('**** Set initial condition and test dhuv0dt =')
143 if theCase==1
```

When not parallel processing, invoke the usual operations. Here add a random noise to the velocity field, but keep $h(x, y, 0)$ smooth as shown by Figure 4.6. The `shiftdim(...,-1)` moves the given row-vector of coefficients into the third dimension to become coefficients of the fields (h, u, v), respectively.

```
155     huv0 = shiftdim([0.5 0 0],-1) ...
156         .*exp(-cos(patches.x)/2-cos(patches.y));
157     huv0 = huv0+0.1*shiftdim([0 1 1],-1).*rand(size(huv0));
158     dhuv0dt = patchSmooth2(0,huv0);
```

With parallel, we must use an `spmd`-block for computations: there is no difference in Cases 2–4 here. Also, we must sometimes explicitly tell functions how to distribute some initial condition arrays over the cpus. Now `patchSmooth2` does not invoke `spmd` so higher level code must, as here. Even if `patches` is global, inside an `spmd`-block we *must* pass `patches` explicitly as a parameter to `patchSmooth2`.

```
172 else, spmd
173     huv0 = shiftdim([0.5 0 0],-1) ...
174         .*exp(-cos(patches.x)/2-cos(patches.y));
175     huv0 = huv0+0.1*rand(size(huv0),patches.codist);
176     dhuv0dt = patchSmooth2(0,huv0,patches)
177     end%spmd
178 end%if theCase
```

Integrate in time, either via the automatic `ode23` or via `RK2mesoPatch` which reduces communication between patches. By default, `RK2mesoPatch` does ten micro-steps for each specified meso-step in `ts`. For stability: with noise up to 0.3, need micro-steps less than 0.0003; with noise 1, need micro-steps less than 0.0001.

```
203 warning('Integrating system in time, wait a minute')
204 ts=0:0.003:0.3;
```

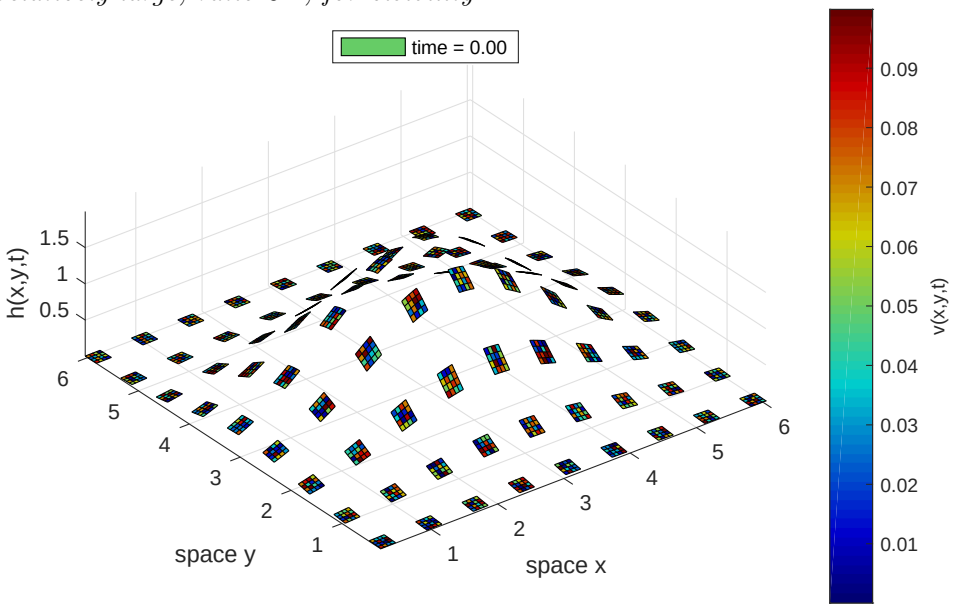
Go to the selected case.

```
210 switch theCase
```

1. For non-parallel, we could use `RK2mesoPatch` as indicated below, but instead choose to use standard `ode23` as here `patchSmooth2` accesses patch information via global `patches`. For post-processing, reshape each and every row of the computed solution to the correct array size—namely that of the initial condition.

```
222 case 1
223 %    tic,[huvs,uerrs] = RK2mesoPatch(ts,huv0);toc
224      [ts,huvs] = ode23(@patchSmooth2,[0 4],huv0(:));
225      huvs=reshape(huvs,[length(ts) size(huv0)]);
```

Figure 4.6: initial field $h(x, y, 0)$ of the patch scheme applied to the heterogeneous, shallow water, rotating substrate, PDE (4.3)*. The micro-scale sub-patch colour displays the initial y-direction velocity field $v(x, y, 0)$. Figure 4.7 plots the roughly smooth field values at time $t = 6$. In this example the patches are relatively large, ratio* 0.4*, for visibility.*



2. In the second case, `RK2mesoPatch` detects a parallel patch code has been requested, but has only one cpu worker, so it auto-initiates an `spmd`-block for the integration. Both this and the next case return *composite* results, so just keep one version of the results.

```
237 case 2
238     huvs = RK2mesoPatch(ts,huv0);
239     huvs = huvs{1};
```

3. In this third case, a user could merge this explicit `spmd`-block with the previous one that sets the initial conditions.

```
248 case 3,spmd
249     huvs = RK2mesoPatch(ts,huv0,[],patches);
250     end%spmd
251     huvs = huvs{1};
```

4. In this fourth case, use Projective Integration (PI). Currently the PI is done serially, with parallel `spmd`-blocks only invoked inside function `aBurst()` (Section 4.4.3) to compute each burst of the micro-scale simulation. The macro-scale time-step needs to be less than about 0.1 (which here is not much projection). The function `microBurst()` interfaces to `aBurst()` (Section 4.4.3) in order to provide shaped initial states, and to provide the patch information.

```
266 case 4
267     microBurst = @(tb0,xb0,bT) ...
```

```
268           aBurst(tb0 ,reshape(xb0,size(huv0)) ,patches);
269       ts = 0:0.1:1
270       huvs = PIRK2(microBurst,ts,gather(huv0(:)));
271       huvs = reshape(huvs,[length(ts) size(huv0)]);
```

End the four cases.

```
278 end%switch theCase
```

4.4.2 Plot the solution

Optionally set to save some plots to file.

```
289 if 0, global OurCf2eps, OurCf2eps=true, end
```

Animate the computed solution field over time

```
295 figure(1), clf, colormap(0.8*jet)
```

First get the x-coordinates and omit the patch-edge values from the plot (because they are not here interpolated).

```
303 if theCase==1, x = patches.x;
304                y = patches.y;
305 else, spmd
306     x = gather( patches.x );
307     y = gather( patches.y );
308     end%spmd
309     x = x{1}; y = y{1};
310 end
311 x([1 end],:,:,:,:,:) = nan;
312 y(:,[1 end],:,:,:,:) = nan;
```

Draw the field values as a patchy surface evolving over 100–200 time steps.

```
319 nTimes = length(ts)
320 for l = 1:ceil(nTimes/200):nTimes
```

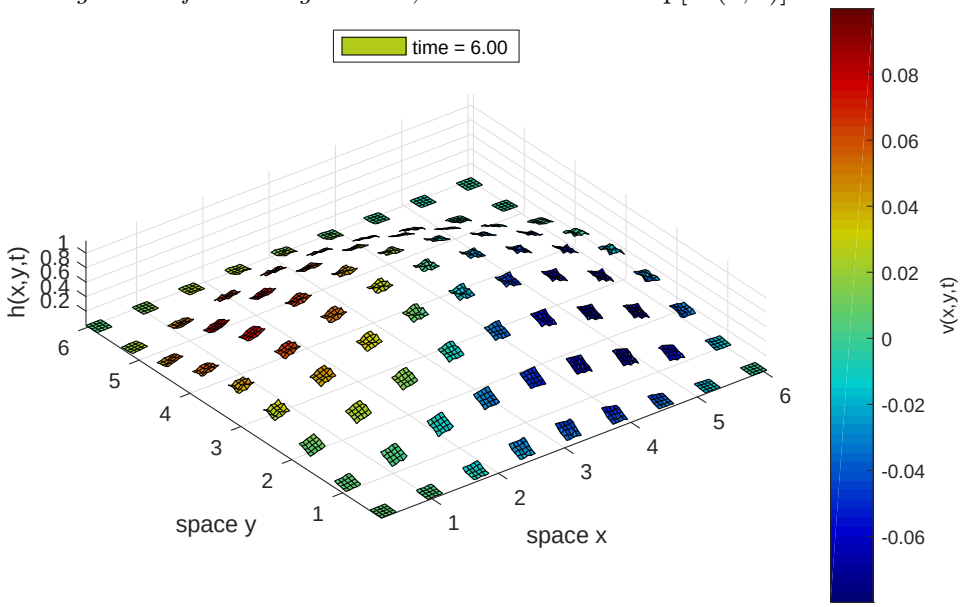
At each time, squeeze sub-patch data fields into three 4D arrays, permute to get all the x/y-variations in the first/last two dimensions, and and then reshape to 2D.

```
328   h = reshape( permute( squeeze( ...
329       huvs(l,:,:,1,1,:,:) ) ,[1 3 2 4]) ,numel(x),numel(y));
330   u = reshape( permute( squeeze( ...
331       huvs(l,:,:,2,1,:,:) ) ,[1 3 2 4]) ,numel(x),numel(y));
332   v = reshape( permute( squeeze( ...
333       huvs(l,:,:,3,1,:,:) ) ,[1 3 2 4]) ,numel(x),numel(y));
```

Draw surface of each patch, to show both micro-scale and macro-scale variation in space. Colour the surface according to the velocity v in the y-direction.

```
341   if l==1
342     hp = surf(x(:),y(:),h',v');
```

Figure 4.7: final field $h(x, y, 6)$, coloured by $v(x, y, 6)$, of the patch scheme applied to the heterogeneous, shallow water, rotating substrate, PDE (4.3) *with heterogeneous factors log-normal, here distributed* $\exp[\mathcal{N}(0, 1)]$.




```
343      axis([0 Len 0 Len 0 max(h(:))])
344      c = colorbar; c.Label.String = 'v(x,y,t)';
345      legend(['time = ' num2str(ts(1),'%4.2f')] ...
346          ,'Location','north')
347      axis equal
348      xlabel('space x'), ylabel('space y'), zlabel('h(x,y,t)')
349      ifOurCf2eps([mfilename 't0'])
350      disp('**** pausing, press blank to begin animation')
351      pause
352    else
353      hp.ZData = h'; hp.CData = v';
354      legend(['time = ' num2str(ts(1),'%4.2f')])
355      pause(0.1)
356    end
```

Finish the animation loop, and optionally save the final plot to file, Figure 4.7.

```
373  end%for over time
374  ifOurCf2eps([mfilename 'tFin'])
```

4.4.3 `microBurst` function for Projective Integration

Projective Integration stability appears to require bursts longer than 0.01. Each burst is done in parallel processing. Here use `RK2mesoPatch` to take take meso-steps, each with default ten micro-steps so the micro-scale step is 0.0003. With macro-step 0.1, these parameters usually give stable projective integration.

```
391 function [tbs,xbs] = aBurst(tb0,xb0,patches)
392     normx=max(abs(xb0(:)));
393     disp(['* aBurst t=' num2str(tb0) '  |x|=' num2str(normx)])
394     assert(normx<20,'solution exploding')
395     tbs = tb0+(0:0.003:0.015);
396     spmd
397       xb0 = codistributed(xb0,patches.codist);
398       xbs = RK2mesoPatch(tbs,xb0,[],patches);
399     end%spmd
400     xbs=reshape(xbs{1},length(tbs),[]);
401 end%function
```

Fin.

4.4.4 `rotFilmMicro()`: 2D shallow water flow on a rotating heterogeneous substrate

This function codes the lattice heterogeneous diffusion inside the patches. For 6D input arrays `huv` (via edge-value interpolation of `patchSmooth2`, Section 3.11), computes the time derivatives (4.3) at each point in the interior of a patch, output in `huvt`. The heterogeneous bed drag and diffusivities, b_{ij} and ν_{ij}, have previously been merged and stored in the array `patches.cs` (2D $\times$ 3): herein `patches` is named `p`.

```
22 function huvt = rotFilmMicro(t,huv,p)
23   [nx,ny,~]=size(huv); % micro-grid points in patches
24   i = 2:nx-1;          % x interior points in a patch
25   j = 2:ny-1;          % y interior points in a patch
26   dx = diff(p.x(2:3)); % x space step
27   dy = diff(p.y(2:3)); % y space step
28   huvt = nan+huv;      % preallocate output array
```

Set indices of fields in the arrays. Need to store different diffusivity values for the x, y-directions as they are evaluated at different points in space.

```
36   h=1; u=2; v=3;
37   b=1; nux=2; nuy=3;
```

Use a staggered grid so that `h(i,j)` $= h_{ij}$, `u(i,j)` $= u_{i+1/2,j}$, and `v(i,j)` $= v_{i,j+1/2}$. We need these averages in order to interpolate some quantities to other points on the staggered grid. But the first two statements fill-in two needed corner values because they are not (currently) interpolated.

```
49 huv(1,ny,u,:,:,:) = huv(2,ny,u,:,:,:)+huv(1,ny-1,u,:,:,:) ...
50                     -huv(2,ny-1,u,:,:,:);
51 huv(nx,1,v,:,:,:) = huv(nx,2,v,:,:,:)+huv(nx-1,1,v,:,:,:) ...
52                     -huv(nx-1,2,v,:,:,:);
53 v4u = (huv(i,j-1,v,:,:,:)+huv(i+1,j,v,:,:,:) ...
54       +huv(i,j,v,:,:,:)+huv(i+1,j-1,v,:,:,:))/4;
55 u4v = (huv(i,j+1,u,:,:,:)+huv(i-1,j,u,:,:,:) ...
56       +huv(i,j,u,:,:,:)+huv(i-1,j+1,u,:,:,:))/4;
```

```
57  h2u = (huv(2:nx,:,h,:,:,:)+huv(1:nx-1,:,h,:,:,:))/2;
58  h2v = (huv(:,2:ny,h,:,:,:)+huv(:,1:ny-1,h,:,:,:))/2;
```

Evaluate conservation of mass PDE (4.3a) (needing averages of h at half-grid points):

```
65    huvt(i,j,h,:,:,:) = ...
66      - (h2u(i,j  ,:,:,:,:).*huv(i  ,j,u,:,:,:) ...
67        -h2u(i-1,j,:,:,:,:).*huv(i-1,j,u,:,:,:) )/dx ...
68      - (h2v(i,j  ,:,:,:,:).*huv(i,j  ,v,:,:,:) ...
69        -h2v(i,j-1,:,:,:,:).*huv(i,j-1,v,:,:,:))/dy ;
```

Evaluate the x-direction momentum PDE (4.3b) (needing to interpolate component v to u-points):

```
77    huvt(i,j,u,:,:,:) = ...
78      - p.cs(i,j,b).*huv(i,j,u,:,:,:) + p.f.*v4u ...
79      - huv(i,j,u,:,:,:).*(huv(i+1,j,u,:,:,:)-huv(i-1,j,u,:,:,:))/(2*dx) ...
80      - v4u.*(huv(i,j+1,u,:,:,:)-huv(i,j-1,u,:,:,:))/(2*dy) ...
81      - p.g*(huv(i+1,j,h,:,:,:)-huv(i,j,h,:,:,:))/dx ...
82      + diff(p.cs(:,j,nux).*diff(huv(:,j,u,:,:,:),[],1),[],1)/dx^2 ...
83      + diff(p.cs(i,:,nuy).*diff(huv(i,:,u,:,:,:),[],2),[],2)/dy^2 ;
```

Evaluate the y-direction momentum PDE (4.3b) (needing to interpolate component u to v-points):

```
91    huvt(i,j,v,:,:,:) = ...
92      - p.cs(i,j,b).*huv(i,j,v,:,:,:) - p.f.*u4v ...
93      - u4v.*(huv(i+1,j,v,:,:,:)-huv(i-1,j,v,:,:,:))/(2*dx) ...
94      - huv(i,j,v,:,:,:).*(huv(i,j+1,v,:,:,:)-huv(i,j-1,v,:,:,:))/(2*dy) ...
95      - p.g*(huv(i,j+1,h,:,:,:)-huv(i,j,h,:,:,:))/dy ...
96      + diff(p.cs(:,j,nux).*diff(huv(:,j,v,:,:,:),[],1),[],1)/dx^2 ...
97      + diff(p.cs(i,:,nuy).*diff(huv(i,:,v,:,:,:),[],2),[],2)/dy^2 ;
98  end% function
```

4.5 To do

- Lots ??

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
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., Divahar, J., Kevrekidis, I. G., Mattner, T. W. & Roberts, A. (2019), Large-scale simulation of shallow water waves with computation only on small staggered patches, Technical report, https://arxiv.org/abs/1912.07815.

Bunder, J. E., Kevrekidis, I. G. & Roberts, A. J. (2020), Equation-free patch scheme for efficient computational homogenisation via self-adjoint coupling, Technical report, http://arxiv.org/abs/2007.06815.

Bunder, J. E. & Roberts, A. J. (2018), Nonlinear emergent macroscale PDEs, with error bound, for nonlinear microscale systems, Technical report, [https://arxiv.org/abs/1806.10297].

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.

Dellar, P. J. & Salmon, R. (2005), 'Shallow water equations with a complete coriolis force and topography', *Phys. Fluids* **17**, 106601.

Frewen, T. A., Hummer, G. & Kevrekidis, I. G. (2009), 'Exploration of effective potential landscapes using coarse reverse integration', *The Journal of Chemical Physics* **131**(13), 134104.

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.

Hereman, W. (2009), Shallow water waves and solitary waves, *in* 'Mathematics of Complexity and Dynamical Systems', Springer, New York, pp. 8112–8125.

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

MacKenzie, T. & Roberts, A. J. (2003), Holistic discretisation of shear dispersion in a two-dimensional channel, *in* K. Burrage & R. B. Sidje, eds, 'Proc. of 10th Computational Techniques and Applications Conference CTAC-2001', Vol. 44, pp. C512–C530.

Maclean, J., Bunder, J. E. & Roberts, A. J. (2020), 'A toolbox of equation-free functions in matlab/octave for efficient system level simulation', *Numerical Algorithms* .

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.

Oron, A., Davis, S. H. & Bankoff, S. G. (1997), 'Long-scale evolution of thin liquid films', *Rev. Mod. Phys.* **69**, 931–980. http://link.aps.org/abstract/RMP/v69/p931.

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. (2003), 'A holistic finite difference approach models linear dynamics consistently', *Mathematics of Computation* **72**, 247–262. http://www.ams.org/mcom/2003-72-241/S0025-5718-02-01448-5

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.

Taylor, G. I. (1953), 'Dispersion of soluble matter in solvent flowing slowly through a tube', *Proc. Roy. Soc. Lond. A* **219**, 186–203.

Watt, S. D. & Roberts, A. J. (1995), 'The accurate dynamic modelling of contaminant dispersion in channels', *SIAM J. Appl. Math.* **55**(4), 1016–1038. http://epubs.siam.org/sam-bin/dbq/article/25797.

Wilson, S. K., Hunt, R. & Duffy, B. R. (2000), 'The rate of spreading in spin coating', *J. Fluid Mech.* **413**, 65–88.