Equation-Free function toolbox for Matlab/Octave:

Full Developers Manual

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

November 26, 2019

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

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

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

§ Appear here for your contribution.

Abstract

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

Contents

3 Patch scheme for given microscale discrete space system

Chapter contents

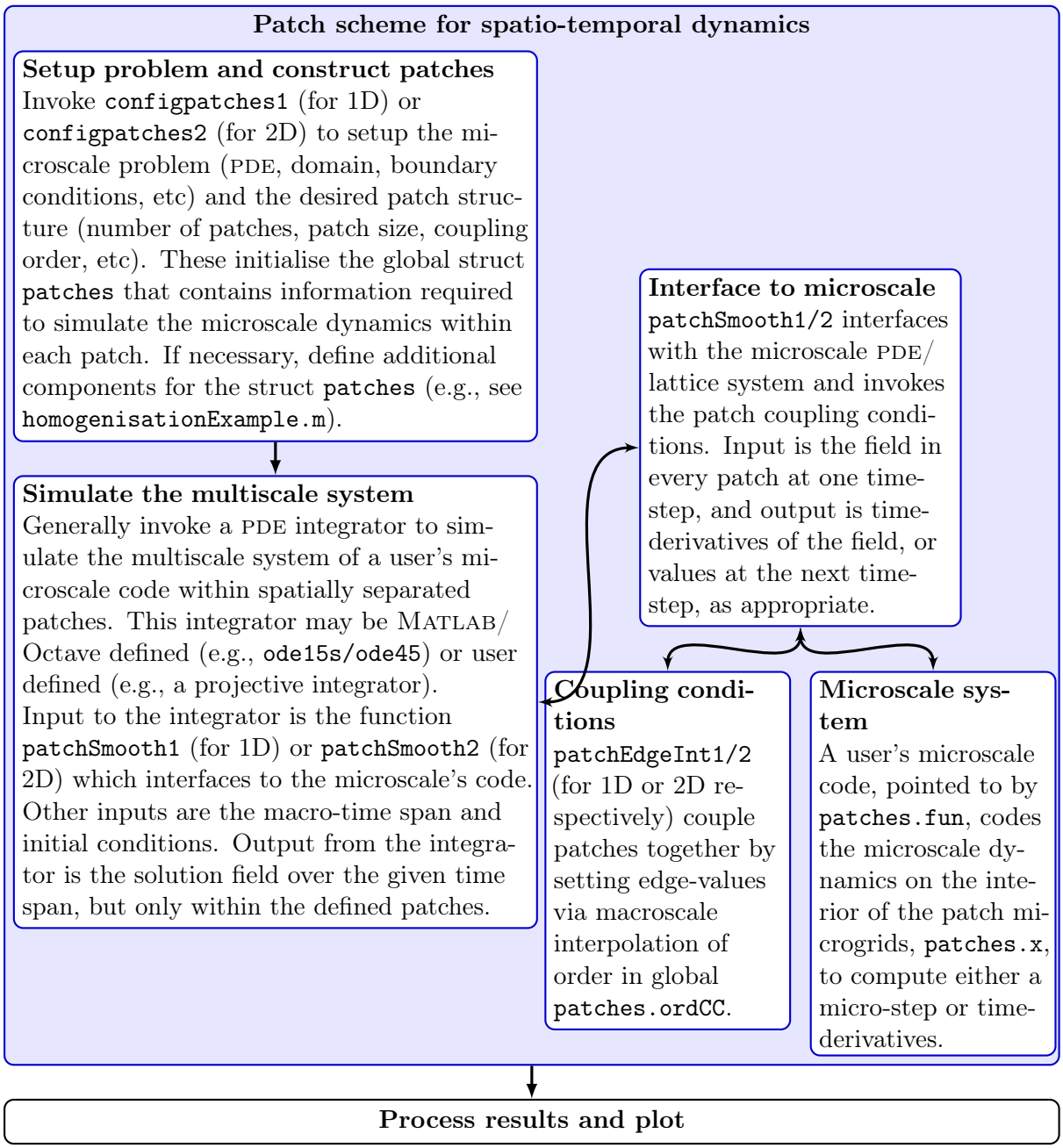
3.1 Introduction

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

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

Quick start See Sections 3.2.2 and 3.10.2 which respectively list example basic code that uses the provided functions to simulate the 1D Burgers' PDE, and a 2D nonlinear 'diffusion' PDE. Then see Figure 3.1.

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.*



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

Section contents

3.2.1 Introduction

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

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

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

- `fun` is the name of the user function, `fun(t,u,x)`, that computes time derivatives (or time-steps) of quantities on the patches.
- `Xlim` give the macro-space spatial domain of the computation: patches are equi-spaced over the interior of the interval [`Xlim(1)`, `Xlim(2)`].
- `BCs` somehow will define the macroscale boundary conditions. Currently, `BCs` is ignored and the system is assumed macro-periodic in the spatial domain.
- `nPatch` is the number of equi-spaced spaced patches.
- `ordCC`, must be ≥ -1, is the 'order' of interpolation across empty space of the macroscale mid-patch values to the edge of the patches for inter-patch coupling: where `ordCC` of 0 or -1 gives spectral interpolation; and `ordCC` being odd is for staggered spatial grids.
- `ratio` (real) is the ratio of the half-width of a patch to the spacing of the patch mid-points: so `ratio` $= \frac{1}{2}$ means the patches abut; `ratio` $= 1$ is overlapping patches as in holistic discretisation; and small `ratio` should greatly reduce computational time.
- `nSubP` is the number of equi-spaced microscale lattice points in each patch. Must be odd so that there is a central lattice point.
- `nEdge` (not yet implemented), *optional*, for each patch, the number of edge values set by interpolation at the edge regions of each patch. May be omitted. The default is one (suitable for microscale lattices with only nearest neighbour interactions).

- `patches.EdgyInt`, *optional*, if non-zero then interpolate to left/right edge-values from right/left next-to-edge values. So far only implemented for spectral interpolation, `ordCC` = 0.

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

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

3.2.2 If no arguments, then execute an example

```
109 if nargin==0
```

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

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

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

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

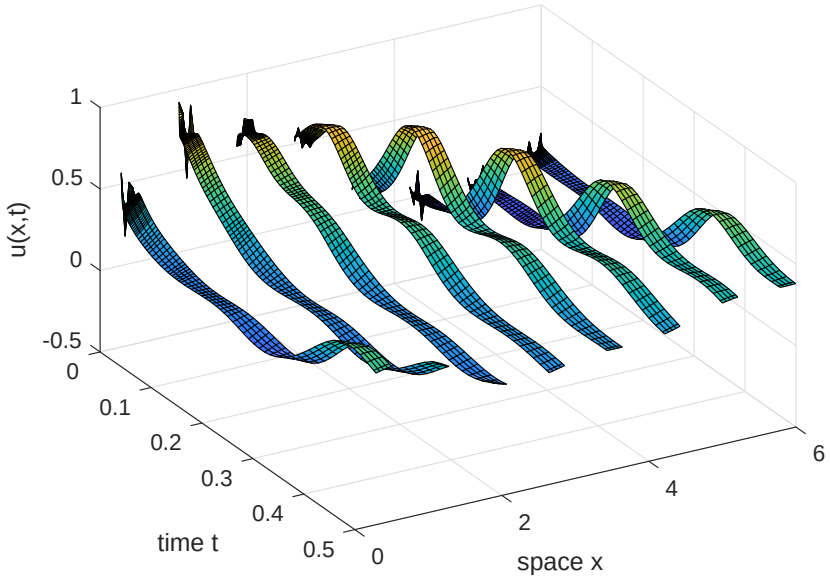
Set an initial condition, with some microscale randomness.

```
134 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 3.3).

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

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



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.

```
158 figure(1),clf
159 if 1, patches.x([1 end],:)=nan;  us=us.';
160 else us=reshape(patchEdgyInt1(us.'),[],length(ts));
161 end
162 surf(ts,patches.x(:),us), view(60,40)
163 title('Example of Burgers PDE on patches in space')
164 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
```

Upon finishing execution of the example, exit this function.

```
175 return
176 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 $\dot{u}_{ij} = \frac{1}{\delta x^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) - 30u_{ij}\frac{1}{2\delta x}(u_{i+1,j} - u_{i-1,j})$.

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

```
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','stiff');
15     xs = lsode(@(x,t) dxdt(t,x),x0,ts);
16 end
```

Check and set default edgy interpolation.

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

For compatibility, by default, do not ensemble average.

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

3.2.3 The code to make patches and interpolation

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

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

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

```
221 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.

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

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

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

Check for staggered grid and periodic case.

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

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

```
255 patches.Cwtsr=zeros(ordCC,1);
256 if patches.alt  % eqn (7) in \cite{Cao2014a}
```

```
257     patches.Cwtsr(1:2:ordCC)=[1 ...
258       cumprod((ratio^2-(1:2:(ordCC-2)).^2)/4)./ ...
259       factorial(2*(1:(ordCC/2-1)))];
260     patches.Cwtsr(2:2:ordCC)=[ratio/2 ...
261       cumprod((ratio^2-(1:2:(ordCC-2)).^2)/4)./ ...
262       factorial(2*(1:(ordCC/2-1))+1)*ratio/2];
263 else %
264     patches.Cwtsr(1:2:ordCC)=(cumprod(ratio^2- ...
265       (((1:(ordCC/2))-1)).^2)./factorial(2*(1:(ordCC/2))-1)/ratio);
266     patches.Cwtsr(2:2:ordCC)=(cumprod(ratio^2- ...
267       (((1:(ordCC/2))-1)).^2)./factorial(2*(1:(ordCC/2))));
268 end
269 patches.Cwtsl=(-1).^((1:ordCC)'-patches.alt).*patches.Cwtsr;
```

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

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

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

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

Fin.

3.3 `patchSmooth1()`: interface to time integrators

To simulate in time with spatial patches we often need to interface a user's time derivative function with time integration routines such as `ode15s` or `PIRK2`. This function provides an interface. It mostly assumes that the sub-patch structure is *smooth* so that the patch centre-values are sensible macroscale variables, and patch edge values are determined by macroscale interpolation of the patch-centre values. However, we have found that microscale heterogeneous systems may be accurately simulated with this function via appropriate interpolation. Communicate patch-design variables to this function using the previously established global struct `patches` (Section 3.2).

```
28 function dudt=patchSmooth1(t,u)
29 global patches
```

Input

- `u` is a vector of length `nSubP` · `nPatch` · `nVars` where there are `nVars` field values at each of the points in the `nSubP` × `nPatch` grid.
- `t` is the current time to be passed to the user's time derivative function.
- `patches` a struct set by `configPatches1()` with the following information used here.
  - `.fun` is the name of the user's function `fun(t,u,x)` that computes the time derivatives on the patchy lattice. The array `u` has size `nSubP` × `nPatch` × `nVars`. Time derivatives must be computed into the same sized array, although herein the patch edge values are overwritten by zeros.
  - `.x` is `nSubP` × `nPatch` array of the spatial locations x_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.

Output

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

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

```
79 u=patchEdgeInt1(u);
```

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

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

Fin.

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

Couples 1D patches across 1D space by computing their edge values from macroscale interpolation of either the mid-patch value, or the patch-core average, or the opposite next-to-edge values (this last choice often maintains symmetry). This function is primarily used by `patchSmooth1()` but is also useful for user graphics. A spatially discrete system could be integrated in time via the patch or gap-tooth scheme (Roberts & Kevrekidis 2007). When using core averages, assumes they are in some sense *smooth* so that these averages are sensible macroscale variables: then patch edge values are determined by macroscale interpolation of the core averages (Bunder et al. 2017). Communicates patch-design variables via the global struct `patches`.

```
28 function u=patchEdgeInt1(u)
29 global patches
```

Input

- `u` is a vector of length `nSubP` · `nPatch` · `nVars` where there are `nVars` field values at each of the points in the `nSubP` × `nPatch` grid.
- `patches` a struct largely set by `configPatches1()`, and which includes the following.
  - `.x` is `nSubP` × `nPatch` array of the spatial locations x_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.ordCC` is order of interpolation integer ≥ -1.
  - `.alt` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation.
  - `.Cwtsr` and `.Cwtsl` define the coupling.
  - `.EdgyInt` in $\{0, 1\}$ is one for interpolating patch-edge values from opposite next-to-edge values (often preserves symmetry).

Output

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

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

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

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

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

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

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

```
108 else r = dx*(nSubP-2)/DX;
109 end
```

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

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

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

```
127 i0 = round((nSubP+1)/2);
128 c = round((patches.nCore-1)/2);
```

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

```
138 if patches.ordCC>0 % then non-spectral interpolation
139   assert(patches.EdgyInt==0, ...
140   'Finite width not yet implemented for Edgy Interpolation')
141   if patches.EnsAve
142     uCore = sum(mean(u((i0-c):(i0+c),j,:),3),1)';
143     dmu = zeros(patches.ordCC,nPatch);
144   else
145     uCore = reshape(sum(u((i0-c):(i0+c),j,:),1),nPatch,nVars);
146     dmu = zeros(patches.ordCC,nPatch,nVars);
147   end;
148   if patches.alt % use only odd numbered neighbours
149     dmu(1,:,:) = (uCore(jp,:)+uCore(jm,:))/2; % \mu
150     dmu(2,:,:) = (uCore(jp,:)-uCore(jm,:)); % \delta
151     jp = jp(jp); jm = jm(jm); % increase shifts to \pm2
152   else % standard
153     dmu(1,j,:) = (uCore(jp,:)-uCore(jm,:))/2; % \mu\delta
154     dmu(2,j,:) = (uCore(jp,:)-2*uCore(j,:)+uCore(jm,:))/2; % \delta^2
155   end% if odd/even
```

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

```
163   for k = 3:patches.ordCC
164     dmu(k,:,:) = dmu(k-2,jp,:)-2*dmu(k-2,j,:)+dmu(k-2,jm,:);
165   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.

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

```
177       -sum(u((nSubP-patches.nCore+1):(nSubP-1),:,:),1);
178     u(1,j,:) = repmat(uCore(j)'*(1-patches.alt) ...
179       +sum(bsxfun(@times,patches.Cwtsl,dmu)),[1,1,nVars]) ...
180       -sum(u(2:patches.nCore,:,:),1);
181   else
182     u(nSubP,j,:) = uCore(j,:)*(1-patches.alt) ...
183       + reshape(-sum(u((nSubP-patches.nCore+1):(nSubP-1),j,:),1) ...
184       +sum(bsxfun(@times,patches.Cwtsr,dmu)),nPatch,nVars);
185     u(1,j,:) = uCore(j,:)*(1-patches.alt) ...
186       +reshape(-sum(u(2:patches.nCore,j,:),1)  ...
187       +sum(bsxfun(@times,patches.Cwtsl,dmu)),nPatch,nVars);
188   end;
```

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

```
195 else% spectral interpolation
```

As the macroscale fields are N-periodic, the macroscale Fourier transform writes the centre-patch values as $U_j = \sum_k C_k e^{ik2\pi j/N}$. Then the edge-patch values $U_{j\pm r} = \sum_k C_k e^{ik2\pi/N(j\pm r)} = \sum_k C'_k e^{ik2\pi j/N}$ where $C'_k = C_k e^{ikr2\pi/N}$. For `nPatch` patches we resolve 'wavenumbers' $|k| <$ `nPatch`$/2$, so set row vector `ks` $= k2\pi/N$ for 'wavenumbers' $k = (0, 1, \ldots, k_{\max}, -k_{\max}, \ldots, -1)$ 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??

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

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

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

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

```
244   if max(abs(imag(u(:))))<1e-9*max(abs(u(:)))
245        uclean=@(u) real(u);
```

```
246   else uclean=@(u) u;
247   end
```

Compute the Fourier transform of the patch centre-values for all the fields. If there are an even number of points, then if complex, treat as positive wavenumber, but if real, treat as cosine.

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

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

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

Restore staggered grid when appropriate.

```
278 if patches.alt
279   nVars = nVars/2;  nPatch = 2*nPatch;
280   v(:,1:2:nPatch,:) = u(:,:,1:nVars);
281   v(:,2:2:nPatch,:) = u(:,:,nVars+1:2*nVars);
282   u = v;
283 end
284 end% if spectral
```

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

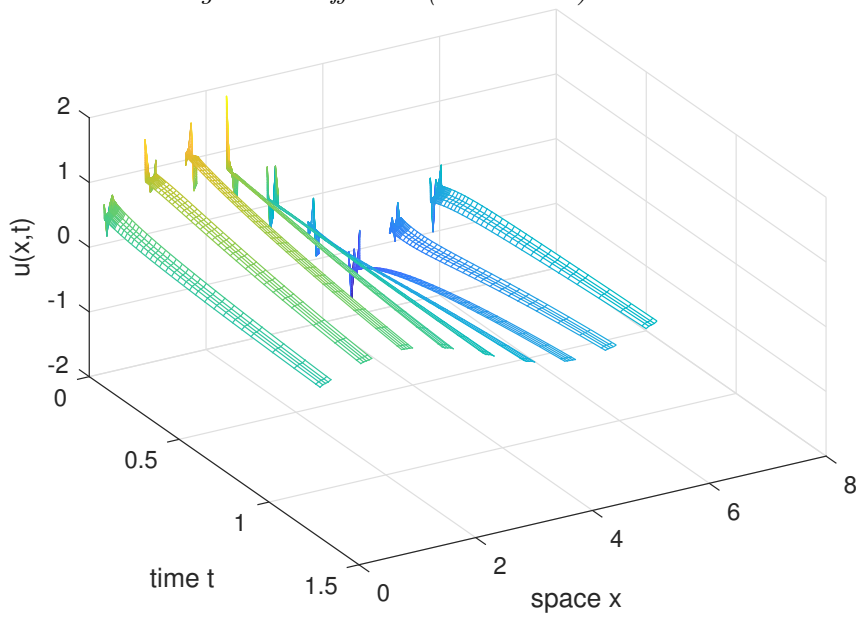
3.5 `homogenisationExample`: simulate heterogeneous diffusion in 1D on patches

Section contents

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

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

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



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.

3.5.1 Script to simulate via stiff or projective integration

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

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

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

```
global patches
nPatch = 9
ratio = 0.2
nSubP = 2*mPeriod+1
```

```
69 Len = 2*pi;
70 ordCC = 4;
71 configPatches1(@heteroDiff,[0 Len],nan,nPatch ...
72     ,ordCC,ratio,nSubP);
```

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

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

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

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

Plot the simulation in Figure 3.3.

```
107 figure(1),clf
108 xs = patches.x;  xs([1 end],:) = nan;
109 mesh(ts,xs(:),ucts'),  view(60,40)
110 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
111 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
112 %print('-depsc2','homogenisationCtsU')
```

The code may invoke this integration interface.

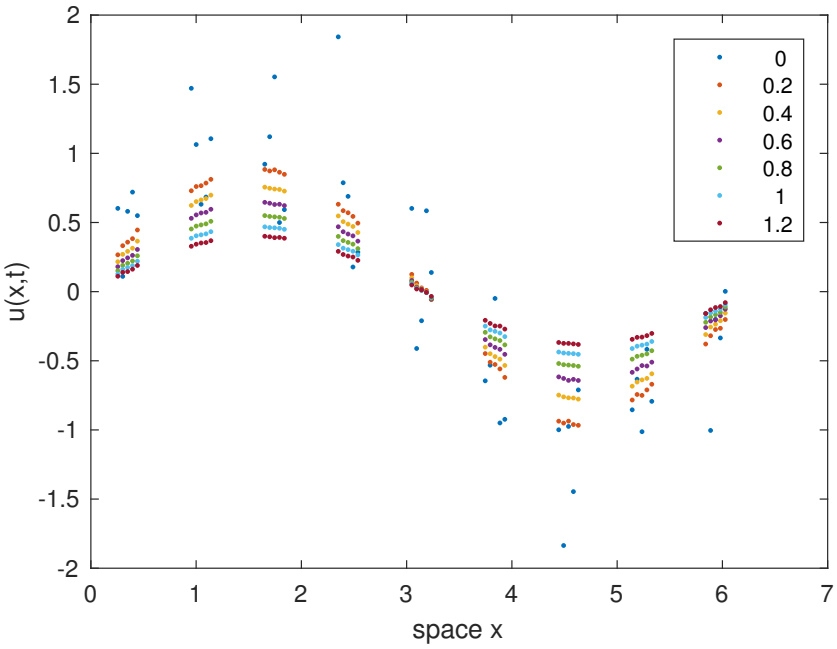
```
10 function [ts,xs] = odeOcts(dxdt,tSpan,x0)
11     if length(tSpan)>2, ts = tSpan;
12     else ts = linspace(tSpan(1),tSpan(end),21)';
13     end
14     lsode_options('integration method','stiff');
15     xs = lsode(@(x,t) dxdt(t,x),x0,ts);
16 end
```

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 3.5.3), as illustrated by Figure 3.4.

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

1. configPatches1 (done in first part)

Figure 3.4: 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.



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.

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

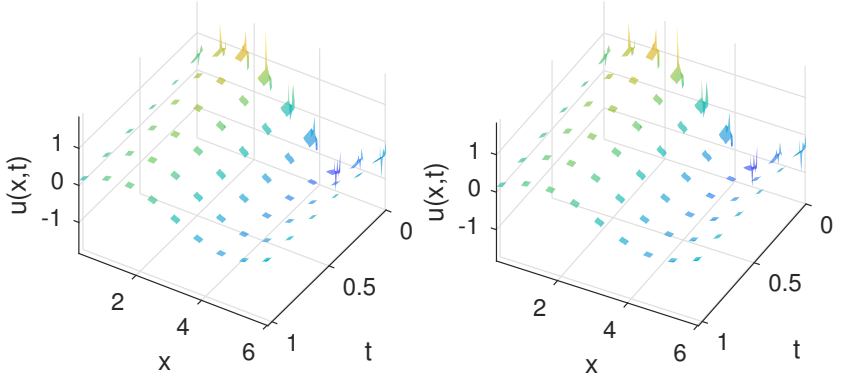
Set the desired macro- and microscale time-steps over the time domain: the macroscale step is in proportion to the effective mean diffusion time on the macroscale; the burst time is proportional to the intra-patch effective diffusion time; and lastly, the microscale time-step is proportional to the diffusion time between adjacent points in the microscale lattice.

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

Plot the macroscale predictions to draw Figure 3.4.

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

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.



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

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

End of this example script.

3.5.2 `heteroDiff()`: heterogeneous diffusion

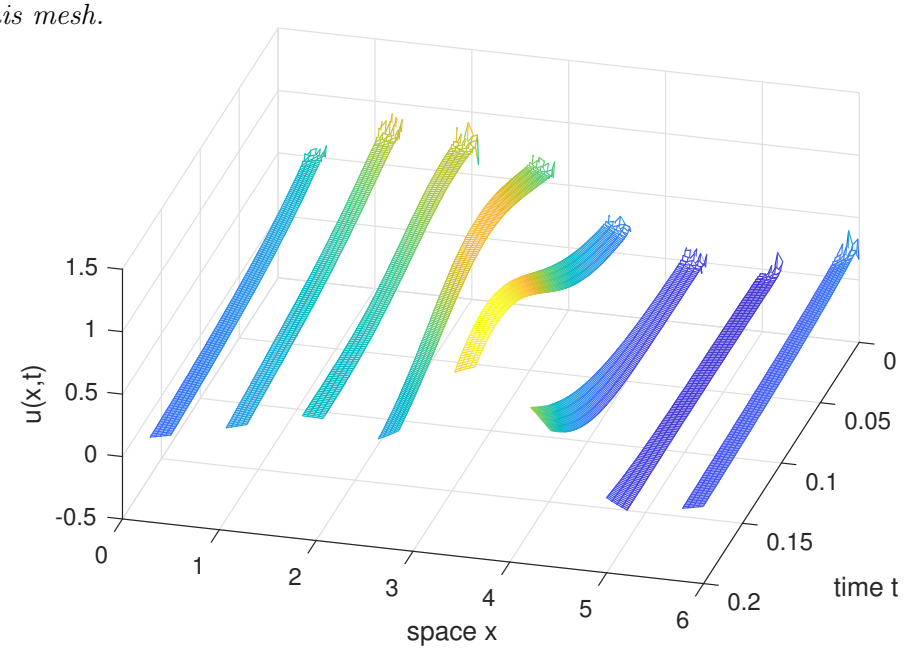
This function codes the lattice heterogeneous diffusion inside the patches. For 2D input arrays `u` and `x` (via edge-value interpolation of `patchSmooth1`, Section 3.3), computes the time derivative (3.1) at each point in the interior of a patch, output in `ut`. The column vector (or possibly array) of diffusion coefficients c_i have previously been stored in struct `patches`.

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

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

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

Figure 3.6: 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.




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

Fin.

3.6 `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 3.6), 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.

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

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

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

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

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

Save the plot to file to form Figure 3.6.

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

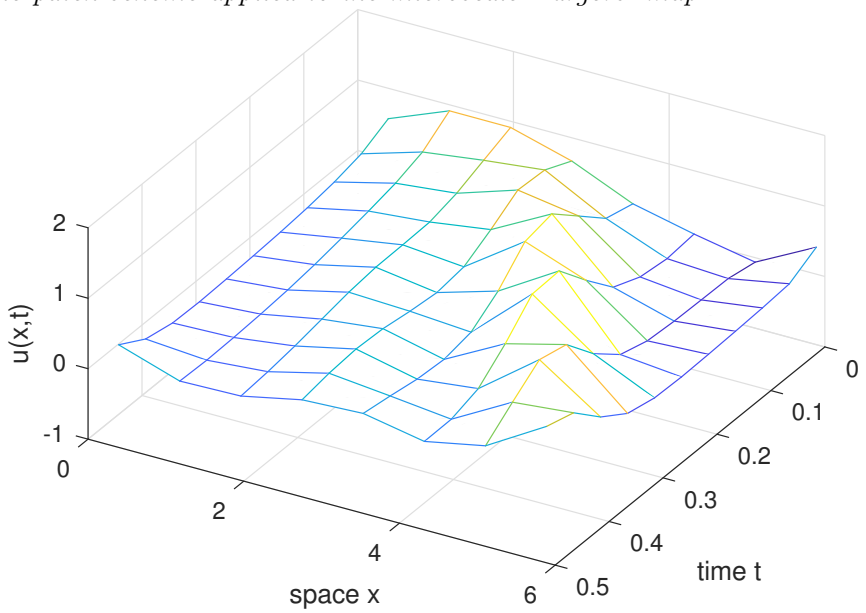
3.6.2 Alternatively use projective integration

Around the microscale burst `burgerBurst()`, wrap the projective integration function `PIRK2()` of Section 2.2. Figure 3.7 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 ↔ burgerBurst ↔ patchSmooth1 ↔ burgersMap
3. process results

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



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

```
117 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.

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

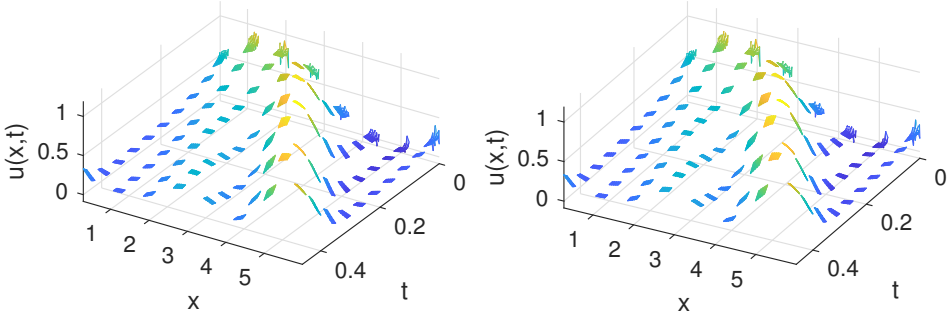
Plot and save the macroscale predictions of the mid-patch values to give the macroscale mesh-surface of Figure 3.7 that shows a progressing wave solution.

```
137 figure(2),clf
138 midP = (nSubP+1)/2;
139 mesh(ts,xs(midP,:),us(:,midP:nSubP:end)')
140 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
141 view(120,50)
142 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
143 %print('-depsc2','BurgersU')
```

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

```
158 figure(3),clf
159 for k = 1:2, subplot(2,2,k)
```

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




```
160   mesh(tss,xs(:),uss')
161   ylabel('x'),xlabel('t'),zlabel('u(x,t)')
162   axis tight, view(126-4*k,50)
163 end
164 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
165 %print('-depsc2','BurgersMicro')
```

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).

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

3.6.4 `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 3.3) to apply the microscale map over all time-steps in the burst. The `patchSmooth1()` interface provides the interpolated edge-values of each patch. Store the results in rows to be consistent with ODE and projective integrators.

```
29   us = nan(ndt+1,numel(ui));
30   us(1,:) = reshape(ui,1,[]);
```

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

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

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

Fin.

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

Section contents

3.7.1 Introduction

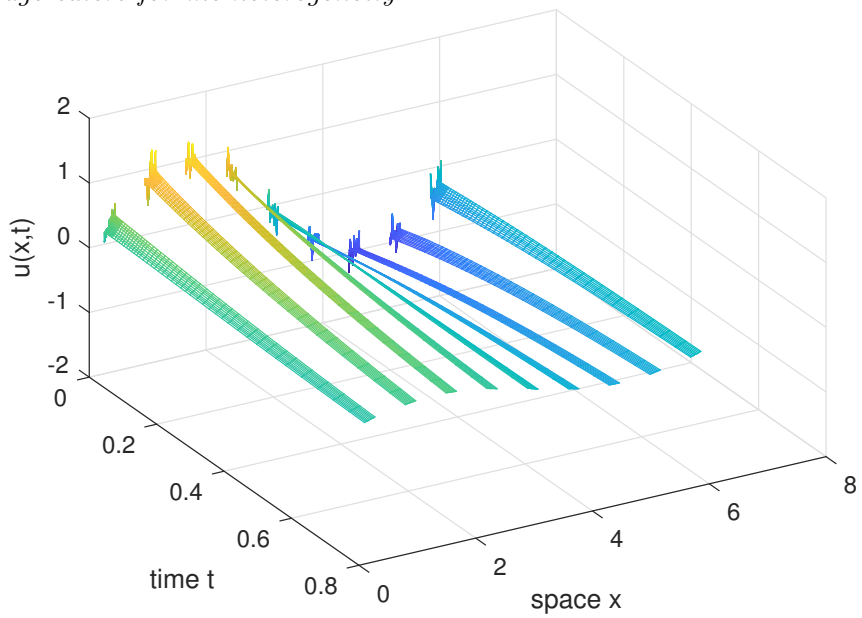
This example is an extension of the homogenisation example of Section 3.5 for heterogeneous diffusion. In cases where the periodicity of the heterogeneous diffusion is known, then Section 3.5 provides a efficient patch dynamics simulation. However, if the diffusion is not completely known or is stochastic,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

For a first comparison, we present a very similar example to that of Section 3.5, but whereas Section 3.5 simulates using only one diffusivity configuration, here we simulate over an ensemble. For example, Figure 3.3 is similar to Figure 3.9, but the latter is an average of an ensemble of eight different simulations with different diffusivity configurations, whereas the former is simulated from just one diffusivity configuration. The main difference between these two is that the average over the ensemble caters for the heterogeneity in the problem.

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

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

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



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

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

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

3.7.2 Script to simulate via stiff or projective integration

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

```
clear all
mPeriod = 4
rand('seed',1);
c = exp(4*rand(mPeriod,1))
cHomo = 1/mean(1./c)
```

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

```
global patches
nPatch = 9
ratio = 0.2
nSubP = 11
Len = 2*pi;
```

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

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

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

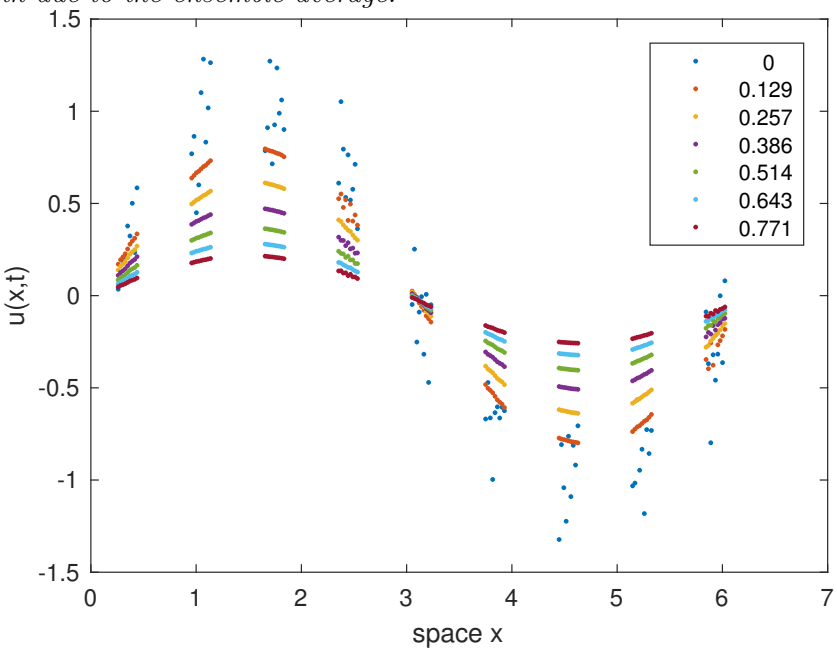
Conventional integration in time Set an initial condition, and here integrate forward in time using a standard method for stiff systems. Integrate the interface `patchSmooth1()` (Section 3.3) to the microscale differential equations.

```
u0 = sin(patches.x)+0.2*randn(nSubP,nPatch);
if patches.EnsAve
  u0 = repmat(u0,[1,1,nVars]);
end
if ~exist('OCTAVE_VERSION','builtin')
    [ts,ucts] = ode15s( @patchSmooth1, [0 2/cHomo], u0(:));
else % octave version is slower
    [ts,ucts] = odeOcts(@patchSmooth1, [0 2/cHomo], u0(:));
end
ucts = reshape(ucts,length(ts),length(patches.x(:)),[]);
```

Plot the ensemble averaged simulation in Figure 3.9.

```
if patches.EnsAve % calculate the ensemble average
  uctsAve = mean(ucts,3);
else
  uctsAve = ucts;
```

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




```
161 end
162 figure(1),clf
163 xs = patches.x;  xs([1 end],:) = nan;
164 mesh(ts,xs(:),uctsAve'),  view(60,40)
165 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
166 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
167 %print('-depsc2','ensAveExCtsU')
```

Use projective integration in time Now consider the interface, `patchSmooth1()`, to the time derivatives, and wrap around it the projective integration `PIRK2` (Section 2.2), of bursts of simulation from `heteroBurst` (Section 3.5.3), as illustrated by Figure 3.10. The rest of this code follows that of Section 3.5, but as we now evaluate an ensemble of field solutions, our final step is always an ensemble average.

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

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

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

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

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

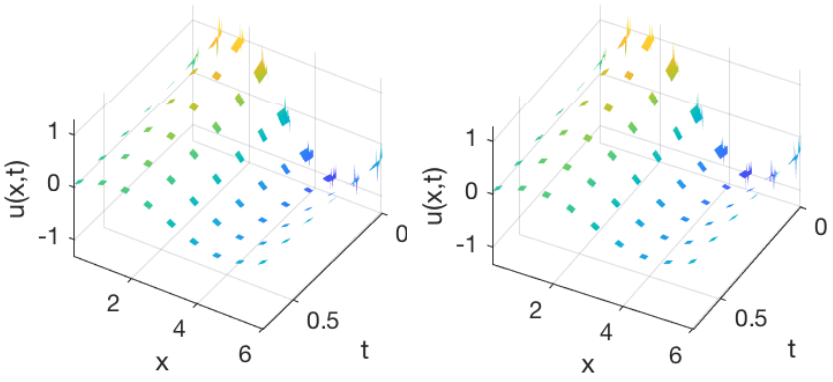


Figure 3.10 shows an average of the ensemble of macroscale predictions.

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

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

```
figure(3),clf
for k = 1:2, subplot(1,2,k)
  surf(tss,xs(:),ussAve',  'EdgeColor','none')
  ylabel('x'), xlabel('t'), zlabel('u(x,t)')
  axis tight, view(126-4*k,45)
end
set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
%print('-depsc2','ensAveExMicro')
```

End of the script.

Sections 3.5.2 and 3.5.3 list the addtional functions used by this script. Fin.

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

Section contents

Figure 3.12: water depth $h(x,t)$ (above) and velocity field $u(x,t)$ (below) of the gap-tooth scheme applied to the ideal 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.*

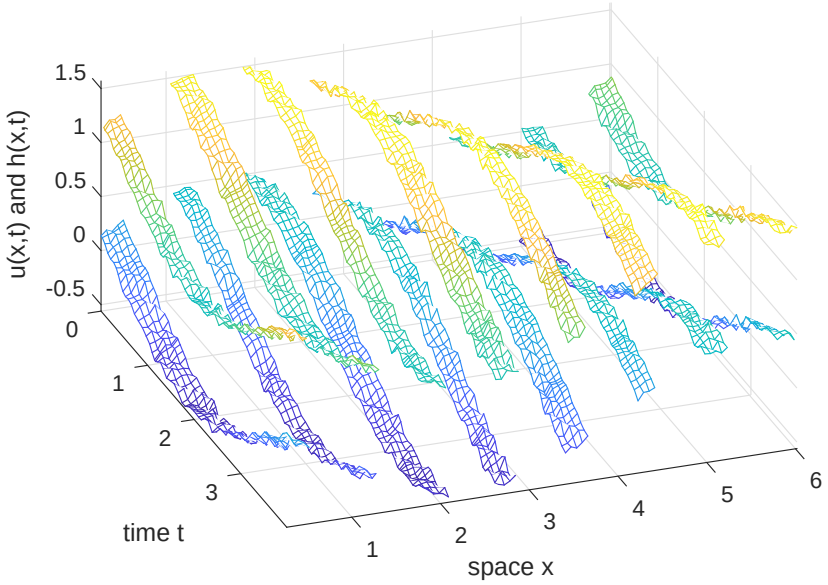


Figure 3.12 shows an example simulation in time generated by the patch scheme applied to an ideal wave PDE (Cao & Roberts 2013). The inter-patch coupling is realised by spectral interpolation of the mid-patch values to the patch edges.

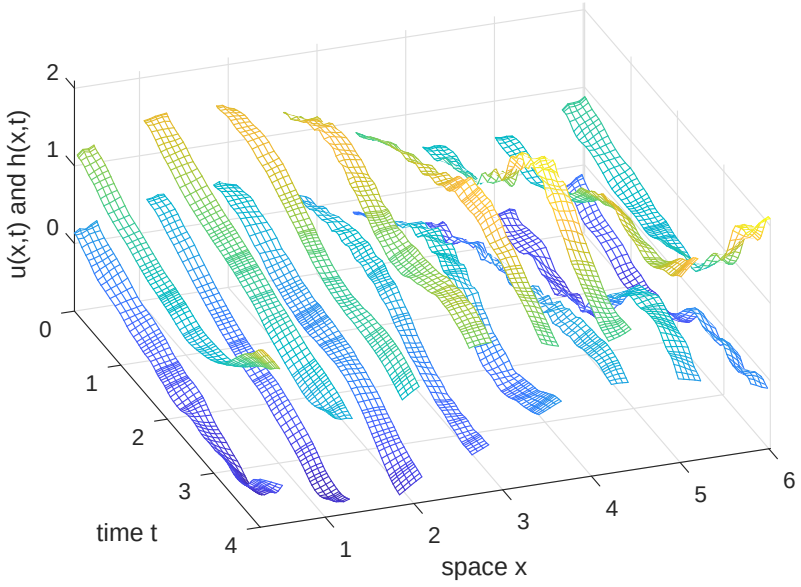
This approach, based upon the differential equations coded in Section 3.8.2, may be adapted by a user to a wide variety of 1D wave and wave-like systems. For example, the differential equations of Section 3.8.3 that describe the nonlinear microscale simulator of the nonlinear shallow water wave PDE derived 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

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

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 3.8.3 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

Figure 3.13: water depth $h(x,t)$ (above) and velocity field $u(x,t)$ (below) of the gap-tooth scheme applied to the Smagorinski shallow water wave PDEs *(3.4). The microscale random initial component decays where the water speed is non-zero due to 'turbulent' dissipation.*



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

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.

3.8.1 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.

```
clear all
global patches
nPatch = 8
ratio = 0.2
nSubP = 11 %of the form 4*n-1
Len = 2*pi;
configPatches1(@idealWavePDE,[0 Len],nan,nPatch,-1,ratio,nSubP);
```

Identify which micro-grid points are h or u values on the staggered micro-grid. Also store the information in the struct `patches` for use by the time derivative function.

```
uPts = mod( bsxfun(@plus,(1:nSubP)',(1:nPatch)) ,2);
hPts = find(uPts==0);
uPts = find(uPts==1);
patches.hPts = hPts; patches.uPts = uPts;
```

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

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

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

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

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

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

Plot the simulation.

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

Optionally save the plot to file.

```
193 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
194 %  if k==1, print('-depsc2','ps1WaveCtsUH')
195 %  else print('-depsc2','ps1WaterWaveCtsUH')
196 %  end
```

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

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

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

3.8.2 `idealWavePDE()`: ideal wave PDE

This function codes the staggered lattice equation inside the patches for the ideal wave PDE system $h_t = -u_x$ and $u_t = -h_x$. Here code for a staggered 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.

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

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

```
33   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.

```
45   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.

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

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

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

3.8.3 `waterWavePDE()`: water wave PDE

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

```
16  function Ut = waterWavePDE(t,U,x)
17    global patches
18    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.

```
26    dx = diff(x(2:3));
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,:);
```

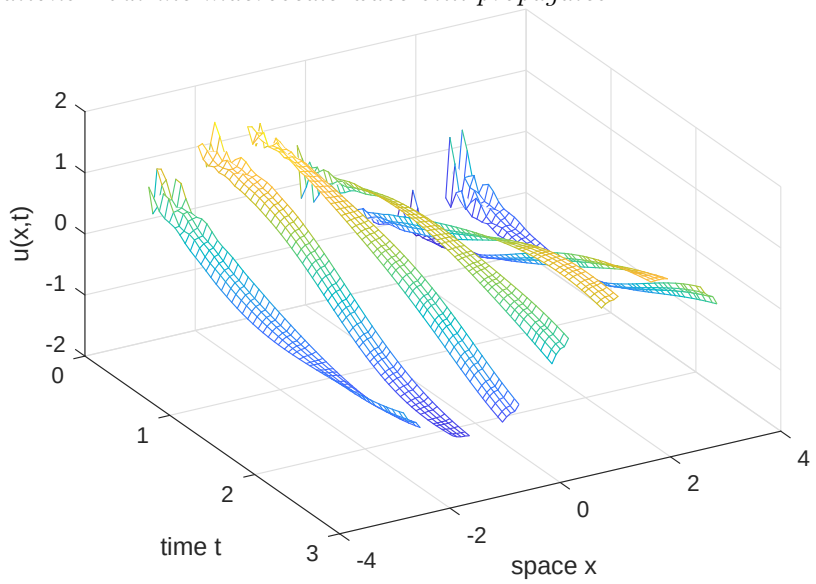
Then estimate $\partial(hu)/\partial x$ from u and the interpolated h at the neighbouring micro-grid points.

```
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;
```

Figure 3.14: 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.



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
```

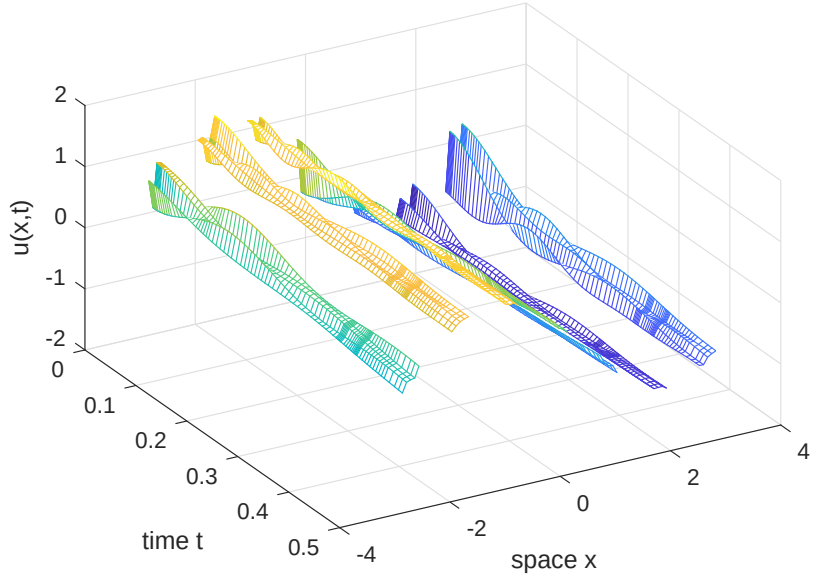
Fin.

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

Figure 3.14 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,

Figure 3.15: 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.



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 3.14 shows one patch simulation of this system: observe the effects of the heterogeneity within each patch.

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 ↔ 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.

```
76 clear all
77 mPeriod = 3
```

```
78 cHetr = exp(1*randn(mPeriod,1));
79 cHetr = cHetr*mean(1./cHetr) % normalise
80 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).

```
93 global patches
94 nPatch = 7
95 ratio = 0.25
96 nSubP = nPeriodsPatch*mPeriod+2
97 patches.EdgyInt = 1; % one to use edges for interpolation
98 configPatches1(@heteroWave,[-pi pi],nan,nPatch ...
99     ,0,ratio,nSubP);
```

Replicate the heterogeneous coefficients across the width of each patch.

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

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

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

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

```
121 if ~exist('OCTAVE_VERSION','builtin')
122     [ts,us] = ode15s(@patchSmooth1, [0 3], [u0(:);v0(:)]);
123 else % octave version
124     [ts,us] = odeOcts(@patchSmooth1, [0 3], [u0(:);v0(:)]);
125 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.

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

Now plot two space-time graphs. The first is every time step over a meso-time to see the oscillation and decay of the fast sub-patch waves. The second is

subsampled surface over the macroscale duration of the simulation to show the propagation of the macroscale wave over the heterogeneous lattice.

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

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

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

Find the eigenvalues of the Jacobian, and list for inspection in Table 3.1.

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

End of the main script.

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

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

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

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

Table 3.1: 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
```

```
27 function ut = heteroWave(t,u,x)
28   global patches
29   dx = diff(x(2:3)); % space step
30   i = 2:size(u,1)-1; % interior points in a patch
31   ut = nan(size(u)); % preallocate output array
32   ut(i,:,1) = u(i,:,2); % du/dt=v then dvdt=
33   ut(i,:,2) = diff(patches.c.*diff(u(:,:,1)))/dx^2 ...
34         +0.02*diff(u(:,:,2),2)/dx^2;
35 end% function
```

Fin.

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

Section contents

3.10.1 Introduction

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSmooth2()`. Section 3.10.2 lists an example of its use.

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

Input If invoked with no input arguments, then executes an example of simulating a nonlinear diffusion PDE relevant to the lubrication flow of a thin layer of fluid—see Section 3.10.2 for the example code.

- `fun` is the name of the user function, `fun(t,u,x,y)`, that computes time-derivatives (or time-steps) of quantities on the 2D micro-grid within all the 2D patches.
- `Xlim` array/vector giving the macro-space domain of the computation: patches are distributed equi-spaced over the interior of the rectangle $[\texttt{Xlim(1)}, \texttt{Xlim(2)}] \times [\texttt{Xlim(3)}, \texttt{Xlim(4)}]$: if `Xlim` is of length two, then the domain is the square of the same interval in both directions.
- `BCs` eventually will define the macroscale boundary conditions. Currently, `BCs` is ignored and the system is assumed macro-periodic in the domain.
- `nPatch` determines the number of equi-spaced spaced patches: if scalar, then use the same number of patches in both directions, otherwise `nPatch(1:2)` gives the number of patches in each direction.
- `ordCC` is the 'order' of interpolation for inter-patch coupling across empty space of the macroscale mid-patch values to the edge-values of the patches: currently must be 0; where 0 gives spectral interpolation.
- `ratio` (real) is the ratio of the half-width of a patch to the spacing of the patch mid-points: so `ratio` $= \frac{1}{2}$ means the patches abut; `ratio` $= 1$ would be overlapping patches as in holistic discretisation; and small `ratio` should greatly reduce computational time. If scalar, then use the same ratio in both directions, otherwise `ratio(1:2)` gives the ratio in each direction.
- `nSubP` is the number of equi-spaced microscale lattice points in each patch: if scalar, then use the same number in both directions, otherwise `nSubP(1:2)` gives the number in each direction. Must be odd so that there is a central micro-grid point in each patch.
- `nEdge`, (not yet implemented) *optional*, is the number of edge values set by interpolation at the edge regions of each patch. The default is one (suitable for microscale lattices with only nearest neighbours. interactions).

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

- `.fun` is the name of the user's function `fun(t,u,x,y)` that computes the time derivatives (or steps) on the patchy lattice.
- `.ordCC` is the specified order of inter-patch coupling.

- `.alt` is true for interpolation using only odd neighbouring patches as for staggered grids, and false for the usual case of all neighbour coupling—not yet implemented.
- `.Cwtsr` and `.Cwtsl` are the `ordCC`-vector of weights for the inter-patch interpolation onto the right and left edges (respectively) with patch:macroscale ratio as specified—not yet implemented.
- `.x` is `nSubP(1)` × `nPatch(1)` array of the regular spatial locations x_{ij} of the microscale grid points in every patch.
- `.y` is `nSubP(2)` × `nPatch(2)` array of the regular spatial locations y_{ij} of the microscale grid points in every patch.
- `.nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.

3.10.2 If no arguments, then execute an example

```
132 if nargin==0
```

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

1. configPatches2
2. ode15s integrator ↔ patchSmooth2 ↔ user's PDE
3. process results

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

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

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

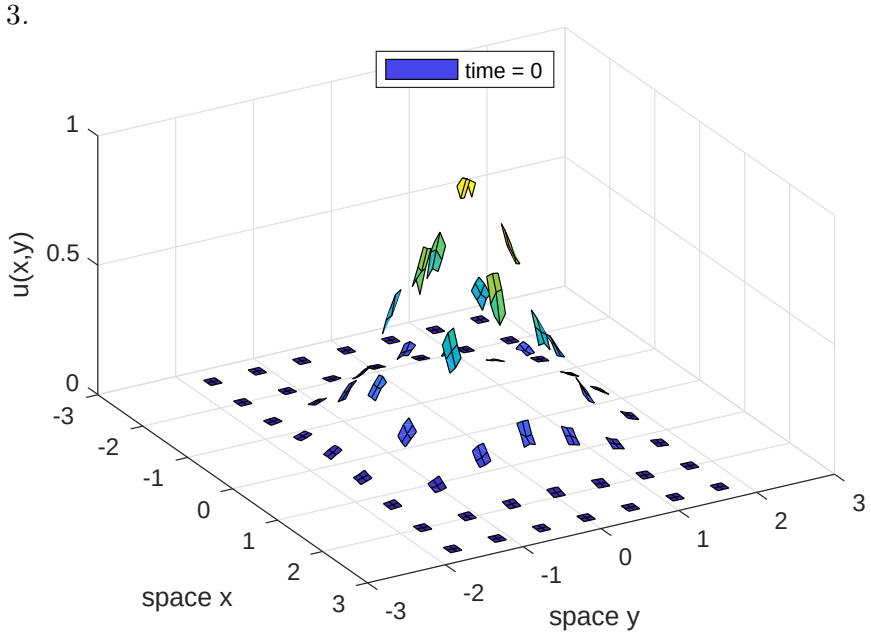
```
163 x = reshape(patches.x,nSubP,1,[],1);
164 y = reshape(patches.y,1,nSubP,1,[]);
165 u0 = exp(-x.^2-y.^2);
166 u0 = u0.*(0.9+0.1*rand(size(u0)));
```

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

```
174 figure(1), clf
175 x = patches.x; y = patches.y;
176 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.

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 $t = 3$.




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

Save the initial condition to file for Figure 3.16.

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

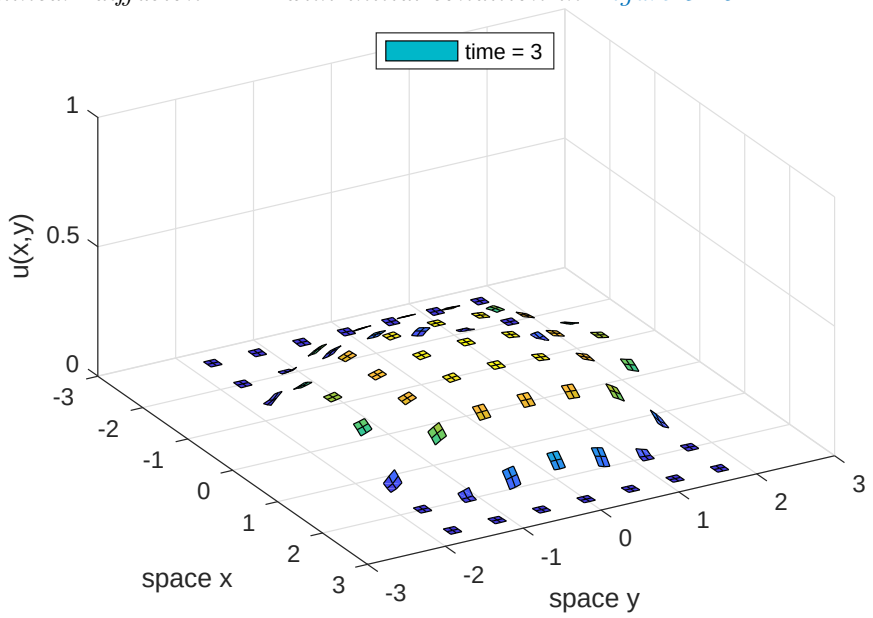
Integrate in time using standard functions.

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

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

```
for i = 1:length(ts)
  u = patchEdgeInt2(us(i,:));
  u = reshape(permute(u,[1 3 2 4]), [numel(x) numel(y)]);
  set(hsurf,'ZData', u');
```

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.




```
229   legend(['time = ' num2str(ts(i),2)])
230   pause(0.1)
231 end
232 %print('-depsc2','configPatches2t3')
```

Upon finishing execution of the example, exit this function.

```
247 return
248 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,x,y)
14   dx = diff(x(1:2));  dy = diff(y(1:2));  % microgrid spacing
15   i = 2:size(u,1)-1;  j = 2:size(u,2)-1;  % interior patch points
16   ut = nan(size(u));  % preallocate storage
17   ut(i,j,:,:) = diff(u(:,j,:,:).^3,2,1)/dx^2 ...
18                +diff(u(i,:,:,:).^3,2,2)/dy^2;
19 end
```

3.10.3 The code to make patches

Initially duplicate parameters for both space dimensions as needed.

```
267 if numel(Xlim)==2, Xlim = repmat(Xlim,1,2); end
268 if numel(nPatch)==1, nPatch = repmat(nPatch,1,2); end
269 if numel(ratio)==1, ratio = repmat(ratio,1,2); end
270 if numel(nSubP)==1, nSubP = repmat(nSubP,1,2); end
```

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

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

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

```
patches.fun = fun;
```

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

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

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

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

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

```
ratio = ratio(:)'; % force to be row vector
if patches.alt  % eqn (7) in \cite{Cao2014a}
  patches.Cwtsr = [1
    ratio/2
    (-1+ratio.^2)/8
    (-1+ratio.^2).*ratio/48
    (9-10*ratio.^2+ratio.^4)/384
    (9-10*ratio.^2+ratio.^4).*ratio/3840
    (-225+259*ratio.^2-35*ratio.^4+ratio.^6)/46080
    (-225+259*ratio.^2-35*ratio.^4+ratio.^6).*ratio/645120 ];
else %
  patches.Cwtsr = [ratio
    ratio.^2/2
    (-1+ratio.^2).*ratio/6
    (-1+ratio.^2).*ratio.^2/24
    (4-5*ratio.^2+ratio.^4).*ratio/120
    (4-5*ratio.^2+ratio.^4).*ratio.^2/720
    (-36+49*ratio.^2-14*ratio.^4+ratio.^6).*ratio/5040
    (-36+49*ratio.^2-14*ratio.^4+ratio.^6).*ratio.^2/40320 ];
end
patches.Cwtsr = patches.Cwtsr(1:ordCC,:);
% maybe should avoid this next implicit auto-replication
patches.Cwtsl = (-1).^((1:ordCC)'-patches.alt).*patches.Cwtsr;
```

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

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

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

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

Fin.

3.11 `patchSmooth2()`: interface to time integrators

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

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

Input

- `u` is a vector of length `prod(nSubP)` · `prod(nPatch)` · `nVars` where there are `nVars` field values at each of the points in the `nSubP(1)` × `nSubP(2)` × `nPatch(1)` × `nPatch(2)` grid.
- `t` is the current time to be passed to the user's time derivative function.
- `patches` a struct set by `configPatches2()` with the following information used here.
  - `.fun` is the name of the user's function `fun(t,u,x,y)` that computes the time derivatives on the patchy lattice. The array `u` has size `nSubP(1)` × `nSubP(2)` × `nPatch(1)` × `nPatch(2)` × `nVars`.

Time derivatives must be computed into the same sized array, but herein the patch edge-values are overwritten by zeros.

- `.x` is `nSubP(1)` × `nPatch(1)` array of the spatial locations x_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
- `.y` is similarly `nSubP(2)` × `nPatch(2)` array of the spatial locations y_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.

Output

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

Reshape the fields `u` as a 4/5D-array, and sets the edge values from macroscale interpolation of centre-patch values. Section 3.12 describes `patchEdgeInt2()`.

```
83 u = patchEdgeInt2(u);
```

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

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

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. Assumes that the sub-patch structure is *smooth* so that the patch centre-values are sensible macroscale variables, and patch edge values are determined by macroscale interpolation of the patch-centre values. Communicate patch-design variables via the global struct `patches`.

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

Input

- `u` is a vector of length `nx` · `ny` · `Nx` · `Ny` · `nVars` where there are `nVars` field values at each of the points in the `nx` × `ny` × `Nx` × `Ny` grid on the `Nx` × `Ny` array of patches.
- `patches` a struct set by `configPatches2()` which includes the following information.

- `.x` is `nx` × `Nx` array of the spatial locations x_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
- `.y` is similarly `ny` × `Ny` array of the spatial locations y_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
- `.ordCC` is order of interpolation, currently only {0}.
- `.Cwtsr` and `.Cwtsl`—not yet used

Output

- `u` is `nx` × `ny` × `Nx` × `Ny` × `nVars` array of the fields with edge values set by interpolation.

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

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

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

```
dx = patches.x(3,1)-patches.x(2,1);
DX = patches.x(2,2)-patches.x(2,1);
rx = dx*(nx-1)/2/DX;
dy = patches.y(3,1)-patches.y(2,1);
DY = patches.y(2,2)-patches.y(2,1);
ry = dy*(ny-1)/2/DY;
```

For the moment assume the physical domain is macroscale periodic so that the coupling formulas are simplest. Should eventually cater for periodic, odd-mid-gap, even-mid-gap, even-mid-patch, Dirichlet, Neumann, Robin?? These index vectors point to patches and their two immediate neighbours—currently not needed.

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

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

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

Lagrange interpolation gives patch-edge values —not yet implemented So compute centred differences of the mid-patch values for the macro-interpolation, of all fields. Assumes the domain is macro-periodic.

```
129 if patches.ordCC>0 % then non-spectral interpolation
130 error('non-spectral interpolation not yet implemented')
131   dmu=nan(patches.ordCC,nPatch,nVars);
132 %  if patches.alt % use only odd numbered neighbours
133 %    dmu(1,:,:)=(u(i0,jp,:)+u(i0,jm,:))/2; % \mu
134 %    dmu(2,:,:)= u(i0,jp,:)-u(i0,jm,:); % \delta
135 %    jp=jp(jp); jm=jm(jm); % increase shifts to \pm2
136 %  else % standard
137      dmu(1,:,:)=(u(i0,jp,:)-u(i0,jm,:))/2; % \mu\delta
138      dmu(2,:,:)=(u(i0,jp,:)-2*u(i0,j,:)+u(i0,jm,:)); % \delta^2
139 %  end% if odd/even
```

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

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

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

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

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

```
182 else% spectral interpolation
```

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

```
192 %  if patches.alt % transform by doubling the number of fields
193 %  error('staggered grid not yet implemented')
194 %    v=nan(size(u)); % currently to restore the shape of u
195 %    u=cat(3,u(:,1:2:nPatch,:),u(:,2:2:nPatch,:));
```

```
196 %    altShift=reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
197 %    iV=[nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
198 %    r=r/2;           % ratio effectively halved
199 %    nPatch=nPatch/2; % halve the number of patches
200 %    nVars=nVars*2;   % double the number of fields
201 %  else % the values for standard spectral
202      altShift = 0;
203      iV = 1:nVars;
204 %  end
```

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

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

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

```
223   if imag(u(i0,j0,:,:,:))==0, uclean = @(u) real(u);
224                             else uclean = @(u) u; end
```

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

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

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

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

Loop over the required IFFTs.

```
255 iFT = 0;
256 for iFTx = 1:nFTx
257 for iFTy = 1:nFTy
258 iFT = iFT+1;
```

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

```
267 for jj = 1:ny
268   ks = (jj-j0)*2/(ny-1)*kry; % fraction of kry along the edge
```

```
269   unj(1,jj,:,:,iV,iFT) = ifft2( bsxfun(@times,Ck ...
270       ,exp(1i*bsxfun(@plus,altShift+krx',ks))));
271   u1j(1,jj,:,:,iV,iFT) = ifft2( bsxfun(@times,Ck ...
272       ,exp(1i*bsxfun(@plus,altShift-krx',ks))));
273 end
```

Second interpolate onto y-limits of the patches.

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

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

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

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

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

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

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

Fin, returning the 4/5D 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 ↔ patchSmooth2 ↔ wavePDE
3. process results

Establish the global data struct `patches` to interface with a function coding the wave PDE: to be solved on 2π-periodic domain, with 9×9 patches, spectral interpolation (0) couples the patches, each patch of half-size ratio 0.25 (big enough for visualisation), and with a 5×5 micro-grid within each patch.

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

3.13.1 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.

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

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

```
small = 1;
jac = nan(length(i));
sizeJacobian = size(jac)
for j = 1:length(i)
  uv = uv0(:);
```

```
66    uv(i(j)) = uv(i(j))+small;
67    tmp = patchSmooth2(0,uv)/small;
68    jac(:,j) = tmp(i);
69  end
```

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

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

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

```
89  kwave = 0:(nPatch-1)/2;
90  freq = sort(reshape(sqrt(kwave'.^2+kwave.^2),1,[]));
91  freq = freq(diff([-1 freq])>1e-9);
92  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: $n_x \times n_y \times N_x \times N_y$.

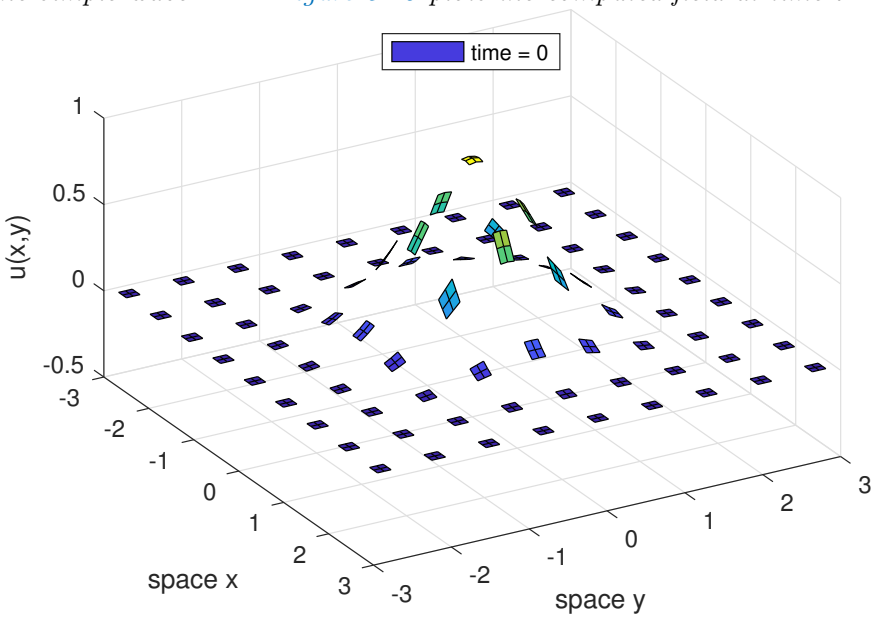
```
107 x = reshape(patches.x,nSubP,1,[],1);
108 y = reshape(patches.y,1,nSubP,1,[]);
109 u0 = exp(-x.^2-y.^2);
110 v0 = zeros(size(u0));
```

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 = patches.x; y = patches.y;
123 x([1 end],:) = nan; y([1 end],:) = nan;
124 u = reshape(permute(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 drawnow
130 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
131 %print('-depsc','wave2Dic')
```

Integrate in time using standard functions.

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 $t=2$.*




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

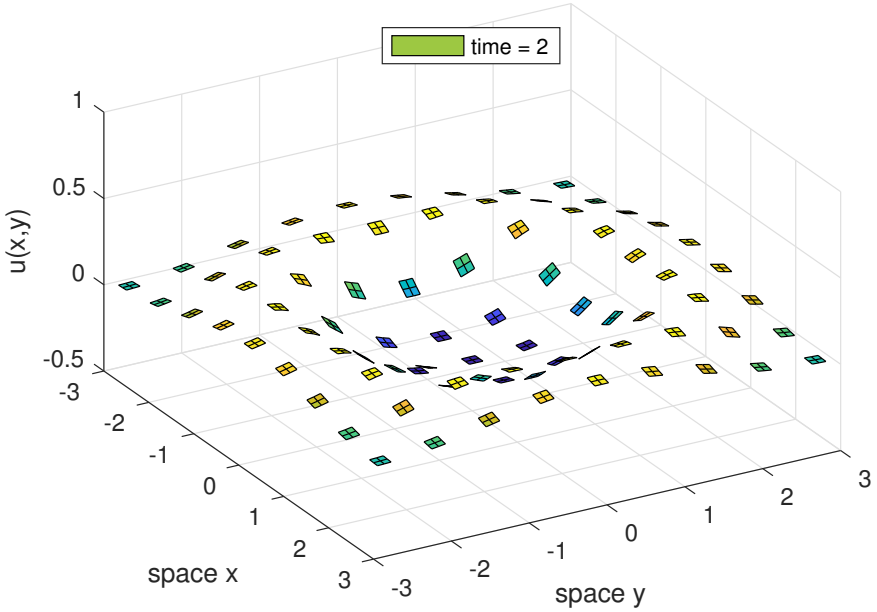
```
157 di = ceil(length(ts)/100);
158 for i = [1:di:length(ts)-1 length(ts)]
159   uv = patchEdgeInt2(uvs(i,:));
160   uv = reshape(permute(uv,[1 3 2 4 5]), [numel(x) numel(y) 2]);
161   set(usurf,'ZData', uv(:,:,1)');
162   legend(['time = ' num2str(ts(i),2)])
163   pause(0.1)
164 end
165 %print('-depsc',['wave2Dt' 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,x,y)
15   if ceil(t+1e-7)-t<2e-2, simTime = t, end %track progress
16   dx = diff(x(1:2));  dy = diff(y(1:2));   % microscale spacing
17   i = 2:size(uv,1)-1;  j = 2:size(uv,2)-1; % interior patch-points
18   uvt = nan(size(uv));  % preallocate storage
```

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




```
19    uvt(i,j,:,:,1) = uv(i,j,:,:,2);
20    uvt(i,j,:,:,2) = diff(uv(:,j,:,:,1),2,1)/dx^2 ...
21                    +diff(uv(i,:,:,:,1),2,2)/dy^2;
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','stiff');
15      xs = lsode(@(x,t) dxdt(t,x),x0,ts);
16  end
```

3.14 To do

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

3.15 Miscellaneous tests

3.15.1 `patchEdgeInt1test`: test the spectral interpolation

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

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

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

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

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

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

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

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

End the for-loop over various geometries.

```
72 end
```

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

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

Identify which microscale grid points are h or u values.

```
92 uPts=mod( bsxfun(@plus,(1:nSubP)',(1:nPatch)) ,2);
93 hPts=find(1-uPts);
94 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.

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

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

```
125 fprintf('Two field-pairs test.\n')
126 x0=patches.x((nSubP+1)/2,1);
127 patches.x=patches.x-x0;
128 for k=1:nPatch/4
129   U0=nan(nSubP,nPatch); V0=U0;
130   U0(hPts)=rand*sin(k*patches.x(hPts)*2*pi/Len);
131   U0(uPts)=rand*sin(k*patches.x(uPts)*2*pi/Len);
132   V0(hPts)=rand*cos(k*patches.x(hPts)*2*pi/Len);
133   V0(uPts)=rand*cos(k*patches.x(uPts)*2*pi/Len);
134   UVi=patchEdgeInt1([U0(:);V0(:)]);
```

```
135   normuError=norm(UVi(:,1:2:nPatch,1)-U0(:,1:2:nPatch))*norm(U0(i0,2:2:nPatch
136       +norm(UVi(:,2:2:nPatch,1)-U0(:,2:2:nPatch))*norm(U0(i0,1:2:nPatch));
137   normuError=norm(UVi(:,1:2:nPatch,2)-V0(:,1:2:nPatch))*norm(V0(i0,2:2:nPatch
138       +norm(UVi(:,2:2:nPatch,2)-V0(:,2:2:nPatch))*norm(V0(i0,1:2:nPatch));
139   if abs(normuError)+abs(normvError)>2e-13
140     normuError=normuError, normvError=normvError
141     error(['failed double field interpolation k=' num2str(k)])
142   end
143 end
```

End for-loop over patches

```
150 end
```

Finish If no error messages, then all OK.

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

3.15.2 `patchEdgeInt2test`: tests 2D spectral interpolation

Try 99 realisations of random tests.

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

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

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

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

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

```
44    % store into 5D array
45    u0s(:,:,:,:,iV)=u0;
46  end
```

Copy and NaN the edges, then interpolate

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

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

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

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Sieber, J., Marschler, C. & Starke, J. (2018), 'Convergence of Equation-Free Methods in the Case of Finite Time Scale Separation with Application to Deterministic and Stochastic Systems', *SIAM Journal on Applied Dynamical Systems* **17**(4), 2574–2614.

Svard, M. & Nordstrom, J. (2006), 'On the order of accuracy for difference approximations of initial-boundary value problems', *Journal of Computational Physics* **218**, 333–352.