Equation-Free function toolbox for Matlab/Octave

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

October 29, 2018

Abstract

This 'equation-free toolbox' facilitates the computer-assisted analysis of complex, multiscale systems. Its aim is to enable microscopic simulators to perform system level tasks and analysis. The methodology bypasses the derivation of macroscopic evolution equations by using only short bursts of microscale simulations which are often the best available description of a system (Kevrekidis & Samaey 2009, Kevrekidis et al. 2004, 2003, e.g.). This suite of functions should empower users to start implementing such methods—but so far we have only just started.

Contents

[*] `http://www.maths.adelaide.edu.au/anthony.roberts`, `http://orcid.org/0000-0001-8930-1552`

[†] `http://www.adelaide.edu.au/directory/john.maclean`

[‡] Appear here for your contribution.

3 Patch scheme for given microscale discrete space system

Section contents

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

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

Quick start For an example, see Section 3.1.1 for basic code that uses the provided functions to simulate Burgers' PDE.

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

Subsection contents

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

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

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

- `fun` is the name of the user function, `fun(t,u,x)`, that computes time derivatives (or time-steps) of quantities on the patches.
- `Xlim` give the macro-space domain of the computation: patches are equi-spaced over the interior of the interval [`Xlim(1)`,`Xlim(2)`].
- `BCs` somehow will define the macroscale boundary conditions. Currently, `BCs` is ignored and the system is assumed macro-periodic in the domain.
- `nPatch` is the number of equi-spaced spaced patches.
- `ordCC` is the 'order' of interpolation across empty space of the macroscale mid-patch values to the edge of the patches for inter-patch coupling: currently must be in $\{-1, 0, \ldots, 8\}$.

- `ratio` (real) is the ratio of the half-width of a patch to the spacing of the patch mid-points: so `ratio` $= \frac{1}{2}$ means the patches abut; and `ratio` $= 1$ is overlapping patches as in holistic discretisation.
- `nSubP` is the number of equi-spaced microscale lattice points in each patch. Must be odd so that there is a central lattice point.

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

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

3.1.1 If no arguments, then execute an example

```
if nargin==0
```

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

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

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

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

Set an initial condition, and integrate in time using standard functions.

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

Plot the simulation using only the microscale values interior to the patches: set x-edges to `nan` to leave the gaps. Figure 7 illustrates an example simulation in time generated by the patch scheme applied to Burgers' PDE.

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

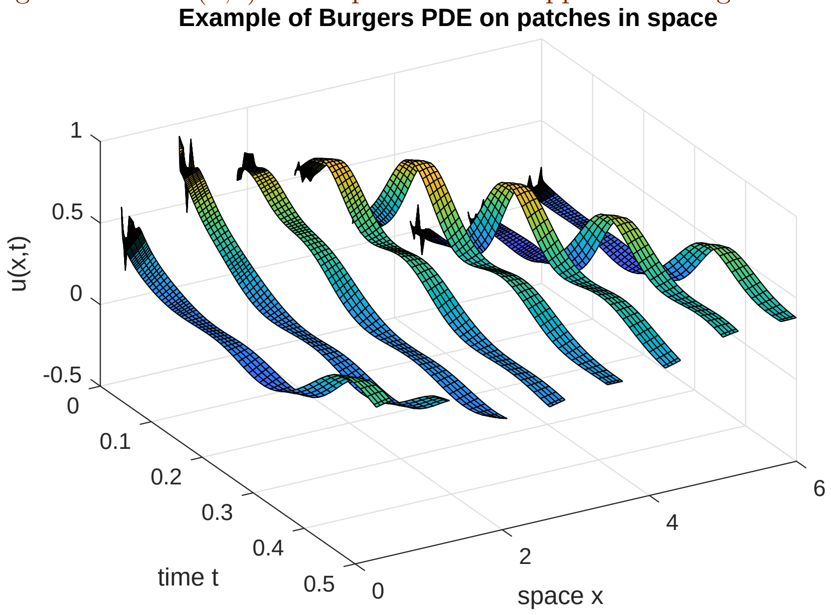
Upon finishing execution of the example, exit this function.

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

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

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

Figure 7: field $u(x,t)$ of the patch scheme applied to Burgers' PDE.



3.1.2 The code to make patches

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

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

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

```
if ~ismember(ordCC,[-1:8])
    error('ordCC out of allowed range [-1:8]')
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;
```

Check for staggered grid and periodic case.

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

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

```
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);
patches.Cwtsl=(-1).^((1:ordCC)'-patches.alt).*patches.Cwtsr;
```

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

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

```
DX=X(2)-X(1);
```

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

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

Fin.

3.2 `patchSmooth1()`: 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 using the previously established global struct `patches`.

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

Input

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

`nSubP` × `nPatch` × `nVars`. Time derivatives must be computed into the same sized array, but 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 micro-scales.

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 3.3 describes `patchEdgeInt1()`.

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

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

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

Fin.

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

Couples patches across space by computing their edge values from macroscale interpolation. Consequently a spatially discrete system could be integrated in time via the patch or gap-tooth scheme (Roberts & Kevrekidis 2007). Assumes that the sub-patch structure is *smooth* so that the patch centre-values are sensible macroscale variables, and patch edge values are determined by macroscale interpolation of the patch-centre values. Communicate patch-design variables via the global struct `patches`.

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

Input

- `u` is a vector of length `nSubP` · `nPatch` · `nVars` where there are `nVars` field values at each of the points in the `nSubP` × `nPatch` grid.
- `patches` a struct set by `configPatches1()` with the following information.
  - `.fun` is the name of the user's function `fun(t,u,x)` that computes the time derivatives (or time-steps) on the patchy lattice. The array `u` has size `nSubP` × `nPatch` × `nVars`. Time derivatives must be computed into the same sized array, but 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 micro-scales.
  - `.ordCC` is order of interpolation, currently in $\{0, 2, 4, 6, 8\}$.
  - `.alt` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation.
  - `.Cwtsr` and `.Cwtsl`

Output

- `u` is `nSubP` × `nPatch` × `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.

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

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);
r=dx*(nSubP-1)/2/DX;
```

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

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

The centre of each patch (as `nSubP` is odd) is at

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

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

```
if patches.ordCC>0 % then non-spectral interpolation
  dmu=nan(patches.ordCC,nPatch,nVars);
  if patches.alt % use only odd numbered neighbours
    dmu(1,:,:)=(u(i0,jp,:)+u(i0,jm,:))/2; % \mu
    dmu(2,:,:)= u(i0,jp,:)-u(i0,jm,:); % \delta
    jp=jp(jp); jm=jm(jm); % increase shifts to \pm2
  else % standard
```

```
    dmu(1,:,:)=(u(i0,jp,:)-u(i0,jm,:))/2; % \mu\delta
    dmu(2,:,:)=(u(i0,jp,:)-2*u(i0,j,:)+u(i0,jm,:)); % \delta^2
  end% if odd/even
```

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

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

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

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

Case of spectral interpolation Assumes the domain is macro-periodic. 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' $k = (0, 1, \ldots, k_{\max}, -k_{\max}, \ldots, -1)$.

```
else% spectral interpolation
```

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

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

Now set wavenumbers.

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

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

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

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

```
  Ck=fft(u(i0,:,:));
  if mod(nPatch,2)==0
    Czz=Ck(1,nPatch/2+1,:)/nPatch;
    Ck(1,nPatch/2+1,:)=0;
  end
```

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

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

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

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

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

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

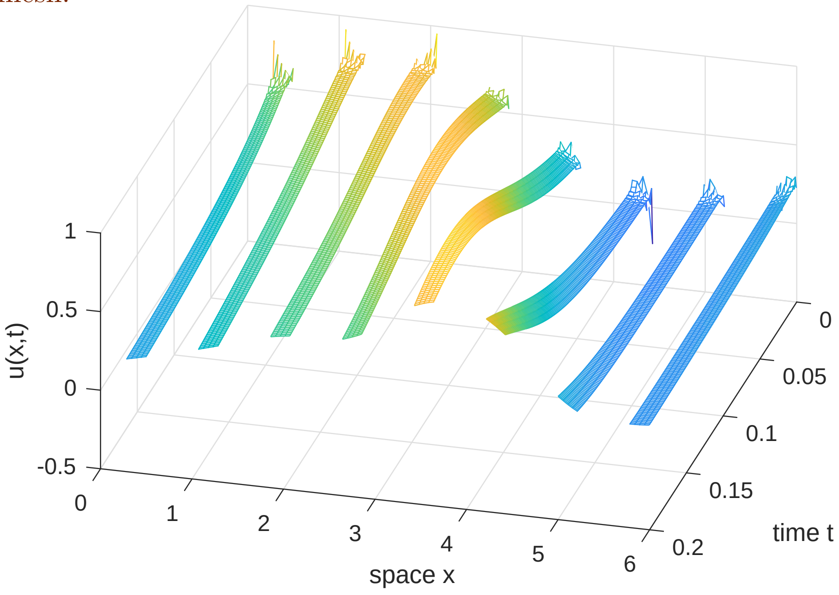
Fin, returning the 2/3D array of field values.

3.4 `BurgersExample`: simulate Burgers' PDE on patches

Subsection contents

Figure 7 shows an example simulation in time generated by the patch scheme function applied to Burgers' PDE. This code similarly applies the Equation-Free functions to a microscale space-time map (Figure 8), a map that happens to be derived as a micro-scale space-time discretisation of Burgers' PDE. Then this example applies projective integration to simulate further in time.

Figure 8: a short time simulation of the Burgers' map (Section 3.4.2) on patches in space. It requires many very small time steps only just visible in this mesh.



3.4.1 Script code to simulate a micro-scale space-time map

This first part of the script implements the following gap-tooth scheme (arrows indicate function recursion).

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

Establish global data struct for the Burgers' map (Section 3.4.2) solved on 2π-periodic domain, with eight patches, each patch of half-size ratio 0.2, with seven points within each patch, and say fourth order interpolation provides edge-values that couple the patches.

```
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 micro-scale space-time map over a time 0.2 using the function burgerBurst() (Section 3.4.3).

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

Plot the simulation. Use only the microscale values interior to the patches via nan in the x-edges 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)
set(gcf,'paperposition',[0 0 14 10])
print('-depsc2','ps1BurgersMapU')
```

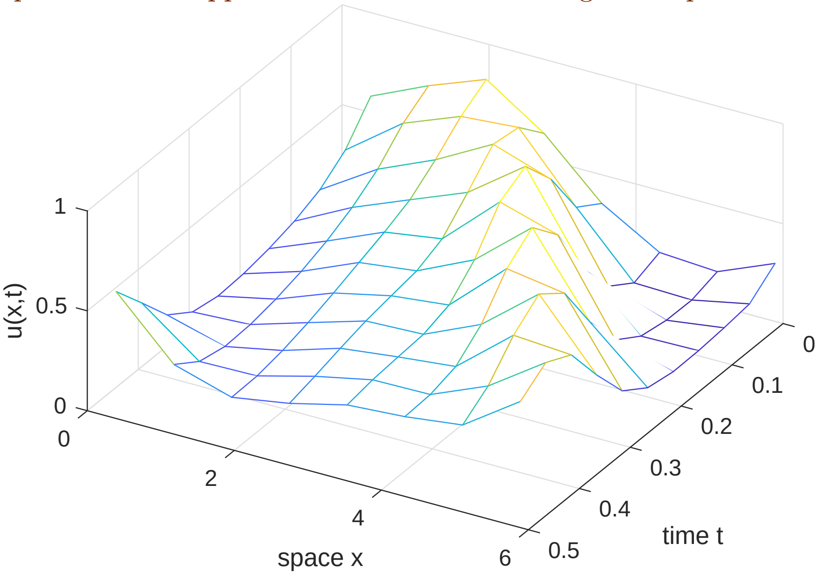
Use projective integration Around the micro-scale burst burgerBurst(), wrap the projective integration function PIRK2() of Section 2.2. Figure 9 shows the macroscale prediction of the patch centre values on macro-scale time-steps.

This second part of the script implements the following design.

1. configPatches1 (done in first part)
2. PIRK2 ↔ burgerBurst ↔ patchSmooth1 ↔ burgersMap
3. process results

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

Figure 9: macro-scale space-time field $u(x,t)$ in a basic projective integration of the patch scheme applied to the micro-scale Burgers' map.



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

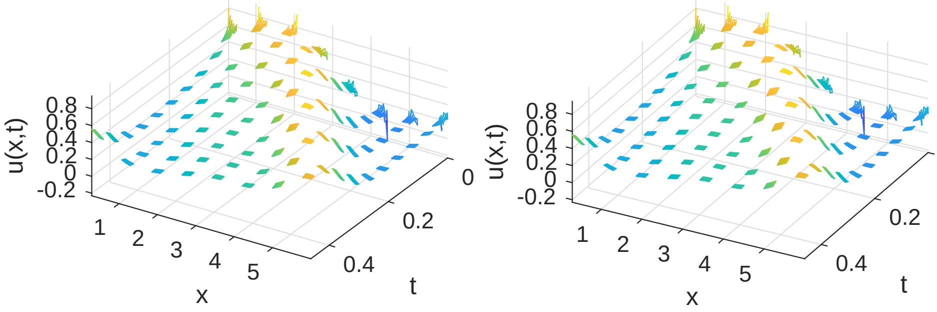
Set the desired macro-scale time-steps, and micro-scale burst length over the time domain. Then projectively integrate in time using PIRK2() which is (roughly) second-order accurate in the macro-scale time-step.

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

Plot the macroscale predictions of the mid-patch values to give the macroscale mesh of Figure 9.

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

Figure 10: the field $u(x,t)$ during each of the microscale bursts used in the projective integration. View this stereo pair cross-eyed.



```
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
view(120,50)
set(gcf,'paperposition',[0 0 14 10])
print('-depsc2','ps1BurgersU')
```

Then plot the microscale mesh of the microscale bursts shown in Figure 10 (a stereo pair). The details of the fine microscale mesh are almost invisible.

```
figure(3),clf
for k = 1:2, subplot(2,2,k)
  mesh(tss,xs(:),uss')
  ylabel('x'),xlabel('t'),zlabel('u(x,t)')
  axis tight, view(126-4*k,50)
end
set(gcf,'paperposition',[0 0 17 12])
print('-depsc2','ps1BurgersMicro')
```

3.4.2 `burgersMap()`: discretise the PDE microscale

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

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

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

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

First find and set the number of micro-scale time-steps.

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

Apply the microscale map over all time-steps in the burst, using `patchSmooth1` (Section 3.2) as the interface that provides the interpolated edge-values of each patch. Store the results in rows to be consistent with ODE and projective integrators.

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

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

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

Fin.

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

Subsection contents

Figure 11 shows an example simulation in time generated by the patch scheme function applied to heterogeneous diffusion. That such simulations of heterogeneous diffusion makes valid predictions was established by Bunder et al. (2017) who proved that the scheme is accurate when the number of points in a patch is one more than an even multiple of the microscale periodicity.

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

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

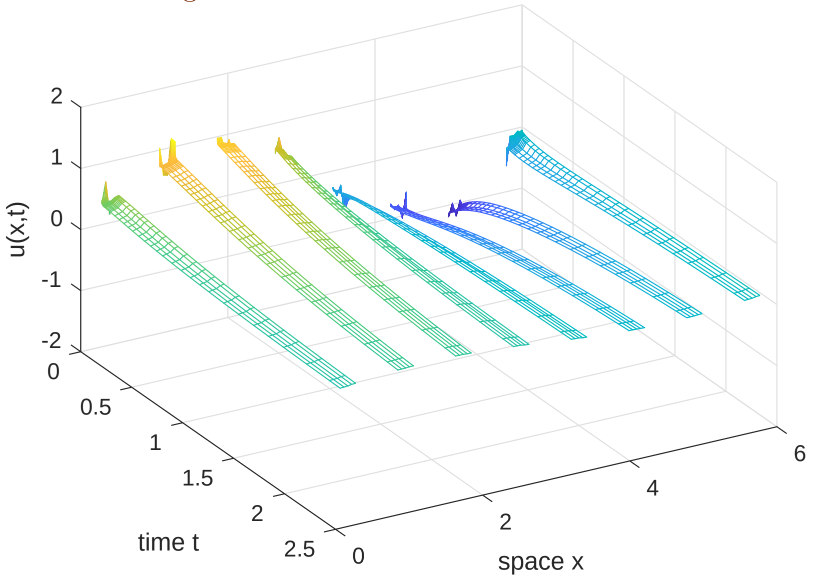
In this 1D space, the macroscale, homogenised, effective diffusion should be the harmonic mean of these coefficients.

3.5.1 Script to simulate via stiff or projective integration

This first part of the script implements the following gap-tooth scheme (arrows indicate function recursion).

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

Figure 11: the diffusing field $u(x,t)$ in the patch (gap-tooth) scheme applied to microscale heterogeneous diffusion.



Set the desired microscale periodicity, and 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 for heterogeneous diffusion solved on 2π-periodic domain, with eight patches, each patch of half-size 0.2, and the number of points in a patch being one more than an even multiple of the microscale periodicity (which Bunder et al. (2017) showed is accurate). Quadratic (fourth-order) interpolation provides values for the inter-patch coupling conditions.

```
global patches
nPatch = 8
```

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

A user can add information to the global data struct `patches` in order to communicate to the time derivative function. Here include the diffusivity coefficients, replicated to fill up a patch.

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

Conventional integration in time Set an initial condition, and here integrate forward in time using a standard method for stiff systems—because of the simplicity of linear problems this method works quite efficiently here. Integrate the interface `patchSmooth1` (Section 3.2) to the microscale differential equations.

```
u0 = sin(patches.x)+0.2*randn(nSubP,nPatch);
[ts,ucts] = ode15s(@patchSmooth1, [0 2/cHomo], u0(:));
```

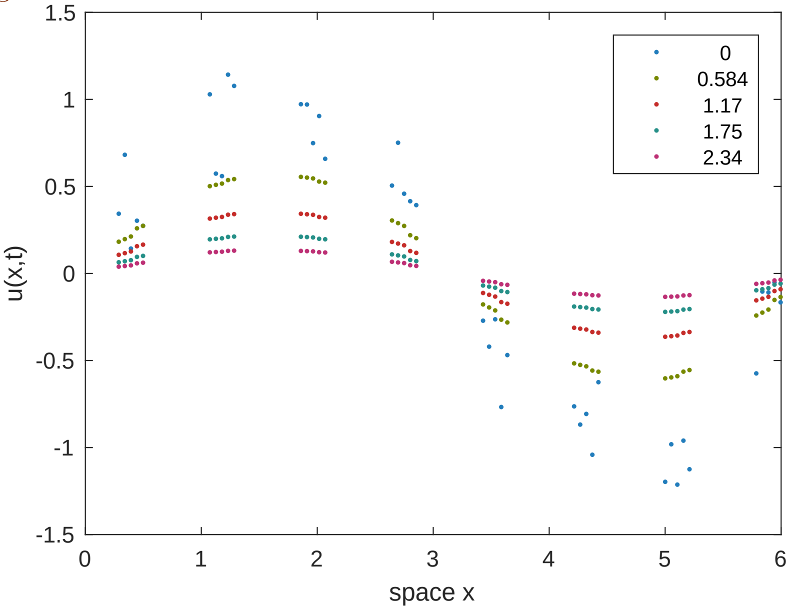
Plot the simulation in Figure 11.

```
figure(1),clf
xs = patches.x;  xs([1 end],:) = nan;
mesh(ts,xs(:),ucts'),  view(60,40)
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
set(gcf,'paperposition',[0 0 14 10])
print('-depsc2','ps1HomogenisationCtsU')
```

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 12.

This second part of the script implements the following design, where the micro-integrator could be, for example, `ode23` or `rk2int`.

Figure 12: field $u(x, t)$ shows basic projective integration of patches of heterogeneous diffusion.



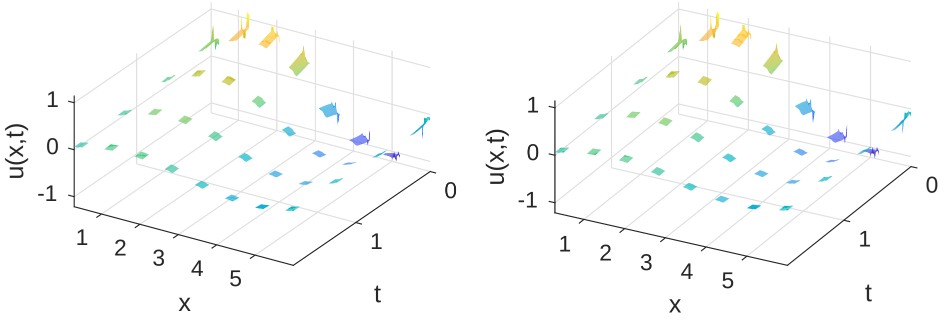
1. configPatches1 (done in first part)
2. PIRK2 ↔ heteroBurst ↔ micro-integrator ↔ patchSmooth1 ↔ heteroDiff
3. process results

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

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

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

Figure 13: stereo pair of the field $u(x,t)$ during each of the microscale bursts used in the projective integration.



```
ts = linspace(0,2/cHomo,5)
bT = 3*( ratio*Len/nPatch )^2/cHomo
addpath('../ProjInt','../RKint')
[us,tss,uss] = PIRK2(@heteroBurst, bT, ts, u0(:));
```

Plot the macroscale predictions to draw Figure 12.

```
figure(2),clf
plot(xs(:),us','.')
ylabel('u(x,t)'), xlabel('space x')
legend(num2str(ts',3))
set(gcf,'paperposition',[0 0 14 10])
print('-depsc2','ps1HomogenisationU')
```

Also plot a surface detailing the microscale bursts as shown in Figure 13.

```
figure(3),clf
for k = 1:2, subplot(2,2,k)
  surf(tss,xs(:),uss',  'EdgeColor','none')
  ylabel('x'), xlabel('t'), zlabel('u(x,t)')
  axis tight, view(126-4*k,45)
end
set(gcf,'paperposition',[0 0 17 12])
print('-depsc2','ps1HomogenisationMicro')
```

End of the script.

3.5.2 `heteroDiff()`: heterogeneous diffusion

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

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

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

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

- `ode23` generates 'noise' that is unsightly at best and may be ruinous;
- `ode15s` does not cater for the NaNs in some components of `u`;
- `rk2int` simple specified step integrator behaves consistently, and so appears best.

```
function [ts, ucts] = heteroBurst(ti, ui, bT)
  switch 'rk2'
  case '23',  [ts,ucts] = ode23 (@patchSmooth1,[ti ti+bT],ui(:));
  case '15s', [ts,ucts] = ode15s(@patchSmooth1,[ti ti+bT],ui(:));
  case 'rk2', ts = linspace(ti,ti+bT,100)';
```

[2]Use `bsxfun()` as pre-2017 Matlab versions may not support auto-replication.

```
            ucts = rk2int(@patchSmooth1,ts,ui(:));
  end
end
```

Fin.

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

Subsection contents

Figure 14 shows an example simulation in time generated by the patch scheme function applied to a simple wave PDE. The inter-patch coupling is realised by spectral interpolation to the patch edges of the mid-patch values.

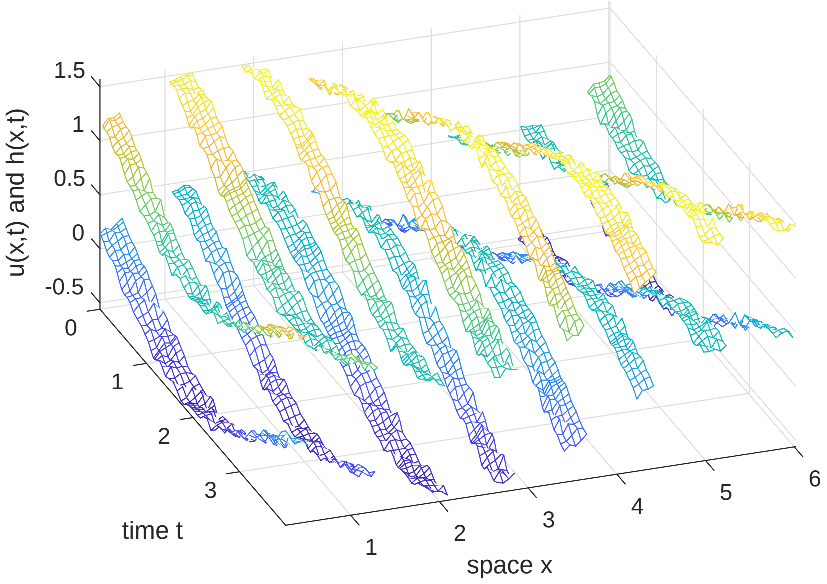
This approach, based upon the differential equations coded in Section 3.6.2, may be adapted by a user to a wide variety of 1D wave and near-wave systems. For example, the differential equations of Section 3.6.3 describes the nonlinear microscale simulator of the nonlinear shallow water wave PDE derived from the Smagorinski model of turbulent flow (Cao & Roberts 2012, 2016*a*).

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

$$\frac{\partial h}{\partial t} = -c_1\frac{\partial u}{\partial x} + f_1[h,u] \quad\text{and}\quad \frac{\partial u}{\partial t} = -c_2\frac{\partial h}{\partial x} + f_2[h,u], \tag{2}$$

where the brackets indicate that the nonlinear functions f_ℓ may involve various spatial derivatives of the fields $h(x,t)$ and $u(x,t)$. For example, Section 3.6.3 encodes a nonlinear Smagorinski model of turbulent shallow

Figure 14: water depth $h(x,t)$ (above) and velocity field $u(x,t)$ (below) of the gap-tooth scheme applied to the simple wave PDE (2), linearised. The micro-scale random component to the initial condition has long lasting effects on the simulation—but the macroscale wave still propagates.



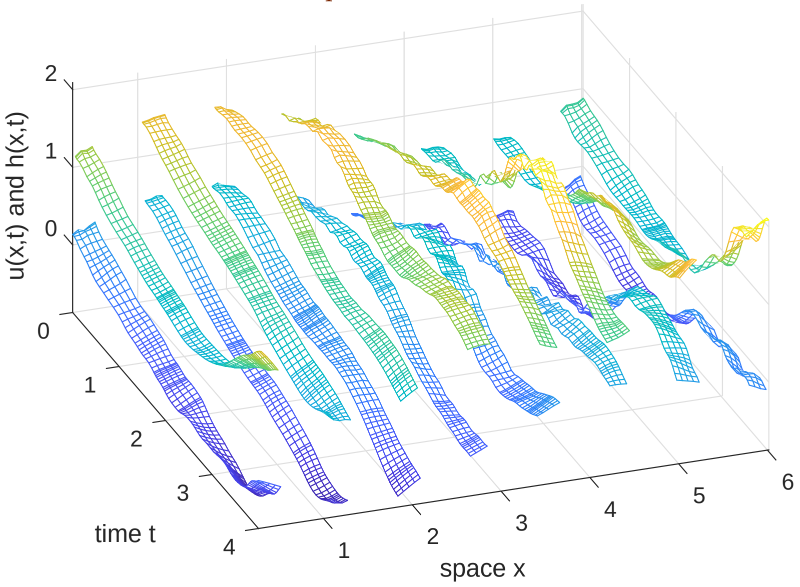
water (Cao & Roberts 2012, 2016*a*, e.g.) along an inclined flat bed: let x measure position along the bed and in terms of fluid depth $h(x,t)$ and depth-averaged lateral velocity $u(x,t)$ the model PDEs are

$$\frac{\partial h}{\partial t} = -\frac{\partial(hu)}{\partial x}\,, \tag{3a}$$

$$\frac{\partial u}{\partial t} = 0.985\left(\tan\theta - \frac{\partial h}{\partial x}\right) - 0.003\frac{u|u|}{h} - 1.045u\frac{\partial u}{\partial x} + 0.26h|u|\frac{\partial^2 u}{\partial x^2}\,, \tag{3b}$$

where $\tan\theta$ is the slope of the bed. Equation (3a) represents conservation of the fluid. The momentum PDE (3b) represents the effects of turbulent bed drag $u|u|/h$, self-advection $u\partial u/\partial x$, nonlinear turbulent dispersion $h|u|\partial^2u/\partial x^2$, and gravitational hydrostatic forcing $\tan\theta - \partial h/\partial x$. Figure 15 shows one simulation of this system—for the same initial condition

Figure 15: water depth $h(x,t)$ (above) and velocity field $u(x,t)$ (below) of the gap-tooth scheme applied to the Smagorinski shallow water wave PDEs (3). The micro-scale random initial component decays where the water speed is non-zero due to 'turbulent' dissipation.



as Figure 14.

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

3.6.1 Script code to simulate wave systems

This script implements the following gap-tooth scheme (arrows indicate function recursion).

1. configPatches1, and add micro-information
2. ode15s ↔ patchSmooth1 ↔ simpleWavePDE

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

Establish the global data struct `paches` for the PDEs (2) (linearised) solved on 2π-periodic domain, with eight patches, each patch of half-size ratio 0.2, with eleven points within each patch, and third-order interpolation to provide edge-values for the inter-patch coupling conditions (higher order interpolation is smoother for smooth initial conditions).

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

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

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

Set an initial condition of a progressive wave, and check evaluation of the time derivative. The capital letter `U` denotes an array of values merged from both u and h fields on the staggered grids (possibly with some optional micro-scale 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 simple wave and the water wave equations in the one loop.

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

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

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

Plot the simulation.

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

Print the output.

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

For the second time through the loop, change to the Smagorinski turbulence model (3) of shallow water flow, keeping other parameters and the initial condition the same.

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

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

3.6.2 simpleWavePDE(): simple wave PDE

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

$$U_{ij} = \begin{cases} u_{ij} & i+j \text{ even}, \\ h_{ij} & i+j \text{ odd}. \end{cases}$$

The output Ut contains the merged time derivatives of the two staggered fields. So set the micro-grid spacing and reserve space for time derivatives.

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

Compute the PDE derivatives at interior points of the patches.

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

Here 'wastefully' compute time derivatives for both PDEs at all grid points—for 'simplicity'—and then merges the staggered results. Since $\dot h_{ij} \approx -(u_{i+1,j} - u_{i-1,j})/(2 \cdot dx) = -(U_{i+1,j} - U_{i-1,j})/(2 \cdot dx)$ as adding/subtracting one from the index of a h-value is the location of the neighbouring u-value on the staggered micro-grid.

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

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

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

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

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

3.6.3 `waterWavePDE()`: water wave PDE

This function codes the staggered lattice equation inside the patches for the nonlinear wave-like PDE system (3). Also, regularise the absolute value appearing the the PDEs via the one-line function `rabs()`.

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

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

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

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

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

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

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

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

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

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

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

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

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

Fin.

3.7 To do

- Testing is so far only qualitative. Need to be quantitative.
- Multiple space dimensions.
- Heterogeneous microscale via averaging regions.
- Parallel processing versions.
- ??
- Adapt to maps in micro-time? Surely easy, just an example.

3.8 Miscellaneous tests

3.8.1 `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
configPatches1(@sin,[0,Len],nan,nPatch,0,ratio,nSubP);
kMax=floor((nPatch-1)/2);
```

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

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

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

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

End the for-loop over various geometries.

```
end
```

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

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

Identify which microscale grid points are h or u values.

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

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

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

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

```
fprintf('Two field-pairs test.\n')
x0=patches.x((nSubP+1)/2,1);
patches.x=patches.x-x0;
for k=1:nPatch/4
  U0=nan(nSubP,nPatch); V0=U0;
  U0(hPts)=rand*sin(k*patches.x(hPts)*2*pi/Len);
  U0(uPts)=rand*sin(k*patches.x(uPts)*2*pi/Len);
  V0(hPts)=rand*cos(k*patches.x(hPts)*2*pi/Len);
  V0(uPts)=rand*cos(k*patches.x(uPts)*2*pi/Len);
  UVi=patchEdgeInt1([U0(:);V0(:)]);
  normuError=norm(UVi(:,1:2:nPatch,1)-U0(:,1:2:nPatch))*norm(U0(i0,2
      +norm(UVi(:,2:2:nPatch,1)-U0(:,2:2:nPatch))*norm(U0(i0,1:2:nPa
  normuError=norm(UVi(:,1:2:nPatch,2)-V0(:,1:2:nPatch))*norm(V0(i0,2
      +norm(UVi(:,2:2:nPatch,2)-V0(:,2:2:nPatch))*norm(V0(i0,1:2:nPa
  if abs(normuError)+abs(normvError)>2e-13
    normuError=normuError, normvError=normvError
    error(['failed double field interpolation k=' num2str(k)])
  end
end
```

End for-loop over patches

```
end
```

Finish If no error messages, then all OK.

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

References

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* **accepted 2 Feb 2017**.

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

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

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

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

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

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

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

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

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

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

Kutz, J. N., Brunton, S. L., Brunton, B. W. & Proctor, J. L. (2016), *Dynamic Mode Decomposition: Data-driven Modeling of Complex Systems*, number 149 *in* 'Other titles in applied mathematics', SIAM, Philadelphia.

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

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

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

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

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

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

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

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.