Equation-free computational homogenisation with various boundaries and various patch spacing

A. J. Roberts[*]

January 26, 2023

Contents

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

Examples

1 `Eckhardt2210eg2`: example of a 1D heterogeneous diffusion by simulation on small patches

Plot an example simulation in time generated by the patch scheme applied to macroscale forced diffusion through a medium with microscale heterogeneity

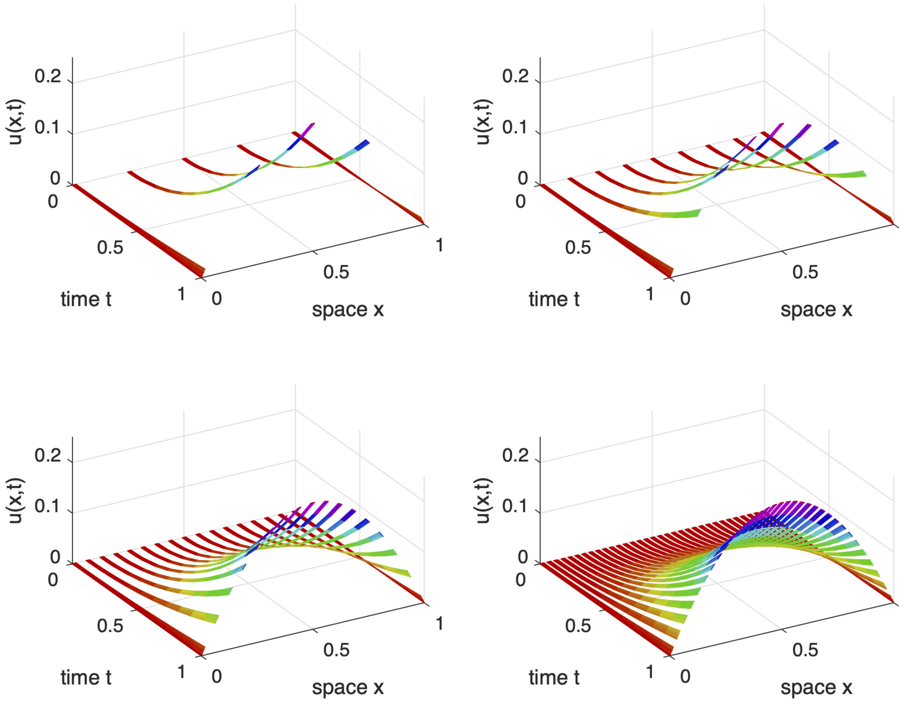


Figure 1: diffusion field $u(x,t)$ of the patch scheme applied to the forced heterogeneous diffusive (1). Simulate for $5, 9, 17, 33$ patches and compare to the full-domain simulation (65 patches, not shown).

in space. This is more-or-less the second example of Eckhardt and Verfürth (2022) [§6.2.1].

Suppose the spatial microscale lattice is at points x_i, with constant spacing dx. With dependent variables $u_i(t)$, simulate the microscale lattice forced diffusion system

$$\frac{\partial u_i}{\partial t} = \frac{1}{dx^2}\delta[a_{i-1/2}\delta u_i] + f_i(t), \tag{1}$$

in terms of the centred difference operator δ. The system has a microscale heterogeneity via the coefficients $a_{i+1/2}$ which has some given known periodicity ϵ.

Here use period $\epsilon = 1/130$ (so that computation completes in seconds). The patch scheme computes only on a fraction of the spatial domain, see Figure 1. Compute *errors* as the maximum difference (at time $t = 1$) between the patch scheme prediction and a full-domain simulation of the same underlying spatial discretisation (which here has space step 0.00128).

patch spacing H	0.25	0.12	0.06	0.03
sine-forcing error	0.0018	0.0009	0.0002	$1.6e{-}5$
parabolic-forcing error	$9.0e{-}9$	$3.7e{-}9$	$0.9e{-}9$	$0.06e{-}9$

The smooth sine-forcing leads to errors that appear due to patch scheme and its interpolation. The parabolic-forcing errors appear to be due to the integration errors of `ode15s` and not at all due to the patch scheme. In comparison, Eckhardt and Verfürth (2022) reported much larger errors in the range 0.001–0.1 (Figure 3).

1.1 Simulate heterogeneous diffusion systems

First establish the microscale heterogeneity has micro-period `mPeriod` on the lattice, and coefficients to match Eckhardt2210.04536 §6.2.1. Set the phase of the heterogeneity so that each patch centre is a point of symmetry of the diffusivity. Then the heterogeneity is repeated to fill each patch.

```
clear all
mPeriod = 6
y = linspace(0,1,mPeriod+1)';
a = 1./(2-cos(2*pi*y(1:mPeriod)))
global microTimePeriod; microTimePeriod=0;
```

Set the spatial period ϵ, via integer $1/\epsilon$, and other parameters.

```
maxLog2Nx = 6
nPeriodsPatch = 2 % any integer
rEpsilon = nPeriodsPatch*(2^maxLog2Nx+1) % up to 200 say
dx = 1/(mPeriod*rEpsilon+1)
nSubP = nPeriodsPatch*mPeriod+2
tol=1e-9;
```

Loop to explore errors on various sized patches.

```
Us=[]; DXs=[]; % for storing results to compare
iPP=0; I=nan;
for log2Nx = 2:maxLog2Nx
nP = 2^log2Nx+1
```

Determine indices of patches that are common in various resolutions

```
if isnan(I), I=1:nP; else I=2*I-1; end
```

Establish the global data struct `patches` for the microscale heterogeneous lattice diffusion system (1) solved on domain $[0, 1]$, with `nP` patches, and say fourth order interpolation to provide the edge-values. Setting `patches.EdgyInt` true means the edge-values come from interpolating the opposite next-to-edge values of the patches (not the mid-patch values).

```
global patches
ordCC = 4
configPatches1(@heteroDiffF,[0 1],'equispaced',nP ...
    ,ordCC,dx,nSubP,'EdgyInt',true,'hetCoeffs',a);
DX = mean(diff(squeeze(patches.x(1,1,1,:))))
DXs=[DXs;DX];
```

Set the forcing coefficients, either the original parabolic, or sinusoidal.

```
if 1 % given forcing
  patches.f1=2*( patches.x-patches.x.^2 );
  patches.f2=2*0.5+0*patches.x;
else% simple sine forcing
  patches.f1=sin(pi*patches.x);
  patches.f2=pi/2*sin(pi*patches.x);
end%if
```

Simulate Set the initial conditions of a simulation to be zero. Integrate to time 1 using standard integrators.

```
u0 = 0*patches.x;
tic
[ts,us] = ode15s(@patchSys1, [0 1], u0(:));
cpuTime=toc
```

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

```
xs = squeeze(patches.x);
us = patchEdgeInt1( permute( reshape(us ...
    ,length(ts),nSubP,1,nP) ,[2 1 3 4]) );
```

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

Plot a space-time surface of field values over the macroscale duration of the simulation.

```
iPP=iPP+1;
if iPP<=4 % only draw four subplots
  figure(1), if iPP==1, clf(), end
  subplot(2,2,iPP)
  mesh(ts,xs(:),uss)
  if iPP==1, uMax=ceil(max(uss(:))*100)/100, end
  view(60,40), colormap(0.8*hsv), zlim([0 uMax])
  xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
  drawnow
end%if
```

At the end of the `log2Nx`-loop, store field at the end-time from centre region of each patch for comparison.

```
i=nPeriodsPatch/2*mPeriod+1+(-mPeriod/2+1:mPeriod/2);
Us(:,:,iPP)=squeeze(us(i,end,I));
Xs=squeeze(patches.x(i,1,1,I));
if iPP>1
   assert(norm(Xs-Xsp)<tol,'sampling error in space')
   end
Xsp=Xs;
end%for log2Nx
ifOurCf2eps(mfilename) %optionally save figure
```

Assess errors by comparing to the full-domain solution

```
DXs=DXs
Uerr=squeeze(max(max(abs(Us-Us(:,:,end)))))
figure(2),clf,
loglog(DXs,Uerr,'o:')
xlabel('H'),ylabel('error')
```

1.2 `heteroDiffF()`: forced heterogeneous diffusion

This function codes the lattice heterogeneous diffusion inside the patches with forcing and with microscale boundary conditions on the macroscale boundaries. Computes the time derivative at each point in the interior of a patch, output in `ut`. The column vector of diffusivities a_i has been stored in struct `patches.cs`, as has the array of forcing coefficients.

```
function ut = heteroDiffF(t,u,patches)
  global microTimePeriod
  % macroscale Dirichlet BCs
  u( 1 ,:,:, 1 )=0; % left-edge of leftmost is zero
  u(end,:,:,end)=0; % right-edge of rightmost is zero
  % interior forced diffusion
  dx = diff(patches.x(2:3));   % space step
  i = 2:size(u,1)-1;     % interior points in a patch
  ut = nan+u;            % preallocate output array
  if microTimePeriod>0 % optional time fluctuations
     at = cos(2*pi*t/microTimePeriod)/30;
  else at=0; end
  ut(i,:,:,:) = diff((patches.cs(:,1,:)+at).*diff(u))/dx^2 ...
      +patches.f2(i,:,:,:)*t^2+patches.f1(i,:,:,:)*t;
end% function
```

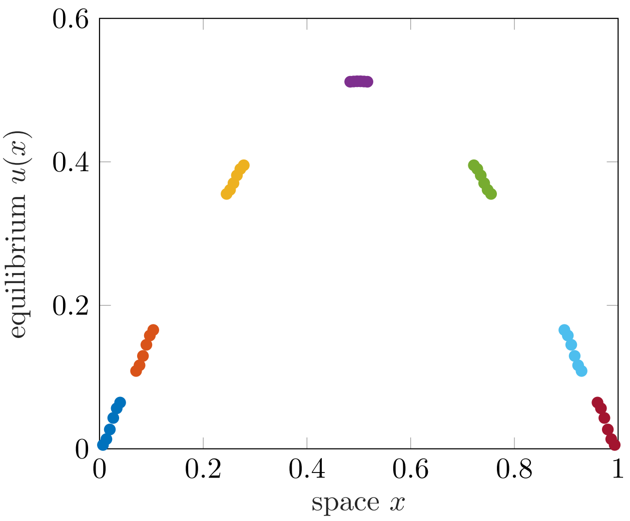
2 EckhartEquilib: find an equilibrium of a 1D heterogeneous diffusion via small patches

Sections 1 and 1.2 describe details of the problem and more details of the following configuration. The aim is to find the equilibrium, Figure 2, of the forced heterogeneous system with a forcing corresponding to that applied at time $t = 1$. Computational efficiency comes from only computing the microscale heterogeneity on small spatially sparse patches, potentially much smaller than those shown in Figure 2.

First configure the patch system Establish the microscale heterogeneity has micro-period `mPeriod` on the lattice, and coefficients to match Eckhardt2210.04536 §6.2.1.

```
mPeriod = 6
y = linspace(0,1,mPeriod+1)';
a = 1./(2-cos(2*pi*y(1:mPeriod)))
global microTimePeriod; microTimePeriod=0;
```

Figure 2: Equilibrium of the heterogeneous diffusion problem with forcing the same as that applied at time $t = 1$, and for relatively large $\epsilon = 0.04$ so we can see the patches. By default this code is for $\epsilon = 0.004$ where the microscale heterogeneity and patches are tiny.



Set the number of patches, the number of periods per patch, and the spatial period ϵ, via integer $1/\epsilon$.

```
nPatch = 7
nPeriodsPatch = 1 % any integer
rEpsilon = 250 % 25 for graphic, up to 2000 say
dx = 1/(mPeriod*rEpsilon+1)
nSubP = nPeriodsPatch*mPeriod+2
```

Establish the global data struct `patches` for the microscale heterogeneous lattice diffusion system (1) solved on domain $[0, 1]$, with Chebyshev-like distribution of patches, and say fourth order interpolation to provide the edge-values. Use 'edgy' interpolation.

```
global patches
ordCC = 4
configPatches1(@heteroDiffF,[0 1],'chebyshev',nPatch ...
    ,ordCC,dx,nSubP,'EdgyInt',true,'hetCoeffs',a);
```

Set the forcing coefficients, either the original parabolic, or sinusoidal. At time $t = 1$ the resultant forcing we actually apply here is simply the sum of the two components.

```
if 0 % given forcing
  patches.f1 = 2*( patches.x-patches.x.^2 );
  patches.f2 = 2*0.5+0*patches.x;
else% simple sine forcing
  patches.f1 = sin(pi*patches.x);
  patches.f2 = pi/2*sin(pi*patches.x);
end%if
```

Find equilibrium with fsolve We seek the equilibrium for the forcing that applies at time $t = 1$ (as if that specific forcing were applying for all time). Execute the function that invokes `fsolve`. For this linear problem, it is computationally quicker using a linear solver, but `fsolve` is quicker in human time, and generalises to nonlinear problems.

```
u = squeeze(execFsolve)
```

Then plot the equilibrium solution (Figure 2).

```
clf, plot(squeeze(patches.x),u,'.')
xlabel('space $x$'),ylabel('equilibrium $u(x)$')
```

Code to execute fsolve We code the function `execFsolve` to execute `fsolve` because easiest if a sub-function that computes the time derivatives has access to variables `u0` and `i`.

```
function [u,normRes] = execFsolve
global patches
```

Start the search from a zero field.

```
u0 = 0*patches.x;
```

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

```
u0([1 end],:,:,:) = nan;
i = find(~isnan(u0));
```

Seek the equilibrium, and report the norm of the residual.

```
[u0(i),res] = fsolve(@duidt,u0(i));
normRes = norm(res)
```

The aim is to zero the time derivatives `duidt` in the following function. First, insert the vector of variables into the patch-array of `u0`. Second, find the time derivatives via the patch scheme, and finally return a vector of those at the patch-internal points.

```
function res = duidt(ui)
  u = u0;   u(i) = ui;
  res = patchSys1(1,u);
  res = res(i);
end%function duidt
end%function execFsolve
```

Fin.

3 `Eckhardt2210eg1`: example of 1D space-time heterogeneous diffusion via computational homogenisation with projective integration and small patches

An example simulation in time generated by projective integration allied with the patch scheme applied to forced diffusion in a medium with microscale heterogeneity in both space and time. This is more-or-less the first example of Eckhardt and Verfürth (2022) [§6.2].

Suppose the spatial microscale lattice is at points x_i, with constant spacing dx. With dependent variables $u_i(t)$, simulate the microscale lattice forced diffusion system

$$\frac{\partial u_i}{\partial t} = \frac{1}{dx^2}\delta[a_{i-1/2}(t)\delta u_i] + f_i(t), \qquad (2)$$

in terms of the centred difference operator δ. The system has a microscale heterogeneity via the coefficients $a_{i+1/2}$ which has given periodicity ϵ in space, and periodicity ϵ^2 in time. Figure 3 shows an example patch simulation.

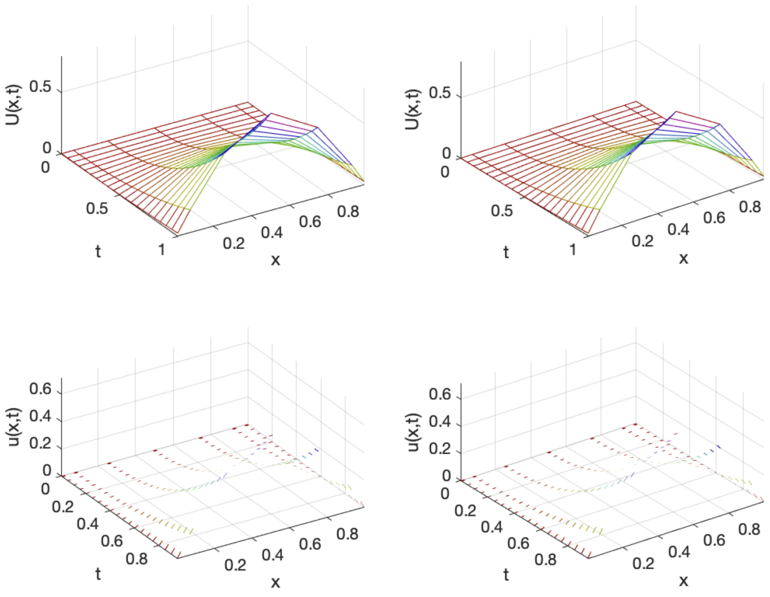
The approximate homogenised PDE is $U_t = A_0 U_{xx} + F$ with $U = 0$ at $x = 0, 1$. Its slowest mode is then $U = \sin(\pi x)e^{-A_0\pi^2 t}$. When $A_0 = 3.3524$ as in Eckhardt then the rate of evolution is about 33 which is relatively fast on the simulation time-scale of $T = 1$. Let's slow down the dynamics by reducing diffusivities by a factor of 30, so effectively $A_0 \approx 0.1$ and $A_0\pi^2 \approx 1$.

Also, in the microscale fluctuations change the time variation to cosine, not its square (because I cannot see the point of squaring it!).

The highest wavenumber mode on the macro-grid of patches, spacing H, is the zig-zag mode on $\dot{U}_i = A_0(U_{I+1} - 2U_I + U_{I-1})/H^2 + F_I$ which evolves like $U_I = (-1)^I e^{-\alpha t}$ for the fastest 'slow rate' of $\alpha = 4A_0^2/H^2$. When $H = 0.2$ and $A_0 \approx 0.1$ this rate is $\alpha \approx 10$.

Here use period $\epsilon = 1/100$ (so that computation completes in seconds, and because we have slowed the dynamics by 30). The patch scheme computes

Figure 3: diffusion field $u(x,t)$ of the patch scheme applied to the forced space-time heterogeneous diffusive (2). Simulate for seven patches (with a 'Chebyshev' distribution): the top stereo pair is a mesh plot of a macroscale value at the centre of each spatial patch at each projective integration time-step; the bottom stereo pair shows the corresponding tiny space-time patches in which microscale computations were carried out.



only on a fraction of the spatial domain. Projective integration computes only on a fraction of the time domain determined by the 'burst length'.

3.1 Simulate heterogeneous diffusion systems

First establish the microscale heterogeneity has micro-period `mPeriod` on the spatial lattice, and coefficients inspired by Eckhardt2210.04536 §6.2. Set the phase of the heterogeneity so that each patch centre is a point of symmetry of the diffusivity. Then the heterogeneity is repeated to fill each patch. If an odd number of odd-periods in a patch, then the centre patch is a grid point of the field u, otherwise the centre patch is at a half-grid point.

```
clear all
mPeriod = 6
y = linspace(0,1,mPeriod+1)';
a = ( 3+cos(2*pi*y(1:mPeriod)) )/30
A0 = 1/mean(1./a) % roughly the effective diffusivity
```

The microscale diffusivity has an additional additive component of $+\frac{1}{30}\cos(2\pi t/\epsilon^2)$ which is coded into time derivative routine via global `microTimePeriod`.

Set the periodicity, via integer $1/\epsilon$, and other parameters.

```
nPeriodsPatch = 2 % any integer
rEpsilon = 100
dx = 1/(mPeriod*rEpsilon+1)
nSubP = nPeriodsPatch*mPeriod+2
tol=1e-9;
```

Set the time periodicity (global).

```
global microTimePeriod
microTimePeriod = 1/rEpsilon^2
```

Establish the global data struct `patches` for the microscale heterogeneous lattice diffusion system (2) solved on macroscale domain $[0,1]$, with `nPatch` patches, and say fourth-order interpolation to provide the edge-values of the inter-patch coupling conditions. Distribute the patches either equispaced or chebyshev. Setting `patches.EdgyInt` true means the edge-values come from interpolating the opposite next-to-edge values of the patches (not the mid-patch values).

```
nPatch = 7
ordCC = 4
Dom = 'chebyshev'
global patches
configPatches1(@heteroDiffF,[0 1],Dom,nPatch ...
    ,ordCC,dx,nSubP,'EdgyInt',true,'hetCoeffs',a);
DX = mean(diff(squeeze(patches.x(1,1,1,:))))
```

Set the forcing coefficients as the odd-periodic extensions, accounting for roundoff error in `f2`.

```
if 0 % given forcing
  patches.f1=2*( patches.x-patches.x.^2 );
  patches.f2=2*0.5+0*patches.x;
else% simple sine forcing
  patches.f1=sin(pi*patches.x);
  patches.f2=pi/2*sin(pi*patches.x);
end%if
```

Simulate Set the initial conditions of a simulation to be zero. Mark that edge of patches are not to be used in the projective extrapolation by setting initial values to NaN.

```
u0 = 0*patches.x;
u0([1 end],:) = nan;
```

Set the desired macro- and microscale time-steps over the time domain. The macroscale step is in proportion to the effective mean diffusion time on the macroscale, here $1/(A_0\pi^2) \approx 1$ so for macro-scale error less than 1% need $\Delta t < 0.24$, so use 0.1 say.

The burst time depends upon the sub-patch effective diffusion rate β where here rate $\beta \approx \pi^2 A_0/h^2 \approx 2000$ for patch width $h \approx 0.02$: use the formula from the Manual, with some extra factor, and rounded to the nearest multiple of the time micro-periodicity.

```
ts = linspace(0,1,21)
h=(nSubP-1)*dx;
beta = pi^2*A0/h^2 % slowest rate of fast modes
burstT = 2.5*log(beta*diff(ts(1:2)))/beta
burstT = max(10,round(burstT/microTimePeriod))*microTimePeriod +1e-12
addpath('../../ProjInt')
```

Time the projective integration simulation.

```
tic
[us,tss,uss] = PIRK2(@heteroBurstF, ts, u0(:), burstT);
cputime=toc
```

Plot space-time surface of the simulation First, just a macroscale mesh plot—stereo pair.

```
xs=squeeze(patches.x);
Xs=mean(xs);
Us=squeeze(mean( reshape(us,length(ts),[],nPatch), 2,'omitnan'));
```

```
figure(1),clf
for k = 1:2, subplot(2,2,k)
  mesh(ts,Xs(:),Us')
  ylabel('x'), xlabel('t'), zlabel('U(x,t)')
  colormap(0.8*hsv), axis tight, view(62-4*k,45)
end
```

Second, plot a surface detailing the microscale bursts—stereo pair. Do not bother with the patch-edge values.

```
xs([1 end],:) = nan;
for k = 1:2, subplot(2,2,2+k)
  surf(tss,xs(:),uss',  'EdgeColor','none')
  ylabel('x'), xlabel('t'), zlabel('u(x,t)')
  colormap(0.7*hsv), axis tight, view(62-4*k,45)
end
```

3.2 `heteroBurstF()`: a burst of heterogeneous diffusion

This code integrates in time the derivatives computed by `heteroDiff` from within the patch coupling of `patchSys1`. Try `ode23`, although `ode45` may give smoother results. Sample every period of the microscale time fluctuations (or, at least, close to the period).

```
function [ts, ucts] = heteroBurstF(ti, ui, bT)
    global microTimePeriod
        [ts,ucts] = ode45( @patchSys1,ti+(0:microTimePeriod:bT),ui(:)
end
```

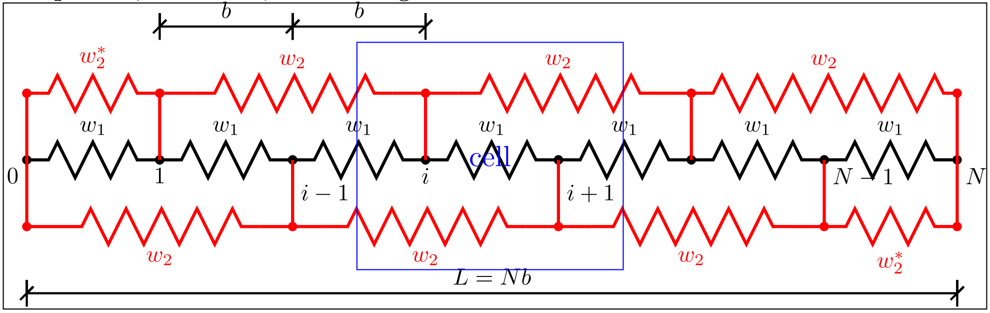
4 `Combescure2022`: example of a 1D heterogeneous toy elasticity by simulation on small patches

Started changing for BCs but nowhere near complete.

Plot an example simulation in time generated by the patch scheme applied to macroscale toy elasticity through a medium with microscale heterogeneity.

Suppose the spatial microscale lattice is at rest at points x_i, with constant spacing b (Figure 4). With displacement variables $u_i(t)$, simulate the microscale lattice toy elasticity system with 2-periodicity: for $p = 1, 2$ (respectively black

Figure 4: 1D arrangement of non-linear springs with connections to (a) next-to-neighbour node (Combescure 2022, Fig. 3(a)). The blue box is one cell of one period, width $2b$, containing an odd and an even i.



and red in Figure 4) and for every i,

$$\epsilon_i^p := \frac{1}{pb}(u_{i+p/2} - u_{i-p/2}), \quad \sigma_i^p := w_p'(\epsilon_i^p), \quad \frac{\partial^2 u_i}{\partial t^2} = \sum_{p=1}^{2} \frac{1}{pb?}(\sigma_{i+p/2}^p - \sigma_{i-p/2}^p). \tag{3}$$

The system has a microscale heterogeneity via the different functions $w_p'(\epsilon) := \epsilon - M_p\epsilon^3 + \epsilon^5$ (Combescure 2022, §4):

- microscale instability with $M_1 := 2$ and $M_2 := 1$; and
- macroscale instability with $M_1 := -1$ and $M_2 := 3$.

4.1 Configure heterogeneous toy elasticity systems

Set some physical parameters.

```
clear all
global b M vis i0 iN
b = 1  % separation of lattice points
N = 40 % # lattice steps in L
L = b*N
M = [0 0] % no cubic spring terms
%M = [2 1] % small scale instability??
%M = [-1 3] % large scale instability??
% see end-heteroToyE for function dLdt of prescribed end movement
vis = 0.01
```

```
tEnd = 130
tol = 1e-9;
```

Patch parameters: here `nSubP` is the number of cells, so `lPatch` is the distance from leftmost odd/even points to the rightmost odd/even points, respectively.

```
edgyInt = true
nSubP = 6, nPatch = 5 % gives ratio=1 for full-domain
%nSubP = 4, nPatch = 3
%H=L/nPatch
%if edgyInt, ratio=2*b*(nSubP-2)/H, end
%nP4ratio1=L/(2*b*(nSubP-2))
```

Establish the global data struct `patches` for the microscale heterogeneous lattice toy elasticity system (3). Solved with `nPatch` patches, and high-order interpolation to provide the edge-values of the inter-patch coupling conditions.

```
global patches
configPatches1(@heteroToyE,[0 L],'equispaced',nPatch ...
    ,0,b,nSubP,'EdgyInt',edgyInt);
assert(abs(2*b-diff(patches.x(1:2)))<tol,'sub-patch grid config error
xx = patches.x+[-1 1]*b/2; % staggered sub-cell positions
```

4.2 Eigenvalues of the Jacobian

Set zero to be the reference equilibrium in this linear problem. Put NaNs on the patch-edges.

```
if 0
u0 = [ 0*xx 0*xx ];
u0([1 end],:,:,:)=nan;
i=find(~isnan(u0));
nJac=length(i)
```

Remove boundary conditions.

```
i0=[]; iN=[];
```

Construct the Jacobian column-wise from the transform of a complete set of unit basis vectors (as this is linear problem at the moment).

```
Jac=nan(nJac);
for j=1:nJac
  uj=u0; uj(i(j))=1;
  dujdt=patchSys1(-1,uj);
  Jac(:,j)=dujdt(i);
end
Jac(abs(Jac)<tol)=0;
figure(3),clf,spy(Jac)
```

Find eigenvalues

```
[evecs,evals]=eig(Jac);
evals=diag(evals);
[~,j]=sort( -real(evals)+0.0001*abs(imag(evals)) );
evals=evals(j);
evecs=evecs(:,j);
leadingEvals=evals(1:18)'
```

Plot spectrum

```
   handle = plot(real(evals),imag(evals),'.');
   xlabel('real-part'), ylabel('imag-part')
   quasiLogAxes(handle,0.1,1);
   drawnow
end%if compute eigenvalues
```

4.3 Simulate in time

Set the initial conditions of a simulation. I choose to store odd i in `u((i+1)/2,1,:)` and even i in `u(i/2,2,:)`, that is, array

$$\mathtt{u} = \begin{bmatrix} u_1 & u_2 \\ u_3 & u_4 \\ u_5 & u_6 \\ \vdots & \vdots \end{bmatrix}.$$

```
u0 = 0*[ sin(pi/L*xx)  -0.14*cos(pi/L*xx) ];
u0 = u0+0.01*( rand(size(u0))-0.5 );
```

But, impose $u_i = 0$ at $x = 0$ which here I translate to mean that $u_i = \dot u_i = 0$ for both $x_i = \pm b/2$. Slightly different to the left-end of Figure 4, but should be near enough. Here find both $u, \dot u$ locations.

```
iO=find(abs([xx xx])<0.6*b);
u(iO)=0;
```

Apply a set force at material originally at $x = L$, so start with $u_i = \dot u_i = 0$ for both $x_i = L \pm b/2$. Subsequently apply an additional and increasing compression force on the points initially at $x = L$. Hmmm: but that is not quite isolating the two sides of $x = L$??

```
iN=find(abs([xx xx]-L)<0.6*b)
u(iN)=0;
```

Integrate some time using standard integrator.

```
tic
[ts,ust] = ode23(@patchSys1, tEnd*linspace(0,1,41), u0(:));
cpuIntegrateTime = toc
```

Plot space-time surface of the simulation We want to see the edge values of the patches, so interpolate and then adjoin a row of `nans` in between patches. Because of the odd/even storage we need to do a lot of permuting and reshaping.

```
xs = reshape( permute( xx ,[2 1 3 4]), 2*nSubP,nPatch);
xs(end+1,:) = nan;
uvs = reshape( permute( reshape(ust ...
      ,length(ts),nSubP,4,1,nPatch) ,[2 3 1 4 5]) ,nSubP,[],1,nPatch)
uvs = reshape( patchEdgeInt1(uvs) ,nSubP,4,[],nPatch);
% extract displacements
us = reshape( permute( uvs(:,1:2,:,:) ...
     ,[2 1 4 3]) ,2*nSubP,nPatch,[]);
us(end+1,:,:) = nan;
us = reshape(us,[],length(ts));
% extract velocities
vs = reshape( permute( uvs(:,3:4,:,:) ...
     ,[2 1 4 3]) ,2*nSubP,nPatch,[]);
vs(end+1,:,:) = nan;
vs = reshape(vs,[],length(ts));
```

Plot evolving function

```
figure(1),clf()
plot(xs(:),vs)
```

```
xlabel('space x')
%ylabel('displacement u')
ylabel('velocity v')
legend(num2str(ts))
```

Plot a space-time surface of displacements over the macroscale duration of the simulation.

```
  figure(2), clf()
  mesh(ts,xs(:),us)
  view(60,40), colormap(0.8*hsv)
  xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
  title(['patch ratio r = ' num2str(ratio)])
  drawnow
```

Similarly plot velocities

```
  figure(3), clf()
  mesh(ts,xs(:),vs)
  view(60,40), colormap(0.8*hsv)
  xlabel('time t'), ylabel('space x'), zlabel('v(x,t)')
  title(['patch ratio r = ' num2str(ratio)])
  drawnow
```

4.4 heteroToyE(): forced heterogeneous toy elasticity

This function codes the lattice heterogeneous toy elasticity inside the patches. Computes the time derivative at each point in the interior of a patch, output in ut.

```
function uvt = heteroToyE(t,uv,patches)
  global b M vis i0 iN
```

Separate state vector into displacement and velocity fields: u_{ijI} is the displacement at the jth point in the ith 2-cell in the Ith patch; similarly for velocity v_{ijI}. That is, physically neighbouring points have different j, whereas physical next-to-neighbours have i different by one.

```
  u=uv(:,1:2,:,:); v=uv(:,3:4,:,:); % separate u and v=du/dt
```

Compute the two different strain fields, and also a first derivative for some optional viscosity.

```
  eps2 = diff(u)/(2*b);
  eps1 = [u(:,2,:,:)-u(:,1,:,:) u([2:end 1],1,:,:)-u(:,2,:,:)]/b;
  eps1(end,2,:,:)=nan; % as this value is fake
  vx1  = [v(:,2,:,:)-v(:,1,:,:) v([2:end 1],1,:,:)-v(:,2,:,:)]/b;
  vx1(end,2,:,:)=nan; % as this value is fake
```

Set corresponding nonlinear stresses

```
  sig2 = eps2-M(2)*eps2.^3+eps2.^5;
  sig1 = eps1-M(1)*eps1.^3+eps1.^5;
```

Preallocate output array, and fill in time derivatives of displacement and velocity, from velocity and gradient of stresses, respectively.

```
  uvt = nan+uv;          % preallocate output array
  i=2:size(uv,1)-1;
  % rate of change of position
  uvt(i,1:2,:,:) = v(i,:,:,:);
  % rate of change of velocity +some artificial viscosity??
  uvt(i,3:4,:,:) = diff(sig2) ...
    +[ sig1(i,1,:,:)-sig1(i-1,2,:,:)  diff(sig1(i,:,:,:),1,2)] ...
  +vis*[ vx1(i,1,:,:)-vx1(i-1,2,:,:)  diff(vx1(i,:,:,:),1,2) ];
```

Maintain boundary value of $u_i, \dot u_i$ by setting them both to be constant in time, for both $x_i = \pm b/2$. If `i0` is empty, then no boundary condition is set.

```
if ~isempty(i0), uvt(i0)=0; end
if ~isempty(iN), uvt(iN(3:4))=dLdt(t); end% vel=d/dt of end displacem
end% function
```

4.5 `dLdt()`: prescribed movement of length

```
function Ld=dLdt(t)
Ld=-0.03*cos(t/20);
end
```

5 `monoscaleDiffEquil2`: equilibrium of a 2D monoscale heterogeneous diffusion via small patches

Here we find the steady state $u(x,y)$ to the heterogeneous PDE(inspired by Freese et al.[1] §5.2)

$$u_t = A(x,y)\vec\nabla\vec\nabla u - f,$$

on domain $[-1,1]^2$ with Dirichlet BCs, for coefficient 'diffusion' matrix

$$A := \begin{bmatrix} 2 & a \\ a & 2 \end{bmatrix} \quad \text{with } a := \operatorname{sign}(xy) \text{ or } a := \sin(\pi x)\sin(\pi y),$$

and for forcing $f(x,y)$ such that the exact equilibrium is

$$u = x\big(1-e^{1-|x|}\big)y\big(1-e^{1-|y|}\big).$$

But for simplicity, let's do $u = x(1-x^2)y(1-y^2)$ for which we code f later—as determined by this computer algebra.

```
on gcd; factor sin;
%let { df(sign(~x),~x)=>0
%     , df(abs(~x),~x)=>sign(x)
%     , abs(~x)^2=>abs(x), sign(~x)^2=>1 };
%u:=x*(1-exp(1-abs(x)))*y*(1-exp(1-abs(y)));
u:=x*(1-x^2)*y*(1-y^2);
a:=sin(pi*x)*sin(pi*y);
f:=2*df(u,x,x)+2*a*df(u,x,y)+2*df(u,y,y);
```

Clear, and initiate globals.

```
clear all
global patches i
```

Patch configuration Initially use 7×7 patches in the square $(-1,1)^2$. For continuous forcing we may have small patches of any reasonable microgrid spacing—here the microgrid error dominates.

```
nPatch = 7
nSubP = 5
dx = 0.03
```

Specify some order of interpolation.

[1] http://arxiv.org/abs/2211.13731

```
configPatches2(@monoscaleDiffForce2,[-1 1 -1 1],'equispace' ...
    ,nPatch ,4 ,dx ,nSubP ,'EdgyInt',true );
```

Compute the time-constant coefficient and time-constant forcing, and store them in struct `patches` for access by the microcode of Section 5.1.

```
  x=patches.x;  y=patches.y;
  patches.A = sin(pi*x).*sin(pi*y);
  patches.fu = ...
    +2*patches.A.*(9*x.^2.*y.^2-3*x.^2-3*y.^2+1) ...
    +12*x.*y.*(x.^2+y.^2-2);
```

By construction, the PDE has analytic solution

```
uAnal = x.*(1-x.^2).*y.*(1-y.^2);
```

Solve for steady state Set initial guess of zero, with `NaN` to indicate patch-edge values. Index `i` are the indices of patch-interior points, and the number of unknowns is then its length.

```
u0 = zeros(nSubP,nSubP,1,1,nPatch,nPatch);
u0([1 end],:,:) = nan;  u0(:,[1 end],:) = nan;
i = find(~isnan(u0));
nVars = numel(i)
```

Solve by iteration. Use `fsolve` for simplicity and robustness (using `optimoptions` to omit its trace information).

```
tic;
uSoln = fsolve(@theRes,u0(i) ...
        ,optimoptions('fsolve','Display','off'));
solnTime = toc
```

Store the solution into the patches, and give magnitudes.

```
u0(i) = uSoln;
normSoln = norm(uSoln)
normResidual = norm(theRes(uSoln))
errors = uAnal(i)-uSoln;
normError = norm(errors)
```

Draw solution profile First reshape arrays to suit 2D space surface plots.

```
figure(1), clf, colormap(hsv)
x = squeeze(patches.x); y = squeeze(patches.y);
u = reshape(permute(squeeze(u0),[1 3 2 4]), [numel(x) numel(y)]);
```

Draw the patch solution surface, with edge-values omitted as already `NaN` by not bothering to interpolate them.

```
surf(x(:),y(:),u'); view(60,55)
xlabel('x'), ylabel('y'), zlabel('u(x,y)')
```

5.1 `monoscaleDiffForce2()`: microscale discretisation inside patches of forced diffusion PDE

This function codes the lattice heterogeneous diffusion of the PDE inside the patches. For 6D input arrays `u`, `x`, and `y`, computes the time derivative at each point in the interior of a patch, output in `ut`.

```
function ut = monoscaleDiffForce2(t,u,patches)
  dx = diff(patches.x(2:3));  % x space step
  dy = diff(patches.y(2:3));  % y space step
  ix = 2:size(u,1)-1; % x interior points in a patch
  iy = 2:size(u,2)-1; % y interior points in a patch
  ut = nan+u;         % preallocate output array
```

Set Dirichlet boundary value of zero around the square domain, or code some function variation.

```
u( 1 ,:,:,:, 1 ,:)=0; % left edge of left patches
u( 1 ,:,:,:, 1 ,:)=(1+patches.y)/2; % or code function of y
u(end,:,:,:,end,:)=0; % right edge of right patches
u(:, 1 ,:,:,:, 1 )=0; % bottom edge of bottom patches
u(:,end,:,:,:,end)=0; % top edge of top patches
u(:,end,:,:,:,end)=1; % or code function of x
```

Compute the time derivatives via stored forcing and coefficients. Easier to code by conflating the last four dimensions into the one `,:`.

```
  ut(ix,iy,:) ...
  = 2*diff(u(:,iy,:),2,1)/dx^2 +2*diff(u(ix,:,:),2,2)/dy^2 ...
   +2*patches.A(ix,iy,:).*( u(ix+1,iy+1,:) -u(ix-1,iy+1,:) ...
      -u(ix+1,iy-1,:) +u(ix-1,iy-1,:) )/(4*dx*dy) ...
   -patches.fu(ix,iy,:);
end%function monoscaleDiffForce2
```

5.2 `theRes()`: function to zero

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

```
function f=theRes(u)
  global patches i
  v=nan(size(patches.x+patches.y));
  v(i)=u;
  f=patchSys2(0,v(:),patches);
  f=f(i);
end%function theRes
```

Fin.

6 `twoscaleDiffEquil2`: equilibrium of a 2D twoscale heterogeneous diffusion via small patches

Here we find the steady state $u(x,y)$ to the heterogeneous PDE (inspired by Freese et al.[2] §5.3.1)

$$u_t = A(x,y)\vec\nabla\vec\nabla u - f,$$

on domain $[-1,1]^2$ with Dirichlet BCs, for coefficient 'diffusion' matrix, varying with period 2ϵ on the microscale $\epsilon = 2^{-7}$, of

$$A := \begin{bmatrix} 2 & a \\ a & 2 \end{bmatrix} \quad \text{with } a := \sin(\pi x/\epsilon)\sin(\pi y/\epsilon),$$

and for forcing $f := (x + \cos 3\pi x)y^3$.

Clear, and initiate globals.

```
clear all
global patches i
```

First establish the microscale heterogeneity has micro-period `mPeriod` on the spatial lattice. Set the phase of the heterogeneity so that each patch centre is a point of symmetry of the diffusivity. Then `configPatches2` replicates the heterogeneity to fill each patch.

[2] http://arxiv.org/abs/2211.13731

```
mPeriod = 6
z = (0.5:mPeriod)'/mPeriod;
A = sin(2*pi*z).*sin(2*pi*z');
```

Set the periodicity, via ϵ, and other microscale parameters.

```
nPeriodsPatch = 1 % any integer
epsilon = 2^(-5) % so we can see patches
dx = (2*epsilon)/mPeriod
nSubP = nPeriodsPatch*mPeriod+2 % for edgy int
```

Patch configuration Say use 7×7 patches in $(-1, 1)^2$, fourth order interpolation, and either 'equispace' or 'chebyshev':

```
nPatch = 7
configPatches2(@twoscaleDiffForce2,[-1 1],'equispace' ...
    ,nPatch ,4 ,dx ,nSubP ,'EdgyInt',true ,'hetCoeffs',A );
```

Compute the time-constant forcing, and store in struct `patches` for access by the microcode of Section 6.1.

```
patches.fu = 100*(patches.x+cos(3*pi*patches.x)).*patches.y.^3;
```

Solve for steady state Set initial guess of zero, with `NaN` to indicate patch-edge values. Index `i` are the indices of patch-interior points, and the number of unknowns is then its length.

```
u0 = zeros(nSubP,nSubP,1,1,nPatch,nPatch);
u0([1 end],:,:) = nan;  u0(:,[1 end],:) = nan;
i = find(~isnan(u0));
nVariables = numel(i)
```

Solve by iteration. Use `fsolve` for simplicity and robustness (and using `optimoptions` to omit trace information).

```
tic;
uSoln = fsolve(@theRes,u0(i) ...
        ,optimoptions('fsolve','Display','off'));
solnTime = toc
```

Store the solution into the patches, and give magnitudes.

```
u0(i) = uSoln;
normSoln = norm(uSoln)
normResidual = norm(theRes(uSoln))
```

Draw solution profile First reshape arrays to suit 2D space surface plots.

```
figure(1), clf, colormap(hsv)
x = squeeze(patches.x); y = squeeze(patches.y);
u = reshape(permute(squeeze(u0),[1 3 2 4]), [numel(x) numel(y)]);
```

Draw the patch solution surface, with edge-values omitted as already `NaN` by not bothering to interpolate them.

```
surf(x(:),y(:),u'); view(60,55)
xlabel('x'), ylabel('y'), zlabel('u(x,y)')
```

6.1 `twoscaleDiffForce2()`: microscale discretisation inside patches of forced diffusion PDE

This function codes the lattice heterogeneous diffusion of the PDE inside the patches. For 6D input arrays `u`, `x`, and `y`, computes the time derivative at each point in the interior of a patch, output in `ut`.

```
function ut = twoscaleDiffForce2(t,u,patches)
  dx = diff(patches.x(2:3));  % x space step
  dy = diff(patches.y(2:3));  % y space step
  ix = 2:size(u,1)-1; % x interior points in a patch
  iy = 2:size(u,2)-1; % y interior points in a patch
  ut = nan+u;         % preallocate output array
```

Set Dirichlet boundary value of zero around the square domain.

```
  u( 1 ,:,:,:, 1 ,:)=0; % left edge of left patches
  u(end,:,:,:,end,:)=0; % right edge of right patches
  u(:, 1 ,:,:,:, 1 )=0; % bottom edge of bottom patches
  u(:,end,:,:,:,end)=0; % top edge of top patches
```

Compute the time derivatives via stored forcing and coefficients. Easier to code by conflating the last four dimensions into the one `,:`.

```
  ut(ix,iy,:) ...
  = 2*diff(u(:,iy,:),2,1)/dx^2 +2*diff(u(ix,:,:),2,2)/dy^2 ...
   +2*patches.cs(ix,iy).*( u(ix+1,iy+1,:) -u(ix-1,iy+1,:) ...
     -u(ix+1,iy-1,:) +u(ix-1,iy-1,:) )/(4*dx*dy) ...
   -patches.fu(ix,iy,:);
end%function twoscaleDiffForce2
```

6.2 theRes(): function to zero

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

```
function f=theRes(u)
  global patches i
  v=nan(size(patches.x+patches.y));
  v(i)=u;
  f=patchSys2(0,v(:),patches);
  f=f(i);
end%function theRes
```

Fin.

7 abdulleDiffEquil2: equilibrium of a 2D twoscale heterogeneous diffusion via small patches

Here we find the steady state $u(x,y)$ to the heterogeneous PDE (inspired by Abdulle, Arjmand, and Paganoni 2020, §5.1)

$$u_t = \vec{\nabla}\cdot[a(x,y)\vec{\nabla}u] + 10,$$

on square domain $[0,1]^2$ with zero-Dirichlet BCs, for coefficient 'diffusion' matrix, varying with period ϵ of (their (45))

$$a := \frac{2 + 1.8\sin 2\pi x/\epsilon}{2 + 1.8\cos 2\pi y/\epsilon} + \frac{2 + \sin \pi y/\epsilon}{2 + 1.8\cos 2\pi x/\epsilon}.$$

The solution shows some nice little microscale wiggles.

Clear, and initiate globals.

```
clear all
global patches i
```

First establish the microscale heterogeneity has micro-period `mPeriod` on the spatial micro-grid lattice. Then `configPatches2` replicates the heterogeneity to fill each patch. (These diffusion coefficients should really recognise the half-grid-point shifts, but let's not bother.)

```
mPeriod = 6
x = (0.5:mPeriod)'/mPeriod;  y=x';
a = (2+1.8*sin(2*pi*x))./(2+1.8*sin(2*pi*y)) ...
   +(2+    sin(2*pi*y))./(2+1.8*sin(2*pi*x));
```

Set the periodicity, via ϵ, and other microscale parameters.

```
nPeriodsPatch = 1 % any integer
epsilon = 2^(-4) % not tiny, so we can see patches
dx = epsilon/mPeriod
nSubP = nPeriodsPatch*mPeriod+2 % when edgy int
```

Patch configuration Choose either Dirichlet (default) or Neumann on the left boundary in coordination with micro-code in Section 7.1

```
Dom.bcOffset = zeros(2);
if 1, Dom.bcOffset(1)=0.5; end% left Neumann
```

Say use 7×7 patches in $(0,1)^2$, fourth order interpolation, and either 'equispace' or 'chebyshev':

```
nPatch = 7
Dom.type='equispace';
configPatches2(@abdulleDiffForce2,[0 1],Dom ...
    ,nPatch ,4 ,dx ,nSubP ,'EdgyInt',true ,'hetCoeffs',a );
```

Solve for steady state Set initial guess of zero, with `NaN` to indicate patch-edge values. Index `i` are the indices of patch-interior points, and the number of unknowns is then its length.

```
u0 = zeros(nSubP,nSubP,1,1,nPatch,nPatch);
u0([1 end],:,:) = nan;  u0(:,[1 end],:) = nan;
i = find(~isnan(u0));
nVariables = numel(i)
```

Solve by iteration. Use `fsolve` for simplicity and robustness (and using `optimoptions` to omit trace information).

```
tic;
uSoln = fsolve(@theRes,u0(i) ...
        ,optimoptions('fsolve','Display','off'));
solnTime = toc
```

Store the solution into the patches, and give magnitudes.

```
u0(i) = uSoln;
normSoln = norm(uSoln)
normResidual = norm(theRes(uSoln))
```

Draw solution profile First reshape arrays to suit 2D space surface plots.

```
figure(1), clf, colormap(hsv)
x = squeeze(patches.x); y = squeeze(patches.y);
u = reshape(permute(squeeze(u0),[1 3 2 4]), [numel(x) numel(y)]);
```

Draw the patch solution surface, with edge-values omitted as already `NaN` by not bothering to interpolate them.

```
surf(x(:),y(:),u'); view(60,55)
xlabel('x'), ylabel('y'), zlabel('u(x,y)')
```

7.1 `abdulleDiffForce2()`: microscale discretisation inside patches of forced diffusion PDE

This function codes the lattice heterogeneous diffusion of the PDE inside the patches. For 6D input arrays `u`, `x`, and `y`, computes the time derivative at each point in the interior of a patch, output in `ut`.

```
function ut = abdulleDiffForce2(t,u,patches)
  dx = diff(patches.x(2:3));  % x space step
  dy = diff(patches.y(2:3));  % y space step
  ix = 2:size(u,1)-1; % x interior points in a patch
  iy = 2:size(u,2)-1; % y interior points in a patch
  ut = nan+u;         % preallocate output array
```

Set Dirichlet boundary value of zero around the square domain, but also cater for zero Neumann condition on the left boundary.

```
  u( 1 ,:,:,:, 1 ,:)=0; % left edge of left patches
  u(end,:,:,:,end,:)=0; % right edge of right patches
  u(:, 1 ,:,:,:, 1 )=0; % bottom edge of bottom patches
  u(:,end,:,:,:,end)=0; % top edge of top patches
  if 1, u(1,:,:,:,1,:)=u(2,:,:,:,1,:); end% left Neumann
```

Compute the time derivatives via stored forcing and coefficients. Easier to code by conflating the last four dimensions into the one `,:`.

```
  ut(ix,iy,:) = diff(patches.cs(:,iy).*diff(u(:,iy,:)))/dx^2 ...
      + diff(patches.cs(ix,:).*diff(u(ix,:,:),1,2),1,2)/dy^2 ...
      + 10;
end%function abdulleDiffForce2
```

7.2 `theRes()`: function to zero

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

```
function f=theRes(u)
  global patches i
  v=nan(size(patches.x+patches.y));
  v(i)=u;
  f=patchSys2(0,v(:),patches);
  f=f(i);
end%function theRes
```

Fin.

8 `randAdvecDiffEquil2`: equilibrium of a 2D random heterogeneous advection-diffusion via small patches

Here we find the steady state $u(x,y)$ of the heterogeneous PDE (inspired by Bonizzoni et al.[3] §6.2)

$$u_t = \mu_1\nabla^2 u - (\cos\mu_2, \sin\mu_2)\cdot\vec\nabla u - u + f\,,$$

on domain $[0,1]^2$ with Neumann BCs, for microscale random diffusion and advection coefficients, $\mu_1 \in [0.01, 0.1]$ and $\mu_2 \in [0, 2\pi)$, and for forcing

$$f := \exp\left[-\frac{(x-\mu_3)^2 + (x-\mu_4)^2}{\mu_5^2}\right],$$

smoothly varying in space for fixed $\mu_3, \mu_4 \in [0.25, 0.75]$ and $\mu_5 \in [0.1, 0.25]$. The above system is dominantly diffusive for lengths scales $\ell < 0.01 = \min\mu_1$.

Clear, and initiate globals.

[3] http://arxiv.org/abs/2211.15221

```
clear all
global patches
```

First establish the microscale heterogeneity has micro-period `mPeriod` on the spatial lattice. Then `configPatches2` replicates the heterogeneity to fill each patch.

```
mPeriod = 4
mu1 = 10.^(-1-rand(mPeriod))
mu2 = 2*pi*rand(mPeriod)
cs = cat(3,mu1,cos(mu2),sin(mu2));
meanDiffAdvec=squeeze(mean(mean(cs)))
```

Set the periodicity, ϵ, and other microscale parameters.

```
nPeriodsPatch = 1 % any integer
epsilon = 2^(-4) % so we can see patches
dx = epsilon/mPeriod
nSubP = nPeriodsPatch*mPeriod+2 % for edgy int
```

Patch configuration Say use 7×7 patches in $(0,1)^2$, fourth order interpolation, either 'equispace' or 'chebyshev', and the offset for Neumann boundary conditions:

```
nPatch = 7
Dom.type= 'equispace';
Dom.bcOffset = 0.5;
configPatches2(@randAdvecDiffForce2,[0 1],Dom ...
    ,nPatch ,4 ,dx ,nSubP ,'EdgyInt',true ,'hetCoeffs',cs );
```

Compute the time-constant forcing, and store in struct `patches` for access by the microcode of Section 8.1.

```
mu = [ 0.25+0.5*rand(1,2) 0.1+0.15*rand ]
patches.fu = exp(-((patches.x-mu(1)).^2+(patches.y-mu(2)).^2)/mu(3)^2
```

Solve for steady state Set initial guess of zero, with `NaN` to indicate patch-edge values. Index `i` are the indices of patch-interior points, store in global patches for access by `theRes`, and the number of unknowns is then its number of elements.

```
u0 = zeros(nSubP,nSubP,1,1,nPatch,nPatch);
u0([1 end],:,:) = nan;  u0(:,[1 end],:) = nan;
patches.i = find(~isnan(u0));
nVariables = numel(patches.i)
```

Solve by iteration. Use `fsolve` for simplicity and robustness (and using `optimoptions` to omit trace information).

```
tic;
uSoln = fsolve(@theRes,u0(patches.i) ...
        ,optimoptions('fsolve','Display','off'));
solnTime = toc
```

Store the solution into the patches, and give magnitudes.

```
u0(patches.i) = uSoln;
normSoln = norm(uSoln)
normResidual = norm(theRes(uSoln))
```

Draw solution profile First reshape arrays to suit 2D space surface plots.

```
figure(1), clf, colormap(hsv)
x = squeeze(patches.x); y = squeeze(patches.y);
u = reshape(permute(squeeze(u0),[1 3 2 4]), [numel(x) numel(y)]);
```

Draw the patch solution surface, with edge-values omitted as already `NaN` by not bothering to interpolate them.

```
surf(x(:),y(:),u'); view(60,55)
xlabel('x'), ylabel('y'), zlabel('u(x,y)')
```

8.1 `randAdvecDiffForce2()`: microscale discretisation inside patches of forced diffusion PDE

This function codes the lattice heterogeneous diffusion of the PDE inside the patches. For 6D input arrays `u`, `x`, and `y`, computes the time derivative at each point in the interior of a patch, output in `ut`.

```
function ut = randAdvecDiffForce2(t,u,patches)
  dx = diff(patches.x(2:3));  % x space step
  dy = diff(patches.y(2:3));  % y space step
  ix = 2:size(u,1)-1; % x interior points in a patch
  iy = 2:size(u,2)-1; % y interior points in a patch
  ut = nan+u;         % preallocate output array
```

Set Neumann boundary condition of zero derivative around the square domain.

```
  u( 1 ,:,:,:, 1 ,:)=u(  2  ,:,:,:, 1 ,:); % left edge of left patche
  u(end,:,:,:,end,:)=u(end-1,:,:,:,end,:); % right edge of right patc
  u(:, 1 ,:,:,:, 1 )=u(:,  2  ,:,:,:, 1 ); % bottom edge of bottom pa
  u(:,end,:,:,:,end)=u(:,end-1,:,:,:,end); % top edge of top patches
```

Compute the time derivatives via stored forcing and coefficients. Easier to code by conflating the last four dimensions into the one `,:`.

```
  ut(ix,iy,:) ...
  = patches.cs(ix,iy,1).*(diff(u(:,iy,:),2,1)/dx^2 ...
                          +diff(u(ix,:,:),2,2)/dy^2)...
   -patches.cs(ix,iy,2).*(u(ix+1,iy,:)-u(ix-1,iy,:))/(2*dx) ...
   -patches.cs(ix,iy,3).*(u(ix,iy+1,:)-u(ix,iy-1,:))/(2*dy) ...
   -u(ix,iy,:) +patches.fu(ix,iy,:);
end%function randAdvecDiffForce2
```

8.2 `theRes()`: function to zero

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

```
function f=theRes(u)
  global patches
  v=nan(size(patches.x+patches.y));
  v(patches.i)=u;
  f=patchSys2(0,v(:),patches);
  f=f(patches.i);
end%function theRes
```

Fin.

9 `homoDiffBdryEquil3`: equilibrium via computational homogenisation of a 3D diffusion on small patches

Find the equilibrium of a forced heterogeneous diffusion in 3D space on 3D patches as an example application.

```
clear all
global patches
```

First set random heterogeneous diffusivities of random (small) period in each of the three directions. Crudely normalise by the harmonic mean so the decay time scale is roughly one.

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

Configure the patch scheme with some arbitrary choices of square domain, patches, and micro-grid spacing 0.05. Use high order interpolation as few patches in each direction. Configure for Dirichlet boundaries except for Neumann on the right x-face.

```
nSubP=mPeriod+2;
nPatch=5;
Dom.type='equispace';
Dom.bcOffset=zeros(2,3); Dom.bcOffset(2)=0.5;
configPatches3(@microDiffBdry3, [-1-0*rand 1+0*rand], Dom ...
    , nPatch, 0, 0.05, nSubP, 'EdgyInt',true  ...
    ,'hetCoeffs',cDiff );
```

Set forcing, and store in global patches for access by the microcode

```
patches.fu = 10*exp(-patches.x.^2-patches.y.^2-patches.z.^2);
patches.fu = patches.fu.*(1+rand(size(patches.fu)));
```

Solve for steady state Set initial guess of zero, with `NaN` to indicate patch-edge values. Index `i` are the indices of patch-interior points, store in global patches for access by `theRes3`, and the number of unknowns is then its number of elements.

```
u0 = zeros([nSubP,1,1,nPatch,nPatch,nPatch]);
u0([1 end],:,:,:) = nan;
u0(:,[1 end],:,:) = nan;
u0(:,:,[1 end],:) = nan;
patches.i = find(~isnan(u0));
nVariables = numel(patches.i)
```

Solve by iteration. Use `fsolve` for simplicity and robustness (optionally `optimoptions` to omit trace information).

```
tic;
uSoln = fsolve(@theRes3,u0(patches.i));% ...
%        ,optimoptions('fsolve','Display','off'));
solveTime = toc
```

Store the solution into the patches, and give magnitudes.

```
u0(patches.i) = uSoln;
normSoln = norm(uSoln)
normResidual = norm(theRes3(uSoln))
```

Plot isosurfaces of the solution

```
figure(1), clf
rgb=get(gca,'defaultAxesColorOrder');
```

Reshape spatial coordinates of patches.

```
x = patches.x(:);
y = patches.y(:);
z = patches.z(:);
```

Draw isosurfaces. Get the solution with interpolated faces, form into a 6D array, and reshape and transpose x and y to suit the isosurface function.

```
  u = squeeze( patchEdgeInt3(u0) );
  u = reshape( permute(u,[2 5 1 4 3 6]) ...
      , [numel(y) numel(x) numel(z)]);
  maxu=max(u(:)),  minu=min(u(:))
```

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

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

Draw cross-eyed stereo view of some isosurfaces.

```
  clf;
  for p=1:2
    subplot(1,2,p)
    for iso=5:-1:1
       isov=(iso-0.5)/5*(maxu-minu)+minu;
       hsurf(iso) = patch(isosurface(x,y,z,u,isov));
       isonormals(x,y,z,u,hsurf(iso))
```

```
        set(hsurf(iso) ,'FaceColor',rgb(iso,:) ...
            ,'EdgeColor','none' ...
            ,'FaceAlpha',iso/5);
        hold on
     end
     axis tight, axis equal, view(45-7*p,25)
     xlabel('x'), ylabel('y'), zlabel('z')
     camlight, lighting gouraud
     hold off
  end% each p
```

9.1 `microDiffBdry3()`: 3D forced heterogeneous diffusion with boundaries

This function codes the lattice forced heterogeneous diffusion inside the 3D patches. For 8D input array `u` (via edge-value interpolation of `patchEdgeInt3`, such as by `patchSys3`, **??**), computes the time derivative at each point in the interior of a patch, output in `ut`. The three 3D array of diffusivities, c^x_{ijk}, c^y_{ijk} and c^z_{ijk}, have previously been stored in `patches.cs` (4D).

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

```
function ut = microDiffBdry3(t,u,patches)
  if nargin<3, global patches, end
```

Microscale space-steps.

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

Code microscale boundary conditions of say Neumann on right, and Dirichlet on left, top, bottom, front, and back (viewed along the z-axis).

```
u( 1 ,:,:,:,:, 1 ,:,:)=0; %left face of leftmost patch
u(end,:,:,:,:,end,:,:)=u(end-1,:,:,:,:,end,:,:); %right face of right
u(:, 1 ,:,:,:,:, 1 ,:)=0; %bottom face of bottommost
u(:,end,:,:,:,:,end,:)=0; %top face of topmost
```

```
u(:,:, 1 ,:,:,:,:, 1 )=0; %front face of frontmost
u(:,:,end,:,:,:,:,end)=0; %back face of backmost
```

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

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

9.2 `theRes3()`: function to zero

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

```
function f=theRes3(u)
  global patches
  v=nan(size(patches.x+patches.y+patches.z));
  v(patches.i)=u;
  f=patchSys3(0,v(:),patches);
  f=f(patches.i);
end%function theRes
```

Fin.

New configuration and interpolation

10 `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). [4]

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

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

Input

- `u` is a vector/array of length `nSubP` · `nVars` · `nEnsem` · `nPatch` where there are `nVars`·`nEnsem` field values at each of the points in the `nSubP` × `nPatch` multiscale spatial grid.
- `patches` a struct largely set by `configPatches1()`, and which includes the following.
  - `.x` is `nSubP` × 1 × 1 × `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).
  - `.nEnsem` the number of realisations in the ensemble.
  - `.parallel` whether serial or parallel.

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

  - `.nCore` [5]
  - [6]

Output

- `u` is 4D array, `nSubP` $\times$ `nVars` $\times$ `nEnsem` $\times$ `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.

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

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

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

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

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

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

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

[5]**ToDo:** introduced sometime but not fully implemented yet, because prefer ensemble
[6]**ToDo:** additional macros bdry info
[7]**ToDo:** Revise??

10.1 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 have to 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' $|k| <$ Nx$/2$, so set row vector ks $= k2\pi/N$ for 'wavenumbers' $k = (0\,, 1\,, \ldots\,, k_{\max}\,, -k_{\max}\,, \ldots\,, -1)$ for odd N, and $k = (0\,, 1\,, \ldots\,, k_{\max}\,, (k_{\max}+1)\,, -k_{\max}\,, \ldots\,, -1)$ for even N.

Deal with staggered grid by doubling the number of fields and halving the number of patches (configPatches1() tests that there are an even number of patches). Then the patch-ratio is effectively halved. The patch edges are near the middle of the gaps and swapped. Have not yet tested whether works for Edgy Interpolation??

```
  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)];
    stagShift = 0.5*[ones(1,nVars) -ones(1,nVars)];
    iV = [nVars+1:2*nVars 1:nVars]; % scatter interp to alternate fie
    r = r/2;           % ratio effectively halved
    Nx = Nx/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
```

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

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

```
if ~patches.EdgyInt
    Cleft = fft(u(i0  ,:,:,:),[],4);
    Cright = Cleft;
else
    Cleft = fft(u(2   ,:,:,:),[],4);
    Cright= fft(u(nx-1,:,:,:),[],4);
end
```

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

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

10.2 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(:,:,:,:) = patches.x([nx-1 2],:,:,I);
  else % interpolate mid-patch values/sums
    F(:,:,:,:,1) = sum( u((i0-c):(i0+c),:,:,I) ,1);
    X(:,:,:,:) = patches.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 = patches.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. First alternative: the case of order p interpolation across the domain, asymmetric near the boundary. Use this first alternative for the moment (Jan 2023).

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

```
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.

```
  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, may override their computed interpolation values with `NaN`.

```
%u( 1,:,:, 1) = nan;
%u(nx,:,:,Nx) = nan;
```

End of the non-periodic interpolation code.

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

Fin, returning the 4D array of field values.

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

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys1()`. Section 11.1 lists an example of its use.

```
function patches = configPatches1(fun,Xlim,Dom ...
    ,nPatch,ordCC,dx,nSubP,varargin)
```

Input If invoked with no input arguments, then executes an example of simulating Burgers' PDE—see Section 11.1 for the example code.

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)`, that computes time derivatives (or time-steps) of quantities on the 1D micro-grid within all the 1D patches.
- `Xlim` give the macro-space spatial domain of the computation, 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. So either `ratio` $= \frac{1}{2}$ means the patches abut and `ratio` $= 1$ is overlapping patches as in holistic discretisation, or `ratio` $= 1$ means the patches abut. Small `ratio` should greatly reduce computational time.

- `nSubP` is the number of equi-spaced microscale lattice points in each patch. If not using `EdgyInt`, then must be odd so that there is a centre-patch lattice point.

- `nEdge` (not yet implemented), *optional*, default=1, for each patch, the number of edge values set by interpolation at the edge regions of each patch. The default is one (suitable for microscale lattices with only nearest neighbour interactions).

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

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

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

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

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

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

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

Output The struct `patches` is created and set with the following components. If no output variable is provided for `patches`, then make the struct available as a global variable.[8]

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

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

- `.ordCC` is the specified order of inter-patch coupling.

[8]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 `nSubP` $\times 1 \times 1 \times$ `nPatch` array of the regular spatial locations x_{iI} of the ith microscale grid point in the Ith patch.
- `.ratio`, 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 `nEnsem` $= 1$, (`nSubP(1)` $- 1) \times n_c$ 2D array of microscale heterogeneous coefficients, or
  - if `nEnsem` > 1, (`nSubP(1)` $- 1) \times n_c \times m_x$ 3D array of m_x ensemble of phase-shifts of the microscale heterogeneous coefficients.
- `.parallel`, logical: true if patches are distributed over multiple CPUs/ cores for the Parallel Computing Toolbox, otherwise false (the default is to activate the *local* pool).
- `.codist`, *optional*, describes the particular parallel distribution of arrays over the active parallel pool.

11.1 If no arguments, then execute an example

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

The code here shows one way to get started: a user's script may have the following three steps (left-right arrows denote 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 0.0.06, and with seven microscale points forming each patch.

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

Set some initial condition, with some microscale randomness.

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

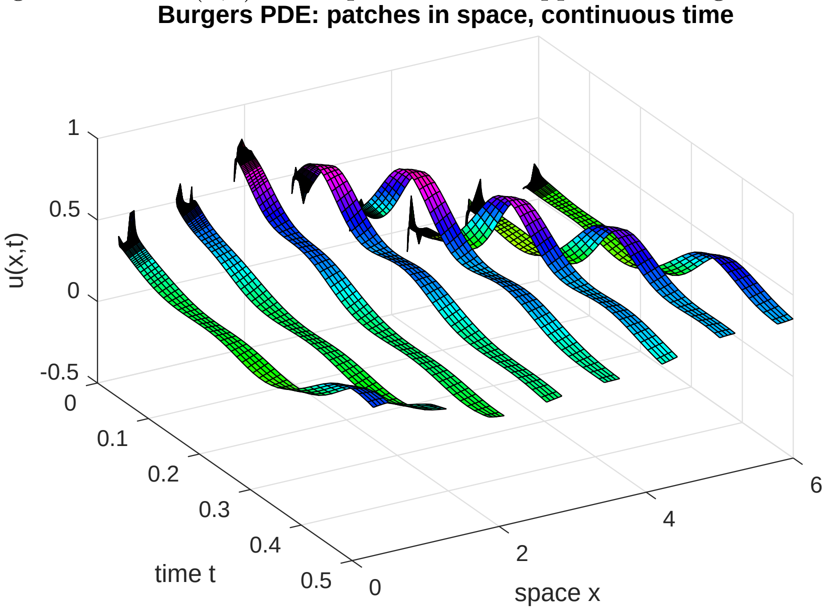
Simulate in time using a standard stiff integrator and the interface function `patchsmooth1()` (??).

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

Plot the simulation using only the microscale values interior to the patches: either set x-edges to `nan` to leave the gaps; or use `patchEdgyInt1` to re-interpolate correct patch edge values and thereby join the patches. Figure 5 illustrates an example simulation in time generated by the patch scheme applied to Burgers' PDE.

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

Figure 5: field $u(x,t)$ of the patch scheme applied to Burgers' PDE.



```
surf(ts,patches.x(:),us)
view(60,40), colormap(0.8*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 5, then exit this function.

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

11.2 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(patches.nEdge==1 ...
      ,'multi-edge-value interp not yet implemented')
assert(2*patches.nEdge+1<=nSubP ...
      ,'too many edge values requested')
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
```

11.3 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
  if pre2023, dx = ratio*DX/(nSubP-1-pEI)*(2-pEI);
  else        ratio = dx/DX*(nSubP-1-pEI)/(2-pEI);  end
  patches.ratio=ratio;
```

In the case of macro-periodicity, precompute the weightings to interpolate field values for coupling. (Might sometime extend to coupling via derivative values.)

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

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'
  X=linspace(Xlim(1)+((nSubP-1)/2-Dom.bcOffset(1))*dx ...
              ,Xlim(2)-((nSubP-1)/2-Dom.bcOffset(2))*dx ,nPatch);
  DX=diff(X(1:2));
  width=(1+patches.EdgyInt)/2*(nSubP-1-patches.EdgyInt)*dx;
  if DX<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, $X_i \propto -cos(i\pi/N)$, but with the extreme edges aligned with the spatial domain boundaries, modified by the offset, and modified by possible 'boundary layers'.[9]

```
case 'chebyshev'
  halfWidth=dx*(nSubP-1)/2;
  X1 = Xlim(1)+halfWidth-Dom.bcOffset(1)*dx;
  X2 = Xlim(2)-halfWidth+Dom.bcOffset(2)*dx;
%  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`.

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

Assign the centre-patch coordinates.

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

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

[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??

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

Fourth, construct the microscale grid in each patch. Reshape the grid to be 4D to suit dimensions (micro,Vars,Ens,macro).

```
assert(patches.EdgyInt | mod(nSubP,2)==1, ...
    'configPatches1: nSubP must be odd')
i0=(nSubP+1)/2;
patches.x = reshape( dx*(-i0+1:i0-1)'+X ,nSubP,1,1,nPatch);
```

11.4 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()`.

```
nE = patches.nEnsem;
patches.le = 1:nE;
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.

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

```
  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.

```
    patches.nEnsem = mx;
    patches.cs = nan(nx-1,nc,mx);
    for i = 1:mx
        is = (i:i+nx-2);
        patches.cs(:,:,i) = cs(is,:);
    end
    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.

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

```
if 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.*[10]

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

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

```
  pari = 1;
  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.

```
  switch pari
    case 1, patches.x=codistributed(patches.x,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
```

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

Couples 2D patches across 2D space by computing their edge values via macroscale interpolation. Research (**Roberts2011a**; **Bunder2019c**) indicates

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

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`.

```
function u = patchEdgeInt2(u,patches)
if nargin<2, global patches, end
%disp('**** Invoking new patchEdgeInt2')
```

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` $\cdot$ `nPatch1` $\cdot$ `nPatch2` multiscale spatial grid on the `nPatch1` $\cdot$ `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.

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

  - `.Cwtsr` and `.Cwtsl` are the coupling coefficients for finite width interpolation in both the x, y-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).
  - `.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 (and corner vales set to `NaN`).

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

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

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

```
[~,ny,~,~,~,Ny] = size(patches.y);
[nx,~,~,~,Nx,~] = size(patches.x);
nEnsem = patches.nEnsem;
nVars = round(numel(u)/numel(patches.x)/numel(patches.y)/nEnsem);
assert(numel(u) == nx*ny*Nx*Ny*nVars*nEnsem ...
  ,'patchEdgeInt2: input u has wrong size for parameters')
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 their four immediate neighbours.

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

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

```
i0 = round((nx+1)/2);
j0 = round((ny+1)/2);
%disp('finished common preamble')
```

12.1 Periodic macroscale interpolation schemes

```
if patches.periodic
```

Get the size ratios of the patches.

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

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
```

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-edge values
    Ux = u([2 nx-1],2:(ny-1),:,:,I,J);
    Uy = u(2:(nx-1),[2 ny-1],:,:,I,J);
  else % interpolate centre-cross values
    Ux = u(i0,2:(ny-1),:,:,I,J);
    Uy = u(2:(nx-1),j0,:,:,I,J);
  end;%if patches.EdgyInt
```

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

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

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

```
  if patches.parallel
    dmux = zeros(szUx0,patches.codist); % 7D
    dmuy = zeros(szUy0,patches.codist); % 7D
  else
    dmux = zeros(szUx0); % 7D
    dmuy = zeros(szUy0); % 7D
  end%if patches.parallel
```

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

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

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

```
  for k = 3:ordCC
    dmux(:,:,:,:,I,:,k)   =     dmux(:,:,:,:,Ip,:,k-2) ...
      -2*dmux(:,:,:,:,I,:,k-2) +dmux(:,:,:,:,Im,:,k-2);
    dmuy(:,:,:,:,:,J,k)   =     dmuy(:,:,:,:,:,Jp,k-2) ...
      -2*dmuy(:,:,:,:,:,J,k-2) +dmuy(:,:,:,:,:,Jm,k-2);
  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,:) ...
  = Ux(1,:,:,:,:,:)*(1-patches.stag) ...
    +sum( shiftdim(patches.Cwtsr(:,1),-6).*dmux(1,:,:,:,:,:,:) ,7);
u(1 ,2:(ny-1),:,patches.le,I,:) ...
  = Ux(k,:,:,:,:,:)*(1-patches.stag) ...
    +sum( shiftdim(patches.Cwtsl(:,1),-6).*dmux(k,:,:,:,:,:,:) ,7);
u(2:(nx-1),ny,:,patches.to,:,J) ...
  = Uy(:,1,:,:,:,:)*(1-patches.stag) ...
    +sum( shiftdim(patches.Cwtsr(:,2),-6).*dmuy(:,1,:,:,:,:,:) ,7);
u(2:(nx-1),1 ,:,patches.bo,:,J) ...
  = Uy(:,k,:,:,:,:)*(1-patches.stag) ...
    +sum( shiftdim(patches.Cwtsl(:,2),-6).*dmuy(:,k,:,:,:,:,:) ,7);
u([1 nx],[1 ny],:,:,:,:)=nan; % remove corner values
```

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

```
else% patches.ordCC<=0, spectral interpolation
%disp('executing spectral interpolation')
```

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

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

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

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

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

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

```
u(nx,iy,:,:,:,:) = uclean( ifft(ifft( ...
    Cle.*exp(1i*(stagShift+krx))  ,[],5),[],6) );
u( 1,iy,:,:,:,:) = uclean( ifft(ifft( ...
    Cri.*exp(1i*(stagShift-krx))  ,[],5),[],6) );
u(ix,ny,:,:,:,:) = uclean( ifft(ifft( ...
    Cbo.*exp(1i*(stagShift+kry))  ,[],5),[],6) );
u(ix, 1,:,:,:,:) = uclean( ifft(ifft( ...
    Cto.*exp(1i*(stagShift-kry))  ,[],5),[],6) );
end% if ordCC>0 else, so spectral
```

12.2 Non-periodic macroscale interpolation

```
else% patches.periodic false
%disp('executing new non-periodic code')
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 Fx and Fy (7D) for interpolating to left/right edges and top/bottom edges, respectively. 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)
```

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-edge values are because their values are to interpolate to the opposite edge of each patch. (Have no plans to implement core averaging as yet.)

```
  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 = patches.x([nx-1 2],:,:,:,:,:);
  else % interpolate mid-patch cross-patch values
    F(:,:,:,:,:,:,1) = u(i0,iy,:,:,:,:);
    X = patches.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 = patches.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
```

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,:,:,:,:,:);
```

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

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

12.2.3 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, may override their computed interpolation values with `NaN`.

```
%u( 1,:,:,:, 1,:) = nan;
%u(nx,:,:,:,Nx,:) = nan;
%u(:, 1,:,:,:, 1) = nan;
%u(:,ny,:,:,:,Ny) = nan;
```

End of the non-periodic interpolation code.

```
%disp('finished new non-periodic code')
end%if patches.periodic else
```

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

```
end% function patchEdgeInt2
```

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

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys2()`. Section 13.1 lists an example of its use.

```
function patches = configPatches2(fun,Xlim,Dom ...
    ,nPatch,ordCC,dx,nSubP,varargin)
```

Input If invoked with no input arguments, then executes an example of simulating a nonlinear diffusion PDE relevant to the lubrication flow of a thin layer of fluid—see Section 13.1 for an example code.

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)`, that computes time-derivatives (or time-steps) of quantities on the 2D micro-grid within all the 2D patches.

- `Xlim` array/vector giving the rectangular macro-space domain of the computation, namely [`Xlim(1)`,`Xlim(2)`] × [`Xlim(3)`,`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 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 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 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 `'usergive` 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 `'usergive` 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. 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 must be odd so that there is/are centre-patch micro-grid point/lines in each patch.

- `nEdge` (not yet implemented), *optional*, default=1, for each patch, the number of edge values set by interpolation at the edge regions of each patch. The default is one (suitable for microscale lattices with only nearest neighbour interactions).

- `EdgyInt`, true/false, *optional*, default=false. If true, then interpolate to left/right/top/bottom edge-values from right/left/bottom/top next-to-edge values. If false or omitted, then interpolate from centre 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 2/3D array of microscale heterogeneous coefficients to be used by the given microscale `fun` in each patch. Say the given array `cs` is of size $m_x \times m_y \times n_c$, where n_c is the number of different sets of coefficients. For example, in heterogeneous diffusion, $n_c = 2$ for the diffusivities in the *two* different spatial directions (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.
  - 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.[12]

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

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

- `.ordCC` is the specified order of inter-patch coupling.

- `.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.

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

- `.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`.

- `.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` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.

- `.le`, `.ri`, `.bo`, `.to` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.

- `.cs` either

  - `[]` 0D, or
  - if `nEnsem` $= 1$, (`nSubP(1)` $- 1) \times$ (`nSubP(2)` $- 1) \times n_c$ 3D array of microscale heterogeneous coefficients, or
  - if `nEnsem` > 1, (`nSubP(1)`$-1)\times$(`nSubP(2)`$-1)\times n_c \times m_x m_y$ 4D array of $m_x m_y$ ensemble of phase-shifts of the microscale heterogeneous coefficients.

- `.parallel`, logical: true if patches are distributed over multiple CPUs/ cores for the Parallel Computing Toolbox, otherwise false (the default is to activate the *local* pool).

- `.codist`, *optional*, describes the particular parallel distribution of arrays over the active parallel pool.

13.1 If no arguments, then execute an example

```
if nargin==0
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 (arrows indicate 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). **Roberts2011a** established that this scheme is consistent with the PDE (as the patch spacing decreases).

```
global patches
patches = configPatches2(@nonDiffPDE,[-3 3 -2 2], [] ...
    , [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)));
```

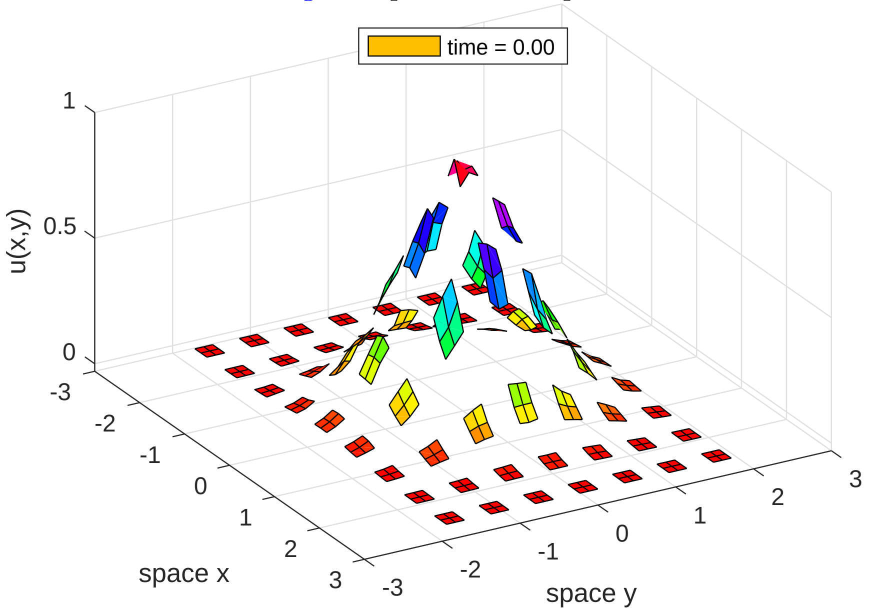
Initiate a plot of the simulation using only the microscale values interior to the patches: optionally set x and y-edges to `nan` to leave the gaps between patches.

```
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 6 while the simulation computes.

```
u = reshape(permute(squeeze(u0) ...
    ,[1 3 2 4]), [numel(x) numel(y)]);
hsurf = surf(x(:),y(:),u');
```

Figure 6: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to a nonlinear 'diffusion' PDE*: Figure 7 plots the computed field at time $t = 3$.*

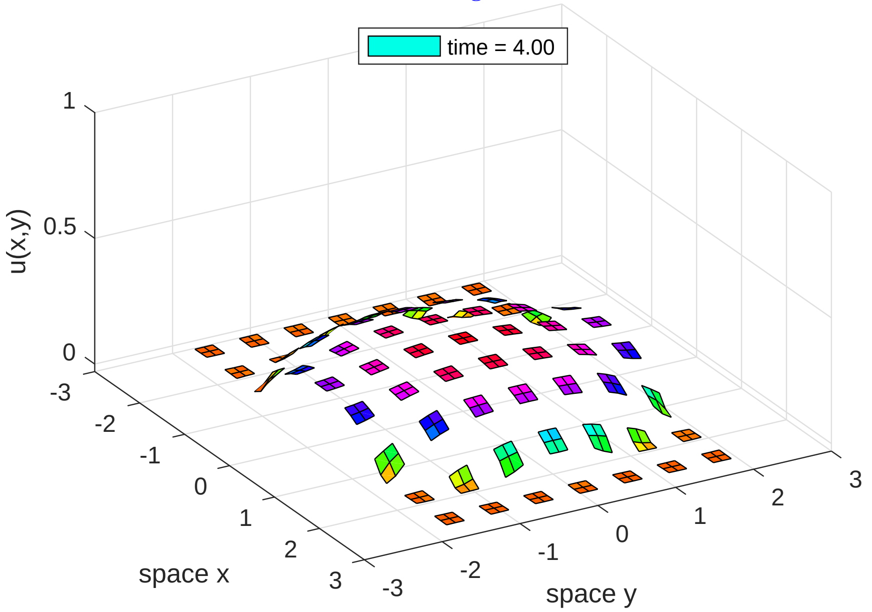


```
axis([-3 3 -3 3 -0.03 1]), view(60,40)
legend('time = 0.00','Location','north')
xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
colormap(hsv)
ifOurCf2eps([mfilename 'ic'])
```

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 (**Maclean2020a**). Ask for output at non-uniform times because the diffusion slows.

```
disp('Wait to simulate nonlinear diffusion h_t=(h^3)_xx+(h^3)_yy')
drawnow
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@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(:));
```

Figure 7: field $u(x, y, t)$ at time $t = 3$ of the patch scheme applied to a nonlinear 'diffusion' PDE *with initial condition in Figure 6.*



```
end
```

Animate the computed simulation to end with Figure 7. Use `patchEdgeInt2` to interpolate patch-edge values (but not corner values, and even if not drawn).

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

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'); % 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); % not yet implemented
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;
```

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(patches.nEdge==1 ...
      ,'multi-edge-value interp not yet implemented')
assert(all(2*patches.nEdge<nSubP) ...
      ,'too many edge values requested')
%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` vector.

```
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 subsequently set corresponding defaults for others fields.

```
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.

```
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 Dirichlet boundaries if no `bcOffset` specified when needed. Do so for both directions independently.

```
patches.periodic=false;
for p=1:2
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
case {'equispace','chebyshev'}
    if ~isfield(Dom,'bcOffset'), Dom.bcOffset=zeros(2,2); end
    % for mixed with usergiven, following should still work
    if numel(Dom.bcOffset)==1
        Dom.bcOffset=repmat(Dom.bcOffset,2,2); end
    if numel(Dom.bcOffset)==2
        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
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
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.

```
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.

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

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
%  sizedx=size(dx), sizenSubP=size(nSubP)
  if pre2023, dx(q) = ratio(q)*DQ/(nSubP(q)-1-pEI)*(2-pEI);
  else        ratio(q) = dx(q)/DQ*(nSubP(q)-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'.[13]

```
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`.

```
  width=(1+patches.EdgyInt)/2*(nSubP(q)-1-patches.EdgyInt)*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
  if DQmin<width*0.999999
     warning('too many Chebyshev patches (mid-domain overlap)')
     end
```

Assign the centre-patch coordinates.

```
  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 ];
```

[13] 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??

The user-given case is entirely up to a user to specify, we just ensure it has 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. Reshape the grid to be 6D to suit dimensions (micro,Vars,Ens,macro).

```
nSubP = reshape(nSubP,1,2); % force to be row vector
assert(patches.EdgyInt | all(mod(nSubP,2)==1), ...
    'configPatches2: nSubP must be odd')
i0 = (nSubP(1)+1)/2;
patches.x = reshape( dx(1)*(-i0+1:i0-1)'+X ...
                   ,nSubP(1),1,1,1,nPatch(1),1);
```

Next the y-direction.

```
i0 = (nSubP(2)+1)/2;
patches.y = reshape( dx(2)*(-i0+1:i0-1)'+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. (Might sometime extend to coupling via derivative values.)

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

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

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

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

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

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

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

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

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

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

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

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

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

```
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.*[14]

```
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
```

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

14 `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 (**Roberts2011a**; **Bunder2019c**), 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. [15]

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

```
function u = patchEdgeInt3(u,patches)
if nargin<2, global patches, end
%disp('**** Invoking new patchEdgeInt3')
```

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` $\times 1\times 1\times 1\times 1\times$ `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` $\times 1\times 1\times 1\times 1\times$ `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 i, but may be variable spaced in macroscale index I.
  - `.z` is similarly $1\times 1\times$ `nSubP3` $\times 1\times 1\times 1\times 1\times$ `nPatch3` array of the spatial locations z_{kK} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index i, but may be variable spaced in macroscale index I.

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

  - `.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).
  - `.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 (edge and corner vales set to `NaN`).

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

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

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

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

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 their six immediate neighbours.

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

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

```
i0 = round((nx+1)/2);
j0 = round((ny+1)/2);
k0 = round((nz+1)/2);
%disp('finished common preamble')
```

14.1 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);
```

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
```

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
    Ux = u([2 nx-1],2:(ny-1),2:(nz-1),:,:,I,J,K);
    Uy = u(2:(nx-1),[2 ny-1],2:(nz-1),:,:,I,J,K);
    Uz = u(2:(nx-1),2:(ny-1),[2 nz-1],:,:,I,J,K);
  else % interpolate centre-cross values
    Ux = u(i0,2:(ny-1),2:(nz-1),:,:,I,J,K);
    Uy = u(2:(nx-1),j0,2:(nz-1),:,:,I,J,K);
    Uz = u(2:(nx-1),2:(ny-1),k0,:,:,I,J,K);
  end;%if patches.EdgyInt
```

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

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

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

```
  if patches.parallel
    dmux = zeros(szUx0,patches.codist); % 9D
    dmuy = zeros(szUy0,patches.codist); % 9D
    dmuz = zeros(szUz0,patches.codist); % 9D
  else
    dmux = zeros(szUx0); % 9D
    dmuy = zeros(szUy0); % 9D
    dmuz = zeros(szUz0); % 9D
  end%if patches.parallel
```

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

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

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

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

Interpolate macro-values to be Dirichlet face values for each patch (Roberts 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`.

```
k=1+patches.EdgyInt; % use centre or two faces
u(nx,2:(ny-1),2:(nz-1),:,patches.ri,I,:,:) ...
```

```
  = Ux(1,:,:,:,:,:,:,:)*(1-patches.stag) ...
  +sum( shiftdim(patches.Cwtsr(:,1),-8).*dmux(1,:,:,:,:,:,:,:,:) ,9);
u(1 ,2:(ny-1),2:(nz-1),:,patches.le,I,:,:) ...
  = Ux(k,:,:,:,:,:,:,:)*(1-patches.stag) ...
  +sum( shiftdim(patches.Cwtsl(:,1),-8).*dmux(k,:,:,:,:,:,:,:,:) ,9);
u(2:(nx-1),ny,2:(nz-1),:,patches.to,:,J,:) ...
  = Uy(:,1,:,:,:,:,:,:)*(1-patches.stag) ...
  +sum( shiftdim(patches.Cwtsr(:,2),-8).*dmuy(:,1,:,:,:,:,:,:,:) ,9);
u(2:(nx-1),1 ,2:(nz-1),:,patches.bo,:,J,:) ...
  = Uy(:,k,:,:,:,:,:,:)*(1-patches.stag) ...
  +sum( shiftdim(patches.Cwtsl(:,2),-8).*dmuy(:,k,:,:,:,:,:,:,:) ,9);
u(2:(nx-1),2:(ny-1),nz,:,patches.fr,:,:,K) ...
  = Uz(:,:,1,:,:,:,:,:)*(1-patches.stag) ...
  +sum( shiftdim(patches.Cwtsr(:,3),-8).*dmuz(:,:,1,:,:,:,:,:,:) ,9);
u(2:(nx-1),2:(ny-1),1 ,:,patches.ba,:,:,K) ...
  = Uz(:,:,k,:,:,:,:,:)*(1-patches.stag) ...
  +sum( shiftdim(patches.Cwtsl(:,3),-8).*dmuz(:,:,k,:,:,:,:,:,:) ,9);
```

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

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

```
 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
```

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

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

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

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

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

```
u(nx,iy,iz,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
    Cle.*exp(1i*(stagShift+krx))  ,[],6),[],7),[],8) );
u( 1,iy,iz,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
    Cri.*exp(1i*(stagShift-krx))  ,[],6),[],7),[],8) );
u(ix,ny,iz,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
    Cbo.*exp(1i*(stagShift+kry))  ,[],6),[],7),[],8) );
u(ix, 1,iz,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
    Cto.*exp(1i*(stagShift-kry))  ,[],6),[],7),[],8) );
u(ix,iy,nz,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
    Cba.*exp(1i*(stagShift+krz))  ,[],6),[],7),[],8) );
u(ix,iy, 1,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
    Cfr.*exp(1i*(stagShift-krz))  ,[],6),[],7),[],8) );
end% if ordCC>0 else, so spectral
```

14.2 Non-periodic macroscale interpolation

```
else% patches.periodic false
%disp('executing new non-periodic code')
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 `Fx`, `Fy` and `Fz` (9D) for interpolating to left/right faces, top/bottom faces, and front/back faces, respectively. 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;
```

14.2.1 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. (Have no plans to implement core averaging as yet.)

```
  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 = patches.x([nx-1 2],:,:,:,:,:,:,:);
  else % interpolate mid-patch cross-patch values
    F(:,:,:,:,:,:,:,:,1) = u(i0,iy,iz,:,:,:,:,:);
    X = patches.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`.

```
Xface = patches.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,:,:,:,:,:,:,:);
```

14.2.2 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 = patches.y(:,[ny-1 2],:,:,:,:,:,:);
  else % interpolate mid-patch cross-patch values
    F(:,:,:,:,:,:,:,:,1) = u(:,j0,iz,:,:,:,:,:);
    Y = patches.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 = patches.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,:,:,:,:,:,:);
```

14.2.3 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 = patches.z(:,:,[nz-1 2],:,:,:,:,:);
  else % interpolate mid-patch cross-patch values
    F(:,:,:,:,:,:,:,:,1) = u(:,:,k0,:,:,:,:,:);
    Z = patches.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 = patches.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,:,:,:,:,:);
```

14.2.4 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, may override their computed interpolation values with `NaN`.

```
%u( 1,:,:,:,:, 1,:,:) = nan;
%u(nx,:,:,:,:,Nx,:,:) = nan;
%u(:, 1,:,:,:,:, 1,:) = nan;
%u(:,ny,:,:,:,:,Ny,:) = nan;
%u(:,:, 1,:,:,:,:, 1) = nan;
%u(:,:,ny,:,:,:,:,Nz) = nan;
```

End of the non-periodic interpolation code.

```
%disp('finished new non-periodic code')
end%if patches.periodic else
```

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

```
end% function patchEdgeInt3
```

15 `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 15.1 and **??** list examples of its use.

```
function patches = configPatches3(fun,Xlim,Dom ...
    ,nPatch,ordCC,dx,nSubP,varargin)
```

Input If invoked with no input arguments, then executes an example of simulating a heterogeneous wave PDE—see Section 15.1 for an example code.

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)`, that computes time-derivatives (or time-steps) of quantities on the 3D micro-grid within all the 3D patches.
- `Xlim` array/vector giving the rectangular-cuboid macro-space domain of the computation: namely [`Xlim(1)`,`Xlim(2)`] × [`Xlim(3)`,`Xlim(4)` × [`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 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 `'usergive` 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 `'usergive` 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 `'usergive` 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. 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 must be odd so that there is/are centre-patch micro-grid point/ planes in each patch.

- `'nEdge'` (not yet implemented), *optional*, default=1, for each patch, the number of face values set by interpolation at the face regions of each patch. The default is one (suitable for microscale lattices with only nearest neighbour interactions).

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

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

- `'hetCoeffs'`, *optional*, default empty. Supply a 3/4D array of microscale heterogeneous coefficients to be used by the given microscale `fun` in each patch. Say the given array `cs` is of size $m_x \times m_y \times m_z \times n_c$, where n_c is the number of different arrays of coefficients. For example, in heterogeneous diffusion, $n_c = 3$ for the diffusivities in the *three* different spatial directions (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.

  – 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.[16]

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

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

- `.ordCC` is the specified order of inter-patch coupling.

- `.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

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

(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` is, for each patch, the number of face values set by interpolation at the face regions of each patch.
- `.le`, `.ri`, `.bo`, `.to`, `.ba`, `.fr` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.
- `.cs` either
  - `[]` 0D, or
  - if `nEnsem` $= 1$, (`nSubP(1)`-1)$\times$(`nSubP(2)`-1)$\times$(`nSubP(3)`-1)$\times n_c$ 4D array of microscale heterogeneous coefficients, or
  - if `nEnsem` > 1, (`nSubP(1)` -1) $\times$ (`nSubP(2)` -1) $\times$ (`nSubP(3)` -1) $\times$ $n_c \times m_x m_y m_z$ 5D array of $m_x m_y m_z$ ensemble of phase-shifts of the microscale heterogeneous coefficients.
- `.parallel`, logical: true if patches are distributed over multiple CPUs/ cores for the Parallel Computing Toolbox, otherwise false (the default is to activate the *local* pool).
- `.codist`, *optional*, describes the particular parallel distribution of arrays over the active parallel pool.

15.1 If no arguments, then execute an example

```
if nargin==0
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 (arrows indicate 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.

```
mPeriod = [2 2 2];
cHetr = exp(0.9*randn([mPeriod 3]));
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.

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

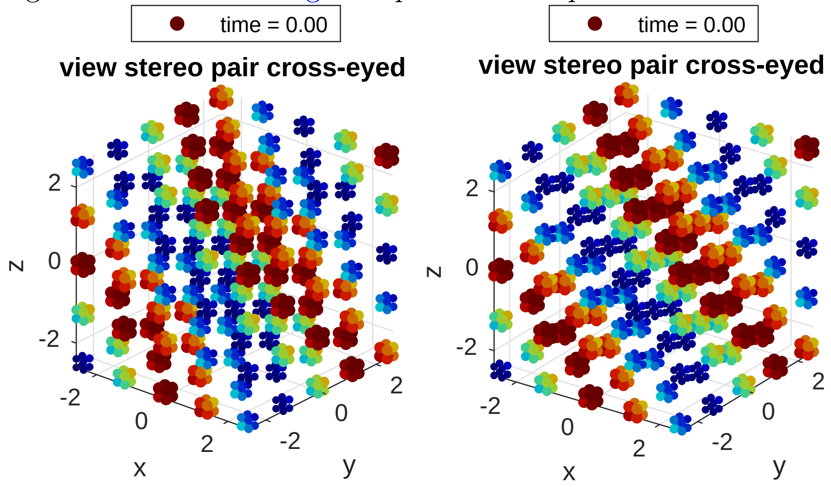
Set a wave initial state using auto-replication of the spatial grid, and as Figure 8 shows. This wave propagates diagonally across space. Concatenate the two u, v-fields to be the two components of the fourth dimension.

```
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 (**Maclean2020a**).

```
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);
```

Figure 8: initial field $u(x,y,z,t)$ at time $t = 0$ of the patch scheme applied to a heterogeneous wave PDE*: Figure 9 plots the computed field at time $t = 6$.*



```
    lsode_options('relative tolerance',1e-4);
    [ts,us] = odeOcts(@patchSys3,[0 1 2],uv0(:));
end
```

Animate the computed simulation to end with Figure 9. Use `patchEdgeInt3` to obtain patch-face values (but not edge nor corner values, and even if not drawn) in order to most easily reconstruct the array data structure.

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

Optionally save the initial condition to graphic file for Figure 6, 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'])
```

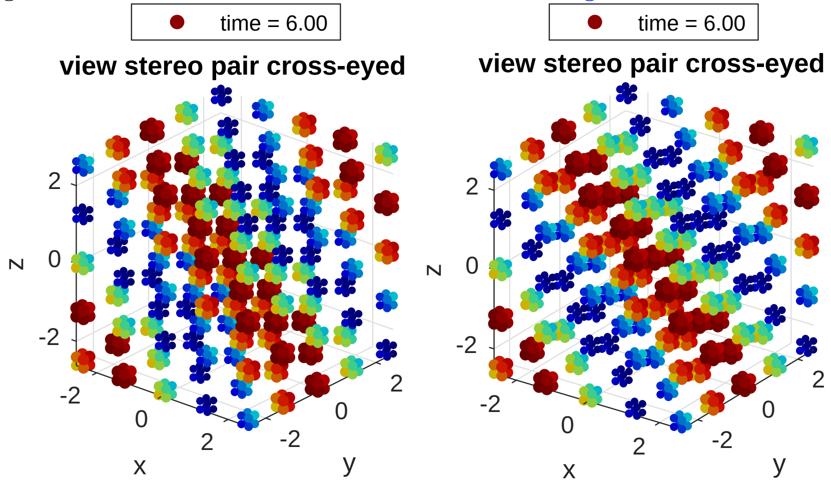
Upon finishing execution of the example, exit this function.

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

15.2 Parse input arguments and defaults

```
p = inputParser;
fnValidation = @(f) isa(f, 'function_handle'); %test for fn name
```

Figure 9: 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 8.*



```
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); % not yet implemented
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;
```

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) ...
      ,'third pair of Xlim must be ordered increasing')
assert(patches.nEdge==1 ...
      ,'multi-edge-value interp not yet implemented')
assert(all(2*patches.nEdge<nSubP) ...
      ,'too many edge values requested')
%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` vector.

```
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 subsequently set corresponding defaults for others fields.

```
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.

```
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(2,2); 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
```

15.3 The code to make patches

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

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

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

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

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=1:3
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
%  sizedx=size(dx), sizenSubP=size(nSubP)
  if pre2023, dx(q) = ratio(q)*DQ/(nSubP(q)-1-pEI)*(2-pEI);
  else        ratio(q) = dx(q)/DQ*(nSubP(q)-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'.[17]

```
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));
```

[17] 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??

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`.

```
  width=(1+patches.EdgyInt)/2*(nSubP(q)-1-patches.EdgyInt)*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%if
```

Assign the centre-patch coordinates.

```
  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 ensure it has the correct shape of a row.

```
case 'usergiven'
  if q==1, Q = reshape(Dom.X,1,[]); end
  if q==2, Q = reshape(Dom.Y,1,[]); end
  if q==3, Q = reshape(Dom.Z,1,[]); end
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
if q==3, Z=Q; end
end%for q
```

Construct the micro-grids Construct the microscale in each patch. Reshape the grid to be 8D to suit dimensions (micro,Vars,Ens,macro).

```
nSubP = reshape(nSubP,1,3); % force to be row vector
assert(patches.EdgyInt | all(mod(nSubP,2)==1), ...
    'configPatches3: nSubP must be odd')
```

```
i0 = (nSubP(1)+1)/2;
patches.x = reshape( dx(1)*(-i0+1:i0-1)'+X ...
                   ,nSubP(1),1,1,1,1,nPatch(1),1,1);
i0 = (nSubP(2)+1)/2;
patches.y = reshape( dx(2)*(-i0+1:i0-1)'+Y ...
                   ,1,nSubP(2),1,1,1,1,nPatch(2),1);
i0 = (nSubP(3)+1)/2;
patches.z = reshape( dx(3)*(-i0+1:i0-1)'+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. (Might sometime extend to coupling via derivative values.)

```
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
```

15.4 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;
```

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

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

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

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

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

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

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

```
    mmx=(0:mx-1)'; mmy=0:my-1; mmz=shiftdim(0:mz-1,-1);
    le = mod(mmx+mod(nx-2,mx),mx)+1;
    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. Need to justify this and the arbitrary threshold more carefully??

```
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.*[18]

```
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.

[18]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
```

References

Abdulle, Assyr, Doghonay Arjmand, and Edoardo Paganoni (2020). *A parabolic local problem with exponential decay of the resonance error for numerical homogenization.* Tech. rep. Institute of Mathematics, École Polytechnique Fédérale de Lausanne (cit. on p. 27).

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. 37, 59, 85).

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. 37, 38, 41, 63, 89).

Combescure, Christelle (Nov. 2022). “Selecting Generalized Continuum Theories for Nonlinear Periodic Solids Based on the Instabilities of the Underlying Microstructure”. In: *Journal of Elasticity*. ISSN: 1573-2681. DOI: 10.1007/s10659-022-09949-6 (cit. on p. 15).

Eckhardt, Daniel and Barbara Verfürth (Oct. 2022). *Fully discrete Heterogeneous Multiscale Method for parabolic problems with multiple spatial and temporal scales*. Tech. rep. http://arxiv.org/abs/2210.04536 (cit. on pp. 3, 10).

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. 37).

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. 37, 41, 63, 89).

Wikipedia (2022). *Divided differences*. https://en.wikipedia.org/wiki/Divided_differences (visited on 12/28/2022) (cit. on pp. 44, 66, 93).