Equation-free computational homogenisation with thicker edge coupling

A. J. Roberts*

April 12, 2023

Contents

*School of Mathematical Sciences, University of Adelaide, South Australia. https://profajroberts.github.io, http://orcid.org/0000-0001-8930-1552

Examples

1 hyperDiffHetero: simulate a heterogeneous hyper-diffusion PDE in 1D on patches

Figure 1 shows an example simulation in time generated by the patch scheme applied to a heterogeneous version of the hyper-diffusion PDE. That such simulations makes valid predictions was established by Bunder, Roberts, and Kevrekidis (2017) who proved that the scheme is accurate when the number of points in a patch is tied to a multiple of the periodicity of the pattern.

We aim to simulate the heterogeneous hyper-diffusion PDE

$$u_t = -D[c_1(x)Du] \quad \text{where operator } D := \partial_x(c_2(x)\partial_x), \tag{1}$$

for microscale periodic coefficients $c_l(x)$, and boundary conditions of $u = u_x = 0$ at $x = 0, L$. In this 1D space, the macroscale, homogenised, effective hyper-diffusion should be some unknown 'average' of these coefficients. We discretise the PDE to a lattice of values $u_i(t)$, with lattice spacing dx, and governed by

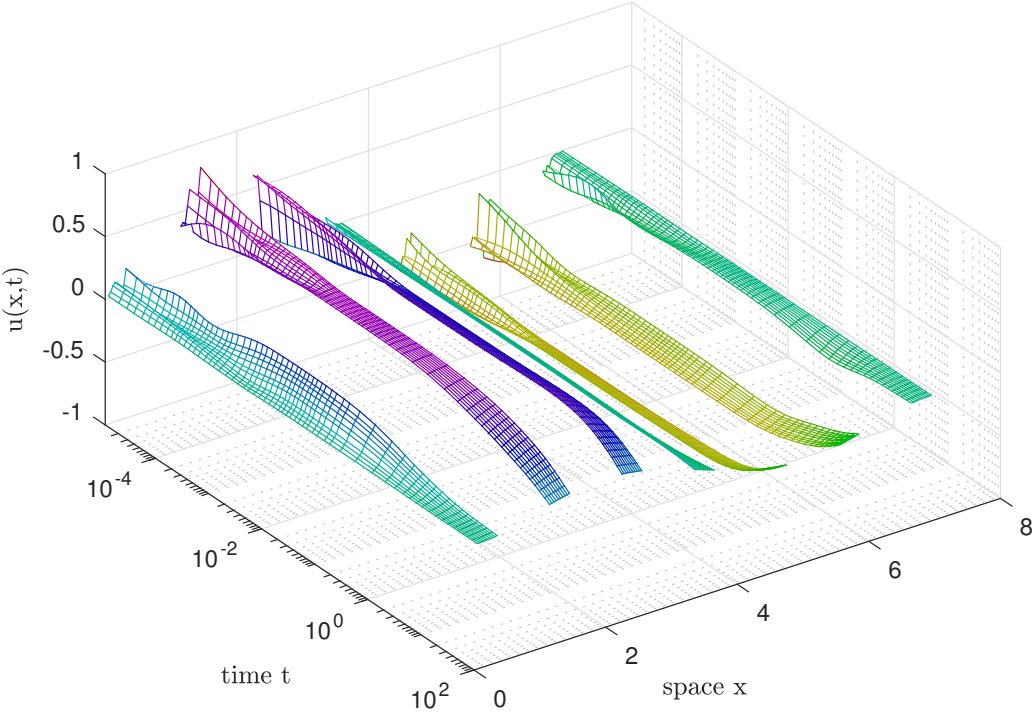
$$\dot{u}_i = -D[c_{i1}Du_i] \quad \text{where operator } D := \delta(c_{i2}\delta)/dx^2$$

in terms of centred difference operator $\delta u_i := u_{i+1/2} - u_{i-1/2}$.

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

```
clear all
basename = mfilename
```

Figure 1: the hyper-diffusing field $u(x,t)$ in the patch (gap-tooth) scheme applied to microscale heterogeneous hyper-diffusion (Section 1). The log-time axis shows: $t < 10^{-2}$, rapid decay of sub-patch micro-structure; $10^{-2} < t < 1$, a quasi-equilibrium; and $1 < t < 10^2$, slow decay of macroscale structures.




```
global OurCf2eps, OurCf2eps=true %optional to save plots
nGap = 3 % controls size of gap between patches
nPtsPeriod = 5
dx = 0.5/nGap/nPtsPeriod
```

Create some random heterogeneous coefficients, log-uniform.

```
csVar = 1
cs = 0.2*exp( -csVar/2+csVar.*rand(nPtsPeriod,2) )
```

Establish global data struct `patches` for heterogeneous hyper-diffusion on a finite domain with, on average, one patch per unit length. Use seven patches, and use high-order interpolation with `ordCC` $= 0$.

```
nPatch = 7
nSubP = 2*nPtsPeriod+4 % or +2 for not-edgyInt
```

```
Len = nPatch;
ordCC = 0;
dom.type = 'equispace';
dom.bcOffset = 0.5 % for BC type
patches = configPatches1(@hyperDiffPDE,[0 Len],dom ...
    ,nPatch,ordCC,dx,nSubP,'EdgyInt',true,'nEdge',2 ...
    ,'hetCoeffs',cs);
xs=squeeze(patches.x);
```

Simulate 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 `patchSys1` (Section 6) to the microscale differential equations.

```
u0 = sin(2*pi/Len*patches.x).*rand(nSubP,1,1,nPatch);
tic
[ts,us] = ode15s(@patchSys1, [0 100], u0(:) ,[],patches);
simulateTime = toc
us = reshape(us,length(ts),numel(patches.x(:)),[]);
```

Plot the simulation in Figure 1, using log-axis for time so we can see a little of both micro- and macro-dynamics.

```
figure(1),clf
xs([1:2 end-1:end],:) = nan;
t0=min(find(ts>1e-5));
mesh(ts(t0:3:end),xs(:),us(t0:3:end,:)'),  view(55,50)
colormap(0.7*hsv)
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
ca=gca;  ca.XScale='log';  ca.XLim=ts([t0 end]);
ifOurCf2eps([basename 'Uxt'])
```

Fin.

1.1 Heterogeneous hyper-diffusion PDE inside patches

As a microscale discretisation of hyper-diffusion PDE (1) $u_t = -D[c_1(x)Du]$, where heterogeneous operator $D = \partial_x(c_2(x)\partial_x)$.

```
function ut=hyperDiffPDE(t,u,patches)
  dx=diff(patches.x(1:2));  % microscale spacing
```

Code Dirichlet boundary conditions of zero function and derivative at left-end of left-patch, and right-end of right-patch. For slightly simpler coding, squeeze out the two singleton dimensions.

```
  u = squeeze(u);
  if ~patches.periodic % discretise BC u=u_x=0
      u(1:2,1)=0;
      u(end-1:end,end)=0;
  end%if
```

Here code straightforward centred discretisation in space.

```
  ut = nan+u;   % preallocate output array
  v = patches.cs(2:end,1).*diff(patches.cs(:,2).*diff(u))/dx^2;
  ut(3:end-2,:) = -diff(patches.cs(2:end-1,2).*diff(v))/dx^2 ;
end
```

2 SwiftHohenbergPattern: simulate patterns of the Swift–Hohenberg PDE in 1D on patches

Figure 2 shows an example simulation in time generated by the patch scheme applied to the patterns arising from the Swift–Hohenberg PDE. That such simulations of patterns makes valid predictions was established by Bunder, Roberts, and Kevrekidis (2017) who proved that the scheme is accurate when the number of points in a patch is just more than a multiple of the periodicity of the pattern.

Consider a lattice of values $u_i(t)$, with lattice spacing dx, and governed by a microscale centred discretisation of the Swift–Hohenberg PDE

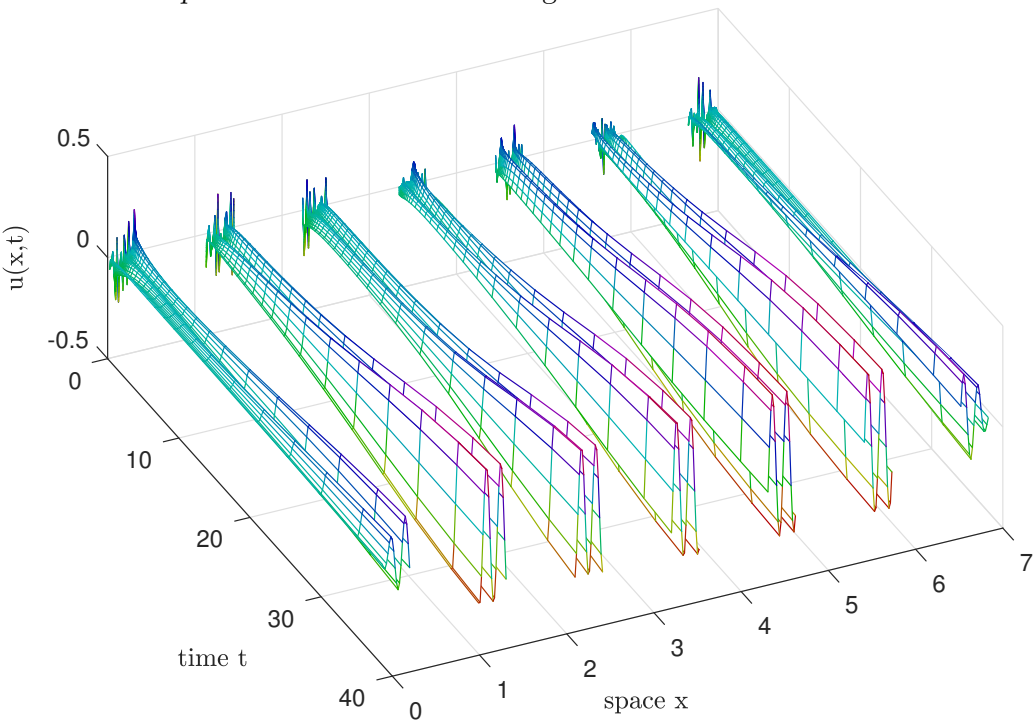
$$\partial_t u = -(1+\partial_x^2/k_0^2)^2 u + \mathrm{Ra}\, u - u^3, \tag{2}$$

with boundary conditions of $u = u_x = 0$ at $x = 0, L$. For Ra just above critical, say $\mathrm{Ra} = 0.1$, the system rapidly evolves to spatial quasi-periodic solutions with period≈ 0.166, wavenumber $k_0 \approx 38$. On medium times these spatial oscillations grow to near equilibrium amplitude of $\sqrt{\mathrm{Ra}}$, and over very long times the phases of the oscillations evolve in space to adapt to the boundaries.

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

```
clear all, close all
%global OurCf2eps, OurCf2eps=true %optional to save plots
```

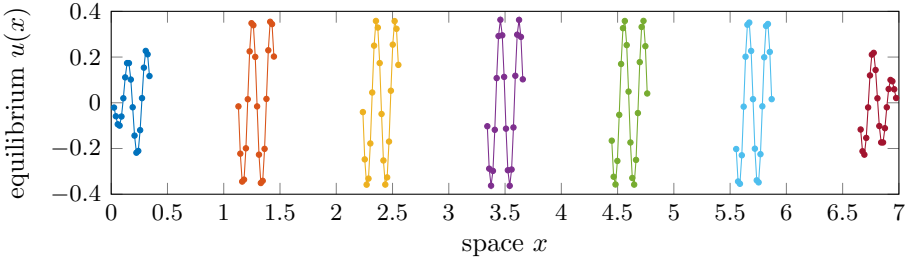
Figure 2: the pattern forming field $u(x,t)$ in the patch (gap-tooth) scheme applied to a microscale discretisation of the Swift–Hohenberg PDE *(Section 2). Physically we see the rapid decay of much microstructure, but also the meso-time growth of sub-patch-scale patterns, wavenumber k_0, that are modulated over the inter-patch distances and over long times.*




```
Ra = 0.1 % Ra>0 leads to patterns
nGap = 3
%waveLength = 0.496688741721854 /nGap %for nPatch==5
waveLength = 0.497630331753555 /nGap %for nPatch==7
%waveLength = 0.5 /nGap %for periodic case
nPtsPeriod = 10
dx = waveLength/nPtsPeriod
k0 = 2*pi/waveLength
```

Establish global data struct `patches` for heterogeneous diffusion on 2π-periodic domain. Use seven patches. Quartic (fourth-order) interpolation `ordCC` $= 4$ provides values for the inter-patch coupling conditions.

Figure 3: an equilibrium of the Swift–Hohenberg PDE *on seven patches in 1D space. In the sub-patch patterns, there is a small phase shift in the patterns from patch to patch. And the amplitude of the pattern has to go to 'zero' at the boundaries.*




```
nPatch = 7
nSubP = 2*nPtsPeriod+4
%nSubP = 2*nGap*nPtsPeriod+4 % full-domain
Len = nPatch;
ordCC = 4;
dom.type='equispace';
dom.bcOffset=0.5
patches = configPatches1(@SwiftHohenbergPDE,[0 Len],dom ...
    ,nPatch,ordCC,dx,nSubP,'EdgyInt',true,'nEdge',2);
xs=squeeze(patches.x);
```

Find equilibrium with fsolve Start the search from some guess.

```
fprintf('\n**** Find equilibrium with fsolve\n')
u = 0.4*sin(k0*patches.x);
```

But set the pairs of patch-edge values to `Nan` in order to use `patches.i` to index the interior sub-patch points as they are the variables.

```
u([1:2 end-1:end],:) = nan;
patches.i = find(~isnan(u));
```

Seek the equilibrium, and report the norm of the residual, via the generic patch system wrapper `theRes` (Section 2.2).

```
tic
[u(patches.i),res] = fsolve(@(v) theRes(v,patches,k0,Ra) ...
```

```
    ,u(patches.i) ,optimoptions('fsolve','Display','off'));
solveTime = toc
normRes = norm(res)
assert(normRes<1e-6,'**** fsolve solution not accurate')
```

Plot the equilibrium see Figure 3.

```
figure(1),clf
subplot(2,1,1)
plot(xs,squeeze(u),'.-')
xlabel('space $x$'),ylabel('equilibrium $u(x)$')
ifOurCf2tex([mfilename 'Equilib'])%optionally save
```

Simulate 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 `patchSys1` (Section 6) to the microscale differential equations.

```
fprintf('\n**** Simulate in time\n')
u0 = 0*patches.x+0.1*randn(nSubP,1,1,nPatch);
tic
[ts,us] = ode15s(@patchSys1, [0 40], u0(:) ,[],patches,k0,Ra);
simulateTime = toc
us = reshape(us,length(ts),numel(patches.x(:)),[]);
```

Plot the simulation in Figure 2.

```
figure(2),clf
xs([1:2 end-1:end],:) = nan;
mesh(ts(1:3:end),xs(:),us(1:3:end,:)'),  view(65,60)
colormap(0.7*hsv)
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
ifOurCf2eps([mfilename 'Uxt'])
```

Fin.

2.1 The Swift–Hohenberg PDE and BCs inside patches

As a microscale discretisation of Swift–Hohenberg PDE $u_t = -(1+\partial_x^2/k_0^2)^2 u + \mathrm{Ra}\, u - u^3$, here code straightforward centred discretisation in space.

```
function ut=SwiftHohenbergPDE(t,u,patches,k0,Ra)
  dx=diff(patches.x(1:2));  % microscale spacing
  i=3:size(u,1)-2;  % interior points in patches
```

Code Dirichlet boundary conditions of zero function and derivative, $u = u_x = 0$, at the left-end of the leftmost-patch, and the right-end of the rightmost-patch. For slightly simpler coding, squeeze out the two singleton dimensions.

```
  u = squeeze(u);
  u(1:2,1)=0;
  u(end-1:end,end)=0;
```

Here code straightforward centred discretisation in space.

```
  ut=nan+u;          % preallocate output array
  v = u(2:end-1,:)+diff(u,2)/dx^2/k0^2;
  ut(i,:) = -( v(2:end-1,:)+diff(v,2)/dx^2/k0^2 ) ...
    +Ra*u(i,:) -u(i,:).^3;
end
```

2.2 `theRes()`: wrapper function to zero for equilibria

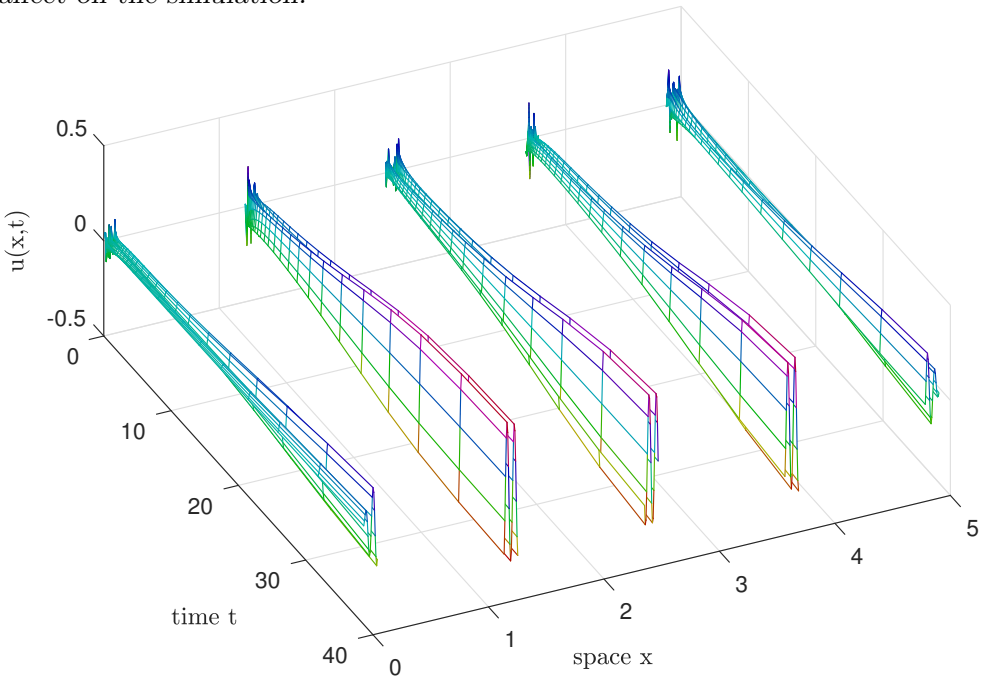
This functions converts a vector of values into the interior values of the patches, then evaluates the time derivative of the system at time zero, and returns the vector of patch-interior time derivatives.

```
function f=theRes(u,patches,k0,Ra)
  v=nan(size(patches.x));
  v(patches.i) = u;
  f = patchSys1(0,v(:),patches,k0,Ra);
  f = f(patches.i);
end%function theRes
```

3 `SwiftHohenbergHetero`: simulate patterns of the Swift–Hohenberg PDE in 1D on patches

Figure 4 shows an example simulation in time generated by the patch scheme applied to the patterns arising from a heterogeneous version of the Swift–Hohenberg PDE. That such simulations of patterns makes valid predictions was established by Bunder, Roberts, and Kevrekidis (2017) who proved that the

Figure 4: the field $u(x,t)$ in the patch (gap-tooth) scheme applied to microscale heterogeneous Swift–Hohenberg PDE *(Section 3). The heterogeneous coefficients are approximately uniform over $[0.9, 1.1]$. This heterogeneity has no noticeable affect on the simulation.*



scheme is accurate when the number of points in a patch is tied to a multiple of the periodicity of the pattern.

Consider a lattice of values $u_i(t)$, with lattice spacing dx, arising from a microscale discretisation of the pattern forming, heterogeneous, Swift–Hohenberg PDE

$$\partial_t u = -D[c_1(x)Du] + \mathrm{Ra}\, u - u^3, \quad D := 1 + \partial_x[c_2(x)\partial_x \cdot]/k_0^2, \tag{3}$$

where $c_\ell(x)$ have period $2\pi/k_0$. Coefficients c_ℓ are chosen iid random, nearly uniform, with mean near one. With mean one, the periodicity of c_ℓ approximately match the periodicity of the resultant spatial pattern.

The current patch scheme coding preserves symmetry in the case of periodic patches (for every order of interpolation). For equispace and chebyshev options, the coupling currently fails symmetry.

Consider the spectrum in the symmetric cases of periodic patches (based

upon only the cases $N = 5, 7$). There are $2N$ small eigenvalues, separated by a gap from the rest. In the homogeneous case, these occur as N pairs. With small heterogeneity, they appear to split into $N - 1$ pairs, and two distinct. With stronger heterogeneity (say 0.5), they *often* appear to also split into two clusters, each of N eigenvalues, with one small-valued cluster, and one meso-valued cluster—curious. Further analysis with sparse approximation of the invariant spaces suggests the following:

- for homogeneous, the $2N$ modes are local oscillations in each patch, with two modes each corresponding to phase shifts of the possible oscillations;
- for heterogeneous
  - N eigenmodes appear to be one phase 'locking' to the heterogeneity; and
  - N eigenmodes appear to be other phase 'locking' to the heterogeneity. Unless it is something to do with the coupling, but then it only appears with heterogeneity.

Consider the spectrum with BCs of $u = u_{xx} = 0$ at ends. Non-symmetric so some eigenvalues are complex! For small or zero heterogeneity find $2N - 2$ eigenvalues are small. Effectively, two modes in each of $N - 2$ interior patches, and one mode each in the two end patches. With increasing heterogeneity (say above 0.3), the gap decreases as a couple (or some) of the small eigenvalues become larger in magnitude.

Consider the spectrum with BCs of $u = u_x = 0$ at ends. Non-symmetric so some eigenvalues are complex! For small or zero heterogeneity find $2N - 4$ eigenvalues are small. Effectively, two modes in each of $N - 2$ interior patches. With increasing heterogeneity (say above 0.4), half $(N - 2)$ of the small eigenvalues become larger in magnitude (presumably some phase 'locking' to the heterogeneity): effectively forms two clusters of modes.

Set the desired microscale periodicity of the patterns, here 0.062, and on the microscale lattice of spacing 0.0062, correspondingly choose random microscale material coefficients. The wavenumber of this microscale patterns is $k_0 \approx 101$.

```
clear all
%global OurCf2eps, OurCf2eps=true %optional to save plots
basename = ['r' num2str(floor(1e5*rem(now,1))) mfilename]
Ra = 0.1  % Ra>0 leads to patterns
nGap = 8 % controls size of gap between patches
```

```
waveLength = 0.496688741721854 /nGap %for nPatch==5
%waveLength = 0.497630331753555 /nGap %for nPatch==7
%waveLength = 0.5 /nGap %for periodic case
nPtsPeriod = 10
dx = waveLength/nPtsPeriod
k0 = 2*pi/waveLength
```

Create some random heterogeneous coefficients.

```
heteroVar = 0.99*[1 1] % must be <2
cl = 1./(1-heteroVar/2+heteroVar.*rand(nPtsPeriod,2));
cRange = quantile(cl,0:0.5:1)
```

Establish global data struct `patches` for heterogeneous Swift–Hohenberg PDE with, on average, one patch per units length. Use seven patches to start with. Quartic (fourth-order) interpolation `ordCC` $= 4$ provides values for the inter-patch coupling conditions. Or use as high-order as possible with `ordCC` $= 0$.

```
nPatch = 5
nSubP = 2*nPtsPeriod+4 % +2 for not-edgyInt
%nSubP = 2*nGap*nPtsPeriod+4 % approx full-domain
Len = nPatch;
ordCC = 0;
dom.type='equispace';
dom.bcOffset=0.5
patches = configPatches1(@heteroSwiftHohenbergPDE,[0 Len],dom ...
    ,nPatch,ordCC,dx,nSubP,'EdgyInt',true,'nEdge',2 ...
    ,'hetCoeffs',cl);
xs=squeeze(patches.x);
```

3.0.1 Explore the Jacobian

Finds that with periodic patches, everything is symmetric. However, for equispace or chebyshev, the patch coupling is not symmetric—is this to be expected?

```
fprintf('\n**** Explore the Jacobian\n')
u0 = 0*patches.x;
u0([1:2 end-1:end],:) = nan;
patches.i = find(~isnan(u0));
```

```
nVars = numel(patches.i)
Jac = nan(nVars);
for j=1:nVars
    Jac(:,j)=theRes((1:nVars)==j,patches,k0,0,0);
end
```

Check on the symmetry of the Jacobian

```
nonSymmetric = norm(Jac-Jac')
Jac(abs(Jac)<1e-12)=0;
antiJac = Jac-Jac';
antiJac(abs(antiJac)<1e-12)=0;
figure(6),clf
spy(Jac,'.'),hold on, spy(antiJac,'rx'),hold off
if nonSymmetric>5e-9, warning('failed symmetry'),
else Jac = (Jac+Jac')/2; %tweak to symmetry
end
```

Compute eigenvalues and eigenvectors.

```
figure(5),clf
[evec,mEval] = eig(-Jac ,'vector');
[~,j]=sort(real(mEval));
mEval=mEval(j); evec=evec(:,j);
loglog(real(mEval),'.')
ylabel('$-\Re\lambda$')
ifOurCf2tex([basename 'Eval'])%optionally save
```

Explore sparse approximations of all the slowest together (lots of iterations required), or separately of the two clusters of the slowest (few iterations needed). First ascertain whether one or two clusters of small eigenvalues.

```
logGaps=diff(log10(real(mEval)));
[~,j]=sort(-logGaps);
%someLogGaps=[logGaps(j(1:5)) j(1:5)]
if logGaps(j(2))<0.4*logGaps(j(1)), nSlow=j(1)
else nSlow=min( sort(j(1:2)) , 3*nPatch)
end
log10Gap=logGaps(nSlow)
smallEvals=-mEval(1:nSlow(end)+2)
```

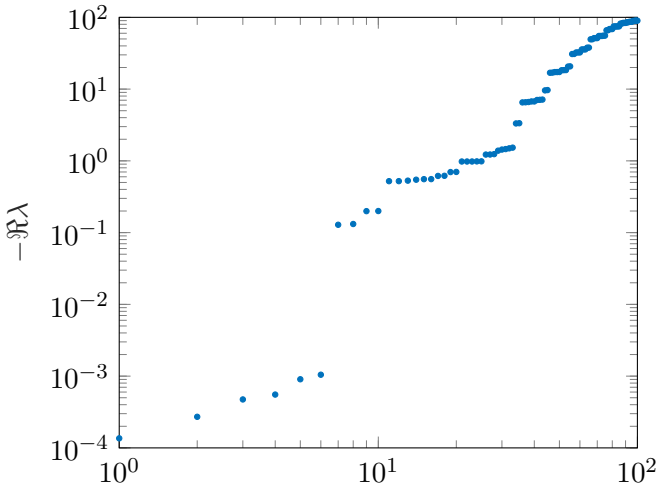


Figure 5: eigenvalues of the patch scheme on the heterogeneous Swift–Hohenberg PDE (linearised). With $N = 5$ patches and BCs of $u = u_x = 0$ at $x \in \{0, 5\}$, there are $2(N-2) = 6$ small eigenvalues, $|\lambda| < 0.001$, corresponding to six slow modes in the interior.

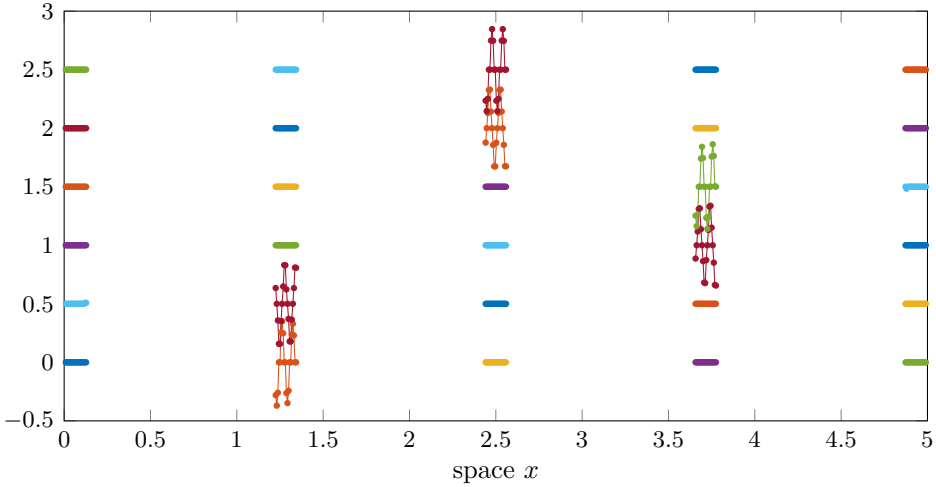
Second, make eigenvectors all real, sparsely approximate cluster modes via an algorithm developed from Hu et al. (2016), and plot. Figure 6 shows that each pair of basis vectors are phase-shifted by 90°.

```
js=find(imag(mEval)>0);
evec(:,js)=imag(evec(:,js));
evec=real(evec);
if numel(nSlow)==1, S = spcart(evec(:,1:nSlow));
else  S = spcart(evec(:,1:nSlow(1)));
   S = [S spcart(evec(:,nSlow(1)+1:nSlow(2))) ];
end;
figure(3),clf
vStep=ceil(max(abs(S(:)))*10+1)/10
for j=1:nSlow(end)
  u0(patches.i)=S(:,j);
  plot(xs,vStep*(j-1)+squeeze(u0),'.-'),hold on
end
hold off, xlabel('space $x$')
ifOurCf2tex([basename 'Evec'])%optionally save
```

Reorganise the eigenvectors to maybe clarify.

```
[i,j]=find(abs(S)>vStep/2);
j=find([1;diff(j)]);
[i,k]=sort(i(j));
```

Figure 6: sparse approximations of the eigenvectors of the six slow modes of Figure 5. Plotted are sparse basis vectors for the invariant space spanned by the six slow eigenvectors: each basis vector shifted vertically to separate. Thus a fair approximation is that there are effectively two modes for each of the $N - 2 = 3$ interior patches.



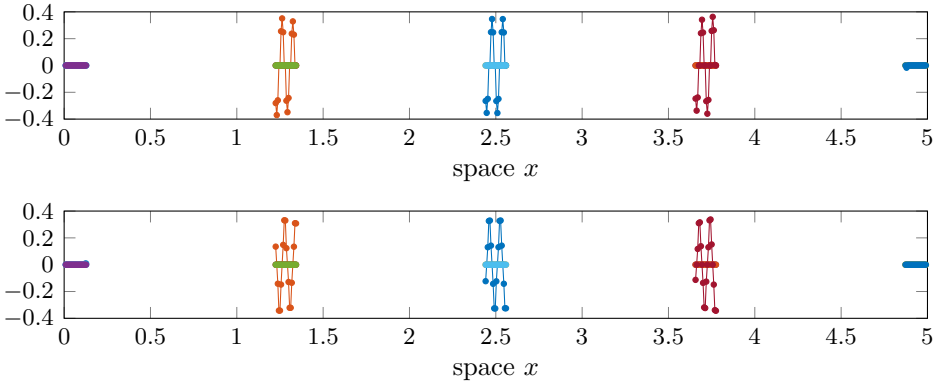

```
figure(4)
for p=1:2
  clf,subplot(2,1,1)
  for j=p:2:numel(k)
    u0(patches.i)=S(:,k(j));
    plot(xs,squeeze(u0),'.-'),hold on
  end% for j
  hold off, xlabel('space $x$')
  ifOurCf2tex([basename 'Evec' num2str(p)])%optionally save
end%for p
```

3.0.2 Find an equilibrium with fsolve

Start the search from some guess.

```
fprintf('\n**** Find equilibrium with fsolve\n')
u = 0.4*sin(2*pi/waveLength*patches.x);
```

Figure 7: sparse basis approximations for the invariant subspace of the six slow modes of Figure 5. A replot of Figure 6 but with three of the basis vectors superimposed in each of the two panels.



But set the pairs of patch-edge values to `Nan` in order to use `patches.i` to index the interior sub-patch points as they are the variables.

```
u([1:2 end-1:end],:) = nan;
patches.i = find(~isnan(u));
```

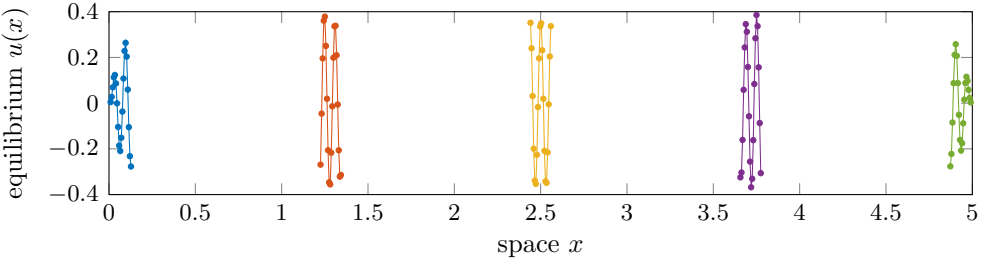
Seek the equilibrium, and report the norm of the residual, via the generic patch system wrapper `theRes` (Section 3.2).

```
tic
[u(patches.i),res] = fsolve(@(v) theRes(v,patches,k0,Ra,1) ...
    ,u(patches.i) ,optimoptions('fsolve','Display','off'));
solveTime = toc
normRes = norm(res)
if normRes>1e-7, warning('residual large: bad equilibrium'),end
```

Plot the equilibrium see Figure 8.

```
figure(1),clf
subplot(2,1,1)
plot(xs,squeeze(u),'.-')
xlabel('space $x$'),ylabel('equilibrium $u(x)$')
ifOurCf2tex([basename 'Equilib'])%optionally save
```

Figure 8: an equilibrium of the heterogeneous Swift–Hohenberg PDE *determined by the patch scheme*



3.0.3 Simulate 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 `patchSys1` (Section 6) to the microscale differential equations.

```
fprintf('\n**** Simulate in time\n')
u0 = 0*sin(2*pi/waveLength*patches.x)+0.1*randn(nSubP,1,1,nPatch);
tic
[ts,us] = ode15s(@patchSys1, [0 40], u0(:) ,[],patches,k0,Ra,1);
simulateTime = toc
us = reshape(us,length(ts),numel(patches.x(:)),[]);
```

Plot the simulation in Figure 4.

```
figure(2),clf
xs([1:2 end-1:end],:) = nan;
mesh(ts(1:3:end),xs(:),us(1:3:end,:)'),  view(65,60)
colormap(0.7*hsv)
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
ifOurCf2eps([basename 'Uxt'])
```

Fin.

3.1 Heterogeneous SwiftHohenberg PDE+BCs inside patches

As a microscale discretisation of Swift–Hohenberg PDE $u_t = -D[c_1(x)Du] + \mathrm{Ra}\,u - u^3$, where heterogeneous operator $D = 1 + \partial_x(c_2(x)\partial_x)/k_0^2$.

```
function ut=heteroSwiftHohenbergPDE(t,u,patches,k0,Ra,cubic)
  dx=diff(patches.x(1:2));  % microscale spacing
  i=3:size(u,1)-2;  % interior points in patches
```

Code a couple of different boundary conditions of zero function and derivative(s) at left-end of left-patch, and right-end of right-patch. For slightly simpler coding, squeeze out the two singleton dimensions.

```
  u = squeeze(u);
  if ~patches.periodic
    switch 1
    case 1 % these are u=u_x=0
      u(1:2,1)=0;
      u(end-1:end,end)=0;
    case 2 % these are u=u_{xx}=0
      u(1:2,1) = [-u(3,1); 0];
      u(end-1:end,end) = [0; -u(end-2,end)];
    end% case
  end%if
```

Here code straightforward centred discretisation in space.

```
  ut = nan+u;          % preallocate output array
  v = u(2:end-1,:)+diff(patches.cs(:      ,2).*diff(u))/dx^2/k0^2;
  v = v.*patches.cs(2:end,1);
  v = v(2:end-1,:)+diff(patches.cs(2:end-1,2).*diff(v))/dx^2/k0^2;
  ut(i,:) = -v +Ra*u(i,:) -cubic*u(i,:).^3;
end
```

3.2 `theRes()`: a wrapper function

This functions converts a vector of values into the interior values of the patches, then evaluates the time derivative of the system at time zero, and returns the vector of patch-interior time derivatives.

```
function f=theRes(u,patches,k0,Ra,cubic)
  v=nan(size(patches.x));
  v(patches.i) = u;
  f = patchSys1(0,v(:),patches,k0,Ra,cubic);
  f = f(patches.i);
end%function theRes
```

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

Couples 1D patches across 1D space by computing their edge values from macroscale interpolation of either the mid-patch value (Roberts 2003; Roberts and Kevrekidis 2007), or the patch-core average (Bunder, Roberts, and Kevrekidis 2017), or the opposite next-to-edge values (Bunder, Kevrekidis, and Roberts 2021)—this last alternative often maintains symmetry. This function is primarily used by `patchSys1()` but is also useful for user graphics. When using core averages (not fully implemented), assumes the averages are sensible macroscale variables: then patch edge values are determined by macroscale interpolation of the core averages (Bunder, Roberts, and Kevrekidis 2017). [1]

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

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

Input

- `u` is a vector/array of length $\texttt{nSubP}\cdot\texttt{nVars}\cdot\texttt{nEnsem}\cdot\texttt{nPatch}$ where there are $\texttt{nVars}\cdot\texttt{nEnsem}$ field values at each of the points in the $\texttt{nSubP}\times\texttt{nPatch}$ multiscale spatial grid.
- `patches` a struct largely set by `configPatches1()`, and which includes the following.
  - `.x` is $\texttt{nSubP}\times 1\times 1\times\texttt{nPatch}$ array of the spatial locations x_{iI} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index i, but may be variable spaced in macroscale index I.
  - `.ordCC` is order of interpolation, integer ≥ -1.
  - `.periodic` indicates whether macroscale is periodic domain, or alternatively that the macroscale has left and right boundaries so interpolation is via divided differences.

[1] Script `patchEdgeInt1test.m` verifies this code.

- `.stag` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation, and zero for ordinary grid.
- `.Cwtsr` and `.Cwtsl` are the coupling coefficients for finite width interpolation—when invoking a periodic domain.
- `.EdgyInt`, true/false, for determining patch-edge values by interpolation: true, from opposite-edge next-to-edge values (often preserves symmetry); false, from centre-patch values (original scheme).
- `.nEdge`, for each patch, the number of edge values set by interpolation at the edge regions of each patch (default is one).
- `.nEnsem` the number of realisations in the ensemble.
- `.parallel` whether serial or parallel.
- `.nCore` [2]

Output

- `u` is 4D array, `nSubP` × `nVars` × `nEnsem` × `nPatch`, of the fields with edge values set by interpolation.

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

```
116   if max(abs(imag(u(:))))<1e-9*max(abs(u(:)))
117        uclean=@(u) real(u);
118   else uclean=@(u) u;
119   end
```

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

```
127 [nx,~,~,Nx] = size(patches.x);
128 nEnsem = patches.nEnsem;
129 nVars = round(numel(u)/numel(patches.x)/nEnsem);
130 assert(numel(u) == nx*nVars*nEnsem*Nx ...
131   ,'patchEdgeInt1: input u has wrong size for parameters')
132 u = reshape(u,nx,nVars,nEnsem,Nx);
```

[2] **ToDo:** introduced sometime but not fully implemented yet, because prefer ensemble

If the user has not defined the patch core, then we assume it to be a single point in the middle of the patch, unless we are interpolating from next-to-edge values.

These index vectors point to patches and their two immediate neighbours, for periodic domain.

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

Implement multiple width edges by folding Subsample x coordinates, noting it is only differences that count *and* the microgrid x spacing must be uniform.

```
x = patches.x;
if patches.nEdge>1
  nEdge = patches.nEdge;
  x = x(1:nEdge:nx,:,:,:);
  nx = nx/nEdge;
  u = reshape(u,nEdge,nx,nVars,nEnsem,Nx);
  nVars = nVars*nEdge;
  u = reshape( permute(u,[2 1 3:5]) ,nx,nVars,nEnsem,Nx);
end%if patches.nEdge
```

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

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

4.1 Periodic macroscale interpolation schemes

```
if patches.periodic
```

Get the size ratios of the patches, then use finite width stencils or spectral.

```
r = patches.ratio(1);
if patches.ordCC>0 % then finite-width polynomial interpolation
```

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

```
198   if patches.EdgyInt % interpolate next-to-edge values
199     Ux = u([2 nx-1],:,:,I);
200   else % interpolate mid-patch values/sums
201     Ux = sum( u((i0-c):(i0+c),:,:,I) ,1);
202   end;
```

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

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

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

```
220   if patches.parallel
221     dmu = zeros(szUx0,patches.codist); % 5D
222   else
223     dmu = zeros(szUx0); % 5D
224   end
```

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

```
231   if patches.stag % use only odd numbered neighbours
232     dmu(:,:,:,I,1) = (Ux(:,:,:,Ip)+Ux(:,:,:,Im))/2; % \mu
233     dmu(:,:,:,I,2) = (Ux(:,:,:,Ip)-Ux(:,:,:,Im)); % \delta
234     Ip = Ip(Ip); Im = Im(Im); % increase shifts to \pm2
235   else % standard
236     dmu(:,:,:,I,1) = (Ux(:,:,:,Ip)-Ux(:,:,:,Im))/2; % \mu\delta
237     dmu(:,:,:,I,2) = (Ux(:,:,:,Ip)-2*Ux(:,:,:,I) ...
238                       +Ux(:,:,:,Im)); % \delta^2
239   end%if patches.stag
```

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

```
246   for k = 3:patches.ordCC
247     dmu(:,:,:,:,k) =       dmu(:,:,:,Ip,k-2) ...
248       -2*dmu(:,:,:,I,k-2) +dmu(:,:,:,Im,k-2);
249   end
```

Interpolate macro-values to be Dirichlet edge values for each patch (Roberts and Kevrekidis 2007; Bunder, Roberts, and Kevrekidis 2017), using weights computed in `configPatches1()`. Here interpolate to specified order.

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

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

For a non-trivial core then more needs doing: the core (one or more) of each patch interpolates to the edge action regions. When more than one in the core, the edge is set depending upon near edge values so the average near the edge is correct.

```
  else% patches.nCore>1
    error('not yet considered, july--dec 2020 ??')
    u(nx,:,:,I) = Ux(:,:,I)*(1-patches.stag) ...
      + reshape(-sum(u((nx-patches.nCore+1):(nx-1),:,:,I),1) ...
      + sum( patches.Cwtsr.*dmu ),Nx,nVars);
    u(1,:,:,I) = Ux(:,:,I)*(1-patches.stag) ...
      + reshape(-sum(u(2:patches.nCore,:,:,I),1)  ...
      + sum( patches.Cwtsl.*dmu ),Nx,nVars);
  end%if patches.nCore
```

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

```
else% patches.ordCC<=0, spectral interpolation
```

As the macroscale fields are N-periodic, the macroscale Fourier transform writes the centre-patch values as $U_j = \sum_k C_k e^{ik2\pi j/N}$. Then the edge-patch values $U_{j\pm r} = \sum_k C_k e^{ik2\pi/N(j\pm r)} = \sum_k C'_k e^{ik2\pi j/N}$ where $C'_k = C_k e^{ikr2\pi/N}$. For `Nx` patches we resolve 'wavenumbers' $|k| < \texttt{Nx}/2$, so set row vector $\texttt{ks} = k2\pi/N$ for 'wavenumbers' $k = (0\,,1\,,\ldots,k_{\max}\,,-k_{\max}\,,\ldots,-1)$ for odd N, and $k = (0\,,1\,,\ldots,k_{\max}\,,(k_{\max}+1)\,,-k_{\max}\,,\ldots,-1)$ 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. [3] [4]

```
323   if patches.stag % transform by doubling the number of fields
324     v = nan(size(u)); % currently to restore the shape of u
325     u = [u(:,:,:,1:2:Nx) u(:,:,:,2:2:Nx)];
326     stagShift = 0.5*[ones(1,nVars) -ones(1,nVars)];
327     iV = [nVars+1:2*nVars 1:nVars]; % scatter interp to alternate fie
328     r = r/2;   % ratio effectively halved
329     Nx = Nx/2; % halve the number of patches
330     nVars = nVars*2;   % double the number of fields
331   else % the values for standard spectral
332     stagShift = 0;
333     iV = 1:nVars;
334   end%if patches.stag
```

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

```
341   kMax = floor((Nx-1)/2);
342   ks = shiftdim( ...
343       2*pi/Nx*(mod((0:Nx-1)+kMax,Nx)-kMax) ...
344       ,-2);
```

Compute the Fourier transform across patches of the patch centre or next-to-edge values for all the fields. If there are an even number of points, then if complex, treat as positive wavenumber, but if real, treat as cosine. When using an ensemble of configurations, different configurations might be coupled to each other, as specified by `patches.le` and `patches.ri`.

```
357 if ~patches.EdgyInt
358     Cleft = fft(u(i0  ,:,:,:),[],4);
359     Cright = Cleft;
360 else
361     Cleft = fft(u(2   ,:,:,:),[],4);
362     Cright= fft(u(nx-1,:,:,:),[],4);
363 end
```

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

[3] **ToDo:** Have not yet tested whether works for Edgy Interpolation.

[4] **ToDo:** Have not yet implemented multiple edge values for a staggered grid as I am uncertain whether it makes any sense.

```
370   u(nx,iV,patches.ri,:) = uclean( ifft( ...
371       Cleft.*exp(1i*ks.*(stagShift+r)) ,[],4));
372   u(1 ,iV,patches.le,:) = uclean( ifft( ...
373       Cright.*exp(1i*ks.*(stagShift-r)) ,[],4));
```

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

```
381 if patches.stag
382   nVars = nVars/2;
383   u=reshape(u,nx,nVars,2,nEnsem,Nx);
384   Nx = 2*Nx;
385   v(:,:,:,1:2:Nx) = u(:,:,1,:,:);
386   v(:,:,:,2:2:Nx) = u(:,:,2,:,:);
387   u = v;
388 end%if patches.stag
389 end%if patches.ordCC
```

4.2 Non-periodic macroscale interpolation

```
397 else% patches.periodic false
398 assert(~patches.stag, ...
399 'not yet implemented staggered grids for non-periodic')
```

Determine the order of interpolation `p`, and hence size of the (forward) divided difference table in `F`.

```
406 if patches.ordCC<1, patches.ordCC = Nx-1; end
407 p = min(patches.ordCC,Nx-1);
408 F = nan(patches.EdgyInt+1,nVars,nEnsem,Nx,p+1);
```

Set function values in first 'column' of the table for every variable and across ensemble. For `EdgyInt`, the 'reversal' of the next-to-edge values are because their values are to interpolate to the opposite edge of each patch.

```
418   if patches.EdgyInt % interpolate next-to-edge values
419     F(:,:,:,:,1) = u([nx-1 2],:,:,I);
420     X(:,:,:,:)   = x([nx-1 2],:,:,I);
421   else % interpolate mid-patch values/sums
422     F(:,:,:,:,1) = sum( u((i0-c):(i0+c),:,:,I) ,1);
423     X(:,:,:,:)   = x(i0,:,:,I);
424   end;
```

Compute table of (forward) divided differences (e.g., Wikipedia 2022) for every variable and across ensemble.

```
for q = 1:p
  i = 1:Nx-q;
  F(:,:,:,i,q+1) = (F(:,:,:,i+1,q)-F(:,:,:,i,q)) ...
                  ./(X(:,:,:,i+q)  -X(:,:,:,i));
end
```

Now interpolate to the edge-values at locations `Xedge`.

```
Xedge = x([1 nx],:,:,:);
```

Code Horner's evaluation of the interpolation polynomials. Indices `i` are those of the left end of each interpolation stencil because the table is of forward differences.[5] First alternative: the case of order p interpolation across the domain, asymmetric near the boundary. Use this first alternative for now.

```
if true
  i = max(1,min(1:Nx,Nx-ceil(p/2))-floor(p/2));
  Uedge = F(:,:,:,i,p+1);
  for q = p:-1:1
    Uedge = F(:,:,:,i,q)+(Xedge-X(:,:,:,i+q-1)).*Uedge;
  end
```

Second alternative: lower the degree of interpolation near the boundary to maintain the band-width of the interpolation. Such symmetry might be essential for multi-D. [6]

```
else%if false
  i = max(1,I-floor(p/2));
```

For the tapering order of interpolation, form the interior mask `Q` (logical) that signifies which interpolations are to be done at order `q`. This logical mask spreads by two as each order `q` decreases.

```
  Q = (I-1>=floor(p/2)) & (Nx-I>=p/2);
  Imid = floor(Nx/2);
```

[5] For EdgyInt, perhaps interpret odd order interpolation in such a way that first-order interpolations reduces to appropriate linear interpolation so that as patches abut the scheme is 'full-domain'. May mean left-edge and right-edge have different indices. Explore sometime??

[6] The aim is to preserve symmetry?? Does it?? As of Jan 2023 it only partially does—fails near boundaries, and maybe fails with uneven spacing.

Initialise to highest divide difference, surrounded by zeros.

```
491   Uedge = zeros(patches.EdgyInt+1,nVars,nEnsem,Nx);
492   Uedge(:,:,:,Q) = F(:,:,:,i(Q),p+1);
```

Complete Horner evaluation of the relevant polynomials.

```
498   for q = p:-1:1
499     Q = [Q(2:Imid) true(1,2) Q(Imid+1:end-1)]; % spread mask
500     Uedge(:,:,:,Q) = F(:,:,:,i(Q),q) ...
501         +(Xedge(:,:,:,Q)-X(:,:,:,i(Q)+q-1)).*Uedge(:,:,:,Q);
502   end%for q
503 end%if
```

Finally, insert edge values into the array of field values, using the required ensemble shifts.

```
511 u(1 ,:,patches.le,I) = Uedge(1,:,:,I);
512 u(nx,:,patches.ri,I) = Uedge(2,:,:,I);
```

We want a user to set the extreme patch edge values according to the microscale boundary conditions that hold at the extremes of the domain. Consequently, unless testing, override their computed interpolation values with `NaN`.

```
522 if isfield(patches,'intTest')&&patches.intTest
523 else % usual case
524   u( 1,:,:, 1) = nan;
525   u(nx,:,:,Nx) = nan;
526 end%if
```

End of the non-periodic interpolation code.

```
532 end%if patches.periodic
```

Unfold multiple edges No need to restore x.

```
539 if patches.nEdge>1
540   nVars = nVars/nEdge;
541   u = reshape( u ,nx,nEdge,nVars,nEnsem,Nx);
542   nx = nx*nEdge;
543   u = reshape( permute(u,[2 1 3:5]) ,nx,nVars,nEnsem,Nx);
544 end%if patches.nEdge
```

Fin, returning the 4D array of field values.

5 configPatches1(): configure spatial patches in 1D

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys1()`. Section 5.1 lists an example of its use.

```
19 function patches = configPatches1(fun,Xlim,Dom ...
20     ,nPatch,ordCC,dx,nSubP,varargin)
21 version = '2023-03-23';
```

Input If invoked with no input arguments, then executes an example of simulating Burgers' PDE—see Section 5.1 for the example code.

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)` or `fun(t,u,patches,...)`, that computes time derivatives (or time-steps) of quantities on the 1D micro-grid within all the 1D patches.
- `Xlim` give the macro-space spatial domain of the computation, namely the interval [`Xlim(1)`, `Xlim(2)`].
- `Dom` sets the type of macroscale conditions for the patches, and reflects the type of microscale boundary conditions of the problem. If `Dom` is `NaN` or `[]`, then the field `u` is macro-periodic in the 1D spatial domain, and resolved on equi-spaced patches. If `Dom` is a character string, then that specifies the `.type` of the following structure, with `.bcOffset` set to the default zero. Otherwise `Dom` is a structure with the following components.
  - `.type`, string, of either `'periodic'` (the default), `'equispace'`, `'chebyshev'`, `'usergiven'`. For all cases except `'periodic'`, users *must* code into `fun` the micro-grid boundary conditions that apply at the left(right) edge of the leftmost(rightmost) patches.
  - `.bcOffset`, optional one or two element array, in the cases of `'equispace'` or `'chebyshev'` the patches are placed so the left/right macroscale boundaries are aligned to the left/right edges of the corresponding extreme patches, but offset by `bcOffset` of the sub-patch micro-grid spacing. For example, use `bcOffset=0` when applying Dirichlet boundary values on the extreme edge micro-grid points, whereas use `bcOffset=0.5` when applying Neumann boundary conditions halfway between the extreme edge micro-grid points.

  - `.X`, optional array, in the case `'usergiven'` it specifies the locations of the centres of the `nPatch` patches—the user is responsible it makes sense.

- `nPatch` is the number of equi-spaced spatial patches.

- `ordCC`, must be ≥ -1, is the 'order' of interpolation across empty space of the macroscale patch values to the edge of the patches for inter-patch coupling: where `ordCC` of 0 or -1 gives spectral interpolation; and `ordCC` being odd specifies staggered spatial grids.

- `dx` (real) is usually the sub-patch micro-grid spacing in x.

  However, if `Dom` is `NaN` (as for pre-2023), then `dx` actually is `ratio`, namely the ratio of (depending upon `EdgyInt`) either the half-width or full-width of a patch to the equi-spacing of the patch mid-points—adjusted a little when `nEdge` > 1. So either `ratio` $= \frac{1}{2}$ means the patches abut and `ratio` $= 1$ is overlapping patches as in holistic discretisation, or `ratio` $= 1$ means the patches abut. Small `ratio` should greatly reduce computational time.

- `nSubP` is the number of equi-spaced microscale lattice points in each patch. If not using `EdgyInt`, then `nSubP/nEdge` must be odd integer so that there is/are centre-patch lattice point(s). So for the defaults of `nEdge` $= 1$ and not `EdgyInt`, then `nSubP` must be odd.

- `nEdge`, *optional*, default=1, for each patch, the number of edge values set by interpolation at the edge regions of each patch. The default is one (suitable for microscale lattices with only nearest neighbour interactions).

- `EdgyInt`, true/false, *optional*, default=false. If true, then interpolate to left/right edge-values from right/left next-to-edge values. If false or omitted, then interpolate from centre-patch values.

- `nEnsem`, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.

- `hetCoeffs`, *optional*, default empty. Supply a 1D or 2D array of microscale heterogeneous coefficients to be used by the given microscale `fun` in each patch. Say the given array `cs` is of size $m_x \times n_c$, where n_c is the number of different sets of coefficients. The coefficients are to be the same for each and every patch; however, macroscale variations are

catered for by the n_c coefficients being n_c parameters in some macroscale formula.

  - If `nEnsem` $= 1$, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the first point in each patch. Best accuracy usually obtained when the periodicity of the coefficients is a factor of `nSubP-2*nEdge` for `EdgyInt`, or a factor of `(nSubP-nEdge)/2` for not `EdgyInt`,
  - If `nEnsem` > 1 (value immaterial), then reset `nEnsem` $:= m_x$ and construct an ensemble of all m_x phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When `EdgyInt` is true, and when the coefficients are diffusivities/elasticities, then this coupling cunningly preserves symmetry.

- `nCore`, *optional-experimental*, default one, but if more, and only for non-EdgyInt, then interpolates from an average over the core of a patch, a core of size ??. Then edge values are set according to interpolation of the averages?? or so that average at edges is the interpolant??

- `'parallel'`, true/false, *optional*, default=false. If false, then all patch computations are on the user's main CPU—although a user may well separately invoke, say, a GPU to accelerate sub-patch computations.

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

Output The struct `patches` is created and set with the following components. If no output variable is provided for `patches`, then make the struct available as a global variable.[7]

```
188 if nargout==0, global patches, end
189 patches.version = version;
```

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

- `.fun` is the name of the user's function `fun(t,u,patches)` or `fun(t,u)` or `fun(t,u,patches,...)`, that computes the time derivatives (or steps) on the patchy lattice.
- `.ordCC` is the specified order of inter-patch coupling.
- `.periodic`: either true, for interpolation on the macro-periodic domain; or false, for general interpolation by divided differences over non-periodic domain or unevenly distributed patches.
- `.stag` is true for interpolation using only odd neighbouring patches as for staggered grids, and false for the usual case of all neighbour coupling.
- `.Cwtsr` and `.Cwtsl`, only for macro-periodic conditions, are the `ordCC`-vector of weights for the inter-patch interpolation onto the right and left edges (respectively) with patch:macroscale ratio as specified or as derived from `dx`.
- `.x` (4D) is $\texttt{nSubP} \times 1 \times 1 \times \texttt{nPatch}$ array of the regular spatial locations x_{iI} of the ith microscale grid point in the Ith patch.
- `.ratio`, only for macro-periodic conditions, is the size ratio of every patch.
- `.nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.
- `.le`, `.ri` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.
- `.cs` either
  - `[]` 0D, or
  - if $\texttt{nEnsem} = 1$, $(\texttt{nSubP(1)} - 1) \times n_c$ 2D array of microscale heterogeneous coefficients, or
  - if $\texttt{nEnsem} > 1$, $(\texttt{nSubP(1)} - 1) \times n_c \times m_x$ 3D array of m_x ensemble of phase-shifts of the microscale heterogeneous coefficients.
- `.parallel`, logical: true if patches are distributed over multiple CPUs/ cores for the Parallel Computing Toolbox, otherwise false (the default is to activate the *local* pool).
- `.codist`, *optional*, describes the particular parallel distribution of arrays over the active parallel pool.

5.1 If no arguments, then execute an example

```
if nargin==0
disp('With no arguments, simulate example of Burgers PDE')
```

The code here shows one way to get started: a user's script may have the following three steps ("$\mapsto$" denotes function recursion).

1. configPatches1
2. ode15s integrator $\mapsto$ patchSys1 $\mapsto$ 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, with micro-grid spacing 0.06, and with seven microscale points forming each patch.

```
global patches
patches = configPatches1(@BurgersPDE, [0 2*pi], ...
    'periodic', 8, 0, 0.06, 7);
```

Set some initial condition, with some microscale randomness.

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

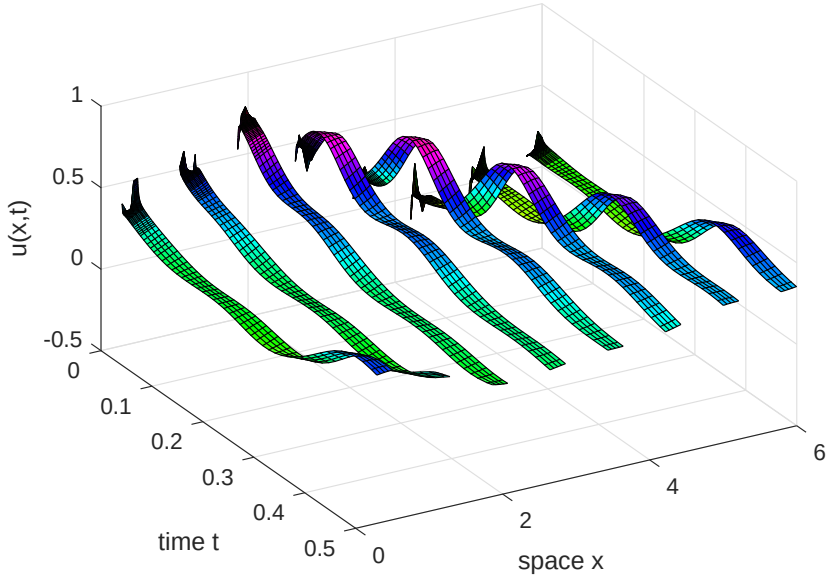
Simulate in time using a standard stiff integrator and the interface function `patchSys1()` (Section 6).

```
if ~exist('OCTAVE_VERSION','builtin')
[ts,us] = ode15s( @patchSys1,[0 0.5],u0(:));
else % octave version
[ts,us] = odeOcts(@patchSys1,[0 0.5],u0(:));
end
```

Plot the simulation using only the microscale values interior to the patches: either set x-edges to `nan` to leave the gaps; or use `patchEdgyInt1` to re-interpolate correct patch edge values and thereby join the patches. Figure 9 illustrates an example simulation in time generated by the patch scheme applied to Burgers' PDE.

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

Figure 9: field $u(x,t)$ of the patch scheme applied to Burgers' PDE.




```
end
mesh(ts,patches.x(:),us)
view(60,40), colormap(0.7*hsv)
title('Burgers PDE: patches in space, continuous time')
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
```

Upon finishing execution of the example, optionally save the graph to be shown in Figure 9, then exit this function.

```
ifOurCf2eps(mfilename)
return
end%if nargin==0
```

5.2 Parse input arguments and defaults

```
p = inputParser;
fnValidation = @(f) isa(f, 'function_handle'); %test for fn name
addRequired(p,'fun',fnValidation);
addRequired(p,'Xlim',@isnumeric);
%addRequired(p,'Dom'); % nothing yet decided
```

```
addRequired(p,'nPatch',@isnumeric);
addRequired(p,'ordCC',@isnumeric);
addRequired(p,'dx',@isnumeric);
addRequired(p,'nSubP',@isnumeric);
addParameter(p,'nEdge',1,@isnumeric);
addParameter(p,'EdgyInt',false,@islogical);
addParameter(p,'nEnsem',1,@isnumeric);
addParameter(p,'hetCoeffs',[],@isnumeric);
addParameter(p,'parallel',false,@islogical);
addParameter(p,'nCore',1,@isnumeric);
parse(p,fun,Xlim,nPatch,ordCC,dx,nSubP,varargin{:});
```

Set the optional parameters.

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

Check parameters.

```
assert(Xlim(1)<Xlim(2) ...
      ,'two entries of Xlim must be ordered increasing')
assert((mod(ordCC,2)==0)|(patches.nEdge==1) ...
      ,'Cannot yet have nEdge>1 and staggered patch grids')
assert(3*patches.nEdge<=nSubP ...
      ,'too many edge values requested')
assert(rem(nSubP,patches.nEdge)==0 ...
      ,'nSubP must be integer multiple of nEdge')
if ~patches.EdgyInt, assert(rem(nSubP/patches.nEdge,2)==1 ...
      ,'for non-edgyInt, nSubP/nEdge must be odd integer')
      end
if (patches.nEnsem>1)&(patches.nEdge>1)
      warning('not yet tested when both nEnsem and nEdge non-one')
      end
if patches.nCore>1
    warning('nCore>1 not yet tested in this version')
    end
```

For compatibility with pre-2023 functions, if parameter `Dom` is `Nan`, then we set the `ratio` to be the value of the so-called `dx` parameter.

```
412 if ~isstruct(Dom), pre2023=isnan(Dom);
413 else pre2023=false; end
414 if pre2023, ratio=dx; dx=nan; end
```

Default macroscale conditions are periodic with evenly spaced patches.

```
422 if isempty(Dom), Dom=struct('type','periodic'); end
423 if (~isstruct(Dom))&isnan(Dom), Dom=struct('type','periodic'); end
```

If `Dom` is a string, then just set type to that string, and then get corresponding defaults for others fields.

```
431 if ischar(Dom), Dom=struct('type',Dom); end
```

Check what is and is not specified, and provide default of Dirichlet boundaries if no `bcOffset` specified when needed.

```
439 patches.periodic=false;
440 switch Dom.type
441 case 'periodic'
442     patches.periodic=true;
443     if isfield(Dom,'bcOffset')
444     warning('bcOffset not available for Dom.type = periodic'), end
445     if isfield(Dom,'X')
446     warning('X not available for Dom.type = periodic'), end
447 case {'equispace','chebyshev'}
448     if ~isfield(Dom,'bcOffset'), Dom.bcOffset=[0;0]; end
449     if length(Dom.bcOffset)==1
450         Dom.bcOffset=repmat(Dom.bcOffset,2,1); end
451     if isfield(Dom,'X')
452     warning('X not available for Dom.type = equispace or chebyshev')
453     end
454 case 'usergiven'
455     if isfield(Dom,'bcOffset')
456     warning('bcOffset not available for usergiven Dom.type'), end
457     assert(isfield(Dom,'X'),'X required for Dom.type = usergiven')
458 otherwise
459     error([Dom.type ' is unknown Dom.type'])
460 end%switch Dom.type
```

5.3 The code to make patches and interpolation

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

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

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

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

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

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

Check for staggered grid and periodic case.

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

Third, set the centre of the patches in the macroscale grid of patches, depending upon `Dom.type`.

```
switch Dom.type
```

The periodic case is evenly spaced within the spatial domain. Store the size ratio in `patches`.

```
case 'periodic'
  X=linspace(Xlim(1),Xlim(2),nPatch+1);
  DX=X(2)-X(1);
  X=X(1:nPatch)+diff(X)/2;
  pEI=patches.EdgyInt;% abbreviation
  pnE=patches.nEdge;  % abbreviation
  if pre2023, dx = ratio*DX/(nSubP-pnE*(1+pEI))*(2-pEI);
  else        ratio = dx/DX*(nSubP-pnE*(1+pEI))/(2-pEI);  end
  patches.ratio=ratio;
```

In the case of macro-periodicity, precompute the weightings to interpolate field values for coupling. [8]

```
532   if ordCC>0
533     [Cwtsr,Cwtsl] = patchCwts(ratio,ordCC,patches.stag);
534     patches.Cwtsr = Cwtsr;  patches.Cwtsl = Cwtsl;
535   end
```

The equi-spaced case is also evenly spaced but with the extreme edges aligned with the spatial domain boundaries, modified by the offset. [9]

```
545 case 'equispace'
546   X=linspace(Xlim(1)+((nSubP-1)/2-Dom.bcOffset(1))*dx ...
547              ,Xlim(2)-((nSubP-1)/2-Dom.bcOffset(2))*dx ,nPatch);
548   DX=diff(X(1:2));
549   width=(1+patches.EdgyInt)/2*(nSubP-1-patches.EdgyInt)*dx;
550   if DX<width*0.999999
551      warning('too many equispace patches (double overlapping)')
552      end
```

The Chebyshev case is spaced according to the Chebyshev distribution in order to reduce macro-interpolation errors, $X_i \propto -\cos(i\pi/N)$, but with the extreme edges aligned with the spatial domain boundaries, modified by the offset, and modified by possible 'boundary layers'. [10]

```
569 case 'chebyshev'
570   halfWidth=dx*(nSubP-1)/2;
571   X1 = Xlim(1)+halfWidth-Dom.bcOffset(1)*dx;
572   X2 = Xlim(2)-halfWidth+Dom.bcOffset(2)*dx;
573 %  X = (X1+X2)/2-(X2-X1)/2*cos(linspace(0,pi,nPatch));
```

Search for total width of 'boundary layers' so that in the interior the patches are non-overlapping Chebyshev. But the width for assessing overlap of patches is the following variable `width`. We need to find `b`, the number of patches 'glued' together at the boundaries.

[8] **ToDo:** Might sometime extend to coupling via derivative values.

[9] **ToDo:** This warning needs refinement for multi-edges??

[10] However, maybe overlapping patches near a boundary should be viewed as some sort of spatial analogue of the 'christmas tree' of projective integration and its projection to a slow manifold. Here maybe the overlapping patches allow for a 'christmas tree' approach to the boundary layers. Needs to be explored??

```
  pEI=patches.EdgyInt;% abbreviation
  pnE=patches.nEdge;  % abbreviation
  width=(1+pEI)/2*(nSubP-pnE-pEI*pnE)*dx;
  for b=0:2:nPatch-2
    DXmin=(X2-X1-b*width)/2*( 1-cos(pi/(nPatch-b-1)) );
    if DXmin>width, break, end
  end
  if DXmin<width*0.999999
     warning('too many Chebyshev patches (mid-domain overlap)')
     end
```

Assign the centre-patch coordinates.

```
  X = [ X1+(0:b/2-1)*width ...
        (X1+X2)/2-(X2-X1-b*width)/2*cos(linspace(0,pi,nPatch-b)) ...
        X2+(1-b/2:0)*width ];
```

The user-given case is entirely up to a user to specify, we just force it to have the correct shape of a row.

```
case 'usergiven'
  X = reshape(Dom.X,1,[]);
end%switch Dom.type
```

Fourth, construct the microscale grid in each patch, centred about the given mid-points `X`. Reshape the grid to be 4D to suit dimensions (micro,Vars,Ens,macro).

```
xs = dx*( (1:nSubP)-mean(1:nSubP) );
patches.x = reshape( xs'+X ,nSubP,1,1,nPatch);
```

5.4 Set ensemble inter-patch communication

For `EdgyInt` or centre interpolation respectively,

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

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

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

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

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

```
651 nE = patches.nEnsem;
652 patches.le = 1:nE;
653 patches.ri = 1:nE;
```

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

```
665 if ~isempty(cs)
666   [mx,nc] = size(cs);
667   nx = nSubP(1);
668   cs = repmat(cs,nSubP,1);
```

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

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

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

```
685     patches.nEnsem = mx;
686     patches.cs = nan(nx-1,nc,mx);
687     for i = 1:mx
688         is = (i:i+nx-2);
689         patches.cs(:,:,i) = cs(is,:);
690     end
691     patches.cs = reshape(patches.cs,nx-1,nc,[]);
```

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

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

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

```
710 if ratio*patches.nEnsem>0.9, warning( ...
711 'Probably poor scale separation in ensemble of coupled phase-shifts')
712 scaleSeparationParameter = ratio*patches.nEnsem
713 end
```

End the two if-statements.

```
719   end%if-else nEnsem>1
720 end%if not-empty(cs)
```

If parallel code then first assume this is not within an `spmd`-environment, and so we invoke `spmd...end` (which starts a parallel pool if not already started). At this point, the global `patches` is copied for each worker processor and so it becomes *composite* when we distribute any one of the fields. Hereafter, *all fields in the global variable* `patches` *must only be referenced within an* `spmd`*-environment.*[11]

```
739 if patches.parallel
740 %  theparpool=gcp()
741   spmd
```

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

```
748   pari = 1;
749   patches.codist=codistributor1d(3+pari);
```

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

```
759   switch pari
760     case 1, patches.x=codistributed(patches.x,patches.codist);
761   otherwise
762     error('should never have bad index for parallel distribution')
763   end%switch
764   end%spmd
```

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

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

```
772 else% not parallel
773   if isfield(patches,'codist'), rmfield(patches,'codist'); end
774 end%if-parallel
```

Fin

```
783 end% function
```

6 patchSys1(): interface 1D space to time integrators

To simulate in time with 1D spatial patches we often need to interface a user's time derivative function with time integration routines such as `ode23` or `PIRK2`. This function provides an interface. Communicate patch-design variables (Section 5) either via the global struct `patches` or via an optional third argument. `patches` is required for the parallel computing of `spmd`, or if parameters are to be passed though to the user microscale function.

```
23 function dudt=patchSys1(t,u,patches,varargin)
24 if nargin<3, global patches, end
```

Input

- `u` is a vector/array of length $\texttt{nSubP} \cdot \texttt{nVars} \cdot \texttt{nEnsem} \cdot \texttt{nPatch}$ where there are $\texttt{nVars} \cdot \texttt{nEnsem}$ field values at each of the points in the $\texttt{nSubP} \times \texttt{nPatch}$ grid.
- `t` is the current time to be passed to the user's time derivative function.
- `patches` a struct set by `configPatches1()` with the following information used here.
  - `.fun` is the name of the user's function `fun(t,u,patches,...)` that computes the time derivatives on the patchy lattice. The array `u` has size $\texttt{nSubP} \times \texttt{nVars} \times \texttt{nEnsem} \times \texttt{nPatch}$. Time derivatives should be computed into the same sized array, then herein the patch edge values are overwritten by zeros.

- `.x` is `nSubP` $\times 1 \times 1 \times$ `nPatch` array of the spatial locations x_i of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale.

- `varargin`, optional, is arbitrary number of parameters to be passed onto the users time-derivative function as specified in configPatches1.

Output

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

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

```
82 sizeu = size(u);
83 u = patchEdgeInt1(u,patches);
```

Ask the user function for the time derivatives computed in the array, overwrite its edge values with the dummy value of zero (as `ode15s` chokes on NaNs), then return to the user/integrator as same sized array as input.

```
93 dudt=patches.fun(t,u,patches,varargin{:});
94 n=patches.nEdge;
95 dudt([1:n end-n+1:end],:,:,:) = 0;
96 dudt=reshape(dudt,sizeu);
```

Fin.

References

Bunder, J. E., I. G. Kevrekidis, and A. J. Roberts (July 2021). "Equation-free patch scheme for efficient computational homogenisation via self-adjoint coupling". In: *Numerische Mathematik* 149.2, pp. 229–272. DOI: 10.1007/s00211-021-01232-5 (cit. on p. 19).

Bunder, J. E., A. J. Roberts, and I. G. Kevrekidis (2017). "Good coupling for the multiscale patch scheme on systems with microscale heterogeneity". In: *J. Computational Physics* 337, pp. 154–174. DOI: 10.1016/j.jcp.2017.02.004 (cit. on pp. 2, 5, 9, 19, 23).

Hu, Zhenfang et al. (Apr. 2016). “Sparse Principal Component Analysis via Rotation and Truncation”. In: *IEEE Transactions on Neural Networks and Learning Systems* 27.4, pp. 875–890. ISSN: 2162-237X. DOI: 10.1109/TNNLS.2015.2427451 (cit. on p. 14).

Roberts, A. J. (2003). “A holistic finite difference approach models linear dynamics consistently”. In: *Mathematics of Computation* 72, pp. 247–262. DOI: 10.1090/S0025-5718-02-01448-5. (Cit. on p. 19).

Roberts, A. J. and I. G. Kevrekidis (2007). “General tooth boundary conditions for equation free modelling”. In: *SIAM J. Scientific Computing* 29.4, pp. 1495–1510. DOI: 10.1137/060654554 (cit. on pp. 19, 23).

Wikipedia (2022). *Divided differences*. https://en.wikipedia.org/wiki/Divided_differences (visited on 12/28/2022) (cit. on p. 26).