Equation-free computational homogenisation with thicker edge coupling

A. J. Roberts*

April 17, 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, but we use the patch scheme to provide a computational homogenisation. 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 $\delta u_i := u_{i+1/2} - u_{i-1/2}$.

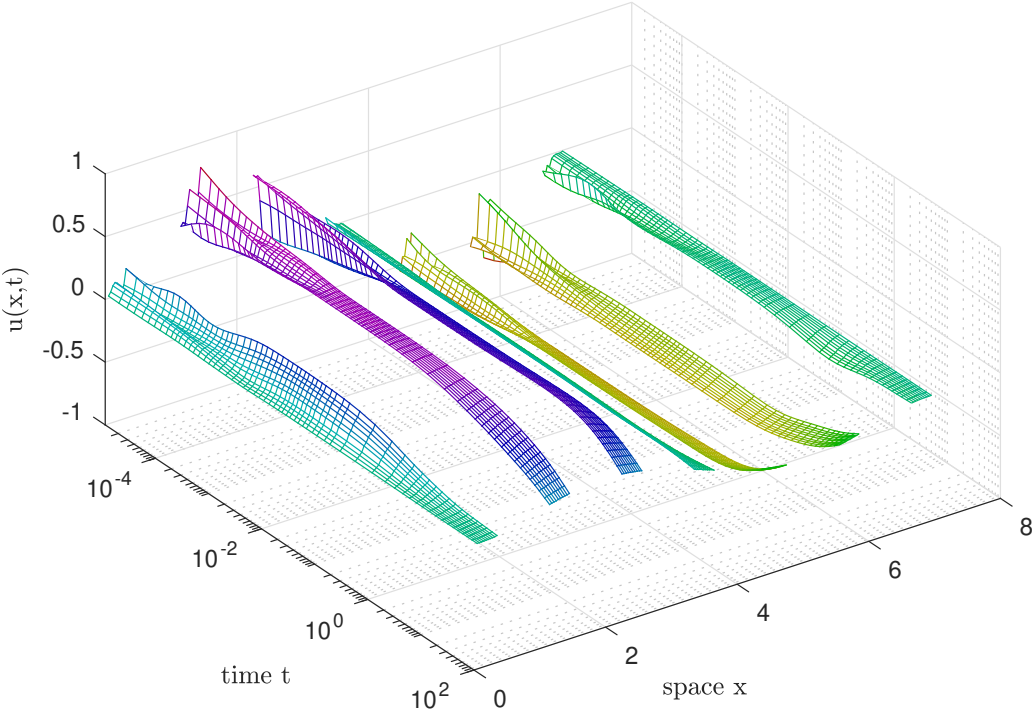
Set the desired microscale periodicity, and correspondingly choose random microscale diffusion coefficients (with some subscripts shifted by a half).

```
clear all
basename = mfilename
%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) )
```

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



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 8) 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: patterns of the Swift–Hohenb 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

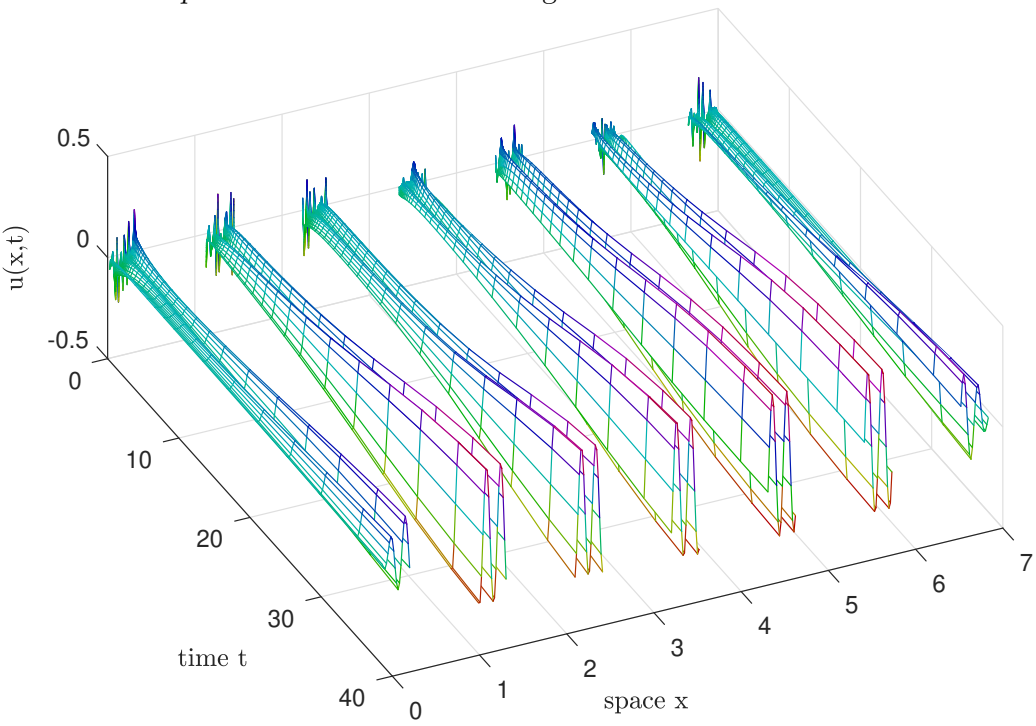
$$\partial_t u = -(1+\partial_x^2/k_0^2)^2 u + \mathrm{Ra}\, u - u^3, \tag{2}$$

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 when wavenumber parameter $k_0 = 38$. 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 of the emergent pattern.

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

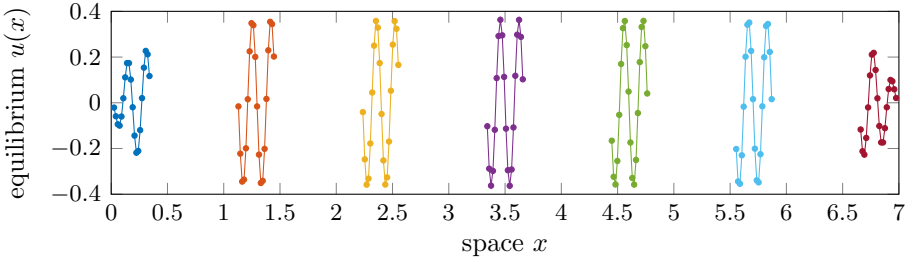
Figure 2: the pattern forming field $u(x,t)$ in the patch (gap-tooth) scheme applied to a microscale discretisation of the Swift–Hohenberg PDE *(Section 2). Physically we see the rapid decay of much microstructure, but also the meso-time growth of sub-patch-scale patterns, wavenumber k_0, that are modulated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



Establish global data struct `patches` for the Swift–Hohenberg PDE on some length domain. Use seven patches. Quartic (fourth-order) interpolation `ordCC` $= 4$ provides values for the inter-patch coupling conditions.

```
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 ...
```

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,ordCC,dx,nSubP,'EdgyInt',true,'nEdge',2);
xs=squeeze(patches.x);
```

2.0.1 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
```

2.0.2 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 8) 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.

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

Here code straightforward centred discretisation in space.

```
206   ut=nan+u;          % preallocate output array
207   v = u(2:end-1,:)+diff(u,2)/dx^2/k0^2;
208   ut(i,:) = -( v(2:end-1,:)+diff(v,2)/dx^2/k0^2 ) ...
209     +Ra*u(i,:) -u(i,:).^3;
210 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.

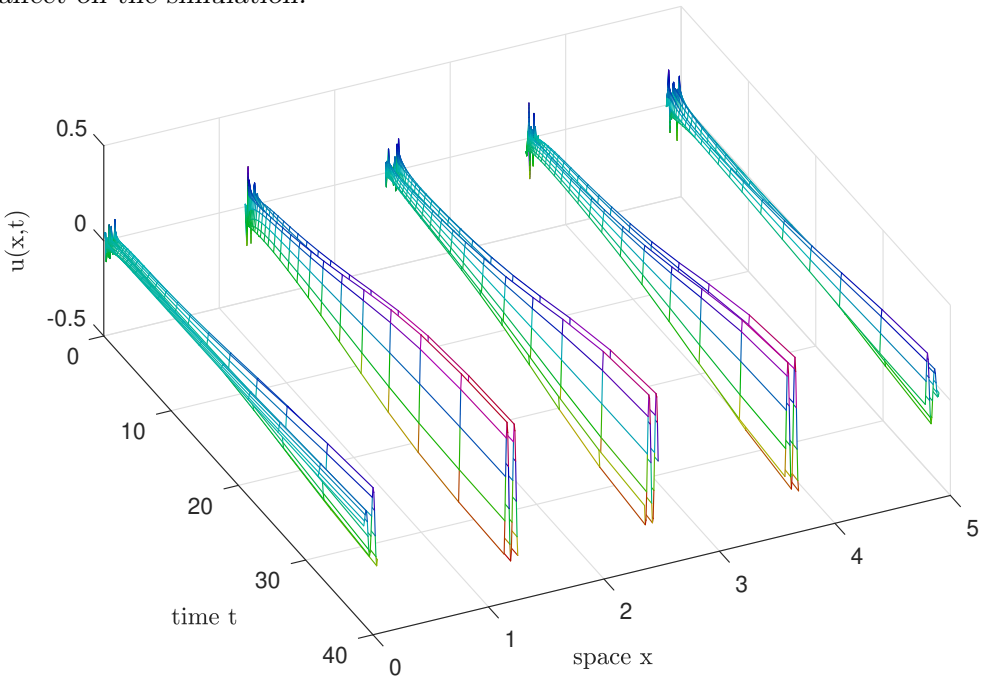
```
225 function f=theRes(u,patches,k0,Ra)
226   v=nan(size(patches.x));
227   v(patches.i) = u;
228   f = patchSys1(0,v(:),patches,k0,Ra);
229   f = f(patches.i);
230 end%function theRes
```

3 `SwiftHohenbergHetero`: patterns of a heterogeneous 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 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

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.*



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 matches 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` $= 0$.

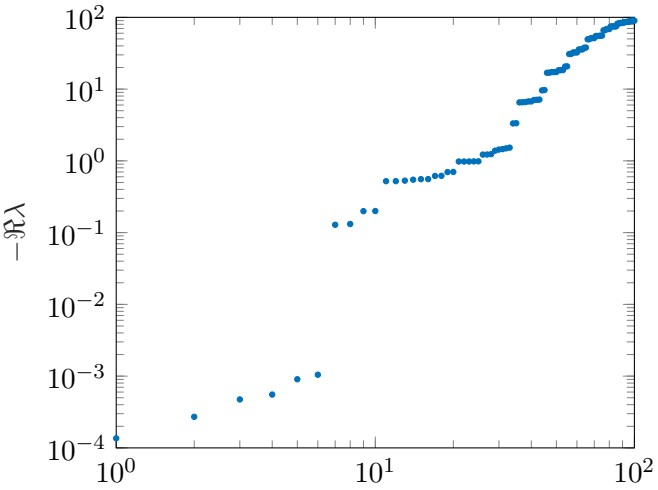
```
nPatch = 5
nSubP = 2*nPtsPeriod+4 % +2 for not-edgyInt
%nSubP = 2*nGap*nPtsPeriod+4 % approx full-domain
Len = nPatch;
ordCC = 0;
dom.type='equispace';
dom.bcOffset=0.5
patches = configPatches1(@heteroSwiftHohenbergPDE,[0 Len],dom ...
    ,nPatch,ordCC,dx,nSubP,'EdgyInt',true,'nEdge',2 ...
    ,'hetCoeffs',cl);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

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.



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)
```

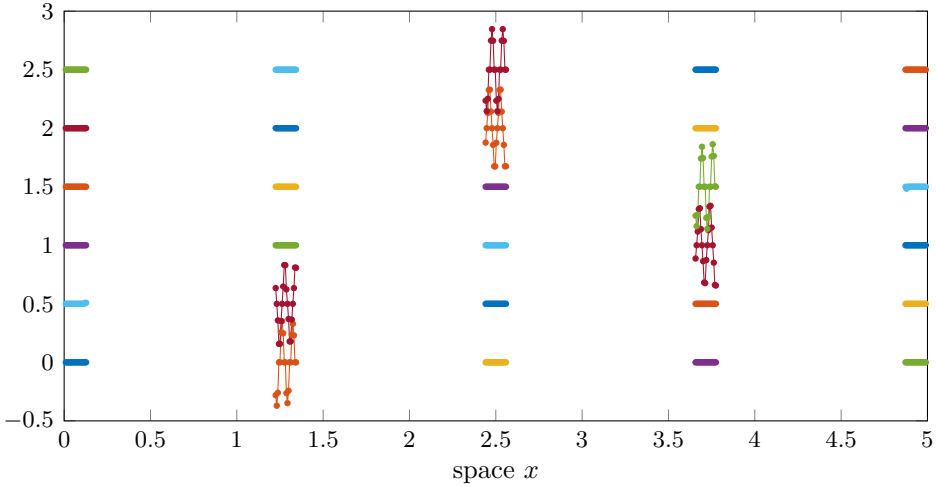
Second, make eigenvectors all real, sparsely approximate cluster modes via an algorithm developed from Hu et al. (2016), and plot. Figure 6 shows that each pair of basis vectors are phase-shifted by 90°.

```
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(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
```

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.




```
  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.

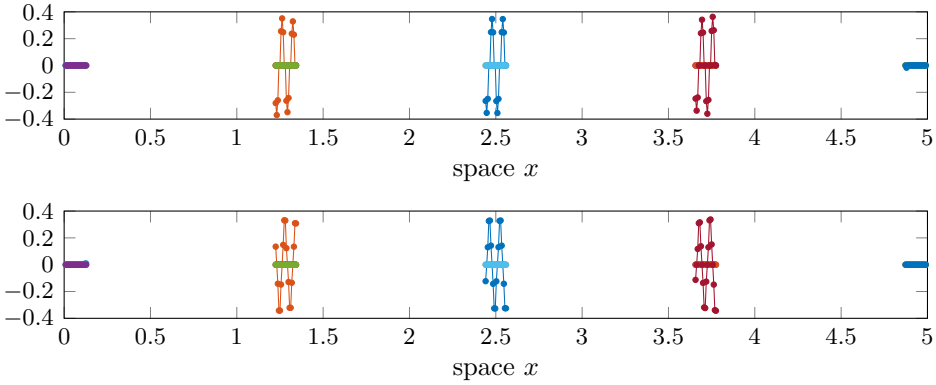
```
fprintf('\n**** Find equilibrium with fsolve\n')
u = 0.4*sin(2*pi/waveLength*patches.x);
```

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).

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.




```
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.

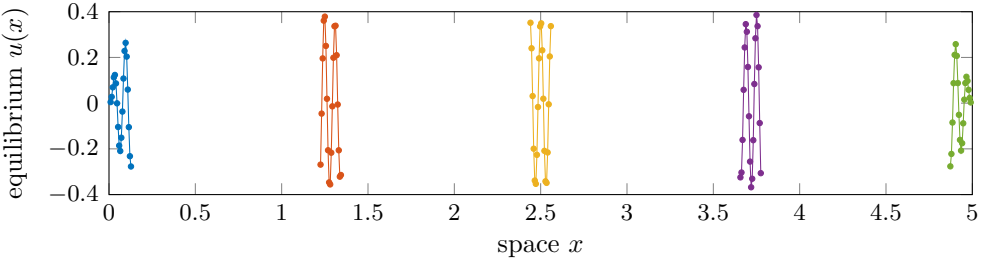
```
figure(1),clf
subplot(2,1,1)
plot(xs,squeeze(u),'.-')
xlabel('space $x$'),ylabel('equilibrium $u(x)$')
ifOurCf2tex([basename 'Equilib'])%optionally save
```

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 8) 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);
```

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




```
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 `SwiftHohenberg2dPattern`: patterns of the Swift–Hohenberg PDE in 2D on patches

Figures 9 to 14 show an example simulation in time generated by the patch scheme applied to the patterns arising from the 2D Swift–Hohenberg PDE.

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

$$\partial_t u = -(1 + \nabla^2/k_0^2)^2 u + \mathrm{Ra}\, u - u^3, \tag{4}$$

with various boundary conditions at $x, y = 0, L$. For Ra just above critical, say $\mathrm{Ra} = 0.1$, the system rapidly evolves to spatial quasi-periodic solutions with period ≈ 0.24 when wavenumber parameter $k_0 = 26$. These spatial oscillations are here resolved on a micro-grid of spacing 0.042. 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
cMap=jet(64); cMap=0.8*cMap(7:end-7,:); % set colormap
basename = ['r' num2str(floor(1e5*rem(now,1))) mfilename]
%global OurCf2eps, OurCf2eps=true %optional to save plots
Ra = 0.2 % Ra>0 leads to patterns
nGapFac = 2
waveLength = 0.5/nGapFac
nPtsPeriod = 6
dx = waveLength/nPtsPeriod
k0 = 2.1*pi/waveLength
```

The above factor 2.1 is close to $3/\sqrt{2} = 2.1213$ for which $(\pm 1, \pm 2)$ modes have same linear growth-rate as $(\pm 2, 0)$ modes.

Establish global data struct `patches` for the Swift–Hohenberg PDE on some square domain. For simplicity, use five patches in each direction. Quartic (fourth-order) interpolation `ordCC` $= 4$ provides values for the inter-patch coupling conditions. Set `bcOffset` for different boundary conditions around the square domain.

```
nPatch = 5
nSubP = 2*nPtsPeriod+4
Len = nPatch;
ordCC = 4;
dom.type='equispace';
dom.bcOffset=[0.5 0.5;1.0 1.5]
patches = configPatches2(@SwiftHohenbergPDE,[0 Len],dom ...
```

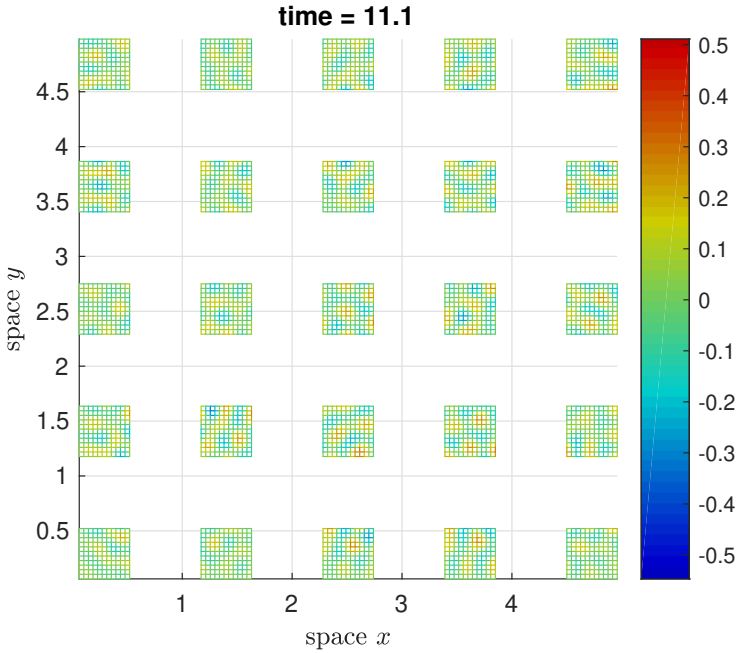


Figure 9: pattern field $u(x, y, t)$ in the patch scheme applied to a microscale discretisation of the 2D Swift–Hohenberg PDE. At this early time much of the random sub-patch microstrucre has decayed leaving some random marginal modes starting to grow.

```
    ,nPatch,ordCC,dx,nSubP,'EdgyInt',true,'nEdge',2);
xs=squeeze(patches.x);
ys=squeeze(patches.y);
```

4.0.1 Simulate in time

Set an initial condition, and here integrate forward in time using a standard method for stiff systems. Integrate the interface `patchSys2` (Section 11) to the microscale differential equations (despite the extreme stiffness, `ode23` is ten times quicker than `ode15s`). Because pattern evolution is eventually phase-diffusion, here sample the pattern at quadratically varying times.

```
fprintf('\n**** Simulate in time\n')
u0 = 0.3*( -1+2*rand(size(patches.x+patches.y)) );
Ts=400*linspace(0,1,97).^2;
tic
[ts,us] = ode23(@patchSys2, Ts, u0(:),[],patches,k0,Ra);
simulateTime = toc
us = reshape(us',nSubP,nSubP,nPatch,nPatch,[]);
```

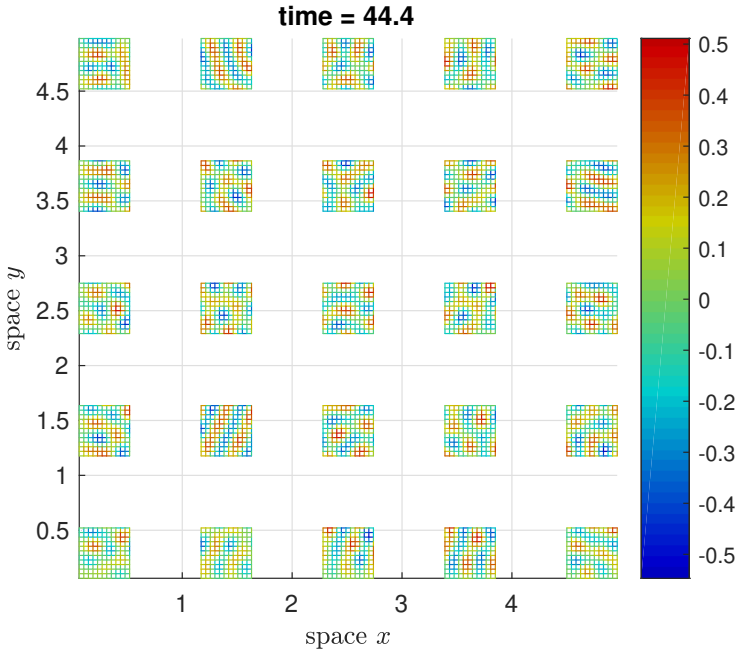


Figure 10: pattern field $u(x, y, t)$ in the patch scheme applied to a microscale discretisation of the 2D Swift–Hohenberg PDE. By now the local subpatch patterns have reached a quasi-equilibrium amplitude.

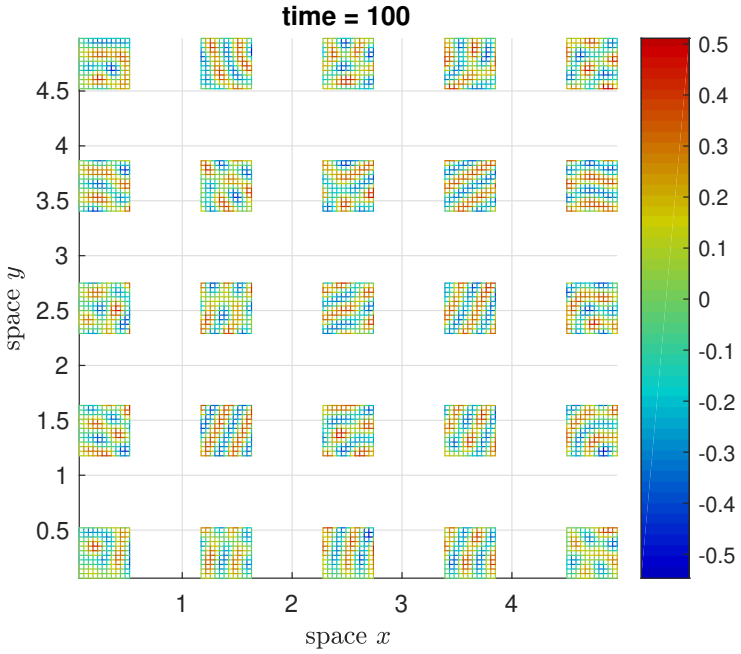


Figure 11: pattern field $u(x, y, t)$ in the patch scheme applied to a microscale discretisation of the 2D Swift–Hohenberg PDE. Patterns within the patches are evolving to the preferred rolls, but with weak coupling to other patches.

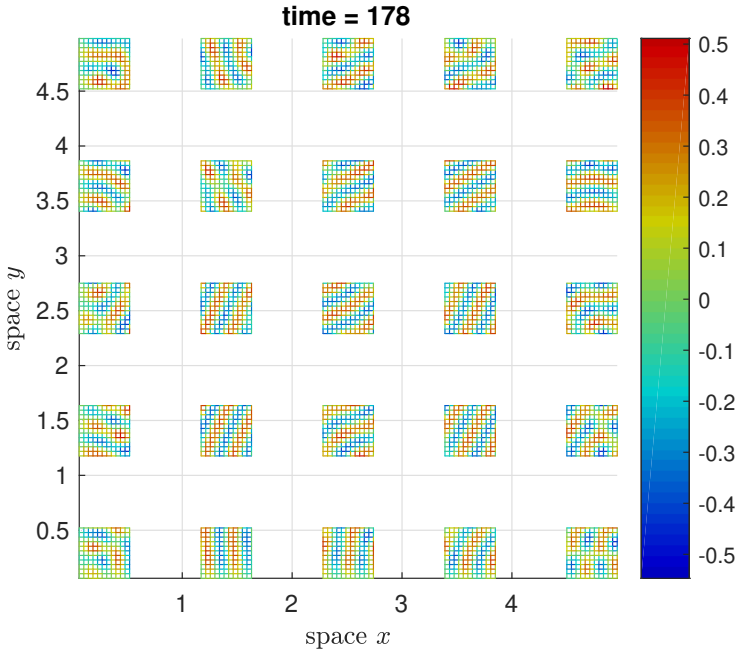


Figure 12: pattern field $u(x, y, t)$ in the patch scheme applied to a microscale discretisation of the 2D Swift–Hohenberg PDE. Can see different effects arising at different types of boundaries.

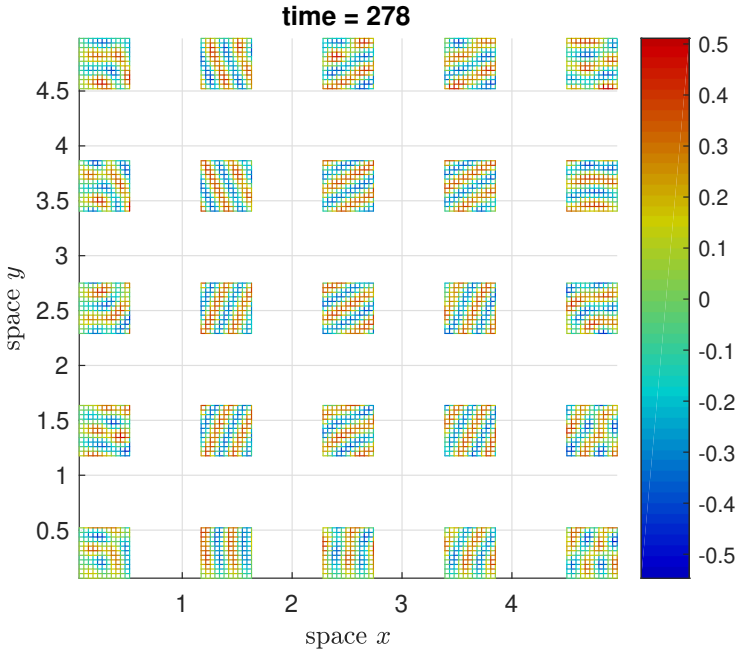


Figure 13: pattern field $u(x, y, t)$ in the patch scheme applied to a microscale discretisation of the 2D Swift–Hohenberg PDE. ...

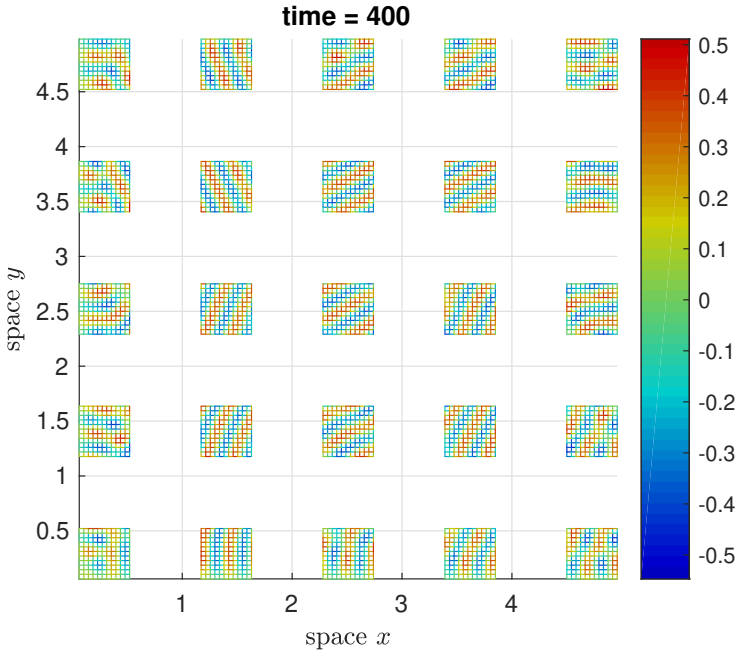


Figure 14: pattern field $u(x, y, t)$ in the patch scheme applied to a microscale discretisation of the 2D Swift–Hohenberg PDE. ...

Plot the simulation such as that shown in Figures 9 to 14 First, reshape the data, omitting edge values.

```
xs([1:2 end-1:end],:) = nan;
ys([1:2 end-1:end],:) = nan;
us = reshape( permute(us,[1 3 2 4 5]) ...
             ,nSubP*nPatch,nSubP*nPatch,[]);
uRange=[min(us(:)) max(us(:))];
```

Second, plot six examples of the evolving pattern, equi-spaced in time-index.

```
plots = round( 1+linspace(0,1,7)*(numel(ts)-1) )
for p=2:numel(plots)
  figure(p),clf
  mesh(xs(:),ys(:),us(:,:,plots(p))')
  axis equal, view(0,90)
  caxis(uRange), colormap(cMap), colorbar
  xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y,t)$')
  title(['time = ' num2str(ts(plots(p)),3)])
  ifOurCf2eps([basename num2str(p)],[12 11])
end%for p
```

Third, plot animation in time: starts after a key press.

```
%%
figure(1),clf
cf=mesh(xs(:),ys(:),us(:,:,1)');
axis equal, view(0,90)
caxis(uRange), colormap(cMap), colorbar
xlabel('space x'), ylabel('space y'), zlabel('$u(x,y,t)$')
title(['time = ' num2str(ts(1),3)])
ca=gca;
disp('Press any key to start animation'),pause
for p=2:numel(ts)
  cf.ZData=us(:,:,p)';
  cf.CData=us(:,:,p)';
  ca.Title.String=['time = ' num2str(ts(p),3)];
  pause(0.1)
end
```

Fin.

4.1 The Swift–Hohenberg PDE and BCs inside patches

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

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

Code various boundary conditions. For slightly simpler coding, squeeze out the two singleton dimensions.

```
  u = squeeze(u);
  u(1:2,:,1,:)=0; % u=u_x=0 at x=0
  u(:,1:2,:,1)=0; % u=u_y=0 at y=0
  u(end-1,:,end,:)=0;                  % u=0   at x=L
  u(end  ,:,end,:)=-u(end-2,:,end,:); % u_x=0 at x=L
  u(:,end-1,:,end)=-u(:,end-2,:,end); % u_y=0   at y=L
  u(:,end  ,:,end)=-u(:,end-3,:,end); % u_yyy=0 at y=L
```

Here code straightforward centred discretisation in space.

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

5 heteroDispersiveWave3: heterogeneous Dispersive Waves from 4th order PDE

This uses small spatial patches to simulate heterogeneous dispersive waves in 3D. The wave equation for $u(x, y, z, t)$ is the fourth-order in space PDE

$$u_{tt} = -\nabla^2(C\nabla^2 u)$$

for microscale variations in scalar $C(x, y, z)$.

Initialise some Matlab aspects.

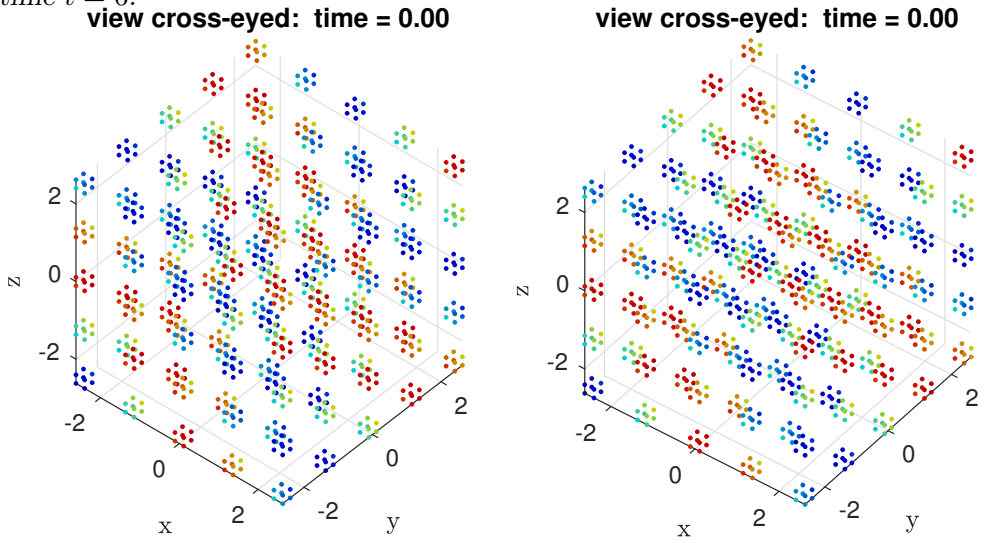
```
clear all
cMap=jet(64); cMap=0.8*cMap(7:end-7,:); % set colormap
basename = [num2str(floor(1e5*rem(now,1))) mfilename]
%global OurCf2eps, OurCf2eps=true %optional to save plots
```

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

```
mPeriod = [2 2 2];
cHetr = exp(0.9*randn(mPeriod));
cHetr = cHetr*mean(1./cHetr(:))
```

Establish global patch data struct to interface with a function coding a fourth-order heterogeneous wave PDE: to be solved on $[-\pi, \pi]^3$-periodic domain, with 5^3 patches, spectral interpolation (0) couples the patches, each patch with micro-grid spacing 0.22 (relatively large for visualisation), and with 6^3 points forming each patch. (Six because two edge layers on each of two faces, and two interior points for the PDE.)

Figure 15: initial field $u(x, y, z, t)$ at time $t = 0$ of the patch scheme applied to a heterogeneous dispersive wave PDE*: Figure 16 plots the computed field at time $t = 6$.*




```
global patches
patches = configPatches3(@heteroDispWave3,[-pi pi] ...
    ,'periodic', 5, 0, 0.22, mPeriod+4 ,'EdgyInt',true ...
    ,'hetCoeffs',cHetr ,'nEdge',2);
```

Set a wave initial state using auto-replication of the spatial grid, and as Figure 15 shows. This wave propagates diagonally across space. Concatenate the two u, v-fields to be the two components of the fourth dimension.

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

Integrate in time to $t = 6$ using standard functions. In Matlab `ode15s` would be natural as the patch scheme is naturally stiff, but `ode23` is much quicker (Maclean, Bunder, and Roberts 2021, Fig. 4).

```
disp('Simulate heterogeneous wave u_tt=delsq[C*delsq(u)]')
tic
[ts,us] = ode23(@patchSys3,linspace(0,6),uv0(:));
simulateTime=toc
```

Animate the computed simulation to end with Figure 16. Use `patchEdgeInt3` to obtain patch-face values in order to most easily reconstruct the array data structure.

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

```
%%
figure(1), clf, colormap(cMap)
xs = patches.x+0*patches.y+0*patches.z;
ys = patches.y+0*patches.x+0*patches.z;
zs = patches.z+0*patches.y+0*patches.x;
if 1, xs([1:2 end-1:end],:,:,:)=nan;
      xs(:,[1:2 end-1:end],:,:)=nan;
      xs(:,:,[1:2 end-1:end],:)=nan;
end;%option
j=find(~isnan(xs));
```

In the scatter plot, `col()` maps the u-data values to the colour of the dots.

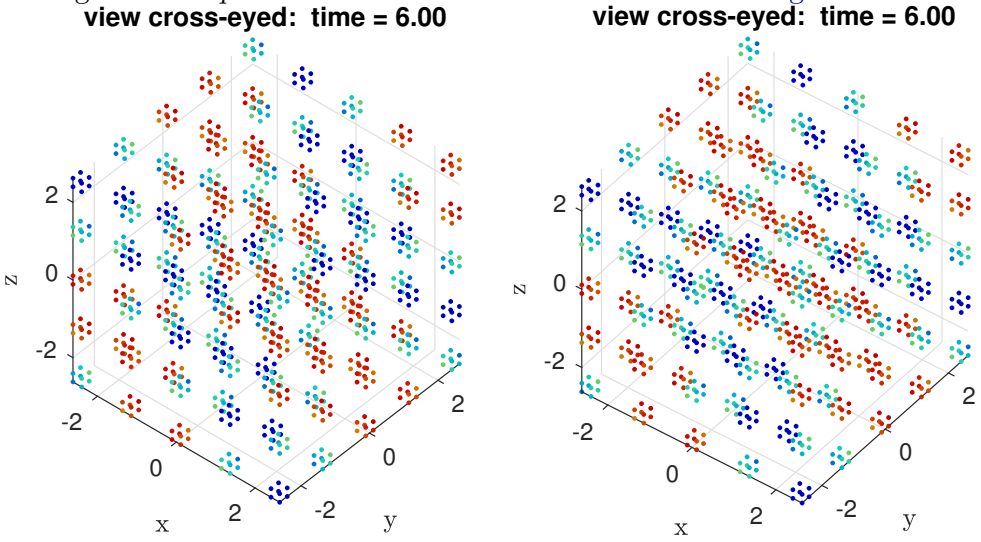
```
col = @(u) sign(u).*abs(u);
```

Loop to plot at each and every time step.

```
for i = 1:length(ts)
  uv = patchEdgeInt3(us(i,:));
  u = uv(:,:,:,1,:);
  for p=1:2
    subplot(1,2,p)
    if (i==1)
      scat(p) = scatter3(xs(j),ys(j),zs(j),'.');
      axis equal, caxis(col([0 1])), view(45-4*p,42)
      xlabel('x'), ylabel('y'), zlabel('z')
    end
    title(['view cross-eyed:  time = ' num2str(ts(i),'%4.2f')])
    set( scat(p),'CData',col(u(j)) );
  end
```

Optionally save the initial condition to graphic file for Figure 15, and optionally save the last plot.

Figure 16: field $u(x, y, z, t)$ at time $t = 6$ of the patch scheme applied to the heterogeneous dispersive wave PDE *with initial condition in Figure 15.*




```
    if i==1,
      ifOurCf2eps([basename 'ic'])
      disp('Type space character to animate simulation')
      pause
    else pause(0.1)
    end
  end% i-loop over all times
  ifOurCf2eps([basename 'fin'])
```

5.1 `heteroDispWave3()`: PDE function of 4th-order heterogeneous dispersive waves

This function codes the lattice heterogeneous waves inside the patches. The wave PDE for $u(x, y, z, t)$ and 'velocity' $v(x, y, z, t)$ is

$$u_t = v, \quad v_t = -\nabla^2(C\nabla^2 u)$$

for microscale variations in scalar $C(x, y, z)$. For 8D input arrays `u`, `x`, `y`, and `z` (via edge-value interpolation of `patchSys3`, Section 14), computes the time derivative at each point in the interior of a patch, output in `ut`. The 3D array of

heterogeneous coefficients, C_{ijk}, c^y_{ijk} and c^z_{ijk}, have been stored in `patches.cs` (3D).

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

```
function ut = heteroDispWave3(t,u,patches)
  if nargin<3, global patches, end
```

Micro-grid space steps.

```
dx = diff(patches.x(2:3));
dy = diff(patches.y(2:3));
dz = diff(patches.z(2:3));
```

First, compute $C\nabla^2 u$ into say `u`, using indices for all but extreme micro-grid points. We use a single colon to represent the last four array dimensions because the result arrays are already dimensioned.

```
I = 2:size(u,1)-1; J = 2:size(u,2)-1;  K = 2:size(u,3)-1;
u(I,J,K,1,:) = patches.cs(I,J,K,1,:).*( diff(u(:,J,K,1,:),2,1)/dx^2 .
    +diff(u(I,:,K,1,:),2,2)/dy^2 +diff(u(I,J,:,1,:),2,3)/dz^2 );
```

Reserve storage, set lowercase indices to non-edge interior, and then assign interior patch values to the heterogeneous diffusion time derivatives.

```
ut = nan+u;  % preallocate output array
i = I(2:end-1); j = J(2:end-1);  k = K(2:end-1);
ut(i,j,k,1,:) = u(i,j,k,2,:);  % du/dt=v
% dv/dt=delta^2 of above C*delta^2
ut(i,j,k,2,:) = -( diff(u(I,j,k,1,:),2,1)/dx^2 ...
  +diff(u(i,J,k,1,:),2,2)/dy^2 +diff(u(i,j,K,1,:),2,3)/dz^2 );
end% function
```

New configuration and interpolation

6 `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.
  - `.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).

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

- `.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);
```

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.

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

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

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);
```

6.1 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.

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

```
szUx0=size(Ux);
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.

```
  if patches.parallel
    dmu = zeros(szUx0,patches.codist); % 5D
  else
    dmu = zeros(szUx0); % 5D
  end
```

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

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

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

```
  for k = 3:patches.ordCC
    dmu(:,:,:,:,k) =       dmu(:,:,:,Ip,k-2) ...
      -2*dmu(:,:,:,I,k-2) +dmu(:,:,:,Im,k-2);
  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' $|k| < \mathtt{Nx}/2$, so set row vector $\mathtt{ks} = k2\pi/N$ 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]

```
  if patches.stag % transform by doubling the number of fields
    v = nan(size(u)); % currently to restore the shape of u
    u = [u(:,:,:,1:2:Nx) u(:,:,:,2:2:Nx)];
```

[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.

```
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.

```
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?

```
if patches.stag
  nVars = nVars/2;
  u=reshape(u,nx,nVars,2,nEnsem,Nx);
  Nx = 2*Nx;
  v(:,:,:,1:2:Nx) = u(:,:,1,:,:);
  v(:,:,:,2:2:Nx) = u(:,:,2,:,:);
  u = v;
end%if patches.stag
end%if patches.ordCC
```

6.2 Non-periodic macroscale interpolation

```
else% patches.periodic false
assert(~patches.stag, ...
'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`.

```
if patches.ordCC<1, patches.ordCC = Nx-1; end
p = min(patches.ordCC,Nx-1);
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.

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

Initialise to highest divide difference, surrounded by zeros.

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

Complete Horner evaluation of the relevant polynomials.

[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.

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

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

```
u(1 ,:,patches.le,I) = Uedge(1,:,:,I);
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`.

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

End of the non-periodic interpolation code.

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

Unfold multiple edges No need to restore x.

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

Fin, returning the 4D array of field values.

7 `configPatches1()`: configure spatial patches in 1D

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys1()`. Section 7.1 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 7.1 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, 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]

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

- `.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.

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

- `.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.

7.1 If no arguments, then execute an example

```
261 if nargin==0
262 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.

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

Set some initial condition, with some microscale randomness.

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

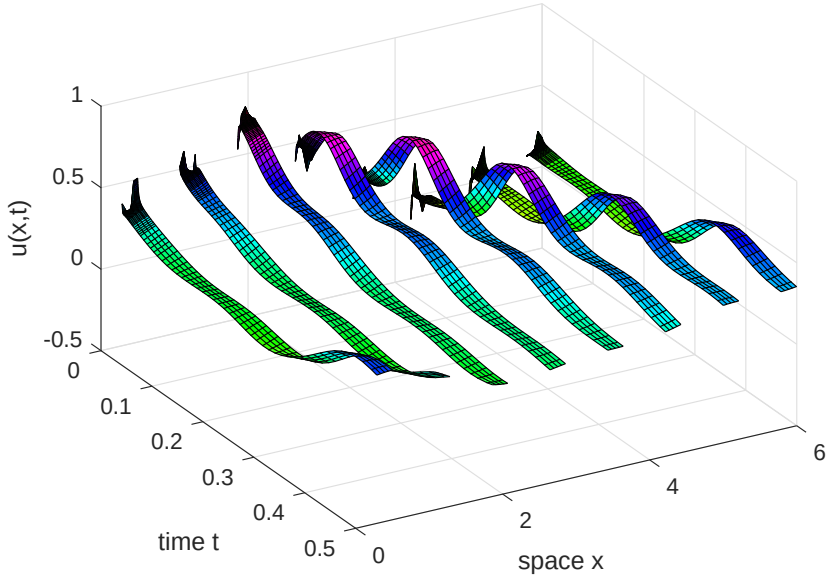
Simulate in time using a standard stiff integrator and the interface function `patchSys1()` (Section 8).

```
298 if ~exist('OCTAVE_VERSION','builtin')
299 [ts,us] = ode15s( @patchSys1,[0 0.5],u0(:));
300 else % octave version
301 [ts,us] = odeOcts(@patchSys1,[0 0.5],u0(:));
302 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 17 illustrates an example simulation in time generated by the patch scheme applied to Burgers' PDE.

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

Figure 17: 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 17, then exit this function.

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

7.2 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.

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

Default macroscale conditions are periodic with evenly spaced patches.

```
if isempty(Dom), Dom=struct('type','periodic'); end
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.

```
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.

```
patches.periodic=false;
switch Dom.type
case 'periodic'
    patches.periodic=true;
    if isfield(Dom,'bcOffset')
    warning('bcOffset not available for Dom.type = periodic'), end
    if isfield(Dom,'X')
    warning('X not available for Dom.type = periodic'), end
case {'equispace','chebyshev'}
    if ~isfield(Dom,'bcOffset'), Dom.bcOffset=[0;0]; end
    if length(Dom.bcOffset)==1
        Dom.bcOffset=repmat(Dom.bcOffset,2,1); end
    if isfield(Dom,'X')
    warning('X not available for Dom.type = equispace or chebyshev')
    end
case 'usergiven'
    if isfield(Dom,'bcOffset')
    warning('bcOffset not available for usergiven Dom.type'), end
    assert(isfield(Dom,'X'),'X required for Dom.type = usergiven')
otherwise
    error([Dom.type ' is unknown Dom.type'])
end%switch Dom.type
```

7.3 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]

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

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

```
544 case 'equispace'
545   X=linspace(Xlim(1)+((nSubP-1)/2-Dom.bcOffset(1))*dx ...
546              ,Xlim(2)-((nSubP-1)/2-Dom.bcOffset(2))*dx ,nPatch);
547   DX=diff(X(1:2));
548   width=(1+patches.EdgyInt)/2*(nSubP-1-patches.EdgyInt)*dx;
549   if DX<width*0.999999
550      warning('too many equispace patches (double overlapping)')
551      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'. [9]

```
568 case 'chebyshev'
569   halfWidth=dx*(nSubP-1)/2;
570   X1 = Xlim(1)+halfWidth-Dom.bcOffset(1)*dx;
571   X2 = Xlim(2)-halfWidth+Dom.bcOffset(2)*dx;
572 %  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] 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%for
  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);
```

7.4 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()`.

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

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

```
675   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.

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

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

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

End the two if-statements.

```
718   end%if-else nEnsem>1
719 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.*[10]

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

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

```
747   pari = 1;
748   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.

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

[10] 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.

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

Fin

```
782 end% function
```

8 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 7) 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 6 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.

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

Couples 2D patches across 2D space by computing their edge values via macroscale interpolation. Research (Roberts, MacKenzie, and Bunder 2014; Bunder et al. 2021) indicates the patch centre-values are sensible macroscale variables, and macroscale interpolation of these determine patch-edge values. However, for computational homogenisation in multi-D, interpolating patch next-to-edge values appears better (Bunder, Kevrekidis, and Roberts 2021).

This function is primarily used by `patchSys2()` but is also useful for user graphics. [11]

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

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

Input

- `u` is a vector/array of length `prod(nSubP)·nVars·nEnsem·prod(nPatch)` where there are `nVars` · `nEnsem` field values at each of the points in the `nSubP1` · `nSubP2` · `nPatch1` · `nPatch2` multiscale spatial grid on the `nPatch1` · `nPatch2` array of patches.

- `patches` a struct set by `configPatches2()` which includes the following information.

  - `.x` is `nSubP1`$\times 1\times 1\times 1\times$`nPatch1`$\times 1$ array of the spatial locations x_{iI} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index i, but may be variable spaced in macroscale index I.
  - `.y` is similarly $1\times$`nSubP2`$\times 1\times 1\times 1\times$`nPatch2` array of the spatial locations y_{jJ} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index j, but may be variable spaced in macroscale index J.
  - `.ordCC` is order of interpolation, currently only $\{0, 2, 4, \ldots\}$
  - `.periodic` indicates whether macroscale is periodic domain, or alternatively that the macroscale has left, right, top and bottom boundaries so interpolation is via divided differences.
  - `.stag` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation. Currently must be zero.
  - `.Cwtsr` and `.Cwtsl` are the coupling coefficients for finite width interpolation in both the x, y-directions—when invoking a periodic domain.

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

- `.EdgyInt`, true/false, for determining patch-edge values by interpolation: true, from opposite-edge next-to-edge values (often preserves symmetry); false, from centre cross-patch values (near original scheme).
- `.nEdge`, two elements, the width of edge values set by interpolation at the x, y-edge regions, respectively, of each patch (default is one for both x, y-edges).
- `.nEnsem` the number of realisations in the ensemble.
- `.parallel` whether serial or parallel.

Output

- `u` is 6D array, `nSubP1` · `nSubP2` · `nVars` · `nEnsem` · `nPatch1` · `nPatch2`, 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.

```
122   if max(abs(imag(u(:))))<1e-9*max(abs(u(:)))
123        uclean=@(u) real(u);
124   else uclean=@(u) u;
125   end
```

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

```
133 [~,ny,~,~,~,Ny] = size(patches.y);
134 [nx,~,~,~,Nx,~] = size(patches.x);
135 nEnsem = patches.nEnsem;
136 nVars = round(numel(u)/numel(patches.x)/numel(patches.y)/nEnsem);
137 assert(numel(u) == nx*ny*Nx*Ny*nVars*nEnsem ...
138   ,'patchEdgeInt2: input u has wrong size for parameters')
139 u = reshape(u,[nx ny nVars nEnsem Nx Ny ]);
```

For the moment assume the physical domain is either macroscale periodic or macroscale rectangle so that the coupling formulas are simplest. These index vectors point to patches and, if periodic, their four immediate neighbours.

```
149 I=1:Nx; Ip=mod(I,Nx)+1; Im=mod(I-2,Nx)+1;
150 J=1:Ny; Jp=mod(J,Ny)+1; Jm=mod(J-2,Ny)+1;
```

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

```
%x = patches.x;
%if patches.nEdge(1)>1
%  m = patches.nEdge(1);
%  x = x(1:m:nx,:,:,:,:,:);
%  nx = nx/m;
%  u = reshape(u,m,nx,ny,nVars,nEnsem,Nx,Ny);
%  nVars = nVars*m;
%  u = reshape( permute(u,[2:3 1 4:7]) ...
%              ,nx,ny,nVars,nEnsem,Nx,Ny);
%end%if patches.nEdge(1)
%y = patches.y;
%if patches.nEdge(2)>1
%  m = patches.nEdge(2);
%  y = y(:,1:m:ny,:,:,:,:);
%  ny = ny/m;
%  u = reshape(u,nx,m,ny,nVars,nEnsem,Nx,Ny);
%  nVars = nVars*m;
%  u = reshape( permute(u,[1 3 2 4:7]) ...
%              ,nx,ny,nVars,nEnsem,Nx,Ny);
%end%if patches.nEdge(2)
x = patches.x;
y = patches.y;
if mean(patches.nEdge)>1
  mx = patches.nEdge(1);
  my = patches.nEdge(2);
  x = x(1:mx:nx,:,:,:,:,:);
  y = y(:,1:my:ny,:,:,:,:);
  nx = nx/mx;
  ny = ny/my;
  u = reshape(u,mx,nx,my,ny,nVars,nEnsem,Nx,Ny);
  nVars = nVars*mx*my;
  u = reshape( permute(u,[2 4 1 3 5:8]) ...
              ,nx,ny,nVars,nEnsem,Nx,Ny);
end%if patches.nEdge
```

The centre of each patch (as `nx` and `ny` are odd for centre-patch interpola-

tion) is at indices

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

9.1 Periodic macroscale interpolation schemes

```
if patches.periodic
```

Get the size ratios of the patches.

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

9.1.1 Lagrange interpolation gives patch-edge values

Compute centred differences of the mid-patch values for the macro-interpolation, of all fields. Here the domain is macro-periodic.

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

Interpolate the three directions in succession, in this way we naturally fill-in corner values. Start with x-direction, and give most documentation for that case as the y-direction is essentially the same.

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

```
  if patches.EdgyInt % interpolate next-to-face values
    U = u([2 nx-1],2:(ny-1),:,:,I,J);
  else % interpolate centre-cross values
    U = u(i0,2:(ny-1),:,:,I,J);
  end;%if patches.EdgyInt
```

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.

```
szUO=size(U); szUO=[szUO ones(1,6-length(szUO)) ordCC];
```

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

```
269   if ~patches.parallel, dmu = zeros(szUO); % 7D
270   else   dmu = zeros(szUO,patches.codist); % 7D
271   end%if patches.parallel
```

First compute differences $\mu\delta$ and δ^2.

```
277   if patches.stag % use only odd numbered neighbours
278     error('polynomial interpolation not yet for staggered patch coupl
279 %    dmux(:,:,:,:,I,:,1) = (Ux(:,:,:,:,Ip,:)+Ux(:,:,:,:,Im,:))/2; % \
280 %    dmux(:,:,:,:,I,:,2) = (Ux(:,:,:,:,Ip,:)-Ux(:,:,:,:,Im,:)); % \de
281 %    Ip = Ip(Ip); Im = Im(Im); % increase shifts to \pm2
282 %    dmuy(:,:,:,:,:,J,1) = (Ux(:,:,:,:,:,Jp)+Ux(:,:,:,:,:,Jm))/2; % \
283 %    dmuy(:,:,:,:,:,J,2) = (Ux(:,:,:,:,:,Jp)-Ux(:,:,:,:,:,Jm)); % \de
284 %    Jp = Jp(Jp); Jm = Jm(Jm); % increase shifts to \pm2
285   else %disp('starting standard interpolation')
286     dmu(:,:,:,:,I,:,1) = (U(:,:,:,:,Ip,:) ...
287                           -U(:,:,:,:,Im,:))/2; %\mu\delta
288     dmu(:,:,:,:,I,:,2) = (U(:,:,:,:,Ip,:) ...
289        -2*U(:,:,:,:,I,:) +U(:,:,:,:,Im,:));   %\delta^2
290   end% if patches.stag
```

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

```
297   for k = 3:ordCC
298     dmu(:,:,:,:,I,:,k) =     dmu(:,:,:,:,Ip,:,k-2) ...
299     -2*dmu(:,:,:,:,I,:,k-2) +dmu(:,:,:,:,Im,:,k-2);
300   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 `configPatches2()`. Here interpolate to specified order.

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

```
k=1+patches.EdgyInt; % use centre or two edges
u(nx,2:(ny-1),:,patches.ri,I,:) ...
  = U(1,:,:,:,:,:)*(1-patches.stag) ...
  +sum( shiftdim(patches.Cwtsr(:,1),-6).*dmu(1,:,:,:,:,:,:) ,7);
u(1 ,2:(ny-1),:,patches.le,I,:,:) ...
  = U(k,:,:,:,:,:)*(1-patches.stag) ...
  +sum( shiftdim(patches.Cwtsl(:,1),-6).*dmu(k,:,:,:,:,:,:) ,7);
```

y-normal edge values Interpolate from either the next-to-edge values, or the centre-cross-line values.

```
  if patches.EdgyInt % interpolate next-to-face values
    U = u(:,[2 ny-1],:,:,I,J);
  else % interpolate centre-cross values
    U = u(:,j0,:,:,I,J);
  end;%if patches.EdgyInt
```

Adjoin extra dimension for the array of differences.

```
szUO=size(U); szUO=[szUO ones(1,6-length(szUO)) ordCC];
```

Store finite differences $(\mu\delta, \delta^2, \mu\delta^3, \delta^4, \ldots)$ in this array.

```
  if ~patches.parallel, dmu = zeros(szUO); % 7D
  else   dmu = zeros(szUO,patches.codist); % 7D
  end%if patches.parallel
```

First compute differences $\mu\delta$ and δ^2.

```
  if patches.stag % use only odd numbered neighbours
    error('polynomial interpolation not yet for staggered patch coupl
  else  %disp('starting standard interpolation')
    dmu(:,:,:,:,:,J,1) = (U(:,:,:,:,:,Jp) ...
                          -U(:,:,:,:,:,Jm))/2; %\mu\delta
    dmu(:,:,:,:,:,J,2) = (U(:,:,:,:,:,Jp) ...
       -2*U(:,:,:,:,:,J) +U(:,:,:,:,:,Jm));   %\delta^2
  end% if stag
```

Recursively take δ^2.

```
  for k = 3:ordCC
    dmu(:,:,:,:,:,J,k) =     dmu(:,:,:,:,:,Jp,k-2) ...
    -2*dmu(:,:,:,:,:,J,k-2) +dmu(:,:,:,:,:,Jm,k-2);
  end
```

Interpolate macro-values using the weights pre-computed by `configPatches2()`. An ensemble of configurations may have cross-coupling.

```
k = 1+patches.EdgyInt; % use centre or two edges
u(:,ny,:,patches.to,:,J) ...
  = U(:,1,:,:,:,:)*(1-patches.stag) ...
  +sum( shiftdim(patches.Cwtsr(:,2),-6).*dmu(:,1,:,:,:,:,:) ,7);
u(:,1 ,:,patches.bo,:,J) ...
  = U(:,k,:,:,:,:)*(1-patches.stag) ...
  +sum( shiftdim(patches.Cwtsl(:,2),-6).*dmu(:,k,:,:,:,:,:) ,7);
```

9.1.2 Case of spectral interpolation

Assumes the domain is macro-periodic.

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

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

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

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

Interpolate the two directions in succession, in this way we naturally fill-in edge-corner values. Start with x-direction, and give most documentation for that case as the other is essentially the same. Need these indices of patch interior.

```
ix = 2:nx-1;   iy = 2:ny-1;
```

x-normal edge values Now set wavenumbers into a vector at the correct dimension. In the case of even N these compute the +-case for the highest wavenumber zig-zag mode, $k = (0\,, 1\,, \ldots, k_{\max}\,, +(k_{\max}+1) - k_{\max}\,, \ldots, -1)$.

```
  kMax = floor((Nx-1)/2);
  kr = shiftdim( rx*2*pi/Nx*(mod((0:Nx-1)+kMax,Nx)-kMax) ,-3);
```

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

```
if ~patches.EdgyInt
     Cm = fft( u(i0,iy,:,:,:,:) ,[],5);
     Cp = Cm;
else
     Cm = fft( u(   2,iy ,:,patches.le,:,:) ,[],5);
     Cp = fft( u(nx-1,iy ,:,patches.ri,:,:) ,[],5);
end%if ~patches.EdgyInt
```

Now invert the Fourier transforms to complete interpolation. Enforce reality when appropriate.

```
u(nx,iy,:,:,:,:) = uclean( ifft( ...
    Cm.*exp(1i*(stagShift+kr))  ,[],5) );
u( 1,iy,:,:,:,:) = uclean( ifft( ...
    Cp.*exp(1i*(stagShift-kr))  ,[],5) );
```

y-normal edge values Set wavenumbers into a vector.

```
  kMax = floor((Ny-1)/2);
  kr = shiftdim( ry*2*pi/Ny*(mod((0:Ny-1)+kMax,Ny)-kMax) ,-4);
```

Compute the Fourier transform of the patch values on the centre-lines for all the fields.

```
if ~patches.EdgyInt
     Cm = fft( u(:,j0,:,:,:,:) ,[],6);
     Cp = Cm;
else
     Cm = fft( u(:,2    ,:,patches.bo,:,:) ,[],6);
     Cp = fft( u(:,ny-1 ,:,patches.to,:,:) ,[],6);
end%if ~patches.EdgyInt
```

Invert the Fourier transforms to complete interpolation.

```
u(:,ny,:,:,:,:) = uclean( ifft( ...
    Cm.*exp(1i*(stagShift+kr))  ,[],6) );
u(:, 1,:,:,:,:) = uclean( ifft( ...
    Cp.*exp(1i*(stagShift-kr))  ,[],6) );

end% if ordCC>0 else, so spectral
```

9.2 Non-periodic macroscale interpolation

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

Determine the order of interpolation `px` and `py` (potentially different in the different directions!), and hence size of the (forward) divided difference tables in `F` (7D) for interpolating to left/right, and top/bottom edges. Because of the product-form of the patch grid, and because we are doing *only* either edgy interpolation or cross-patch interpolation (*not* just the centre patch value), the interpolations are all 1D interpolations.

```
if patches.ordCC<1
     px = Nx-1; py = Ny-1;
else px = min(patches.ordCC,Nx-1);
     py = min(patches.ordCC,Ny-1);
end
ix=2:nx-1;  iy=2:ny-1; % indices of edge 'interior' (ix n/a)
```

9.2.1 x-direction values

Set function values in first 'column' of the tables 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. [12]

```
  F = nan(patches.EdgyInt+1,ny-2,nVars,nEnsem,Nx,Ny,px+1);
  if patches.EdgyInt % interpolate next-to-edge values
    F(:,:,:,:,:,:,1) = u([nx-1 2],iy,:,:,:,:);
    X = x([nx-1 2],:,:,:,:,:);
  else % interpolate mid-patch cross-patch values
    F(:,:,:,:,:,:,1) = u(i0,iy,:,:,:,:);
    X = x(i0,:,:,:,:,:);
  end%if patches.EdgyInt
```

Form tables of divided differences Compute tables of (forward) divided differences (e.g., Wikipedia 2022) for every variable, and across ensemble, and for left/right edges. Recursively find all divided differences.

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

Interpolate with divided differences Now interpolate to find the edge-values on left/right edges at `Xedge` for every interior `Y`.

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

Code Horner's recursive evaluation of the interpolation polynomials. Indices `i` are those of the left edge of each interpolation stencil, because the table is of forward differences. This alternative: the case of order p_x and p_y interpolation across the domain, asymmetric near the boundaries of the rectangular domain.

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

[12] **ToDo:** Have no plans to implement core averaging as yet.

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

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

9.2.2 y-direction values

Set function values in first 'column' of the tables for every variable and across ensemble.

```
  F = nan(nx,patches.EdgyInt+1,nVars,nEnsem,Nx,Ny,py+1);
  if patches.EdgyInt % interpolate next-to-edge values
    F(:,:,:,:,:,:,1) = u(:,[ny-1 2],:,:,:,:);
    Y = y(:,[ny-1 2],:,:,:,:);
  else % interpolate mid-patch cross-patch values
    F(:,:,:,:,:,:,1) = u(:,j0,:,:,:,:);
    Y = y(:,j0,:,:,:,:);
  end;
```

Form tables of divided differences.

```
for q = 1:py
  j = 1:Ny-q;
  F(:,:,:,:,:,j,q+1) ...
  = (F(:,:,:,:,:,j+1 ,q)-F(:,:,:,:,:,j,q)) ...
  ./(Y(:,:,:,:,:,j+q)   -Y(:,:,:,:,:,j));
end
```

Interpolate to find the edge-values on top/bottom edges `Yedge` for every x.

```
Yedge = y(:,[1 ny],:,:,:,:);
```

Code Horner's recursive evaluation of the interpolation polynomials. Indices `j` are those of the bottom edge of each interpolation stencil, because the table is of forward differences.

```
  j = max(1,min(1:Ny,Ny-ceil(py/2))-floor(py/2));
  Uedge = F(:,:,:,:,:,j,py+1);
  for q = py:-1:1
    Uedge = F(:,:,:,:,:,j,q)+(Yedge-Y(:,:,:,:,:,j+q-1)).*Uedge;
  end
```

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

```
679 u(:,1 ,:,patches.bo,:,:) = Uedge(:,1,:,:,:,:);
680 u(:,ny,:,patches.to,:,:) = Uedge(:,2,:,:,:,:);
```

9.2.3 Optional NaNs for safety

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

```
692 if isfield(patches,'intTest')&&patches.intTest
693 else % usual case
694     u( 1,:,:,:, 1,:) = nan;
695     u(nx,:,:,:,Nx,:) = nan;
696     u(:, 1,:,:,:, 1) = nan;
697     u(:,ny,:,:,:,Ny) = nan;
698 end%if
```

End of the non-periodic interpolation code.

```
705 end%if patches.periodic else
```

Unfold multiple edges No need to restore x, y.

```
712 if mean(patches.nEdge)>1
713   nVars = nVars/(mx*my);
714   u = reshape( u ,nx,ny,mx,my,nVars,nEnsem,Nx,Ny);
715   nx = nx*mx;
716   ny = ny*my;
717   u = reshape( permute(u,[3 1 4 2 5:8]) ...
718               ,nx,ny,nVars,nEnsem,Nx,Ny);
719 end%if patches.nEdge
```

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

```
727 end% function patchEdgeInt2
```

10 configPatches2(): configures spatial patches in 2D

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys2()`. Section 10.1 lists an example of its use.

```
19 function patches = configPatches2(fun,Xlim,Dom ...
20     ,nPatch,ordCC,dx,nSubP,varargin)
21 version = '2023-04-12';
```

Input If invoked with no input arguments, then executes an example of simulating a nonlinear diffusion PDE relevant to the lubrication flow of a thin layer of fluid—see Section 10.1 for an 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 2D micro-grid within all the 2D patches.
- `Xlim` array/vector giving the rectangular macro-space domain of the computation, namely $[\texttt{Xlim(1)},\texttt{Xlim(2)}]\times[\texttt{Xlim(3)},\texttt{Xlim(4)}]$. If `Xlim` has two elements, then the domain is the square domain of the same interval in both directions.
- `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 doubly macro-periodic in the 2D 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/bottom/top edges of the leftmost/rightmost/bottommost/topmost patches, respectively.
  - `.bcOffset`, optional one, two or four element vector/array, in the cases of `'equispace'` or `'chebyshev'` the patches are placed so the left/right/top/bottom macroscale boundaries are aligned to the left/right/top/bottom edges of the corresponding extreme patches,

but offset by `.bcOffset` of the sub-patch micro-grid spacing. For example, use `bcOffset=0` when the micro-code applies Dirichlet boundary values on the extreme edge micro-grid points, whereas use `bcOffset=0.5` when the microcode applies Neumann boundary conditions halfway between the extreme edge micro-grid points. Similarly for the top and bottom edges.

If `.bcOffset` is a scalar, then apply the same offset to all boundaries. If two elements, then apply the first offset to both x-boundaries, and the second offset to both y-boundaries. If four elements, then apply the first two offsets to the respective x-boundaries, and the last two offsets to the respective y-boundaries.

- `.X`, optional vector/array with `nPatch(1)` elements, in the case `'usergiven'` it specifies the x-locations of the centres of the patches—the user is responsible the locations makes sense.
- `.Y`, optional vector/array with `nPatch(2)` elements, in the case `'usergiven'` it specifies the y-locations of the centres of the patches—the user is responsible the locations makes sense.

- `nPatch` sets the number of equi-spaced spatial patches: if scalar, then use the same number of patches in both directions, otherwise `nPatch(1:2)` gives the number of patches (≥ 1) in each direction.

- `ordCC` is the 'order' of interpolation for inter-patch coupling across empty space of the macroscale patch values to the edge-values of the patches: currently must be $0, 2, 4, \ldots$; where 0 gives spectral interpolation.

- `dx` (real—scalar or two element) is usually the sub-patch micro-grid spacing in x and y. If scalar, then use the same `dx` in both directions, otherwise `dx(1:2)` gives the spacing in each of the two directions.

  However, if `Dom` is `NaN` (as for pre-2023), then `dx` actually is `ratio` (scalar or two element), 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 scalar, then use the same number in both directions, otherwise

`nSubP(1:2)` gives the number in each direction. If not using `EdgyInt`, then `nSubP./nEdge` must be odd integer(s) so that there is/are centre-patch lattice lines. So for the defaults of `nEdge` = 1 and not `EdgyInt`, then `nSubP` must be odd.

- `'nEdge'`, *optional* (integer—scalar or two element), default=1, the width of edge values set by interpolation at the edge regions of each patch. If two elements, then respectively the width in x, y-directions. The default is one (suitable for microscale lattices with only nearest neighbour interactions).

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

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

- `hetCoeffs`, *optional*, default empty. Supply a 2D or 3D array of microscale heterogeneous coefficients to be used by the given microscale `fun` in each patch. Say the given array `cs` is of size $m_x \times m_y \times n_c$, where n_c is the number of different sets of coefficients. For example, in heterogeneous diffusion, $n_c = 2$ for the diffusivities in the *two* different spatial directions (or $n_c = 3$ for the diffusivity tensor). The coefficients are to be the same for each and every patch; however, macroscale variations are catered for by the n_c coefficients being n_c parameters in some macroscale formula.

  - If `nEnsem` = 1, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the $(1, 1)$-point in each patch. 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` $:= m_x \cdot m_y$ and construct an ensemble of all $m_x \cdot m_y$ phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When `EdgyInt` is true, and when the coefficients are diffusivities/elasticities in x and y directions, respectively, then this coupling cunningly preserves symmetry.

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

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

Output The struct `patches` is created and set with the following components. If no output variable is provided for `patches`, then make the struct available as a global variable.[13]

```
213 if nargout==0, global patches, end
214 patches.version = version;
```

- `.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—not yet implemented.

- `.Cwtsr` and `.Cwtsl`, only for macro-periodic conditions, are the `ordCC`$\times 2$-array of weights for the inter-patch interpolation onto the right/top and left/bottom edges (respectively) with patch:macroscale ratio as specified or as derived from `dx`.

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

- `.x` (6D) is `nSubP(1)` $\times 1 \times 1 \times 1 \times$ `nPatch(1)` $\times 1$ array of the regular spatial locations x_{iI} of the microscale grid points in every patch.
- `.y` (6D) is $1 \times$ `nSubP(2)` $\times 1 \times 1 \times 1 \times$ `nPatch(2)` array of the regular spatial locations y_{jJ} of the microscale grid points in every patch.
- `.ratio` 1×2, only for macro-periodic conditions, are the size ratios of every patch.
- `.nEdge` 1×2, is the width of edge values set by interpolation at the edge regions of each patch, in the x, y-directions respectively.
- `.le`, `.ri`, `.bo`, `.to` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.
- `.cs` either
  - `[]` 0D, or
  - if `nEnsem` $= 1$, (`nSubP(1)` $- 1$) $\times$ (`nSubP(2)` $- 1$) $\times n_c$ 3D array of microscale heterogeneous coefficients, or
  - if `nEnsem` > 1, (`nSubP(1)` -1) $\times$ (`nSubP(2)` -1) $\times n_c \times m_x m_y$ 4D array of $m_x m_y$ ensemble of phase-shifts of the microscale heterogeneous coefficients.
- `.parallel`, logical: true if patches are distributed over multiple CPUs/ cores for the Parallel Computing Toolbox, otherwise false (the default is to activate the *local* pool).
- `.codist`, *optional*, describes the particular parallel distribution of arrays over the active parallel pool.

10.1 If no arguments, then execute an example

```
298 if nargin==0
299 disp('With no arguments, simulate example of nonlinear diffusion')
```

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

1. configPatches2
2. ode23 integrator $\mapsto$ patchSys2 $\mapsto$ user's PDE
3. process results

Establish global patch data struct to interface with a function coding a nonlinear 'diffusion' PDE: to be solved on 6×4-periodic domain, with 9×7 patches, spectral interpolation (0) couples the patches, with 5×5 points forming the micro-grid in each patch, and a sub-patch micro-grid spacing of 0.12 (relatively large for visualisation). Roberts, MacKenzie, and Bunder (2014) established that this scheme is consistent with the PDE (as the patch spacing decreases).

```
global patches
patches = configPatches2(@nonDiffPDE,[-3 3 -2 2] ...
    ,'periodic', [9 7], 0, 0.12, 5 ,'EdgyInt',false);
```

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

```
u0 = exp(-patches.x.^2-patches.y.^2);
u0 = u0.*(0.9+0.1*rand(size(u0)));
```

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

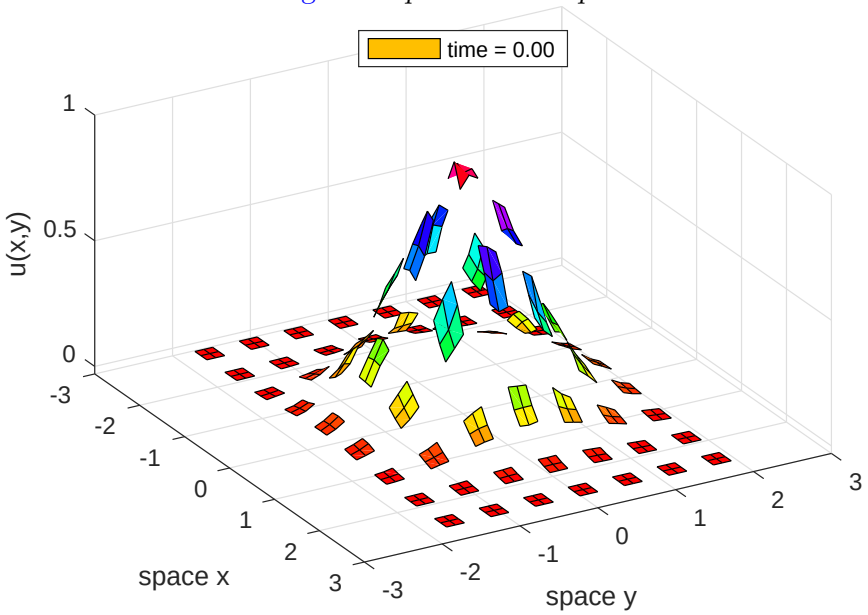
```
figure(1), clf, colormap(0.8*hsv)
x = squeeze(patches.x);  y = squeeze(patches.y);
if 1, x([1 end],:) = nan; y([1 end],:) = nan; end
```

Start by showing the initial conditions of Figure 18 while the simulation computes.

```
u = reshape(permute(squeeze(u0) ...
    ,[1 3 2 4]), [numel(x) numel(y)]);
hsurf = mesh(x(:),y(:),u');
axis([-3 3 -3 3 -0.03 1]), view(60,40)
legend('time = 0.00','Location','north')
xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
colormap(hsv)
ifOurCf2eps([mfilename 'ic'])
```

Integrate in time to $t = 4$ using standard functions. In MATLAB `ode15s` would be natural as the patch scheme is naturally stiff, but `ode23` is quicker (Maclean, Bunder, and Roberts 2021, Fig. 4). Ask for output at non-uniform times because the diffusion slows.

Figure 18: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to a nonlinear 'diffusion' PDE: Figure 19 plots the computed field at time $t = 3$.

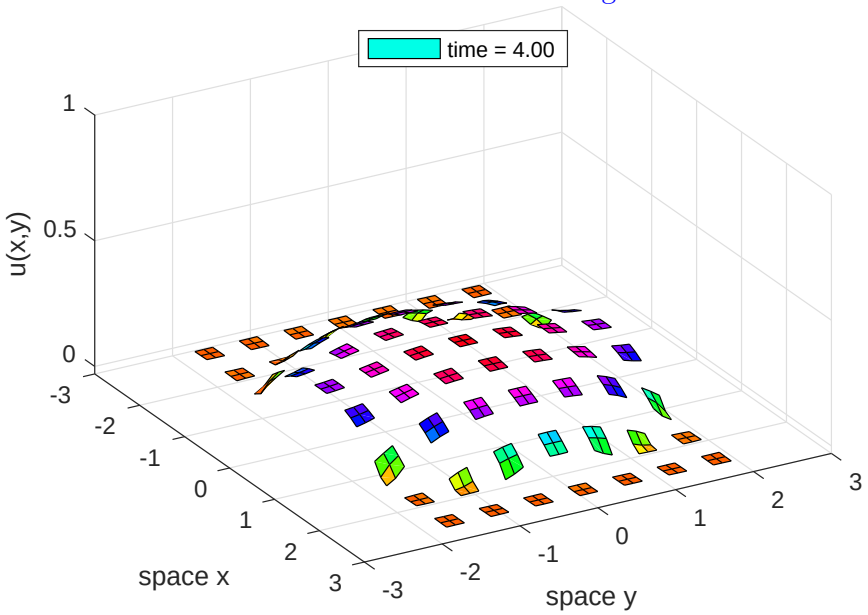



```
disp('Wait to simulate nonlinear diffusion h_t=(h^3)_xx+(h^3)_yy')
drawnow
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@patchSys2,linspace(0,2).^2,u0(:));
else % octave version is quite slow for me
    lsode_options('absolute tolerance',1e-4);
    lsode_options('relative tolerance',1e-4);
    [ts,us] = odeOcts(@patchSys2,[0 1],u0(:));
end
```

Animate the computed simulation to end with Figure 19. Use `patchEdgeInt2` to interpolate patch-edge values.

```
for i = 1:length(ts)
  u = patchEdgeInt2(us(i,:));
  u = reshape(permute(squeeze(u) ...
      ,[1 3 2 4]), [numel(x) numel(y)]);
  set(hsurf,'ZData', u');
  legend(['time = ' num2str(ts(i),'%4.2f')])
```

Figure 19: field $u(x, y, t)$ at time $t = 3$ of the patch scheme applied to a nonlinear 'diffusion' PDE with initial condition in Figure 18.




```
395   pause(0.1)
396 end
397 ifOurCf2eps([mfilename 't3'])
```

Upon finishing execution of the example, exit this function.

```
412 return
413 end%if no arguments
```

10.2 Parse input arguments and defaults

```
427 p = inputParser;
428 fnValidation = @(f) isa(f, 'function_handle');%test for fn name
429 addRequired(p,'fun',fnValidation);
430 addRequired(p,'Xlim',@isnumeric);
431 %addRequired(p,'Dom'); % nothing yet decided
432 addRequired(p,'nPatch',@isnumeric);
433 addRequired(p,'ordCC',@isnumeric);
434 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); % not yet implemented
parse(p,fun,Xlim,nPatch,ordCC,dx,nSubP,varargin{:});
```

Set the optional parameters.

```
patches.nEdge = p.Results.nEdge;
if numel(patches.nEdge)==1
    patches.nEdge = repmat(patches.nEdge,1,2);
    end
patches.EdgyInt = p.Results.EdgyInt;
patches.nEnsem = p.Results.nEnsem;
cs = p.Results.hetCoeffs;
patches.parallel = p.Results.parallel;
%patches.nCore = p.Results.nCore;
```

Initially duplicate parameters for both space dimensions as needed.

```
if numel(Xlim)==2,   Xlim = repmat(Xlim,1,2); end
if numel(nPatch)==1, nPatch = repmat(nPatch,1,2); end
if numel(dx)==1,     dx = repmat(dx,1,2); end
if numel(nSubP)==1,  nSubP = repmat(nSubP,1,2); end
```

Check parameters.

```
assert(Xlim(1)<Xlim(2) ...
      ,'first pair of Xlim must be ordered increasing')
assert(Xlim(3)<Xlim(4) ...
      ,'second pair of Xlim must be ordered increasing')
assert((mod(ordCC,2)==0)|all(patches.nEdge==1) ...
      ,'Cannot yet have nEdge>1 and staggered patch grids')
assert(all(3*patches.nEdge<=nSubP) ...
      ,'too many edge values requested')
assert(all(rem(nSubP,patches.nEdge)==0) ...
      ,'nSubP must be integer multiple of nEdge')
if ~patches.EdgyInt, assert(all(rem(nSubP./patches.nEdge,2)==1) ...
```

```
485       ,'for non-edgyInt, nSubP./nEdge must be odd integer')
486       end
487 if (patches.nEnsem>1)&all(patches.nEdge>1)
488       warning('not yet tested when both nEnsem and nEdge non-one')
489       end
490 %if patches.nCore>1
491 %    warning('nCore>1 not yet tested in this version')
492 %    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` vector.

```
503 if ~isstruct(Dom), pre2023=isnan(Dom);
504 else pre2023=false; end
505 if pre2023, ratio=dx; dx=nan; end
```

Default macroscale conditions are periodic with evenly spaced patches.

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

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

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

We allow different macroscale domain conditions in the different directions. But for the moment do not allow periodic to be mixed with the others (as the interpolation mechanism is different code)—hence why we choose `periodic` be seven characters, whereas the others are eight characters. The different conditions are coded in different rows of `Dom.type`, so we duplicate the string if only one row specified.

```
535 if size(Dom.type,1)==1, Dom.type=repmat(Dom.type,2,1); end
```

Check what is and is not specified, and provide default of zero (Dirichlet boundaries) if no `bcOffset` specified when needed. Do so for both directions independently.

```
544 patches.periodic=false;
545 for p=1:2
546 switch Dom.type(p,:)
547 case 'periodic'
```

```
548     patches.periodic=true;
549     if isfield(Dom,'bcOffset')
550     warning('bcOffset not available for Dom.type = periodic'), end
551     msg=' not available for Dom.type = periodic';
552     if isfield(Dom,'X'), warning(['X' msg]), end
553     if isfield(Dom,'Y'), warning(['Y' msg]), end
554 case {'equispace','chebyshev'}
555     if ~isfield(Dom,'bcOffset'), Dom.bcOffset=zeros(2,2); end
556     % for mixed with usergiven, following should still work
557     if numel(Dom.bcOffset)==1
558         Dom.bcOffset=repmat(Dom.bcOffset,2,2); end
559     if numel(Dom.bcOffset)==2
560         Dom.bcOffset=repmat(Dom.bcOffset(:)',2,1); end
561     msg=' not available for Dom.type = equispace or chebyshev';
562     if (p==1)& isfield(Dom,'X'), warning(['X' msg]), end
563     if (p==2)& isfield(Dom,'Y'), warning(['Y' msg]), end
564 case 'usergiven'
565 %    if isfield(Dom,'bcOffset')
566 %    warning('bcOffset not available for usergiven Dom.type'), end
567     msg=' required for Dom.type = usergiven';
568     if p==1, assert(isfield(Dom,'X'),['X' msg]), end
569     if p==2, assert(isfield(Dom,'Y'),['Y' msg]), end
570 otherwise
571     error([Dom.type ' is unknown Dom.type'])
572 end%switch Dom.type
573 end%for p
```

10.3 The code to make patches

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

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

Second, store the order of interpolation that is to provide the values for the inter-patch coupling conditions. Spectral coupling is `ordCC` of 0 or (not yet??) -1. [14]

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

[14] **ToDo:** Perhaps implement staggered spectral coupling.

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

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

Check for staggered grid and periodic case.

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

Set the macro-distribution of patches Third, set the centre of the patches in the macroscale grid of patches. Loop over the coordinate directions, setting the distribution into `Q` and finally assigning to array of corresponding direction.

```
for q=1:2
qq=2*q-1;
```

Distribution depends upon `Dom.type`:

```
switch Dom.type(q,:)
```

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

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

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

```
case 'equispace'
  Q=linspace(Xlim(qq)+((nSubP(q)-1)/2-Dom.bcOffset(qq))*dx(q) ...
          ,Xlim(qq+1)-((nSubP(q)-1)/2-Dom.bcOffset(qq+1))*dx(q) ...
          ,nPatch(q));
  DQ=diff(Q(1:2));
  width=(1+patches.EdgyInt)/2*(nSubP(q)-1-patches.EdgyInt)*dx;
  if DQ<width*0.999999
     warning('too many equispace patches (double overlapping)')
     end
```

The Chebyshev case is spaced according to the Chebyshev distribution in order to reduce macro-interpolation errors, $Q_i \propto -\cos(i\pi/N)$, but with the extreme edges aligned with the spatial domain boundaries, modified by the offset, and modified by possible 'boundary layers'. [15]

```
case 'chebyshev'
  halfWidth=dx(q)*(nSubP(q)-1)/2;
  Q1 = Xlim(1)+halfWidth-Dom.bcOffset(qq)*dx(q);
  Q2 = Xlim(2)-halfWidth+Dom.bcOffset(qq+1)*dx(q);
%  Q = (Q1+Q2)/2-(Q2-Q1)/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`.

```
  pEI=patches.EdgyInt; % abbreviation
  pnE=patches.nEdge(q);% abbreviation
  width=(1+pEI)/2*(nSubP(q)-pnE*(1+pEI))*dx(q);
  for b=0:2:nPatch(q)-2
    DQmin=(Q2-Q1-b*width)/2*( 1-cos(pi/(nPatch(q)-b-1)) );
    if DQmin>width, break, end
  end%for
  if DQmin<width*0.999999
     warning('too many Chebyshev patches (mid-domain overlap)')
     end
```

Assign the centre-patch coordinates.

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

```
  Q =[ Q1+(0:b/2-1)*width ...
       (Q1+Q2)/2-(Q2-Q1-b*width)/2*cos(linspace(0,pi,nPatch(q)-b)) ..
       Q2+(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'
  if q==1, Q = reshape(Dom.X,1,[]);
  else     Q = reshape(Dom.Y,1,[]);
  end%if
end%switch Dom.type
```

Assign Q-coordinates to the correct spatial direction. At this stage they are all rows.

```
if q==1, X=Q; end
if q==2, Y=Q; end
end%for q
```

Construct the micro-grids Fourth, construct the microscale grid in each patch, centred about the given mid-points `X,Y`. Reshape the grid to be 6D to suit dimensions (micro,Vars,Ens,macro).

```
xs = dx(1)*( (1:nSubP(1))-mean(1:nSubP(1)) );
patches.x = reshape( xs'+X ...
                    ,nSubP(1),1,1,1,nPatch(1),1);
ys = dx(2)*( (1:nSubP(2))-mean(1:nSubP(2)) );
patches.y = reshape( ys'+Y ...
                    ,1,nSubP(2),1,1,1,nPatch(2));
```

Pre-compute weights for macro-periodic In the case of macro-periodicity, precompute the weightings to interpolate field values for coupling. [16]

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

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

```
775       patches.Cwtsr = Cwtsr;  patches.Cwtsl = Cwtsl;
776     end%if
777   end%if patches.periodic
```

10.4 Set ensemble inter-patch communication

For `EdgyInt` or centre interpolation respectively,

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

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

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

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

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

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

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

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

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

```
838     patches.nEnsem = mx*my;
839     patches.cs = nan(nx-1,ny-1,nc,mx,my);
840     for j = 1:my
841         js = (j:j+ny-2);
842         for i = 1:mx
843             is = (i:i+nx-2);
844             patches.cs(:,:,:,i,j) = cs(is,js,:);
845         end
846     end
847     patches.cs = reshape(patches.cs,nx-1,ny-1,nc,[]);
```

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

```
857     le = mod((0:mx-1)+mod(nx-2,mx),mx)+1;
858     patches.le = reshape(  le'+mx*(0:my-1)  ,[],1);
859     ri = mod((0:mx-1)-mod(nx-2,mx),mx)+1;
860     patches.ri = reshape(  ri'+mx*(0:my-1)  ,[],1);
861     bo = mod((0:my-1)+mod(ny-2,my),my)+1;
862     patches.bo = reshape( (1:mx)'+mx*(bo-1) ,[],1);
863     to = mod((0:my-1)-mod(ny-2,my),my)+1;
864     patches.to = reshape( (1:mx)'+mx*(to-1) ,[],1);
```

Issue warning if the ensemble is likely to be affected by lack of scale separation. [17]

```
872 if prod(ratio)*patches.nEnsem>0.9, warning( ...
873 'Probably poor scale separation in ensemble of coupled phase-shifts')
874 scaleSeparationParameter = ratio*patches.nEnsem
875 end
```

End the two if-statements.

```
881   end%if-else nEnsem>1
882 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

[17] **ToDo:** Maybe need to justify this and the arbitrary threshold more carefully??

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.*[18]

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

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

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

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

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

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

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

Fin

```
end% function
```

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

11 patchSys2(): interface 2D space to time integrators

To simulate in time with 2D spatial patches we often need to interface a users time derivative function with time integration routines such as `ode23` or `PIRK2`. This function provides an interface. Communicate patch-design variables (Section 10) 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 = patchSys2(t,u,patches,varargin)
24 if nargin<3, global patches, end
```

Input

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

Output

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

Reshape the fields `u` as a 6D-array, and sets the edge values from macroscale interpolation of centre-patch values. Section 9 describes `patchEdgeInt2()`.

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

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

```
104 dudt = patches.fun(t,u,patches,varargin{:});
105 m = patches.nEdge(1);
106 dudt([1:m end-m+1:end],:,:) = 0;
107 m = patches.nEdge(2);
108 dudt(:,[1:m end-m+1:end],:) = 0;
109 dudt = reshape(dudt,sizeu);
```

Fin.

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

Couples 3D patches across 3D space by computing their face values via macroscale interpolation. Assumes patch face values are determined by macroscale interpolation of the patch centre-plane values (Roberts, MacKenzie, and Bunder 2014; Bunder et al. 2021), or patch next-to-face values which appears better (Bunder, Kevrekidis, and Roberts 2021). This function is primarily used by `patchSys3()` but is also useful for user graphics. [19]

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

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

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

Input

- `u` is a vector/array of length `prod(nSubP)`$\cdot$`nVars`$\cdot$`nEnsem`$\cdot$`prod(nPatch)` where there are `nVars` $\cdot$ `nEnsem` field values at each of the points in the `nSubP1` $\cdot$ `nSubP2` $\cdot$ `nSubP3` $\cdot$ `nPatch1` $\cdot$ `nPatch2` $\cdot$ `nPatch3` multiscale spatial grid on the `nPatch1` $\cdot$ `nPatch2` $\cdot$ `nPatch3` array of patches.

- `patches` a struct set by `configPatches3()` which includes the following information.

  - `.x` is `nSubP1` $\times 1 \times 1 \times 1 \times$ `nPatch1` $\times 1 \times 1$ array of the spatial locations x_{iI} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index i, but may be variable spaced in macroscale index I.
  - `.y` is similarly $1 \times$ `nSubP2` $\times 1 \times 1 \times 1 \times$ `nPatch2` $\times 1$ array of the spatial locations y_{jJ} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index j, but may be variable spaced in macroscale index J.
  - `.z` is similarly $1 \times 1 \times$ `nSubP3` $\times 1 \times 1 \times 1 \times$ `nPatch3` array of the spatial locations z_{kK} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index k, but may be variable spaced in macroscale index K.
  - `.ordCC` is order of interpolation, currently only $\{0, 2, 4, \ldots\}$
  - `.periodic` indicates whether macroscale is periodic domain, or alternatively that the macroscale has left, right, top, bottom, front and back boundaries so interpolation is via divided differences.
  - `.stag` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation. Currently must be zero.
  - `.Cwtsr` and `.Cwtsl` are the coupling coefficients for finite width interpolation in each of the x, y, z-directions—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 cross-patch values (near original scheme).
  - `.nEdge`, three elements, the width of edge values set by interpolation at the x, y, z-face regions, respectively, of each patch (default is one all x, y, z-faces).

– `.nEnsem` the number of realisations in the ensemble.

– `.parallel` whether serial or parallel.

Output

- `u` is 8D array, `nSubP1·nSubP2·nSubP3·nVars·nEnsem·nPatch1·nPatch2·nPatch3`, of the fields with face values set by interpolation.

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.

```
129   if max(abs(imag(u(:))))<1e-9*max(abs(u(:)))
130        uclean=@(u) real(u);
131   else uclean=@(u) u;
132   end
```

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

```
140 [~,~,nz,~,~,~,~,Nz] = size(patches.z);
141 [~,ny,~,~,~,~,Ny,~] = size(patches.y);
142 [nx,~,~,~,~,Nx,~,~] = size(patches.x);
143 nEnsem = patches.nEnsem;
144 nVars = round( numel(u)/numel(patches.x) ...
145     /numel(patches.y)/numel(patches.z)/nEnsem );
146 assert(numel(u) == nx*ny*nz*Nx*Ny*Nz*nVars*nEnsem ...
147   ,'patchEdgeInt3: input u has wrong size for parameters')
148 u = reshape(u,[nx ny nz nVars nEnsem Nx Ny Nz]);
```

For the moment assume the physical domain is either macroscale periodic or macroscale rectangle so that the coupling formulas are simplest. These index vectors point to patches and, if periodic, their six immediate neighbours.

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

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

```
%x = patches.x;
%if patches.nEdge(1)>1
%  m = patches.nEdge(1);
%  x = x(1:m:nx,:,:,:,:,:,:,:);
%  nx = nx/m;
%  u = reshape(u,m,nx,ny,nz,nVars,nEnsem,Nx,Ny,Nz);
%  nVars = nVars*m;
%  u = reshape( permute(u,[2:4 1 5:9]) ...
%              ,nx,ny,nz,nVars,nEnsem,Nx,Ny,Nz);
%end%if patches.nEdge(1)
%y = patches.y;
%if patches.nEdge(2)>1
%  m = patches.nEdge(2);
%  y = y(:,1:m:ny,:,:,:,:,:,:);
%  ny = ny/m;
%  u = reshape(u,nx,m,ny,nz,nVars,nEnsem,Nx,Ny,Nz);
%  nVars = nVars*m;
%  u = reshape( permute(u,[1 3:4 2 5:9]) ...
%              ,nx,ny,nz,nVars,nEnsem,Nx,Ny,Nz);
%end%if patches.nEdge(2)
%z = patches.z;
%if patches.nEdge(3)>1
%  m = patches.nEdge(3);
%  z = z(:,:,1:m:nz,:,:,:,:,:);
%  nz = nz/m;
%  u = reshape(u,nx,ny,m,nz,nVars,nEnsem,Nx,Ny,Nz);
%  nVars = nVars*m;
%  u = reshape( permute(u,[1:2 4 3 5:9]) ...
%              ,nx,ny,nz,nVars,nEnsem,Nx,Ny,Nz);
%end%if patches.nEdge(3)
x = patches.x;
y = patches.y;
z = patches.z;
if mean(patches.nEdge)>1
  mx = patches.nEdge(1);
  my = patches.nEdge(2);
  mz = patches.nEdge(3);
  x = x(1:mx:nx,:,:,:,:,:,:,:);
  y = y(:,1:my:ny,:,:,:,:,:,:);
```

```
  z = z(:,:,1:mz:nz,:,:,:,:,:);
  nx = nx/mx;
  ny = ny/my;
  nz = nz/mz;
  u = reshape(u,mx,nx,my,ny,mz,nz,nVars,nEnsem,Nx,Ny,Nz);
  nVars = nVars*mx*my*mz;
  u = reshape( permute(u,[2:2:6 1:2:5 7:11]) ...
              ,nx,ny,nz,nVars,nEnsem,Nx,Ny,Nz);
end%if patches.nEdge
```

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

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

12.1 Periodic macroscale interpolation schemes

```
if patches.periodic
```

Get the size ratios of the patches in each direction.

```
rx = patches.ratio(1);
ry = patches.ratio(2);
rz = patches.ratio(3);
```

12.1.1 Lagrange interpolation gives patch-face values

Compute centred differences of the mid-patch values for the macro-interpolation, of all fields. Here the domain is macro-periodic.

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

Interpolate the three directions in succession, in this way we naturally fill-in face-edge and corner values. Start with x-direction, and give most documentation for that case as the others are essentially the same.

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

```
  if patches.EdgyInt % interpolate next-to-face values
    U = u([2 nx-1],2:(ny-1),2:(nz-1),:,:,I,J,K);
  else % interpolate centre-cross values
    U = u(i0,2:(ny-1),2:(nz-1),:,:,I,J,K);
  end;%if patches.EdgyInt
```

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.

```
szU0=size(U); szU0=[szU0 ones(1,8-length(szU0)) ordCC];
```

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

```
  if ~patches.parallel, dmu = zeros(szU0); % 9D
  else   dmu = zeros(szU0,patches.codist); % 9D
  end%if patches.parallel
```

First compute differences $\mu\delta$ and δ^2.

```
  if patches.stag % use only odd numbered neighbours
    error('polynomial interpolation not yet for staggered patch coupl
%    dmux(:,:,:,:,:,I,:,:,1) = (Ux(:,:,:,:,:,Ip,:,:) +Ux(:,:,:,:,:,Im
%    dmux(:,:,:,:,:,I,:,:,2) = (Ux(:,:,:,:,:,Ip,:,:) -Ux(:,:,:,:,:,Im
%    Ip = Ip(Ip); Im = Im(Im); % increase shifts to \pm2
%    dmuy(:,:,:,:,:,:,J,:,1) = (Ux(:,:,:,:,:,:,Jp,:)+Ux(:,:,:,:,:,:,J
%    dmuy(:,:,:,:,:,:,J,:,2) = (Ux(:,:,:,:,:,:,Jp,:)-Ux(:,:,:,:,:,:,J
%    Jp = Jp(Jp); Jm = Jm(Jm); % increase shifts to \pm2
%    dmuz(:,:,:,:,:,:,:,K,1) = (Ux(:,:,:,:,:,:,:,Kp)+Ux(:,:,:,:,:,:,:
%    dmuz(:,:,:,:,:,:,:,K,2) = (Ux(:,:,:,:,:,:,:,Kp)-Ux(:,:,:,:,:,:,:
%    Kp = Kp(Kp); Km = Km(Km); % increase shifts to \pm2
  else  %disp('starting standard interpolation')
    dmu(:,:,:,:,:,I,:,:,1) = (U(:,:,:,:,:,Ip,:,:) ...
                              -U(:,:,:,:,:,Im,:,:))/2; %\mu\delta
    dmu(:,:,:,:,:,I,:,:,2) = (U(:,:,:,:,:,Ip,:,:) ...
       -2*U(:,:,:,:,:,I,:,:) +U(:,:,:,:,:,Im,:,:));   %\delta^2
  end% if stag
```

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

```
326   for k = 3:ordCC
327     dmu(:,:,:,:,:,I,:,:,k) =     dmu(:,:,:,:,:,Ip,:,:,k-2) ...
328     -2*dmu(:,:,:,:,:,I,:,:,k-2) +dmu(:,:,:,:,:,Im,:,:,k-2);
329   end
```

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

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

```
345 k=1+patches.EdgyInt; % use centre or two faces
346 u(nx,2:(ny-1),2:(nz-1),:,patches.ri,I,:,:) ...
347   = U(1,:,:,:,:,:,:,:)*(1-patches.stag) ...
348   +sum( shiftdim(patches.Cwtsr(:,1),-8).*dmu(1,:,:,:,:,:,:,:,:) ,9);
349 u(1 ,2:(ny-1),2:(nz-1),:,patches.le,I,:,:) ...
350   = U(k,:,:,:,:,:,:,:)*(1-patches.stag) ...
351   +sum( shiftdim(patches.Cwtsl(:,1),-8).*dmu(k,:,:,:,:,:,:,:,:) ,9);
```

y-normal face values Interpolate from either the next-to-edge-face values, or the centre-cross-plane values.

```
363   if patches.EdgyInt % interpolate next-to-face values
364     U = u(:,[2 ny-1],2:(nz-1),:,:,I,J,K);
365   else % interpolate centre-cross values
366     U = u(:,j0,2:(nz-1),:,:,I,J,K);
367   end;%if patches.EdgyInt
```

Adjoin extra dimension for the array of differences.

```
373 szU0=size(U); szU0=[szU0 ones(1,8-length(szU0)) ordCC];
```

Store finite differences $(\mu\delta, \delta^2, \mu\delta^3, \delta^4, \ldots)$ in this array.

```
380   if ~patches.parallel, dmu = zeros(szU0); % 9D
381   else   dmu = zeros(szU0,patches.codist); % 9D
382   end%if patches.parallel
```

First compute differences $\mu\delta$ and δ^2.

```
388   if patches.stag % use only odd numbered neighbours
389     error('polynomial interpolation not yet for staggered patch coupl
390   else  %disp('starting standard interpolation')
391     dmu(:,:,:,:,:,:,J,:,1) = (U(:,:,:,:,:,:,:,Jp,:) ...
392                               -U(:,:,:,:,:,:,:,Jm,:))/2; %\mu\delta
393     dmu(:,:,:,:,:,:,J,:,2) = (U(:,:,:,:,:,:,:,Jp,:) ...
394        -2*U(:,:,:,:,:,:,:,J,:) +U(:,:,:,:,:,:,:,Jm,:));   %\delta^2
395   end% if stag
```

Recursively take δ^2.

```
401   for k = 3:ordCC
402     dmu(:,:,:,:,:,:,J,:,k) =     dmu(:,:,:,:,:,:,:,Jp,:,k-2) ...
403     -2*dmu(:,:,:,:,:,:,:,J,:,k-2) +dmu(:,:,:,:,:,:,:,Jm,:,k-2);
404   end
```

Interpolate macro-values using the weights pre-computed by `configPatches3()`. An ensemble of configurations may have cross-coupling.

```
412 k=1+patches.EdgyInt; % use centre or two faces
413 u(:,ny,2:(nz-1),:,patches.to,:,J,:) ...
414   = U(:,1,:,:,:,:,:,:)*(1-patches.stag) ...
415   +sum( shiftdim(patches.Cwtsr(:,2),-8).*dmu(:,1,:,:,:,:,:,:,:) ,9);
416 u(:,1 ,2:(nz-1),:,patches.bo,:,J,:) ...
417   = U(:,k,:,:,:,:,:,:)*(1-patches.stag) ...
418   +sum( shiftdim(patches.Cwtsl(:,2),-8).*dmu(:,k,:,:,:,:,:,:,:) ,9);
```

z-normal face values Interpolate from either the next-to-edge-face values, or the centre-cross-plane values.

```
429   if patches.EdgyInt % interpolate next-to-face values
430     U = u(:,:,[2 nz-1],:,:,I,J,K);
431   else % interpolate centre-cross values
432     U = u(:,:,k0,:,:,I,J,K);
433   end;%if patches.EdgyInt
```

Adjoin extra dimension for the array of differences.

```
439 szUO=size(U); szUO=[szUO ones(1,8-length(szUO)) ordCC];
```

Store finite differences $(\mu\delta, \delta^2, \mu\delta^3, \delta^4, \ldots)$ in this array.

```
  if ~patches.parallel, dmu = zeros(szUO); % 9D
  else   dmu = zeros(szUO,patches.codist); % 9D
  end%if patches.parallel
```

First compute differences $\mu\delta$ and δ^2.

```
  if patches.stag % use only odd numbered neighbours
    error('polynomial interpolation not yet for staggered patch coupl
  else  %disp('starting standard interpolation')
    dmu(:,:,:,:,:,:,:,:,K,1) = (U(:,:,:,:,:,:,:,:,Kp) ...
                                -U(:,:,:,:,:,:,:,:,Km))/2; %\mu\delta
    dmu(:,:,:,:,:,:,:,:,K,2) = (U(:,:,:,:,:,:,:,:,Kp) ...
       -2*U(:,:,:,:,:,:,:,:,K) +U(:,:,:,:,:,:,:,:,Km));   %\delta^2
  end% if stag
```

Recursively take δ^2.

```
  for k = 3:ordCC
    dmu(:,:,:,:,:,:,:,:,K,k) =     dmu(:,:,:,:,:,:,:,:,Kp,k-2) ...
    -2*dmu(:,:,:,:,:,:,:,:,K,k-2) +dmu(:,:,:,:,:,:,:,:,Km,k-2);
  end
```

Interpolate macro-values using the weights pre-computed by `configPatches3()`. An ensemble of configurations may have cross-coupling.

```
k=1+patches.EdgyInt; % use centre or two faces
u(:,:,nz,:,patches.fr,:,:,K) ...
  = U(:,:,1,:,:,:,:,:)*(1-patches.stag) ...
  +sum( shiftdim(patches.Cwtsr(:,3),-8).*dmu(:,:,1,:,:,:,:,:,:) ,9);
u(:,:,1 ,:,patches.ba,:,:,K) ...
  = U(:,:,k,:,:,:,:,:)*(1-patches.stag) ...
  +sum( shiftdim(patches.Cwtsl(:,3),-8).*dmu(:,:,k,:,:,:,:,:,:) ,9);
```

12.1.2 Case of spectral interpolation

Assumes the domain is macro-periodic.

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

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

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

```
517  if patches.stag % transform by doubling the number of fields
518  error('staggered grid not yet implemented??')
519    v=nan(size(u)); % currently to restore the shape of u
520    u=cat(3,u(:,1:2:nPatch,:),u(:,2:2:nPatch,:));
521    stagShift=reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
522    iV=[nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
523    r=r/2;           % ratio effectively halved
524    nPatch=nPatch/2; % halve the number of patches
525    nVars=nVars*2;   % double the number of fields
526  else % the values for standard spectral
527     stagShift = 0;
528     iV = 1:nVars;
529  end%if patches.stag
```

Interpolate the three directions in succession, in this way we naturally fill-in face-edge and corner values. Start with x-direction, and give most documentation for that case as the others are essentially the same. Need these indices of patch interior.

```
539 ix = 2:nx-1;   iy = 2:ny-1;   iz = 2:nz-1;
```

x-normal face values Now set wavenumbers into a vector at the correct dimension. In the case of even N these compute the $+$-case for the highest wavenumber zig-zag mode, $k = (0\,, 1\,, \ldots, k_{\max}\,, +(k_{\max}+1) - k_{\max}\,, \ldots, -1)$.

```
552   kMax = floor((Nx-1)/2);
553   kr = shiftdim( rx*2*pi/Nx*(mod((0:Nx-1)+kMax,Nx)-kMax) ,-4);
```

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

```
567 if ~patches.EdgyInt
568      Cm = fft( u(i0,iy,iz,:,:,:,:,:) ,[],6);
569      Cp = Cm;
570 else
571      Cm = fft( u(   2,iy,iz ,:,patches.le,:,:,:) ,[],6);
572      Cp = fft( u(nx-1,iy,iz ,:,patches.ri,:,:,:) ,[],6);
573 end%if ~patches.EdgyInt
```

Now invert the Fourier transforms to complete interpolation. Enforce reality when appropriate.

```
580 u(nx,iy,iz,:,:,:,:,:) = uclean( ifft( ...
581     Cm.*exp(1i*(stagShift+kr))  ,[],6) );
582 u( 1,iy,iz,:,:,:,:,:) = uclean( ifft( ...
583     Cp.*exp(1i*(stagShift-kr))  ,[],6) );
```

y-normal face values Set wavenumbers into a vector.

```
593   kMax = floor((Ny-1)/2);
594   kr = shiftdim( ry*2*pi/Ny*(mod((0:Ny-1)+kMax,Ny)-kMax) ,-5);
```

Compute the Fourier transform of the patch values on the centre-planes for all the fields.

```
601 if ~patches.EdgyInt
602      Cm = fft( u(:,j0,iz,:,:,:,:,:) ,[],7);
603      Cp = Cm;
604 else
605      Cm = fft( u(:,2   ,iz ,:,patches.bo,:,:,:) ,[],7);
606      Cp = fft( u(:,ny-1,iz ,:,patches.to,:,:,:) ,[],7);
607 end%if ~patches.EdgyInt
```

Invert the Fourier transforms to complete interpolation.

```
u(:,ny,iz,:,:,:,:,:) = uclean( ifft( ...
    Cm.*exp(1i*(stagShift+kr))  ,[],7) );
u(:, 1,iz,:,:,:,:,:) = uclean( ifft( ...
    Cp.*exp(1i*(stagShift-kr))  ,[],7) );
```

z-normal face values Set wavenumbers into a vector.

```
  kMax = floor((Nz-1)/2);
  kr = shiftdim( rz*2*pi/Nz*(mod((0:Nz-1)+kMax,Nz)-kMax) ,-6);
```

Compute the Fourier transform of the patch values on the centre-planes for all the fields.

```
if ~patches.EdgyInt
     Cm = fft( u(:,:,k0,:,:,:,:,:) ,[],8);
     Cp = Cm;
else
     Cm = fft( u(:,:,2    ,:,patches.ba,:,:,:) ,[],8);
     Cp = fft( u(:,:,nz-1 ,:,patches.fr,:,:,:) ,[],8);
end%if ~patches.EdgyInt
```

Invert the Fourier transforms to complete interpolation.

```
u(:,:,nz,:,:,:,:,:) = uclean( ifft( ...
    Cm.*exp(1i*(stagShift+kr))  ,[],8) );
u(:,:, 1,:,:,:,:,:) = uclean( ifft( ...
    Cp.*exp(1i*(stagShift-kr))  ,[],8) );

end% if ordCC>0
```

12.2 Non-periodic macroscale interpolation

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

Determine the order of interpolation `px`, `py` and `pz` (potentially different in the different directions!), and hence size of the (forward) divided difference tables in `F` (9D) for interpolating to left/right, top/bottom, and front/back faces. Because of the product-form of the patch grid, and because we are doing *only* either edgy interpolation or cross-patch interpolation (*not* just the centre patch value), the interpolations are all 1D interpolations.

```
if patches.ordCC<1
     px = Nx-1;  py = Ny-1;  pz = Nz-1;
else px = min(patches.ordCC,Nx-1);
     py = min(patches.ordCC,Ny-1);
     pz = min(patches.ordCC,Nz-1);
end
% interior indices of faces  (ix n/a)
ix=2:nx-1;  iy=2:ny-1;  iz=2:nz-1;
```

12.2.1 x-direction values

Set function values in first 'column' of the tables for every variable and across ensemble. For `EdgyInt`, the 'reversal' of the next-to-face values are because their values are to interpolate to the opposite face of each patch. [20]

```
  F = nan(patches.EdgyInt+1,ny-2,nz-2,nVars,nEnsem,Nx,Ny,Nz,px+1);
  if patches.EdgyInt % interpolate next-to-face values
    F(:,:,:,:,:,:,:,:,1) = u([nx-1 2],iy,iz,:,:,:,:,:);
    X = x([nx-1 2],:,:,:,:,:,:,:);
  else % interpolate mid-patch cross-patch values
    F(:,:,:,:,:,:,:,:,1) = u(i0,iy,iz,:,:,:,:,:);
    X = x(i0,:,:,:,:,:,:,:);
  end%if patches.EdgyInt
```

Form tables of divided differences Compute tables of (forward) divided differences (e.g., Wikipedia 2022) for every variable, and across ensemble, and in both directions, and for all three types of faces (left/right, top/bottom, and front/back). Recursively find all divided differences in the respective direction.

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

Interpolate with divided differences Now interpolate to find the face-values on left/right faces at `Xface` for every interior `Y,Z`.

[20] **ToDo:** Have no plans to implement core averaging as yet.

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

Code Horner's recursive evaluation of the interpolation polynomials. Indices `i` are those of the left face of each interpolation stencil, because the table is of forward differences. This alternative: the case of order p_x, p_y and p_z interpolation across the domain, asymmetric near the boundaries of the rectangular domain.

```
  i = max(1,min(1:Nx,Nx-ceil(px/2))-floor(px/2));
  Uface = F(:,:,:,:,:,i,:,:,px+1);
  for q = px:-1:1
    Uface = F(:,:,:,:,:,i,:,:,q) ...
    +(Xface-X(:,:,:,:,:,i+q-1,:,:)).*Uface;
  end
```

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

```
u(1 ,iy,iz,:,patches.le,:,:,:) = Uface(1,:,:,:,:,:,:,:,:);
u(nx,iy,iz,:,patches.ri,:,:,:) = Uface(2,:,:,:,:,:,:,:,:);
```

12.2.2 y-direction values

Set function values in first 'column' of the tables for every variable and across ensemble.

```
  F = nan(nx,patches.EdgyInt+1,nz-2,nVars,nEnsem,Nx,Ny,Nz,py+1);
  if patches.EdgyInt % interpolate next-to-face values
    F(:,:,:,:,:,:,:,:,:,1) = u(:,[ny-1 2],iz,:,:,:,:,:,:);
    Y = y(:,[ny-1 2],:,:,:,:,:,:,:);
  else % interpolate mid-patch cross-patch values
    F(:,:,:,:,:,:,:,:,:,1) = u(:,j0,iz,:,:,:,:,:,:);
    Y = y(:,j0,:,:,:,:,:,:,:);
  end%if patches.EdgyInt
```

Form tables of divided differences.

```
for q = 1:py
  j = 1:Ny-q;
  F(:,:,:,:,:,:,:,j,:,q+1) ...
  = ( F(:,:,:,:,:,:,:,j+1,:,q)-F(:,:,:,:,:,:,:,j,:,q)) ...
   ./(Y(:,:,:,:,:,:,:,j+q,:)  -Y(:,:,:,:,:,:,:,j,:));
end
```

Interpolate to find the top/bottom faces `Yface` for every x and interior z.

```
Yface = y(:,[1 ny],:,:,:,:,:,:);
```

Code Horner's recursive evaluation of the interpolation polynomials. Indices `j` are those of the bottom face of each interpolation stencil, because the table is of forward differences.

```
  j = max(1,min(1:Ny,Ny-ceil(py/2))-floor(py/2));
  Uface = F(:,:,:,:,:,:,j,:,py+1);
  for q = py:-1:1
    Uface = F(:,:,:,:,:,:,j,:,q) ...
    +(Yface-Y(:,:,:,:,:,:,j+q-1,:)).*Uface;
  end
```

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

```
u(:,1 ,iz,:,patches.bo,:,:,:) = Uface(:,1,:,:,:,:,:,:);
u(:,ny,iz,:,patches.to,:,:,:) = Uface(:,2,:,:,:,:,:,:);
```

12.2.3 z-direction values

Set function values in first 'column' of the tables for every variable and across ensemble.

```
  F = nan(nx,ny,patches.EdgyInt+1,nVars,nEnsem,Nx,Ny,Nz,pz+1);
  if patches.EdgyInt % interpolate next-to-face values
    F(:,:,:,:,:,:,:,:,1) = u(:,:,[nz-1 2],:,:,:,:,:);
    Z = z(:,:,[nz-1 2],:,:,:,:,:);
  else % interpolate mid-patch cross-patch values
    F(:,:,:,:,:,:,:,:,1) = u(:,:,k0,:,:,:,:,:);
    Z = z(:,:,k0,:,:,:,:,:);
  end%if patches.EdgyInt
```

Form tables of divided differences.

```
for q = 1:pz
  k = 1:Nz-q;
  F(:,:,:,:,:,:,:,k,q+1) ...
  = ( F(:,:,:,:,:,:,:,k+1,q)-F(:,:,:,:,:,:,:,k,q)) ...
   ./(Z(:,:,:,:,:,:,:,k+q)  -Z(:,:,:,:,:,:,:,k));
end
```

Interpolate to find the face-values on front/back faces `Zface` for every x, y.

```
Zface = z(:,:,[1 nz],:,:,:,:,:);
```

Code Horner's recursive evaluation of the interpolation polynomials. Indices `k` are those of the bottom face of each interpolation stencil, because the table is of forward differences.

```
  k = max(1,min(1:Nz,Nz-ceil(pz/2))-floor(pz/2));
  Uface = F(:,:,:,:,:,:,:,k,pz+1);
  for q = pz:-1:1
    Uface = F(:,:,:,:,:,:,:,k,q) ...
    +(Zface-Z(:,:,:,:,:,:,:,k+q-1)).*Uface;
  end
```

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

```
u(:,:,1 ,:,patches.fr,:,:,:) = Uface(:,:,1,:,:,:,:,:);
u(:,:,nz,:,patches.ba,:,:,:) = Uface(:,:,2,:,:,:,:,:);
```

12.2.4 Optional NaNs for safety

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

```
if isfield(patches,'intTest')&&patches.intTest
else % usual case
    u( 1,:,:,:,:, 1,:,:) = nan;
    u(nx,:,:,:,:,Nx,:,:) = nan;
    u(:, 1,:,:,:,:, 1,:) = nan;
    u(:,ny,:,:,:,:,Ny,:) = nan;
    u(:,:, 1,:,:,:,:, 1) = nan;
    u(:,:,nz,:,:,:,:,Nz) = nan;
end%if
```

End of the non-periodic interpolation code.

```
end%if patches.periodic else
```

Unfold multiple edges No need to restore x, y, z.

```
911 if mean(patches.nEdge)>1
912   nVars = nVars/(mx*my*mz);
913   u = reshape( u ,nx,ny,nz,mx,my,mz,nVars,nEnsem,Nx,Ny,Nz);
914   nx = nx*mx;
915   ny = ny*my;
916   nz = nz*mz;
917   u = reshape( permute(u,[4 1 5 2 6 3 7:11]) ...
918               ,nx,ny,nz,nVars,nEnsem,Nx,Ny,Nz);
919 end%if patches.nEdge
```

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

```
927 end% function patchEdgeInt3
```

13 configPatches3(): configures spatial patches in 3D

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys3()`, and possibly other patch functions. Section 13.1 and **??** list examples of its use.

```
20 function patches = configPatches3(fun,Xlim,Dom ...
21     ,nPatch,ordCC,dx,nSubP,varargin)
22 version = '2023-04-12';
```

Input If invoked with no input arguments, then executes an example of simulating a heterogeneous wave PDE—see Section 13.1 for an 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 3D micro-grid within all the 3D patches.
- `Xlim` array/vector giving the rectangular-cuboid macro-space domain of the computation: namely $[$`Xlim(1)`$,$`Xlim(2)`$] \times [$`Xlim(3)`$,$`Xlim(4)`$ \times [$`Xlim(5)`$,$`Xlim(6)`$]$. If `Xlim` has two elements, then the domain is the cubic domain of the same interval in all three directions.
- `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 triply macro-periodic in the 3D 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/bottom/top/back/front faces of the leftmost/rightmost/bottommost/topmost/backmost/frontmost patches, respectively.
- `.bcOffset`, optional one, three or six element vector/array, in the cases of `'equispace'` or `'chebyshev'` the patches are placed so the left/right macroscale boundaries are aligned to the left/right faces of the corresponding extreme patches, but offset by `bcOffset` of the sub-patch micro-grid spacing. For example, use `bcOffset=0` when the micro-code applies Dirichlet boundary values on the extreme face micro-grid points, whereas use `bcOffset=0.5` when the microcode applies Neumann boundary conditions halfway between the extreme face micro-grid points. Similarly for the top, bottom, back, and front faces.

  If `.bcOffset` is a scalar, then apply the same offset to all boundaries. If three elements, then apply the first offset to both x-boundaries, the second offset to both y-boundaries, and the third offset to both z-boundaries. If six elements, then apply the first two offsets to the respective x-boundaries, the middle two offsets to the respective y-boundaries, and the last two offsets to the respective z-boundaries.
- `.X`, optional vector/array with `nPatch(1)` elements, in the case `'usergiven'` it specifies the x-locations of the centres of the patches—the user is responsible the locations makes sense.
- `.Y`, optional vector/array with `nPatch(2)` elements, in the case `'usergiven'` it specifies the y-locations of the centres of the patches—the user is responsible the locations makes sense.
- `.Z`, optional vector/array with `nPatch(3)` elements, in the case `'usergiven'` it specifies the z-locations of the centres of the patches—the user is responsible the locations makes sense.

- `nPatch` sets the number of equi-spaced spatial patches: if scalar, then use the same number of patches in all three directions, otherwise `nPatch(1:3)` gives the number (≥ 1) of patches in each direction.

- `ordCC` is the 'order' of interpolation for inter-patch coupling across empty space of the macroscale patch values to the face-values of the patches: currently must be $0, 2, 4, \ldots$; where 0 gives spectral interpolation.

- `dx` (real—scalar or three elements) is usually the sub-patch micro-grid spacing in x, y and z. If scalar, then use the same `dx` in all three directions, otherwise `dx(1:3)` gives the spacing in each of the three directions.

  However, if `Dom` is `NaN` (as for pre-2023), then `dx` actually is `ratio` (scalar or three elements), 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 scalar, then use the same number in all three directions, otherwise `nSubP(1:3)` gives the number in each direction. If not using `EdgyInt`, then `nSubP./nEdge` must be odd integer(s) so that there is/are centre-patch lattice planes. So for the defaults of `nEdge` $= 1$ and not `EdgyInt`, then `nSubP` must be odd.

- `'nEdge'`, *optional* (integer—scalar or three element), default=1, the width of face values set by interpolation at the face regions of each patch. If two elements, then respectively the width in x, y-directions. The default is one (suitable for microscale lattices with only nearest neighbour interactions).

- `'EdgyInt'`, true/false, *optional*, default=false. If true, then interpolate to left/right/top/bottom/front/back face-values from right/left/bottom/top/back/front next-to-face values. If false or omitted, then interpolate from centre-patch planes.

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

- `'hetCoeffs'`, *optional*, default empty. Supply a 3D or 4D array of microscale heterogeneous coefficients to be used by the given microscale `fun` in each patch. Say the given array `cs` is of size $m_x \times m_y \times m_z \times n_c$, where n_c is the number of different arrays of coefficients. For example, in heterogeneous diffusion, $n_c = 3$ for the diffusivities in the *three* different spatial directions (or $n_c = 6$ for the diffusivity tensor). The coefficients are to be the same for each and every patch. However, macroscale variations are catered for by the n_c coefficients being n_c parameters in some macroscale formula.

  - If `nEnsem` $= 1$, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the $(1, 1, 1)$-point in each patch. 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` $:= m_x \cdot m_y \cdot m_z$ and construct an ensemble of all $m_x \cdot m_y \cdot m_z$ phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When `EdgyInt` is true, and when the coefficients are diffusivities/elasticities in x, y, z-directions, respectively, then this coupling cunningly preserves symmetry.

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

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

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

as a global variable.[21]

```
225 if nargout==0, global patches, end
226 patches.version = version;
```

- `.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—not yet implemented.
- `.Cwtsr` and `.Cwtsl` are the `ordCC` $\times$ 3-array of weights for the inter-patch interpolation onto the right/top/front and left/bottom/back faces (respectively) with patch:macroscale ratio as specified or as derived from `dx`.
- `.x` (8D) is `nSubP(1)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(1)` $\times 1 \times 1$ array of the regular spatial locations x_{iI} of the microscale grid points in every patch.
- `.y` (8D) is $1 \times$ `nSubP(2)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(2)` $\times 1$ array of the regular spatial locations y_{jJ} of the microscale grid points in every patch.
- `.z` (8D) is $1 \times 1 \times$ `nSubP(3)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(3)` array of the regular spatial locations z_{kK} of the microscale grid points in every patch.
- `.ratio` 1×3, only for macro-periodic conditions, are the size ratios of every patch.
- `.nEdge` 1×3, is the width of face values set by interpolation at the face regions of each patch, in the x, y, z-directions respectively.
- `.le`, `.ri`, `.bo`, `.to`, `.ba`, `.fr` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.

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

- `.cs` either
  - `[]` 0D, or
  - if `nEnsem` $= 1$, (`nSubP(1)` -1) $\times$ (`nSubP(2)` -1) $\times$ (`nSubP(3)` -1) $\times n_c$ 4D array of microscale heterogeneous coefficients, or
  - if `nEnsem` > 1, (`nSubP(1)` -1) $\times$ (`nSubP(2)` -1) $\times$ (`nSubP(3)` -1) $\times$ $n_c \times m_x m_y m_z$ 5D array of $m_x m_y m_z$ ensemble of phase-shifts of the microscale heterogeneous coefficients.
- `.parallel`, logical: true if patches are distributed over multiple CPUs/ cores for the Parallel Computing Toolbox, otherwise false (the default is to activate the *local* pool).
- `.codist`, *optional*, describes the particular parallel distribution of arrays over the active parallel pool.

13.1 If no arguments, then execute an example

```
315 if nargin==0
316 disp('With no arguments, simulate example of heterogeneous wave')
```

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

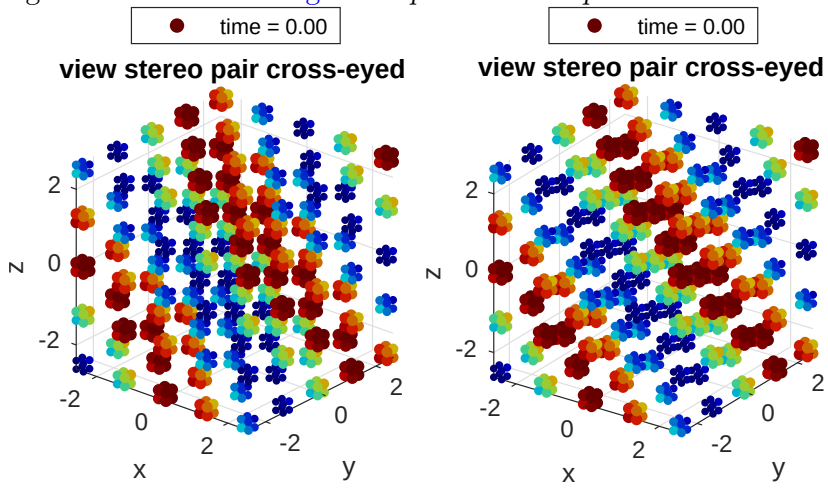
1. configPatches3
2. ode23 integrator $\mapsto$ patchSys3 $\mapsto$ user's PDE
3. process results

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

```
334 mPeriod = [2 2 2];
335 cHetr = exp(0.9*randn([mPeriod 3]));
336 cHetr = cHetr*mean(1./cHetr(:))
```

Establish global patch data struct to interface with a function coding a nonlinear 'diffusion' PDE: to be solved on $[-\pi, \pi]^3$-periodic domain, with 5^3 patches, spectral interpolation (0) couples the patches, each patch with micro-grid spacing 0.22 (relatively large for visualisation), and with 4^3 points forming each patch.

Figure 20: initial field $u(x, y, z, t)$ at time $t = 0$ of the patch scheme applied to a heterogeneous wave PDE*: Figure 21 plots the computed field at time $t = 6$.*




```
global patches
patches = configPatches3(@heteroWave3,[-pi pi] ...
    ,'periodic' , 5, 0, 0.22, mPeriod+2 ,'EdgyInt',true ...
    ,'hetCoeffs',cHetr);
```

Set a wave initial state using auto-replication of the spatial grid, and as Figure 20 shows. This wave propagates diagonally across space. Concatenate the two u, v-fields to be the two components of the fourth dimension.

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

Integrate in time to $t = 6$ using standard functions. In Matlab `ode15s` would be natural as the patch scheme is naturally stiff, but `ode23` is much quicker (Maclean, Bunder, and Roberts 2021, Fig. 4).

```
disp('Simulate heterogeneous wave u_tt=div[C*grad(u)]')
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@patchSys3,linspace(0,6),uv0(:));
else %disp('octave version is very slow for me')
    lsode_options('absolute tolerance',1e-4);
    lsode_options('relative tolerance',1e-4);
```

```
    [ts,us] = odeOcts(@patchSys3,[0 1 2],uv0(:));
end
```

Animate the computed simulation to end with Figure 21. Use `patchEdgeInt3` to obtain patch-face values in order to most easily reconstruct the array data structure.

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

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

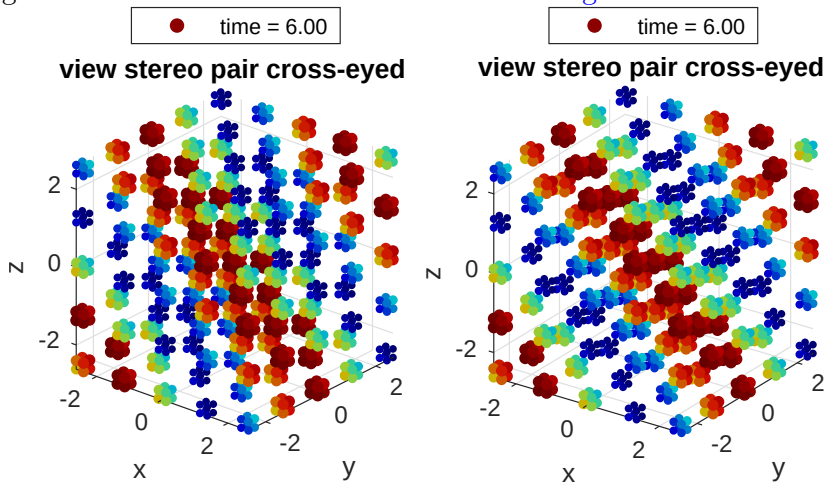
In the scatter plot, these functions `pix()` and `col()` map the u-data values to the size of the dots and to the colour of the dots, respectively.

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

Loop to plot at each and every time step.

```
for i = 1:length(ts)
  uv = patchEdgeInt3(us(i,:));
  u = uv(:,:,:,1,:);
  for p=1:2
    subplot(1,2,p)
    if (i==1)| exist('OCTAVE_VERSION','builtin')
      scat(p) = scatter3(xs(j),ys(j),zs(j),'filled');
      axis equal, caxis(col([0 1])), view(45-5*p,25)
      xlabel('x'), ylabel('y'), zlabel('z')
      title('view stereo pair cross-eyed')
    end % in matlab just update values
    set(scat(p),'CData',col(u(j)) ...
       ,'SizeData',pix((8+xs(j)-ys(j)+zs(j))/6+0*u(j)));
```

Figure 21: field $u(x, y, z, t)$ at time $t = 6$ of the patch scheme applied to the heterogeneous wave PDE *with initial condition in Figure 20.*




```
    legend(['time = ' num2str(ts(i),'%4.2f')],'Location','north')
  end
```

Optionally save the initial condition to graphic file for Figure 18, and optionally save the last plot.

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

Upon finishing execution of the example, exit this function.

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

13.2 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'); % too flexible
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); % not yet implemented
parse(p,fun,Xlim,nPatch,ordCC,dx,nSubP,varargin{:});
```

Set the optional parameters.

```
patches.nEdge = p.Results.nEdge;
if numel(patches.nEdge)==1
    patches.nEdge = repmat(patches.nEdge,1,3);
    end
patches.EdgyInt = p.Results.EdgyInt;
patches.nEnsem = p.Results.nEnsem;
cs = p.Results.hetCoeffs;
patches.parallel = p.Results.parallel;
%patches.nCore = p.Results.nCore;
```

Initially duplicate parameters for three space dimensions as needed.

```
if numel(Xlim)==2,   Xlim = repmat(Xlim,1,3); end
if numel(nPatch)==1, nPatch = repmat(nPatch,1,3); end
if numel(dx)==1,     dx = repmat(dx,1,3); end
if numel(nSubP)==1,  nSubP = repmat(nSubP,1,3); end
```

Check parameters.

```
assert(Xlim(1)<Xlim(2) ...
      ,'first pair of Xlim must be ordered increasing')
assert(Xlim(3)<Xlim(4) ...
      ,'second pair of Xlim must be ordered increasing')
assert(Xlim(5)<Xlim(6) ...
```

```
538       ,'third pair of Xlim must be ordered increasing')
539 assert((mod(ordCC,2)==0)|all(patches.nEdge==1) ...
540       ,'Cannot yet have nEdge>1 and staggered patch grids')
541 assert(all(3*patches.nEdge<=nSubP) ...
542       ,'too many edge values requested')
543 assert(all(rem(nSubP,patches.nEdge)==0) ...
544       ,'nSubP must be integer multiple of nEdge')
545 if ~patches.EdgyInt, assert(all(rem(nSubP./patches.nEdge,2)==1) ...
546       ,'for non-edgyInt, nSubP./nEdge must be odd integer')
547       end
548 if (patches.nEnsem>1)&all(patches.nEdge>1)
549       warning('not yet tested when both nEnsem and nEdge non-one')
550       end
551 %if patches.nCore>1
552 %    warning('nCore>1 not yet tested in this version')
553 %    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` vector.

```
564 if ~isstruct(Dom), pre2023=isnan(Dom);
565 else pre2023=false; end
566 if pre2023, ratio=dx; dx=nan; end
```

Default macroscale conditions are periodic with evenly spaced patches.

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

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

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

We allow different macroscale domain conditions in the different directions. But for the moment do not allow periodic to be mixed with the others (as the interpolation mechanism is different code)—hence why we choose `periodic` be seven characters, whereas the others are eight characters. The different conditions are coded in different rows of `Dom.type`, so we duplicate the string if only one row specified.

```
597 if size(Dom.type,1)==1, Dom.type=repmat(Dom.type,3,1); end
```

Check what is and is not specified, and provide default of Dirichlet boundaries if no `bcOffset` specified when needed. Do so for all three directions independently.

```
patches.periodic=false;
for p=1:3
switch Dom.type(p,:)
case 'periodic'
    patches.periodic=true;
    if isfield(Dom,'bcOffset')
    warning('bcOffset not available for Dom.type = periodic'), end
    msg=' not available for Dom.type = periodic';
    if isfield(Dom,'X'), warning(['X' msg]), end
    if isfield(Dom,'Y'), warning(['Y' msg]), end
    if isfield(Dom,'Z'), warning(['Z' msg]), end
case {'equispace','chebyshev'}
    if ~isfield(Dom,'bcOffset'), Dom.bcOffset=zeros(2,3); end
    % for mixed with usergiven, following should still work
    if numel(Dom.bcOffset)==1
        Dom.bcOffset=repmat(Dom.bcOffset,2,3); end
    if numel(Dom.bcOffset)==3
        Dom.bcOffset=repmat(Dom.bcOffset(:)',2,1); end
    msg=' not available for Dom.type = equispace or chebyshev';
    if (p==1)& isfield(Dom,'X'), warning(['X' msg]), end
    if (p==2)& isfield(Dom,'Y'), warning(['Y' msg]), end
    if (p==3)& isfield(Dom,'Z'), warning(['Z' msg]), end
case 'usergiven'
%    if isfield(Dom,'bcOffset')
%    warning('bcOffset not available for usergiven Dom.type'), end
    msg=' required for Dom.type = usergiven';
    if p==1, assert(isfield(Dom,'X'),['X' msg]), end
    if p==2, assert(isfield(Dom,'Y'),['Y' msg]), end
    if p==3, assert(isfield(Dom,'Z'),['Z' msg]), end
otherwise
    error([Dom.type ' is unknown Dom.type'])
end%switch Dom.type
end%for p
```

13.3 The code to make patches

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

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

Second, store the order of interpolation that is to provide the values for the inter-patch coupling conditions. Spectral coupling is `ordCC` of 0 or (not yet??) -1.

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

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

```
668 patches.stag = mod(ordCC,2);
669 assert(patches.stag==0,'staggered not yet implemented??')
670 ordCC = ordCC+patches.stag;
671 patches.ordCC = ordCC;
```

Check for staggered grid and periodic case.

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

Set the macro-distribution of patches Third, set the centre of the patches in the macroscale grid of patches. Loop over the coordinate directions, setting the distribution into `Q` and finally assigning to array of corresponding direction.

```
694 for q=1:3
695 qq=2*q-1;
```

Distribution depends upon `Dom.type`:

```
701 switch Dom.type(q,:)
```

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

```
709 case 'periodic'
710   Q=linspace(Xlim(qq),Xlim(qq+1),nPatch(q)+1);
711   DQ=Q(2)-Q(1);
712   Q=Q(1:nPatch(q))+diff(Q)/2;
713   pEI=patches.EdgyInt;% abbreviation
714   pnE=patches.nEdge(q);% abbreviation
715   if pre2023, dx(q) = ratio(q)*DQ/(nSubP(q)-pnE*(1+pEI))*(2-pEI);
716   else        ratio(q) = dx(q)/DQ*(nSubP(q)-pnE*(1+pEI))/(2-pEI);
717   end
718   patches.ratio=ratio;
```

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

```
727 case 'equispace'
728   Q=linspace(Xlim(qq)+((nSubP(q)-1)/2-Dom.bcOffset(qq))*dx(q) ...
729           ,Xlim(qq+1)-((nSubP(q)-1)/2-Dom.bcOffset(qq+1))*dx(q) ...
730           ,nPatch(q));
731   DQ=diff(Q(1:2));
732   width=(1+patches.EdgyInt)/2*(nSubP(q)-1-patches.EdgyInt)*dx;
733   if DQ<width*0.999999
734      warning('too many equispace patches (double overlapping)')
735      end
```

The Chebyshev case is spaced according to the Chebyshev distribution in order to reduce macro-interpolation errors, $Q_i \propto -\cos(i\pi/N)$, but with the extreme edges aligned with the spatial domain boundaries, modified by the offset, and modified by possible 'boundary layers'. [22]

```
752 case 'chebyshev'
753   halfWidth=dx(q)*(nSubP(q)-1)/2;
754   Q1 = Xlim(1)+halfWidth-Dom.bcOffset(qq)*dx(q);
755   Q2 = Xlim(2)-halfWidth+Dom.bcOffset(qq+1)*dx(q);
756 %  Q = (Q1+Q2)/2-(Q2-Q1)/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`.

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

```
765   pEI=patches.EdgyInt; % abbreviation
766   pnE=patches.nEdge(q);% abbreviation
767   width=(1+pEI)/2*(nSubP(q)-pnE*(1+pEI))*dx(q);
768   for b=0:2:nPatch(q)-2
769     DQmin=(Q2-Q1-b*width)/2*( 1-cos(pi/(nPatch(q)-b-1)) );
770     if DQmin>width, break, end
771   end%for
772   if DQmin<width*0.999999
773     warning('too many Chebyshev patches (mid-domain overlap)')
774   end%if
```

Assign the centre-patch coordinates.

```
780   Q =[ Q1+(0:b/2-1)*width ...
781        (Q1+Q2)/2-(Q2-Q1-b*width)/2*cos(linspace(0,pi,nPatch(q)-b)) ..
782        Q2+(1-b/2:0)*width ];
```

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

```
791 case 'usergiven'
792   if q==1, Q = reshape(Dom.X,1,[]); end
793   if q==2, Q = reshape(Dom.Y,1,[]); end
794   if q==3, Q = reshape(Dom.Z,1,[]); end
795 end%switch Dom.type
```

Assign Q-coordinates to the correct spatial direction. At this stage they are all rows.

```
802 if q==1, X=Q; end
803 if q==2, Y=Q; end
804 if q==3, Z=Q; end
805 end%for q
```

Construct the micro-grids Fourth, construct the microscale grid in each patch, centred about the given mid-points `X,Y,Z`. Reshape the grid to be 8D to suit dimensions (micro,Vars,Ens,macro).

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

```
ys = dx(2)*( (1:nSubP(2))-mean(1:nSubP(2)) );
patches.y = reshape( ys'+Y ...
                    ,1,nSubP(2),1,1,1,1,nPatch(2),1);
zs = dx(3)*( (1:nSubP(3))-mean(1:nSubP(3)) );
patches.z = reshape( zs'+Z ...
                    ,1,1,nSubP(3),1,1,1,1,nPatch(3));
```

Pre-compute weights for macro-periodic In the case of macro-periodicity, precompute the weightings to interpolate field values for coupling. [23]

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

13.4 Set ensemble inter-patch communication

For `EdgyInt` or centre interpolation respectively,

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

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

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

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

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

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

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

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

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

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

```
913     patches.nEnsem = mx*my*mz;
914     patches.cs = nan(nx-1,ny-1,nz-1,nc,mx,my,mz);
915     for k = 1:mz
916       ks = (k:k+nz-2);
917       for j = 1:my
918         js = (j:j+ny-2);
919         for i = 1:mx
920           is = (i:i+nx-2);
921           patches.cs(:,:,:,:,i,j,k) = cs(is,js,ks,:);
922         end
923       end
924     end
925     patches.cs = reshape(patches.cs,nx-1,ny-1,nz-1,nc,[]);
```

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

```
935     mmx=(0:mx-1)'; mmy=0:my-1; mmz=shiftdim(0:mz-1,-1);
936     le = mod(mmx+mod(nx-2,mx),mx)+1;
937     patches.le = reshape(  le+mx*(mmy+my*mmz)  ,[],1);
```

```
    ri = mod(mmx-mod(nx-2,mx),mx)+1;
    patches.ri = reshape(  ri+mx*(mmy+my*mmz)  ,[],1);
    bo = mod(mmy+mod(ny-2,my),my)+1;
    patches.bo = reshape( 1+mmx+mx*(bo-1+my*mmz) ,[],1);
    to = mod(mmy-mod(ny-2,my),my)+1;
    patches.to = reshape( 1+mmx+mx*(to-1+my*mmz) ,[],1);
    ba = mod(mmz+mod(nz-2,mz),mz)+1;
    patches.ba = reshape( 1+mmx+mx*(mmy+my*(ba-1)) ,[],1);
    fr = mod(mmz-mod(nz-2,mz),mz)+1;
    patches.fr = reshape( 1+mmx+mx*(mmy+my*(fr-1)) ,[],1);
```

Issue warning if the ensemble is likely to be affected by lack of scale separation. [24]

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

End the two if-statements.

```
  end%if-else nEnsem>1
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.*[25]

```
if patches.parallel
  spmd
```

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

[24] **ToDo:** Need to justify this and the arbitrary threshold more carefully??

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

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

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

```
  switch pari
    case 1, patches.x=codistributed(patches.x,patches.codist);
    case 2, patches.y=codistributed(patches.y,patches.codist);
    case 3, patches.z=codistributed(patches.z,patches.codist);
  otherwise
    error('should never have bad index for parallel distribution')
  end%switch
  end%spmd
```

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

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

Fin

```
end% function
```

14 patchSys3(): interface 3D space to time integrators

To simulate in time with 3D spatial patches we often need to interface a users time derivative function with time integration routines such as `ode23` or `PIRK2`. This function provides an interface. Communicate patch-design variables (Section 13) 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.

```
function dudt = patchSys3(t,u,patches,varargin)
if nargin<3, global patches, end
```

Input

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

Output

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

Sets the edge-face values from macroscale interpolation of centre-patch values, and if necessary, reshapes the fields `u` as a 8D-array. Section 12 describes `patchEdgeInt3()`.

```
sizeu = size(u);
u = patchEdgeInt3(u,patches);
```

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

```
dudt = patches.fun(t,u,patches,varargin{:});
m = patches.nEdge(1);
dudt([1:m end-m+1:end],:,:,:) = 0;
m = patches.nEdge(2);
dudt(:,[1:m end-m+1:end],:,:) = 0;
m = patches.nEdge(3);
dudt(:,:,[1:m end-m+1:end],:) = 0;
dudt = reshape(dudt,sizeu);
```

Fin.

15 New interpolation tests

15.1 `patchEdgeInt1test`: test the 1D patch coupling

A script to test the spectral and finite-order polynomial interpolation of function `patchEdgeInt1()`. Tests one or several variables, normal and staggered grids, and also tests centre and edge interpolation. But does not yet test core averaging, nor divided differences on staggered, etc.

Start by establishing global data struct, and the number of realisations of cases.

```
clear all, close all
global patches
nRealise = 20
```

15.1.1 Check divided difference interpolation

```
fprintf('\n\n**** Check divided difference interpolation\n')
pause(1)
```

But not yet implemented staggered grid version?? Check over various types and orders of interpolation, numbers of patches, random domain lengths, random ratios, and randomised distribution of patches. (The `@sin` is a dummy.)

```
for iReal=1:nRealise
    nEdge = randi(3)% =1,2, or 3
    edgyInt = rand<0.5
    nSubP = nEdge*( (2-edgyInt)*randi(2)+1+edgyInt )
    ordCC = 2*randi(4)
    nPatch = ordCC+randi([2 4])
    Domain=5*[-rand rand]
    dx=rand*diff(Domain)/nPatch/nSubP
    configPatches1(@sin,Domain,'equispace',nPatch,ordCC,dx,nSubP ...
        ,'EdgyInt',edgyInt,'nEdge',nEdge);
    patches.intTest = true;
```

Displace patches to a random non-uniform spacing.

```
H = diff(patches.x(1,:,:,1:2));
patches.x = patches.x+0.8*H*(rand(1,1,1,nPatch)-0.5);
%H = squeeze( diff(patches.x(1,:,:,:)) )% for information only
```

Check multiple fields simultaneously Set profiles to be various powers of x, `ps`, and store as different 'variables' at each point.

```
    ps=1:ordCC
    cs=randn(size(ps));
    u0=patches.x.^ps.*cs+randn;
```

Copy data, and set edges to `inf` so we can be certain that interpolation is computing the required edge values.

```
    u=u0;  u([1:nEdge  end-nEdge+1:end],:)=inf;
```

Then evaluate the interpolation and squeeze the singleton dimension of an 'ensemble'.

```
    ui=patchEdgeInt1(u(:));
    ui=squeeze(ui);
```

All patches should have zero error: but need to either in `patchEdgeInt1` comment out `NaN` assignment of boundary values, or not test the two extreme patches here, or add code to omit NaNs here. High-order interpolation seems to be more affected by round-off so relax error size.

```
    j=1:nPatch;
    iError=ui(:,:,j)-u0(:,:,j);
    hist(log10(abs(iError(abs(iError)>0))),-17:-9)
    xlabel('log10 iError'),pause(0.3)%??
    normError=norm(iError(:))
    assert(normError<1e-13*4^ordCC ...
    ,'failed divided difference interpolation')
```

End the for-loop over random parameters.

```
end%for iReal
fprintf('\n\nPassed all divided difference interpolation\n')
```

15.1.2 Test standard spectral interpolation

```
fprintf('\n\n**** Test standard spectral interpolation\n')
pause(1)
```

Test over random numbers of patches, random domain lengths, random microscale spacing, random choice of `edgyInt`. Say do fifteen realisations.

```
for iReal=1:nRealise
    nEdge=randi(3)% =1,2, or 3
    edgyInt = rand<0.5
    nSubP = nEdge*( (2-edgyInt)*randi(2)+1+edgyInt )
    nPatch=randi([5 10])
    Len=10*rand
    dx=0.5*rand*Len/nPatch/nSubP
    configPatches1(@sin,[0 Len],'periodic',nPatch,0,dx,nSubP ...
        ,'EdgyInt',edgyInt,'nEdge',nEdge); % random Edgy or not
    if mod(nPatch,2)==0, fprintf('\nAvoiding highest wavenumber\n'),
    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);
  u=u0;  u([1:nEdge  end-nEdge+1:end],:)=nan;
  ui=patchEdgeInt1(u(:));
  normError=rms(ui(:)-u0(:));
  if abs(normError)>5e-14
```

```
166       normError=normError, k=k
167       error(['failed single var interpolation k=' num2str(k)])
168     end
169   end
```

Test multiple fields Use this to measure some of the errors in order to omit singleton dimensions,

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

Set a profile, and evaluate the interpolation. For the case of the highest wavenumber, squash the error when the centre-patch values are all zero by multiplying by result norm. Not yet working for edgy interpolation.

```
188 for k=1:(nPatch-1)/2 % not checking the highest wavenumber
189   u0=sin(k*patches.x*2*pi/Len);
190   v0=cos(k*patches.x*2*pi/Len);
191   uvi=patchEdgeInt1( reshape([u0 v0],[],1) );
192   normuError=normDiff(uvi(:,1,:,:),u0);
193   normvError=normDiff(uvi(:,2,:,:),v0);
194   if abs(normuError)+abs(normvError)>2e-13
195     normuError=normuError, normvError=normvError
196     error(['failed double field interpolation k=' num2str(k)])
197   end
198 end
```

End the for-loop over various geometries.

```
205 end
206 fprintf('\nPassed standard spectral interpolation tests\n')
```

15.1.3 Now test spectral interpolation on staggered grid

```
221 fprintf('\n\n**** Test spectral interpolation on staggered\n')
222 pause(1)
```

Must have even number of patches for a staggered grid. Have not yet implemented multiple edge values for a staggered grid as I am uncertain whether it makes any sense—certainly this test fails anyway.

```
for iReal=1:nRealise
    nEdge = 1 % required
    edgyInt = rand<0.5
    nPatch=2*randi([3 10])
    nSubP=7 % of form 4*N-1
    Len=10*rand
    dx=0.5*rand*Len/nPatch/nSubP
    configPatches1(@simpleWavepde,[0 Len],'periodic' ...
        ,nPatch,-1,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);
    if mod(nPatch,4)==0, fprintf('\nAvoiding highest wavenumber\n'),
    kMax=floor((nPatch/2-1)/2)
```

Identify which microscale grid points are h or u values.

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

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

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

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

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

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

```
fprintf('Staggered: Two field-pairs test.\n')
x0=patches.x((nSubP+1)/2,1);
patches.x=patches.x-x0;
oddP=1:2:nPatch; evnP=2:2:nPatch;
for k=1:kMax
  U0=nan(nSubP,1,1,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);
  U=U0; U([1:nEdge  end-nEdge+1:end],:)=nan;
  V0(hPts)=rand*cos(k*patches.x(hPts)*2*pi/Len);
  V0(uPts)=rand*cos(k*patches.x(uPts)*2*pi/Len);
  V=V0; V([1:nEdge  end-nEdge+1:end],:)=nan;
  UVi=patchEdgeInt1([U0 V0]);
  normuError=[normDiff(UVi(:,1,:,oddP),U0(:,:,:,oddP),U0(:,:,:,evnP))
      normDiff(UVi(:,1,:,evnP),U0(:,:,:,evnP),U0(:,:,:,oddP))]';
  normvError=[normDiff(UVi(:,2,:,oddP),V0(:,:,:,oddP),V0(:,:,:,evnP))
      normDiff(UVi(:,2,:,evnP),V0(:,:,:,evnP),V0(:,:,:,oddP))]';
  if norm(normuError)+norm(normvError)>2e-13
    normuError=normuError, normvError=normvError
    patches=patches
    error(['staggered: failed double field interpolation k=' num2str(
  end
end
```

End for-loop over patches

```
end
```

15.1.4 Check standard finite width interpolation

```
fprintf('\n\n**** Check standard finite width interpolation\n')
pause(1)
```

Check over various types and orders of interpolation, numbers of patches, random domain lengths and random ratios. (The `@sin` is a dummy.)

```
for iReal=1:nRealise
    nEdge=randi(3)% =1,2, or 3
    edgyInt = rand<0.5
    nSubP = nEdge*( (2-edgyInt)*randi(2)+1+edgyInt )
    ordCC = 2*randi(4)
    nPatch = ordCC+randi([2 4])
    Domain=5*[-rand rand]
    dx=0.5*rand*diff(Domain)/nPatch/nSubP
    configPatches1(@sin,Domain,'periodic',nPatch,ordCC,dx,nSubP ...
        ,'EdgyInt',edgyInt,'nEdge',nEdge);
```

Check multiple fields simultaneously Set profiles to be various powers of x, `ps`, and store as different 'variables' at each point.

```
    ps=1:ordCC
    cs=randn(size(ps));
    u0=patches.x.^ps.*cs+randn;
```

Copy data, and set edges to NaN so we can be certain that interpolation is computing the required edge values.

```
    u=u0;  u([1:nEdge  end-nEdge+1:end],:)=nan;
```

Then evaluate the interpolation and squeeze the singleton dimension of an 'ensemble'.

```
    ui=patchEdgeInt1(u(:));
    ui=squeeze(ui);
```

The interior patches should have zero error.

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

End the for-loops over various parameters.

```
end%for iReal
fprintf('\nPassed all standard polynomial interpolation\n')
```

15.1.5 Now test finite width interpolation on staggered grid

```
413 fprintf('\n\n**** Check finite width staggered\n')
414 pause(1)
```

Must have even number of patches for a staggered grid.

```
420 for iReal=1:nRealise
421     nEdge = 1 % required for now
422     edgyInt = rand<0.5
423     nPatch=2*randi([3 10])
424     nSubP=3; % of form 4*N-1
425     Len=10*rand
426     dx=0.5*rand*Len/nPatch/nSubP
427     configPatches1(@simpleWavepde,[0 Len],'periodic' ...
428         ,nPatch,-1,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);
429     kMax=floor((nPatch/2-1)/2)
```

Identify which microscale grid points are h or u values.

```
436     uPts=mod( (1:nSubP)'+(1:nPatch) ,2);
437     hPts=find(1-uPts);
438     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.

```
446 fprintf('\nSingle field-pair test.\n')
447 for k=-kMax:kMax
448   U0=nan(nSubP,nPatch);
449   U0(hPts)=rand*exp(+1i*k*patches.x(hPts)*2*pi/Len);
450   U0(uPts)=rand*exp(-1i*k*patches.x(uPts)*2*pi/Len);
451   Ui=squeeze(patchEdgeInt1(U0(:)));
452   normError=norm(Ui-U0);
453   if abs(normError)>5e-14
454     normError=normError
455     error(['failed single sys interpolation k=' num2str(k)])
456   end
457 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.

```
i0=(nSubP+1)/2; % centre-patch index
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,1,1,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]);
  Ui=squeeze(UVi(:,1,1,:));
  Vi=squeeze(UVi(:,2,1,:));
  normuError=norm(Ui(:,1:2:nPatch)-U0(:,1:2:nPatch))*norm(U0(i0,2:2:n
            +norm(Ui(:,2:2:nPatch)-U0(:,2:2:nPatch))*norm(U0(i0,1:2:n
  normvError=norm(Vi(:,1:2:nPatch)-V0(:,1:2:nPatch))*norm(V0(i0,2:2:n
            +norm(Vi(:,2:2:nPatch)-V0(:,2:2:nPatch))*norm(V0(i0,1:2:n
  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 the realisations

```
end
```

15.1.6 Finish

If no error messages, then all OK.

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

15.2 `patchEdgeInt2test`: tests 2D patch coupling

A script to test the spectral, finite-order, and divided difference, polynomial interpolation of function `patchEdgeInt2()`. Tests one or several variables,

normal grids, and also tests centre and edge interpolation. But does not yet test staggered grids, core averaging, etc as they are not yet implemented.

Start by establishing global data struct for the range of various cases. Choose a number of realisations for every type.

```
clear all, close all
global patches
nRealise = 20
```

15.2.1 Check divided difference interpolation

```
fprintf('\n\n**** Check divided difference interpolation\n')
pause(1)
```

Check over various types and orders of interpolation, numbers of patches, random domain lengths, random ratios, and randomised distribution of patches. (The `@sin` is a dummy.)

```
maxErrors=[];
for realisation = 1:nRealise
    nEdge = randi(3,1,2)% =1,2, or 3
    edgyInt = (rand>0.5)
    Lx = 1+3*rand; Ly = 1+3*rand;
    xyLim = [0 Lx 0 Ly]-[rand*[1 1] rand*[1 1]]
    nSubP = nEdge.*( (2-edgyInt)*randi(3,1,2)+1+edgyInt )
    ordCC = 2*randi(4)
    nPatch = ordCC+randi(4,1,2)
    dx = [Lx Ly]./nPatch./nSubP.*rand(1,2)/2
    configPatches2(@sin,xyLim,'equispace',nPatch,ordCC ...
        ,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);
```

Second, displace patches to a random non-uniform spacing.

```
Hx = diff(patches.x(1,1,:,:,1:2,1));
patches.x = patches.x+0.8*Hx*(rand(1,1,1,1,nPatch(1),1)-0.5);
Hx = squeeze( diff(patches.x(1,1,:,:,:,1)) );% for information only
Hy = diff(patches.y(1,1,:,:,:,1:2));
patches.y = patches.y+0.8*Hy*(rand(1,1,1,1,1,nPatch(2))-0.5);
Hy = squeeze( diff(patches.y(1,1,:,:,1,:)) );% for information only
```

Check multiple fields simultaneously Set profiles to be various powers of x and y, `ps` and `qs`, and store as different 'variables' at each point. First, limit the order of test polynomials by the order of interpolation and by the number of patches.

```
    ox=min(ordCC,nPatch(1)-1);
    oy=min(ordCC,nPatch(2)-1);
    [ps,qs]=ndgrid(0:ox,0:oy);
    ps=reshape(ps,1,1,[]);
    qs=reshape(qs,1,1,[]);
    cs=2*rand(size(ps))-1;
    u0=cs.*patches.x.^ps.*patches.y.^qs;
```

Then evaluate the interpolation, setting edges to `inf` for error checking.

```
    u=u0;
    u([1:nEdge(1)  end-nEdge(1)+1:end],:,:)=inf;
    u(:,[1:nEdge(2)  end-nEdge(2)+1:end],:)=inf;
    ui=patchEdgeInt2(u(:));
```

All patches should have zero error: but need to either in `patchEdgeInt2` comment out `NaN` assignment of boundary values, or not test the two extreme patches here, or add code to omit NaNs here. High-order interpolation seems to be more affected by round-off so relax error size.

```
    error = ui-u0;
    hist(log10(abs(error(abs(error)>1e-20))),-20:-7)
    xlabel('log10 error'), pause(0.3)%??
    maxError=max(abs(error(:)))
    maxErrors=[maxErrors maxError];
    assert(maxError<3e-12*4^ordCC ...
    ,'failed divided difference interpolation')
    disp('*** This divided difference test passed')
```

End the for-loops over various parameters.

```
end% for realisation
maxMaxErrorDividedDiffs = max(maxErrors)
disp('***** Passed all divided difference interpolation')
pause(1)
```

15.2.2 Test standard spectral interpolation

```
fprintf('\n\n**** Test standard spectral interpolation\n')
pause(1)
```

Test over various numbers of patches, random domain lengths and random ratios. Try realisations of random tests.

```
for realisation=1:nRealise
```

Choose and configure random sized domains, random sub-patch resolution, random size-ratios, random number of periodic-patches, randomly edge or mid-patch interpolation.

```
nEdge=randi(3,1,2)% =1,2, or 3
edgyInt = (rand>0.5)
Lx = 1+3*rand, Ly = 1+3*rand
xyLim = [0 Lx 0 Ly]-[rand*[1 1] rand*[1 1]]
nSubP = nEdge.*( (2-edgyInt)*randi(3,1,2)+1+edgyInt )
nPatch = randi([3 6],1,2)
dx = [Lx Ly]./nPatch./nSubP.*rand(1,2)/2
configPatches2(@sin,xyLim,'periodic',nPatch,0 ...
    ,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);
```

Choose a random number of fields, then generate trigonometric shape with random wavenumber and random phase shift. But if an even number of patches in either direction, then do not test the highest wavenumber because of aliasing problem.

```
nV=randi(3)
[nx,Nx]=size(squeeze(patches.x));
[ny,Ny]=size(squeeze(patches.y));
u0=nan(nx,ny,nV,1,Nx,Ny);
for iV=1:nV
  kx=randi([0 floor((nPatch(1)-1)/2)])
  ky=randi([0 floor((nPatch(2)-1)/2)])
  phix=pi*rand*(2*kx~=nPatch(1))
  phiy=pi*rand*(2*ky~=nPatch(2))
  % generate 6D array via auto-replication
  u0(:,:,iV,1,:,:)=sin(2*pi*kx*patches.x/Lx+phix) ...
                  .*sin(2*pi*ky*patches.y/Ly+phiy);
end
```

Copy and nan the edges, then interpolate

```
u=u0;
u([1:nEdge(1)  end-nEdge(1)+1:end],:,:)=nan;
u(:,[1:nEdge(2)  end-nEdge(2)+1:end],:)=nan;
u=patchEdgeInt2(u(:));
```

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

```
error = u-u0;
assert(all(~isnan(error(:))),'found nans in the error!')
hist(log10(abs(error(abs(error)>1e-20))),-20:-7)
xlabel('log10 error'), pause(0.3)%??
normError=norm(error(:))
assert(normError<1e-12, '2D spectral interpolation failed')
disp('*** This spectral test passed')
```

End the for-loop over realisations

```
end
disp('***** All the spectral tests passed')
pause(1)
```

15.2.3 Check polynomial finite width interpolation

Check over various types and orders of interpolation, numbers of patches, random domain lengths and random ratios. (The `@sin` is a dummy.)

```
for realisations=1:nRealise
    nEdge = randi(3,1,2)% =1,2, or 3
    edgyInt = (rand>0.5)
    nSubP = nEdge.*( (2-edgyInt)*randi(3,1,2)+1+edgyInt )
    ordCC = 2*randi(4)
    nPatch = ordCC+randi(4,1,2)
    xyLim=5*[-rand(1,2); rand(1,2)]
    dx = diff(xyLim)./nPatch./nSubP.*rand(1,2)/2
    configPatches2(@sin,xyLim,'periodic',nPatch,ordCC ...
        ,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);
```

Check multiple fields simultaneously Set profiles to be various powers of x, `ps`, and store as different 'variables' at each point.

```
252     [ps,qs]=meshgrid(0:ordCC);
253     ps=reshape(ps,1,1,[]); qs=reshape(qs,1,1,[]);
254     cs=2*rand(size(ps))-1;
255     u0=cs.*patches.x.^ps.*patches.y.^qs;
```

Then evaluate the interpolation.

```
261     ui=patchEdgeInt2(u0(:));
```

The interior patches should have zero error. Appear to need error tolerance of 10^{-8} because of the size of the domain and the high order of interpolation.

```
269     I=ordCC/2+1:nPatch(1)-ordCC/2;
270     J=ordCC/2+1:nPatch(2)-ordCC/2;
271     error=ui(:,:,:,:,I,J)-u0(:,:,:,:,I,J);
272     assert(all(~isnan(error(:))),'found nans in the error!')
273     hist(log10(abs(error(abs(error)>1e-20))),-20:-7)
274     xlabel('log10 error'), pause(0.3)%??
275     normError=norm(error(:))
276     assert(normError<5e-9 ...
277     ,'failed finite stencil interpolation')
278     disp('*** This finite stencil test passed')
```

End the for-loops over various parameters.

```
285 end %for realisations
286 disp('***** Passed all standard polynomial interpolation')
```

15.2.4 Finished

If no error messages, then all OK.

```
300 disp('******* All the interpolation tests successful')
```

15.3 `patchEdgeInt3test`: tests 3D patch coupling

A script to test the spectral, finite-order, and divided difference, polynomial interpolation of function `patchEdgeInt3()`. Tests one or several variables, normal grids, and also tests centre and edge interpolation. But does not yet test staggered grids, core averaging, etc as they are not yet implemented.

Start by establishing global data struct for the range of various cases. Choose a number of realisations for every type, but beware that some realisations take several minutes.

```
clear all, close all
global patches
nRealise = 10
```

15.3.1 Check divided difference interpolation

```
fprintf('\n\n**** Check divided difference interpolation\n')
pause(1)
```

Check over various types and orders of interpolation, numbers of patches, random domain lengths, random ratios, and randomised distribution of patches. (The `@sin` is a dummy.)

```
maxErrors=[];
for realisation = 1:nRealise
    nEdge = randi(3,1,3)% =1,2, or 3
    edgyInt = (rand>0.5)
    Lx = 1+3*rand; Ly = 1+3*rand; Lz = 1+3*rand;
    xyzLim = [0 Lx 0 Ly 0 Lz]-[rand*[1 1] rand*[1 1] rand*[1 1]]
    nSubP = nEdge.*( (2-edgyInt)*randi(2,1,3)+1+edgyInt )
    ordCC = 2*randi(3)
    nPatch = ordCC+randi(4,1,3)
    dx = [Lx Ly Lz]./nPatch./nSubP.*rand(1,3)/2
    configPatches3(@sin,xyzLim,'equispace',nPatch,ordCC ...
        ,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);
```

Second, displace patches to a random non-uniform spacing.

```
Hx = diff(patches.x(1,1,1,:,:,1:2,1,1));
patches.x = patches.x+0.8*Hx*(rand(1,1,1,1,1,nPatch(1),1,1)-0.5);
Hx = squeeze( diff(patches.x(1,1,1,:,:,:,1,1)) )';% for information o
Hy = diff(patches.y(1,1,1,:,:,1,1:2,1));
patches.y = patches.y+0.8*Hy*(rand(1,1,1,1,1,1,nPatch(2),1)-0.5);
Hy = squeeze( diff(patches.y(1,1,1,:,:,1,:,1)) )';% for information o
Hz = diff(patches.z(1,1,1,:,:,1,1,1:2));
patches.z = patches.z+0.8*Hz*(rand(1,1,1,1,1,1,1,1,nPatch(3))-0.5);
Hz = squeeze( diff(patches.z(1,1,1,:,:,1,1,:)) )';% for information o
```

Check multiple fields simultaneously Set profiles to be various powers of x, y and z, `ps`, `qs` and `rs`, and store as different 'variables' at each point. First, limit the order of test polynomials by the order of interpolation and by the number of patches.

```
    ox=min(ordCC,nPatch(1)-1);
    oy=min(ordCC,nPatch(2)-1);
    oz=min(ordCC,nPatch(3)-1);
    [ps,qs,rs]=ndgrid(0:ox,0:oy,0:oz);
    ps=reshape(ps,1,1,1,[]);
    qs=reshape(qs,1,1,1,[]);
    rs=reshape(rs,1,1,1,[]);
    cs=2*rand(size(ps))-1;
    u0=cs.*patches.x.^ps.*patches.y.^qs.*patches.z.^rs;
```

Then evaluate the interpolation, setting faces to `inf` for error checking.

```
    u=u0;
    u([1:nEdge(1)  end-nEdge(1)+1:end],:,:,:)=inf;
    u(:,[1:nEdge(2)  end-nEdge(2)+1:end],:,:)=inf;
    u(:,:,[1:nEdge(3)  end-nEdge(3)+1:end],:)=inf;
    ui=patchEdgeInt3(u(:));
```

All patches should have zero error: but need to either in `patchEdgeInt3` comment out `NaN` assignment of boundary values, or not test the two extreme patches here, or add code to omit NaNs here. High-order interpolation seems to be more affected by round-off so relax error size.

```
    error = ui-u0;
    hist(log10(abs(error(abs(error)>1e-20))),-20:-6)
    xlabel('log10 error'), pause(0.3)%??
    maxError=max(abs(error(:)))
    maxErrors=[maxErrors maxError];
    assert(maxError<1e-10*4^ordCC ...
    ,'failed divided difference interpolation')
    disp('*** This divided difference test passed')
```

End the for-loops over various parameters.

```
end% for realisation
maxMaxErrorDividedDiffs = max(maxErrors)
disp('***** Passed all divided difference interpolation')
pause(1)
```

15.3.2 Test standard spectral interpolation

```
fprintf('\n\n**** Test standard spectral interpolation\n')
pause(1)
```

Test over various numbers of patches, random domain lengths and random ratios. Try realisations of random tests.

```
for realisation=1:nRealise
```

Choose and configure random sized domains, random sub-patch resolution, random size-ratios, random number of periodic-patches, randomly edge or mid-patch interpolation.

```
nEdge=randi(3,1,3)% =1,2, or 3
edgyInt = (rand>0.5)
Lx = 1+3*rand, Ly = 1+3*rand, Lz = 1+3*rand
xyzLim = [0 Lx 0 Ly 0 Lz]-[rand*[1 1] rand*[1 1] rand*[1 1]]
nSubP = nEdge.*( (2-edgyInt)*randi(3,1,3)+1+edgyInt )
nPatch = randi([3 6],1,3)
dx = [Lx Ly Lz]./nPatch./nSubP.*rand(1,3)/2
configPatches3(@sin,xyzLim,'periodic',nPatch,0 ...
    ,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);
```

Choose a random number of fields, then generate trigonometric shape with random wavenumber and random phase shift. But if an even number of patches in either direction, then do not test the highest wavenumber because of aliasing problem.

```
nV=randi(3)
[nx,Nx]=size(squeeze(patches.x));
[ny,Ny]=size(squeeze(patches.y));
[nz,Nz]=size(squeeze(patches.z));
u0=nan(nx,ny,nz,nV,1,Nx,Ny,Nz);
for iV=1:nV
  ks=floor( rand(1,3).*floor(([Nx Ny Nz]+1)/2) )
  phis = pi*rand(1,3).*(2*ks~=[Nx Ny Nz])
  % generate 8D array via auto-replication
  u0(:,:,:,iV,1,:,:,:)=sin(2*pi*ks(1)*patches.x/Lx+phis(1)) ...
                      .*sin(2*pi*ks(2)*patches.y/Ly+phis(2)) ...
                      .*sin(2*pi*ks(3)*patches.z/Lz+phis(3));
end
```

Copy and `nan` the faces, then interpolate

```
u=u0;
u([1:nEdge(1)  end-nEdge(1)+1:end],:,:,:)=nan;
u(:,[1:nEdge(2)  end-nEdge(2)+1:end],:,:)=nan;
u(:,:,[1:nEdge(3)  end-nEdge(3)+1:end],:)=nan;
u=patchEdgeInt3(u(:));
```

Compute difference, ignoring the nans which should only be in the corners. If there is an error in the interpolation, then abort the script for checking: please record parameter values and inform us.

```
error = u-u0;
assert(all(~isnan(error(:))),'found nans in the error!')
hist(log10(abs(error(abs(error)>1e-20))),-20:-6)
xlabel('log10 error'), pause(0.3)%??
normError=norm(error(:))
assert(normError<1e-10, '3D spectral interpolation failed')
disp('*** This spectral test passed')
```

End the for-loop over realisations

```
end
disp('***** All the spectral tests passed')
pause(1)
```

15.3.3 Check polynomial finite width interpolation

Check over various types and orders of interpolation, numbers of patches, random domain lengths and random ratios. (The `@sin` is a dummy.)

```
for realisations=1:nRealise
    nEdge = randi(3,1,3)% =1,2, or 3
    edgyInt = (rand>0.5)
    nSubP = nEdge.*( (2-edgyInt)*randi(2,1,3)+1+edgyInt )
    ordCC = 2*randi(3)
    nPatch = ordCC+randi(3,1,3)
    xyzLim=5*[-rand(1,3); rand(1,3)]
    dx = diff(xyzLim)./nPatch./nSubP.*rand(1,3)/2
    configPatches3(@sin,xyzLim,'periodic',nPatch,ordCC ...
        ,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);
```

Check multiple fields simultaneously Set profiles to be various powers of x, y, z, namely `ps`, `qs`, `rs`, and store as different 'variables' at each point.

```
    [ps,qs,rs]=meshgrid(0:ordCC);
    ps=reshape(ps,1,1,1,[]);
    qs=reshape(qs,1,1,1,[]);
    rs=reshape(rs,1,1,1,[]);
    cs=2*rand(size(ps))-1;
    u0=cs.*patches.x.^ps.*patches.y.^qs.*patches.z.^rs;
```

Then evaluate the interpolation.

```
    ui=patchEdgeInt3(u0(:));
```

The interior patches should have zero error. Appear to need error tolerance of 10^{-8} because of the size of the domain and the high order of interpolation.

```
    I=ordCC/2+1:nPatch(1)-ordCC/2;
    J=ordCC/2+1:nPatch(2)-ordCC/2;
    K=ordCC/2+1:nPatch(3)-ordCC/2;
    error=ui(:,:,:,:,:,I,J,K)-u0(:,:,:,:,:,I,J,K);
    assert(all(~isnan(error(:))),'found nans in the error!')
    hist(log10(abs(error(abs(error)>1e-20))),-20:-6)
    xlabel('log10 error'), pause(0.3)%??
    normError=norm(error(:))
    assert(normError<5e-8 ...
    ,'failed finite stencil polynomial interpolation')
    disp('*** This finite stencil test passed')
```

End the for-loops over various parameters.

```
end%for realisation
disp('***** Passed all polynomial interpolation tests')
```

15.3.4 Finished

If no error messages, then all OK.

```
disp('******* All types of interpolation tests successful')
```

16 quasiLogAxes(): transforms some axes of a plot to quasi-log

This function rescales some coordinates and labels the axes of the given 2D or 3D plot. The original aim was to effectively show the complex spectrum of multiscale systems such as the patch scheme. The eigenvalues are over a wide range of magnitudes, but are signed. So we use a nonlinear asinh transformation of the axes, and then label the axes with reasonable ticks. The nonlinear rescaling is useful in other scenarios also.

```
21 function quasiLogAxes(handle,xScale,yScale,zScale,cScale)
```

Input

- `handle`: handle to your plot to transform, for example, obtained by `handle=plot(...)`
- `xScale` (optional, default inf): if inf, then no transformation is done in this coordinate. Otherwise, with x denoting every horizontal coordinate, then transforms the plot-data with the asinh() function so that
  - for $|x| \lesssim x_{\text{scale}}$ the x-axis scaling is approximately linear, whereas
  - for $|x| \gtrsim x_{\text{scale}}$ the x-axis scaling is approximately signed-logarithmic.
- `yScale` (optional, default inf): corresponds to `xScale` for the second axis scaling.
- `zScale` (optional, default inf): corresponds to `xScale` for a third axis scaling if it exists.
- `cScale` (optional, default inf): corresponds to `cScale` for a colormap, and colorbar scaling if it exists.
- axis limits (optional): if the axis limits of the plot do not 'fit' the plot data, then we assume you have set the axis limits, in which case your limits are used (each direction considered separately).

Output None, just the transformed plot.

Example If invoked with no arguments, then execute an example.

```
if nargin==0
   % generate some data
   n=99;  fast=(rand(n,1)<0.8);
   z = -rand(n,1).*(1+1e3*fast)+1i*randn(n,1).*(5+1e2*fast);
   % plot data and transform axes
   handle = plot(real(z),imag(z),'.');
   xlabel('real-part'), ylabel('imag-part')
   quasiLogAxes(handle,1,10);
   return
end% example
```

Default values for scaling, `inf` denotes no transformation of that axis.

```
if nargin<5, cScale=inf; end
if nargin<4, zScale=inf; end
if nargin<3, yScale=inf; end
if nargin<2, xScale=inf; end
```

Get current limits of the plot so we can attempt to detect if a user has set some limits that we should keep. And also get the pointer to the axes and to the figure of the plot.

```
xlim0=xlim; ylim0=ylim; zlim0=zlim; clim0=caxis;
theAxes = get(handle(1),'parent');
theFig = get(theAxes,'parent');
```

Find overall factors so the data is nonlinearly mapped to order oneish—so that then pgfplots et al. do not think there is an overall scaling factor on the axes.

```
xFac=1e-99; yFac=xFac; zFac=xFac; cFac=xFac;
for k=1:length(handle)
    if ~isinf(xScale)
    temp = asinh(handle(k).XData/xScale);
    xFac = max(xFac, max(abs(temp(:)),[],'omitnan') );
    end
    if ~isinf(yScale)
    temp = asinh(handle(k).YData/yScale);
    yFac = max(yFac, max(abs(temp(:)),[],'omitnan') );
    end
```

```
115     if ~isinf(zScale)
116     temp = asinh(handle(k).ZData/zScale);
117     zFac = max(zFac, max(abs(temp(:)),[],'omitnan') );
118     end
119     if ~isinf(cScale)
120     temp = asinh(handle(k).CData/cScale);
121     cFac = max(cFac, max(abs(temp(:)),[],'omitnan') );
122     end
123 end%for
124 xFac=9/xFac; yFac=9/yFac; zFac=9/zFac; cFac=9/cFac;
```

Scale the plot data in the plot `handle`. Give an error if it appears that the plot-data has already been transformed. Color data has to be transformed first because usually there is automatic flow from z-data to c-data.

```
134 for k=1:length(handle)
135     assert(~strcmp(handle(k).UserData,'quasiLogAxes'), ...
136        'Replot graph---it appears plot data is already transformed')
137     if ~isinf(cScale)
138     handle(k).CData = cFac*asinh(handle(k).CData/cScale);
139     end
140     if ~isinf(xScale)
141     handle(k).XData = xFac*asinh(handle(k).XData/xScale);
142     end
143     if ~isinf(yScale)
144     handle(k).YData = yFac*asinh(handle(k).YData/yScale);
145     end
146     if ~isinf(zScale)
147     handle(k).ZData = zFac*asinh(handle(k).ZData/zScale);
148     end
149     handle(k).UserData = 'quasiLogAxes';
150 end%for
151 if ~isinf(xScale), xlim0=xFac*asinh(xlim0/xScale); end
152 if ~isinf(yScale), ylim0=yFac*asinh(ylim0/yScale); end
153 if ~isinf(zScale), zlim0=zFac*asinh(zlim0/zScale); end
154 if ~isinf(cScale), clim0=cFac*asinh(clim0/cScale); end
```

Get limits of nonlinearly transformed data, and reset with 4% padding around all margins—crude but serviceable.

```
axis tight;
xlim1=xlim+0.04*diff(xlim)*[-1 1];
ylim1=ylim+0.04*diff(ylim)*[-1 1];
zlim1=zlim+0.04*diff(zlim)*[-1 1];
clim1=caxis+ 0*diff(caxis)*[-1 1];
```

But if the scaled range is too different from the original, then restore the original. Then set the scaled limits.

```
if diff(xlim1)<0.5*diff(xlim0) | diff(xlim1)>2*diff(xlim0)
    xlim1=xlim0; end
if diff(ylim1)<0.5*diff(ylim0) | diff(ylim1)>2*diff(ylim0)
    ylim1=ylim0; end
if diff(zlim1)<0.5*diff(zlim0) | diff(zlim1)>2*diff(zlim0)
    zlim1=zlim0; end
if diff(clim1)<0.5*diff(clim0) | diff(clim1)>2*diff(clim0)
    clim1=clim0; end
xlim(xlim1); ylim(ylim1); zlim(zlim1); caxis(clim1);
```

Tick marks on the axes

```
if ~isinf(xScale)
    tickingQuasiLogAxes(theAxes,'X',xlim1,xScale,xFac)
end%if
if ~isinf(yScale)
    tickingQuasiLogAxes(theAxes,'Y',ylim1,yScale,yFac)
end%if
if ~isinf(zScale)
    tickingQuasiLogAxes(theAxes,'Z',zlim1,zScale,zFac)
end%if
```

But for color, only if we can find a colorbar.

```
if ~isinf(cScale)
  for p=1:numel(theFig.Children)
    ca = theFig.Children(p);
    if class(ca) == "matlab.graphics.illustration.ColorBar"
      tickingQuasiLogAxes(ca,'C',clim1,cScale,cFac)
      break
    end
  end
end%if
```

Turn the grid on by default.

```
grid on
end%function
```

16.1 tickingQuasiLogAxes(): typeset ticks and labels on an axis

```
function tickingQuasiLogAxes(ca,Q,qlim1,qScale,qFac)
```

Input

- ca: pointer to axes/colorbar dataset.
- Q: character, either X,Y,Z,C.
- qlim1: the scaled limits of the axis.
- qScale: the scaling parameter for the axis.
- qFac: the scaling factor for the axis.

Output None, just the ticked and labelled axes.

Get the order of magnitude of the horizontal data.

```
 qmax=max(abs(qlim1));
 qmag=floor(log10(qScale*sinh(qmax/qFac)));
```

Form a range of ticks, geometrically spaced, trim off the small values that would be too dense near zero (omit those within 6% of qmax).

```
 ticks=10.^(qmag+(-7:0));
 j=find(ticks>qScale*sinh(0.06*qmax/qFac));
 nj=length(j);
 if nj<3,     ticks=[1;2;5]*ticks(j);
 elseif nj<5, ticks=[1;3]*ticks(j);
 else         ticks=ticks(j);
 end
 ticks=sort([0;ticks(:);-ticks(:)]);
```

Set the ticks in place according to the transformation.

```
    if Q=='C', p='s'; Q=''; else p=''; end
    set(ca,[Q 'Tick' p],qFac*asinh(ticks/qScale) ...
          ,[Q 'TickLabel' p],cellstr(num2str(ticks,4)))
    if Q=='X', set(ca,[Q 'TickLabelRotation'],40), end
end%function qScaling
```

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 pp. 32, 55, 86).

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. 4, 7, 11, 31, 32, 35, 60, 92).

Bunder, J.E. et al. (2021). "Large-scale simulation of shallow water waves with computation only on small staggered patches". In: *International Journal for Numerical Methods in Fluids* 93.4, pp. 953–977. DOI: 10.1002/fld.4915 (cit. on pp. 55, 86).

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. 16).

Maclean, John, J. E. Bunder, and A. J. Roberts (2021). "A toolbox of Equation-Free functions in Matlab/Octave for efficient system level simulation". In: *Numerical Algorithms* 87, pp. 1729–1748. DOI: 10.1007/s11075-020-01027-z (cit. on pp. 28, 73, 108).

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. 31).

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. 31, 35, 60, 92).

Roberts, A. J., Tony MacKenzie, and Judith Bunder (2014). "A dynamical systems approach to simulating macroscale spatial dynamics in multiple dimensions". In: *J. Engineering Mathematics* 86.1, pp. 175–207. DOI: 10.1007/s10665-013-9653-6 (cit. on pp. 55, 73, 86).

Wikipedia (2022). *Divided differences*. https://en.wikipedia.org/wiki/Divided_differences (visited on 12/28/2022) (cit. on pp. 38, 65, 98).