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

A. J. Roberts*

February 1, 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 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).

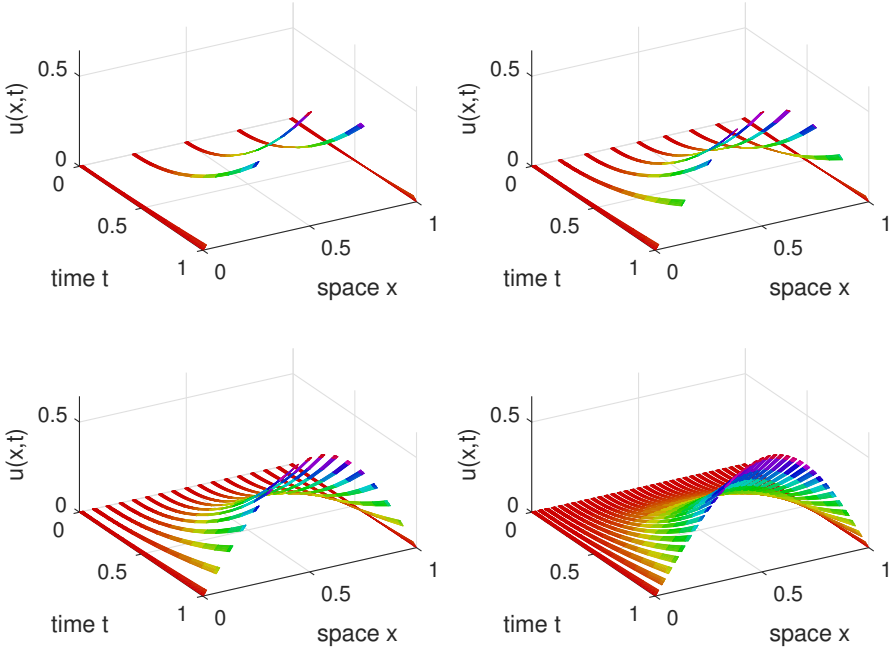
patch spacing H	0.25	0.12	0.06	0.03
exp-sine-forcing error	8E−3	2E−3	3E−4	2E−5
parabolic-forcing error	9E−9	4E−9	1E−9	0.06E−9

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.

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).




```
clear all
%global OurCf2eps, OurCf2eps=true %option to save plots
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],'equispace',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 exp-sinusoidal.

```
    if 0 % given forcing is exact
      patches.f1=2*( patches.x-patches.x.^2 );
      patches.f2=2*0.5+0*patches.x;
    else% simple exp.sine forcing
      patches.f1=sin(pi*patches.x).*exp(patches.x);
      patches.f2=pi/2*sin(pi*patches.x).*exp(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 `nan`s in between patches. For the field values (which are rows in `us`) we need to reshape, permute, interpolate to get edge values, pad with `nan`s, and reshape again.

```
    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 plot
```

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')
ifOurCf2eps([mfilename 'Errs']) %optionally save plot
```

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

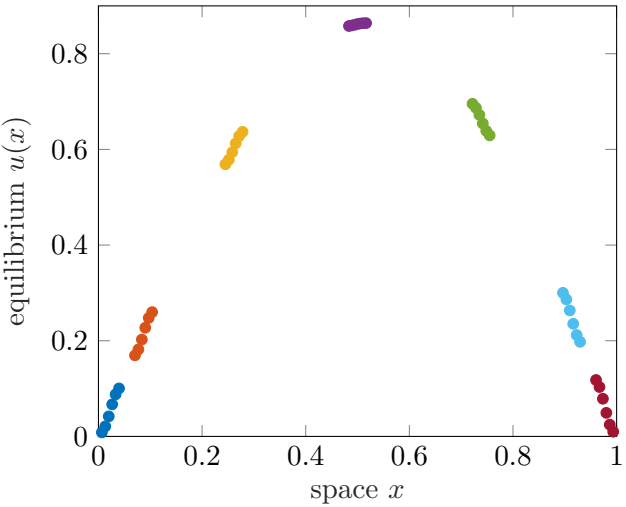


Figure 2: Equilibrium of the heterogeneous diffusion problem with forcing the same as that applied at time $t = 1$, and for relatively large $\epsilon = 0.04$ so we can see the patches. By default this code sets $\epsilon = 0.004$ whence the microscale heterogeneity and patches are tiny.

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 Eckhardt and Verfürth (2022) [§6.2.1].

```
clear all
%global OurCf2eps, OurCf2eps=true %option to save plots
mPeriod = 6
y = linspace(0,1,mPeriod+1)';
a = 1./(2-cos(2*pi*y(1:mPeriod)))
global microTimePeriod; microTimePeriod=0;
```

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 = 25 % 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 distri-

bution 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 exp-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 exp-sine forcing
  patches.f1 = sin(pi*patches.x).*exp(patches.x);
  patches.f2 = pi/2*sin(pi*patches.x).*exp(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 $u(x)$')
ifOurCf2tex(mfilename)%optionally save
```

2.1 One way 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 `EckhartEquilibErrs`: explore errors in equilibria of a 1D heterogeneous diffusion on small patches

Figure 2 finds the equilibrium, 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. Here we explore the errors as the number N of patches increases.

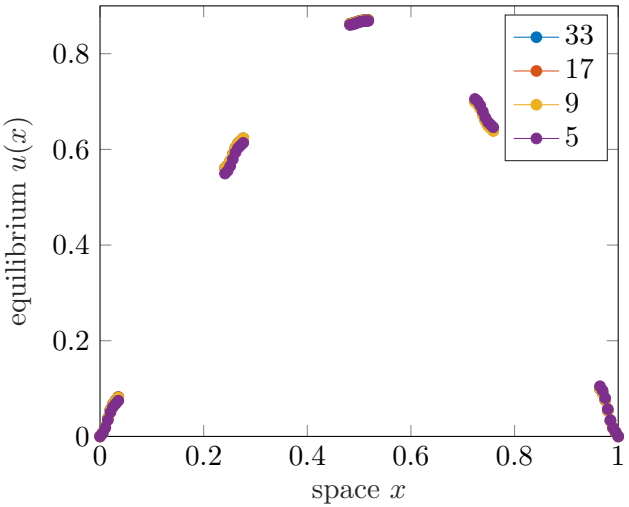


Figure 3: Equilibrium of the heterogeneous diffusion problem for relatively large $\epsilon = 0.03$ so we can see the patches. The solution is obtained with various numbers of patches, but we only compare solutions in these five common patches.

Find mean-abs errors to be the following:

	$N =$	5	9	17	33	65
equispace	second-order	8E−3	1E−2	1E−2	4E−3	9E−4
equispace	fourth-order	2E−3	7E−4	1E−4	9E−6	5E−7
equispace	sixth-order	2E−3	2E−5	4E−7	1E−8	2E−10
chebyshev	second-order	4E−2	6E−2	3E−2	2E−2	2E−2
chebyshev	fourth-order	9E−4	3E−3	6E−4	3E−4	2E−4
chebyshev	sixth-order	9E−4	3E−5	1E−5	4E−6	1E−6
usergiven	second-order	4E−2	6E−2	3E−2	9E−3	2E−3
usergiven	fourth-order	8E−4	3E−3	6E−4	4E−5	2E−6
usergiven	sixth-order	8E−4	3E−5	1E−5	2E−7	3E−9

For 'chebyshev' this assessment of errors is a bit dodgy as it is based only on the centre and boundary patches. The 'usergiven' distribution is for overlapping patches with Chebyshev distribution of centres—a spatial 'christmas tree'[1]. Curiously, and with above caveats, here my 'smart' chebyshev is the worst, the overlapping Chebyshev is good, but equispace is usually the best.

The above errors are for simple sin forcing. What if we make not so simple with exp modification of the forcing? The errors shown below are very little

[1]But the error assessment is with respect to finest patch-grid, no longer with a full domain solution

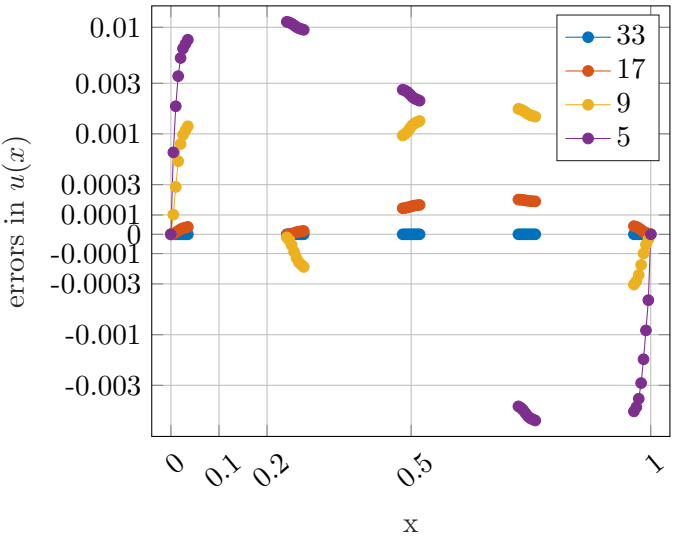


Figure 4: Errors in the equilibrium of the heterogeneous diffusion problem for relatively large $\epsilon = 0.03$. The solution is obtained with various numbers of patches, but we only plot the errors within these five common patches.

different (despite the magnitude of the solution being a little larger).

	$N =$	5	9	17	33	65
equispace	fourth-order	4E−3	7E−4	1E−4	8E−6	5E−7
chebyshev	fourth-order	7E−4	2E−3	5E−4	3E−4	1E−4
usergiven	fourth-order	2E−3	3E−3	5E−4	4E−5	2E−6

Clear, and initiate global patches. Choose the type of patch distribution to be either 'equispace', 'chebyshev', or 'usergiven'. Also set order of interpolation (fourth-order is good start).

```
clear all
global patches
%global OurCf2eps, OurCf2eps=true %option to save plots
switch 1
    case 1, Dom.type = 'equispace'
    case 2, Dom.type = 'chebyshev'
    case 3, Dom.type = 'usergiven'
    end% switch
ordInt = 4
```

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
z = (0.5:mPeriod)'/mPeriod;
a = 1./(2-cos(2*pi*z))
global microTimePeriod; microTimePeriod=0;
```

To use a hierarchy of patches with `nPatch` of 5, 9,17, ..., we need up to N patches plus one `dx` to fit into the domain interval. Cater for up to some full-domain simulation—can compute `log2Nmax` = 129 ($\epsilon = 0.008$) in a few seconds:

```
log2Nmax = 7 % 5 for plots, 7 for choice
nPatchMax=2^log2Nmax+1
```

Set the periodicity ϵ, and other microscale parameters.

```
nPeriodsPatch = 1 % any integer
nSubP = nPeriodsPatch*mPeriod+2 % for edgy int
epsilon = 1/(nPatchMax*nPeriodsPatch+1/mPeriod)
dx = epsilon/mPeriod
```

For various numbers of patches Choose five to be the coarsest number of patches. Want place to store common results for the solutions. Assign `Ps` to be the indices of the common patches: for equispace set to the five common patches, but for chebyshev the only common ones are the three centre and boundary-adjacent patches.

```
us=[]; xs=[]; nPs=[];
for log2N=log2Nmax:-1:2
    if log2N==log2Nmax
         Ps=linspace(1,nPatchMax ...
           ,5-2*all(Dom.type=='chebyshev'))
    else Ps=(Ps+1)/2
    end
```

Set the number of patches in $(0, 1)$:

```
    nPatch = 2^log2N+1
```

In the case of 'usergiven', we choose standard Chebyshev distribution of the centre of the patches, which involves overlapping of patches near the boundaries! (instead of the coded chebyshev which has a boundary layer of non-overlapping patches and a Chebyshev within the interior).

```
195     if all(Dom.type=='usergiven')
196       halfWidth=dx*(nSubP-1)/2;
197       X1 = 0+halfWidth; X2 = 1-halfWidth;
198       Dom.X = (X1+X2)/2-(X2-X1)/2*cos(linspace(0,pi,nPatch));
199     end
```

Configure the patches:

```
205     configPatches1(@heteroDiffF,[0 1],Dom,nPatch ...
206         ,ordInt,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.

```
215     if 0 %given forcing gives exact answers for ordInt=4 !
216       patches.f1 = 2*( patches.x-patches.x.^2 );
217       patches.f2 = 2*0.5+0*patches.x;
218     else% simple exp-sine forcing
219       patches.f1 = sin(pi*patches.x).*exp(patches.x);
220       patches.f2 = pi/2*sin(pi*patches.x).*exp(patches.x);
221     end%if
```

Solve for steady state Set initial guess of either zero or a subsample of the next finer solution, 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.

```
233     if log2N==log2Nmax
234       u0 = zeros(nSubP,1,1,nPatch);
235     else u0 = u0(:,:,:,1:2:end);
236     end
237     u0([1 end],:) = nan;
238     patches.i = find(~isnan(u0));
239     nVariables = numel(patches.i)
```

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

```
246     tic;
247     uSoln = fsolve(@theRes1,u0(patches.i) ...
248         ,optimoptions('fsolve','Display','off'));
249     fsolveTime = toc
```

Store the solution into the patches, and give magnitudes—Inf norm is max(abs()).

```
256     normSoln = norm(uSoln,Inf)
257     normResidual = norm(theRes1(uSoln),Inf)
258     u0(patches.i) = uSoln;
259     u0 = patchEdgeInt1(u0);
260     u0( 1 ,:,:, 1 ) = 0;
261     u0(end,:,:,end) = 0;
```

Concatenate the solution on common patches into stores.

```
267     us=cat(3,us,squeeze(u0(:,:,:,Ps)));
268     xs=cat(3,xs,squeeze(patches.x(:,:,:,Ps)));
269     nPs = [nPs;nPatch];
```

End loop. Check grids were aligned, then compute errors compared to the full-domain solution.

```
277 end%for log2N
278 assert(max(abs(reshape(diff(xs,1,3),[],1)))<1e-12,'x-coord failure')
279 errs = us-us(:,:,1);
280 meanAbsErrs = mean(abs(reshape(errs,[],size(us,3))))
281 ratioErrs = meanAbsErrs(2:end)./meanAbsErrs(1:end-1)
```

Plot solution in common patches First adjoin NaNs to separate patches, and reshape.

```
291 x=xs(:,:,1); u=us;
292 x(end+1,:)=nan; u(end+1,:)=nan;
293 u=reshape(u,numel(x),[]);
```

Reshape solution field.

```
299 figure(1),clf
300 plot(x(:),u,'.-'), legend(num2str(nPs))
301 xlabel('space $x$'),ylabel('equilibrium $u(x)$')
302 ifOurCf2tex([mfilename 'us'])%optionally save
```

Plot errors Use quasi-log axis to separate the errors.

```
310 err = u(:,1)-u;
311 figure(2), clf
```

```
h=plot(x(:),err,'.-'); legend(num2str(nPs))
quasiLogAxes(h,10,sqrt(prod(meanAbsErrs(2:3))))
xlabel('space $x$'), ylabel('errors in $u(x)$')
ifOurCf2tex(mfilename)%optionally save
```

3.1 `theRes1()`: 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=theRes1(u)
  global patches
  v=nan(size(patches.x));
  v(patches.i)=u;
  f=patchSys1(1,v(:),patches);
  f=f(patches.i);
end%function theRes1
```

4 `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].

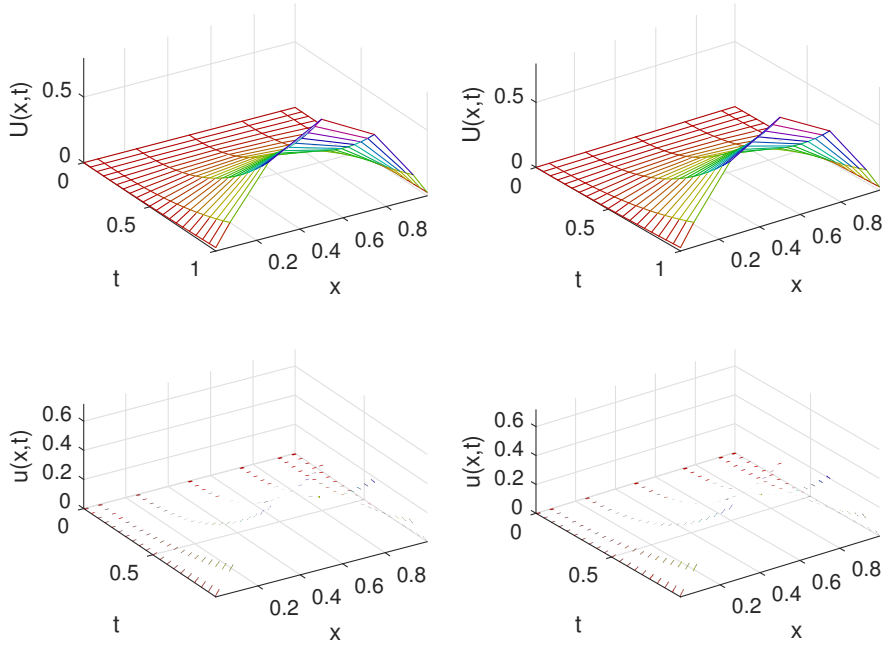
Suppose the spatial microscale lattice is at points x_i, with constant spacing dx. With dependent variables $u_i(t)$, simulate the microscale lattice forced diffusion system

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

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 5 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

Figure 5: 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.



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

4.1 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
%global OurCf2eps, OurCf2eps=true %option to save plots
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. Optionally save to Figs folder.

```
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
ifOurCf2eps(mfilename)
```

4.2 `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
```

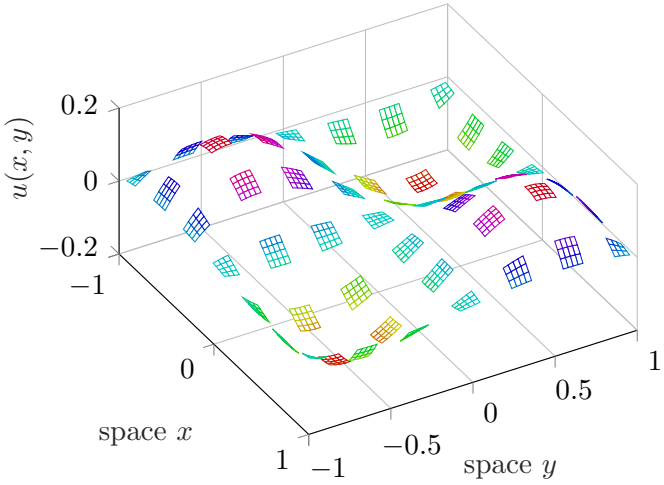


Figure 6: Equilibrium of the macroscale diffusion problem of Freese with Dirichlet zero-value boundary conditions (Section 5). The patch size is not small so we can see the patches.

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

Here we find the steady state $u(x, y)$, see Figure 6, to the heterogeneous PDE(inspired by Freese et al.[2] §5.2)

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

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

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

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 obtain $u = x(1 - x^2)y(1 - y^2)$ for which we code f later—as determined by this Reduce algebra code.

```
on gcd; factor sin;
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.

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

```
clear all
global patches
%global OurCf2eps, OurCf2eps=true %option to save plot
```

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.

```
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;
patches.i = find(~isnan(u0));
nVariables = numel(patches.i)
```

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

```
tic;
uSoln = fsolve(@theRes1,u0(patches.i) ...
        ,optimoptions('fsolve','Display','off'));
solnTime = toc
normResidual = norm(theRes1(uSoln))
normSoln = norm(uSoln)
normError = norm(uSoln-uAnal(patches.i))
```

Store the solution vector into the patches, and interpolate, but have not bothered to set boundary values so they stay NaN from the interpolation.

```
u0(patches.i) = uSoln;
u0 = patchEdgeInt2(u0);
```

Draw solution profile Separate patches with NaNs, then reshape arrays to suit 2D space surface plots.

```
figure(1), clf, colormap(0.8*hsv)
x(end+1,:,:)=nan;  u0(end+1,:,:)=nan;
y(:,end+1,:)=nan;  u0(:,end+1,:)=nan;
u = reshape(permute(squeeze(u0),[1 3 2 4]), [numel(x) numel(y)]);
```

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

```
mesh(x(:),y(:),u'); view(60,55)
xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')
ifOurCf2tex(mfilename)%optionally save
```

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
  i = 2:size(u,1)-1; % x interior points in a patch
  j = 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.

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

Or code some function variation around the boundary, such as a function of y on the left boundary, and a (constant) function of x at the top boundary.

```
199   if 0
200     u(1,:,:,:,1,:)=(1+patches.y)/2; % left edge of left patches
201     u(:,end,:,:,:,end)=1; % top edge of top patches
202   end%if
```

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

```
210   ut(i,j,:) ...
211   = 2*diff(u(:,j,:),2,1)/dx^2 +2*diff(u(i,:,:),2,2)/dy^2 ...
212    +2*patches.A(i,j,:).*( u(i+1,j+1,:) -u(i-1,j+1,:) ...
213      -u(i+1,j-1,:) +u(i-1,j-1,:) )/(4*dx*dy) ...
214    -patches.fu(i,j,:);
215 end%function monoscaleDiffForce2
```

5.2 `theRes1()`: 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.

```
227 function f=theRes1(u)
228   global patches
229   v=nan(size(patches.x+patches.y));
230   v(patches.i)=u;
231   f=patchSys2(0,v(:),patches);
232   f=f(patches.i);
233 end%function theRes1
```

Fin.

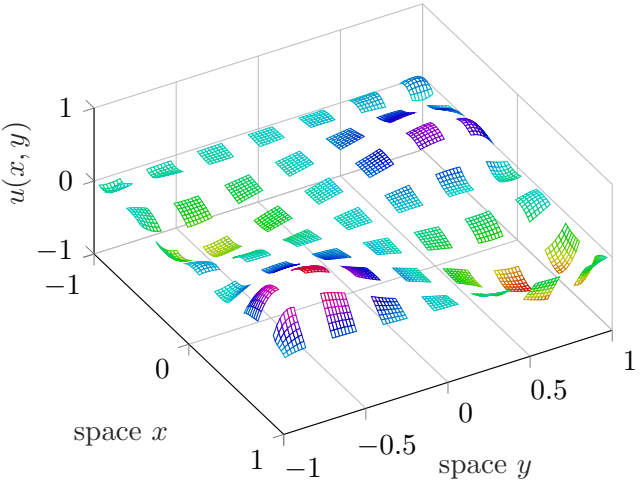


Figure 7: Equilibrium of the multiscale diffusion problem of Freese with Dirichlet zero-value boundary conditions (Section 6). The patch size is not small so we can see the patches and the sub-patch grid. The solution $u(x,y)$ is boringly smooth.

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.[3] §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
%global OurCf2eps, OurCf2eps=true %option to save plot
```

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.

[3] 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^(-6) % 4 or 5 to see the 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 7.1.

```
x = patches.x;  y = patches.y;
patches.fu = 100*(x+cos(3*pi*x)).*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;
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), and give magnitudes.

```
tic;
uSoln = fsolve(@theRes2,u0(patches.i) ...
        ,optimoptions('fsolve','Display','off'));
solveTime = toc
normResidual = norm(theRes2(uSoln))
normSoln = norm(uSoln)
```

Store the solution vector into the patches, and interpolate, but have not bothered to set boundary values so they stay NaN from the interpolation.

```
u0(patches.i) = uSoln;
u0 = patchEdgeInt2(u0);
```

Draw solution profile Separate patches with NaNs, then reshape arrays to suit 2D space surface plots.

```
figure(1), clf, colormap(0.8*hsv)
x(end+1,:,:)=nan;  u0(end+1,:,:)=nan;
y(:,end+1,:)=nan;  u0(:,end+1,:)=nan;
u = reshape(permute(squeeze(u0),[1 3 2 4]), [numel(x) numel(y)]);
```

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

```
mesh(x(:),y(:),u'); view(60,55)
xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')
ifOurCf2tex(mfilename)%optionally save
```

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
  i = 2:size(u,1)-1; % x interior points in a patch
  j = 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(i,j,:) ...
  = 2*diff(u(:,j,:),2,1)/dx^2 +2*diff(u(i,:,:),2,2)/dy^2 ...
   +2*patches.cs(i,j).*( u(i+1,j+1,:) -u(i-1,j+1,:) ...
     -u(i+1,j-1,:) +u(i-1,j-1,:) )/(4*dx*dy) ...
   -patches.fu(i,j,:);
end%function twoscaleDiffForce2
```

6.2 `theRes2()`: 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=theRes2(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 theRes2
```

7 `twoscaleDiffEquil2Errs`: errors in equilibria 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.[4] §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 some microscale period ϵ (here $\epsilon \approx 0.24, 0.12, 0.06, 0.03$), of

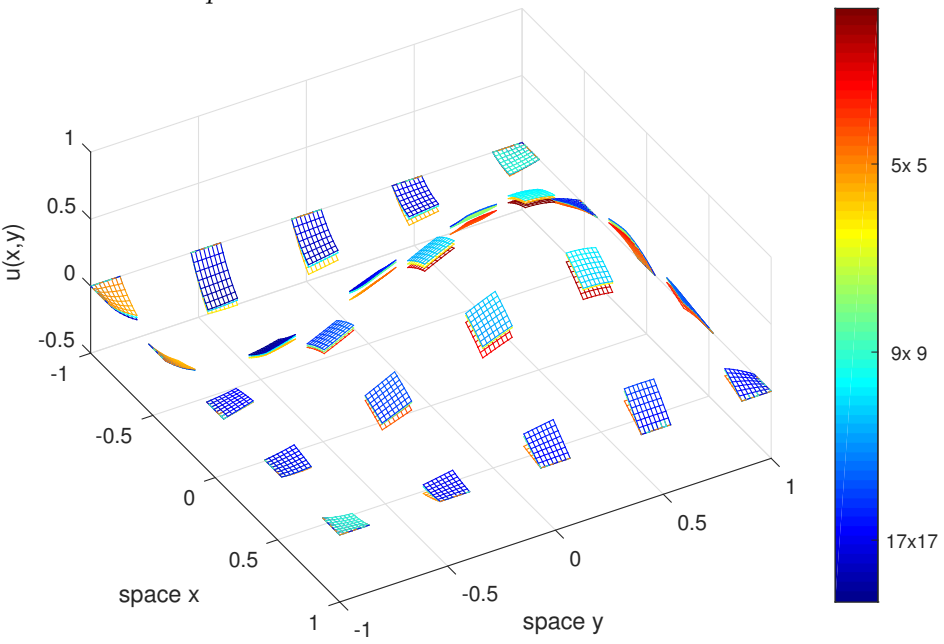
$$A := \begin{bmatrix} 2 & a \\ a & 2 \end{bmatrix} \quad \text{with } a := \sin(\pi x/\epsilon)\sin(\pi y/\epsilon),$$

and for forcing $f := 10(x + y + \cos \pi x)$ (for which the solution has magnitude up to one).[5]

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

[5] Freese et al. had forcing $f := (x + \cos 3\pi x)y^3$, but here we want smoother forcing so we get meaningful results in a minute or two computation.[6] For the same reason we do not

Figure 8: For various numbers of patches as indicated on the colorbar, plot the equilibrium of the multiscale diffusion problem of Freese with Dirichlet zero-value boundary conditions (Section 7). We only compare solutions only in these 25 common patches.



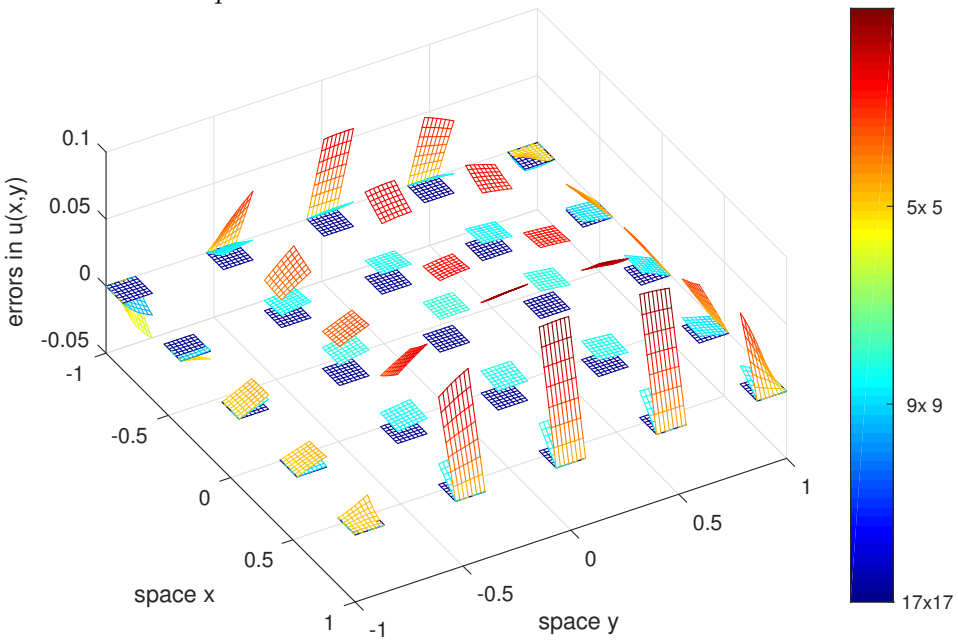
Here we explore the errors for increasing number N of patches (in both directions). Find mean-abs errors to be the following (for different orders of interpolation and patch distribution):

N	5	9	17	33
equispace, 2nd-order	6E−2	3E−2	1E−2	3E−3
equispace, 4th-order	3E−2	8E−3	7E−4	7E−5
chebyshev, 4th-order	1E−2	2E−2	6E−3	2E−3
usergiven, 4th-order	1E−2	2E−2	4E−3	n/a
equispace, 6th-order	3E−2	1E−3	1E−4	2E−5

Script start Clear, and initiate global patches. Choose the type of patch distribution to be either 'equispace', 'chebyshev', or 'usergiven'. Also set order of interpolation (fourth-order is good start).

invoke their smaller $\epsilon \approx 0.01$.

Figure 9: For various numbers of patches as indicated on the colorbar, plot the equilibrium of the multiscale diffusion problem of Freese with Dirichlet zero-value boundary conditions (Section 7). We only compare solutions only in these 25 common patches.




```
clear all
global patches
%global OurCf2eps, OurCf2eps=true %option to save plot
switch 1
    case 1, Dom.type = 'equispace'
    case 2, Dom.type = 'chebyshev'
    case 3, Dom.type = 'usergiven'
end% switch
ordInt = 4
```

First configure the patch system Establish the microscale heterogeneity has micro-period `mPeriod` on the spatial lattice. Then `configPatches2` replicates the heterogeneity as needed to fill each patch.

```
mPeriod = 6
z = (0.5:mPeriod)'/mPeriod;
```

```
A = sin(2*pi*z).*sin(2*pi*z');
```

To use a hierarchy of patches with `nPatch` of 5, 9, 17, ..., we need up to N patches plus one `dx` to fit into the domain interval. Cater for up to some full-domain simulation—can compute `log2Nmax` $= 5$ ($\epsilon = 0.06$) within minutes:

```
log2Nmax = 4 % >2 up to 6 OKish
nPatchMax=2^log2Nmax+1
```

Set the periodicity ϵ, and other microscale parameters.

```
nPeriodsPatch = 1 % any integer
nSubP = nPeriodsPatch*mPeriod+2 % for edgy int
epsilon = 2/(nPatchMax*nPeriodsPatch+1/mPeriod)
dx = epsilon/mPeriod
```

For various numbers of patches Choose five patches to be the coarsest number of patches. Define variables to store common results for the solutions from differing patches. Assign `Ps` to be the indices of the common patches: for equispace set to the five common patches, but for 'chebyshev' the only common ones are the three centre and boundary-adjacent patches.

```
us=[]; xs=[]; ys=[]; nPs=[];
for log2N=log2Nmax:-1:2
    if log2N==log2Nmax
         Ps=linspace(1,nPatchMax ...
           ,5-2*all(Dom.type=='chebyshev'))
    else Ps=(Ps+1)/2
    end
```

Set the number of patches in $(-1, 1)$:

```
    nPatch = 2^log2N+1
```

In the case of 'usergiven', we set the standard Chebyshev distribution of the patch-centres, which involves overlapping of patches near the boundaries! (instead of the coded chebyshev which has boundary layers of abutting patches, and non-overlapping Chebyshev between the boundary layers).

```
    if all(Dom.type=='usergiven')
      halfWidth = dx*(nSubP-1)/2;
      X1 = -1+halfWidth;  X2 = 1-halfWidth;
```

```
162       Dom.X = (X1+X2)/2-(X2-X1)/2*cos(linspace(0,pi,nPatch));
163       Dom.Y = Dom.X;
164     end
```

Configure the patches:

```
170     configPatches2(@twoscaleDiffForce2,[-1 1],Dom,nPatch  ...
171         ,ordInt ,dx ,nSubP ,'EdgyInt',true ,'hetCoeffs',A );
```

Compute the time-constant forcing, and store in struct `patches` for access by the microcode of Section 7.1.

```
179     if 1
180       patches.fu = 10*(patches.x+cos(pi*patches.x)+patches.y);
181     else patches.fu = 8+0*patches.x+0*patches.y;
182     end
```

Solve for steady state Set initial guess of either zero or a subsample of the previous, next-finer, solution. `NaN` indicates patch-edge values. Index `i` are the indices of patch-interior points, and the number of unknowns is then its length.

```
193     if log2N==log2Nmax
194          u0 = zeros(nSubP,nSubP,1,1,nPatch,nPatch);
195     else u0 = u0(:,:,:,:,1:2:end,1:2:end);
196     end
197     u0([1 end],:,:) = nan;  u0(:,[1 end],:) = nan;
198     patches.i = find(~isnan(u0));
199     nVariables = numel(patches.i)
```

First try to solve via iterative solver `bicgstab`.

```
205     tic;
206     maxIt = ceil(nVariables/10);
207     rhsb = theRes2( zeros(size(patches.i)) );
208     [uSoln,flag] = bicgstab(@(u) rhsb-theRes2(u),rhsb ...
209                     ,1e-9,maxIt,[],[],u0(patches.i));
210     bicgTime = toc
```

However, the above often fails (and `fsolve` sometimes takes too long here), so then try a preconditioned version of `bicgstab`. The preconditioner is derived from the Jacobian which is expensive to find (four minutes for $N = 33$, one hour for $N = 65$), but we do so as follows.

```
220     if flag>0, disp('**** bicg failed, trying ILU preconditioner')
221         disp(['Computing Jacobian: wait roughly ' ...
222               num2str(nPatch^4/4500,2) ' secs'])
223         tic
224         Jac=sparse(nVariables,nVariables);
225         for j=1:nVariables
226             Jac(:,j)=sparse( rhsb-theRes2((1:nVariables)'==j) );
227         end
228         formJacTime=toc
```

Compute an incomplete LU-factorization, and use it as preconditioner to `bicgstab`.

```
235         tic
236         [L,U] = ilu(Jac,struct('type','ilutp','droptol',1e-4));
237         LUfillFactor = (nnz(L)+nnz(U))/nnz(Jac)
238         [uSoln,flag] = bicgstab(@(u) rhsb-theRes2(u),rhsb ...
239                     ,1e-9,maxIt,L,U,u0(patches.i));
240         precondSolveTime=toc
241         assert(flag==0,'preconditioner fails bicgstab. Lower droptol?
242     end%if flag
```

Store the solution into the patches, and give magnitudes—Inf norm is max(abs()).

```
249     normResidual = norm(theRes2(uSoln),Inf)
250     normSoln = norm(uSoln,Inf)
251     u0(patches.i) = uSoln;
252     u0 = patchEdgeInt2(u0);
253     u0( 1 ,:,:,:, 1 ,:)=0; % left edge of left patches
254     u0(end,:,:,:,end,:)=0; % right edge of right patches
255     u0(:, 1 ,:,:,:, 1 )=0; % bottom edge of bottom patches
256     u0(:,end,:,:,:,end)=0; % top edge of top patches
257     assert(normResidual<1e-5,'poor--bad solution found')
```

Concatenate the solution on common patches into stores.

```
263     us=cat(5,us,squeeze(u0(:,:,:,:,Ps,Ps)));
264     xs=cat(3,xs,squeeze(patches.x(:,:,:,:,Ps,:)));
265     ys=cat(3,ys,squeeze(patches.y(:,:,:,:,:,Ps)));
266     nPs = [nPs;nPatch];
```

End loop. Check micro-grids are aligned, then compute errors compared to the full-domain solution (or the highest resolution solution for the case of 'usergiven').

```
end%for log2N
assert(max(abs(reshape(diff(xs,1,3),[],1)))<1e-12,'x-coord failure')
assert(max(abs(reshape(diff(ys,1,3),[],1)))<1e-12,'y-coord failure')
errs = us-us(:,:,:,:,1);
meanAbsErrs = mean(abs(reshape(errs,[],size(us,5))))
ratioErrs = meanAbsErrs(2:end)./meanAbsErrs(1:end-1)
```

Plot solution in common patches First reshape arrays to suit 2D space surface plots, inserting nans to separate patches.

```
x = xs(:,:,1); y = ys(:,:,1); u=us;
x(end+1,:)=nan;   y(end+1,:)=nan;
u(end+1,:,:)=nan; u(:,end+1,:)=nan;
u = reshape(permute(u,[1 3 2 4 5]),numel(x),numel(y),[]);
```

Plot the patch solution surfaces, with colour offset between surfaces (best if u-field has a range of one): blues are the full-domain solution, reds the coarsest patches.

```
figure(1), clf, colormap(jet)
for p=1:size(u,3)
     mesh(x(:),y(:),u(:,:,p)',p+u(:,:,p)');
     hold on;
end, hold off
view(60,55)
colorbar('Ticks',1:size(u,3) ...
    ,'TickLabels',[num2str(nPs) ['x';'x';'x'] num2str(nPs)]);
xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
ifOurCf2eps([mfilename 'us'])%optionally save
```

Plot error surfaces Plot the error surfaces, with colour offset between surfaces (best if u-field has a range of one): dark blue is the full-domain zero error, reds the coarsest patches.

```
err=u(:,:,1)-u;
maxAbsErr=max(abs(err(:)));
figure(2), clf, colormap(jet)
```

```
for p=1:size(u,3)
     mesh(x(:),y(:),err(:,:,p)',p+err(:,:,p)'/maxAbsErr);
     hold on;
end, hold off
view(60,55)
colorbar('Ticks',1:size(u,3) ...
    ,'TickLabels',[num2str(nPs) ['x';'x';'x'] num2str(nPs)]);
xlabel('space x'), ylabel('space y')
zlabel('errors in u(x,y)')
ifOurCf2eps(mfilename)%optionally save
```

7.1 `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
  i = 2:size(u,1)-1; % x interior points in a patch
  j = 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(i,j,:) ...
  = 2*diff(u(:,j,:),2,1)/dx^2 +2*diff(u(i,:,:),2,2)/dy^2 ...
   +2*patches.cs(i,j).*( u(i+1,j+1,:) -u(i-1,j+1,:) ...
     -u(i+1,j-1,:) +u(i-1,j-1,:) )/(4*dx*dy) ...
   +patches.fu(i,j,:);
end%function twoscaleDiffForce2
```

7.2 `theRes2()`: 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=theRes2(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 theRes2
```

8 `abdulleDiffEquil2`: equilibrium of a 2D multiscale 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))

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

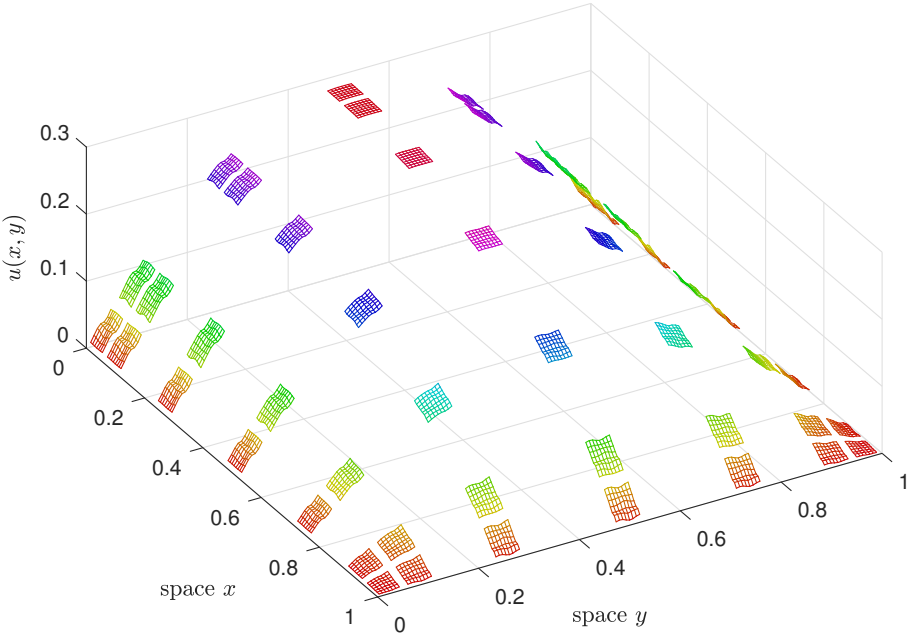
Figure 10 shows solutions have some nice microscale wiggles reflecting the heterogeneity.

Clear, and initiate globals.

```
clear all
global patches
%global OurCf2eps, OurCf2eps=true %option to save plot
```

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.)

Figure 10: Equilibrium of the macroscale diffusion problem of Abdulle with boundary conditions of Dirichlet zero-value except for $x = 0$ which is Neumann (Section 8). Here the patches have a Chebyshev-like spatial distribution. The patch size is chosen large enough to see within.




```
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));
diffusivityRange = [min(a(:)) max(a(:))]
```

Set the periodicity ϵ, here big enough so we can see the patches, and other microscale parameters.

```
epsilon = 0.04
dx = epsilon/mPeriod
nPeriodsPatch = 1 % any integer
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 8.1

```
78 Dom.bcOffset = zeros(2);
79 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':

```
86 nPatch = 7
87 Dom.type='chebyshev';
88 configPatches2(@abdulleDiffForce2,[0 1],Dom ...
89     ,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.

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

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

```
113 tic;
114 uSoln = fsolve(@theRes2,u0(patches.i) ...
115         ,optimoptions('fsolve','Display','off'));
116 solnTime = toc
117 normResidual = norm(theRes2(uSoln))
118 normSoln = norm(uSoln)
```

Store the solution vector into the patches, and interpolate, but have not bothered to set boundary values so they stay NaN from the interpolation.

```
126 u0(patches.i) = uSoln;
127 u0 = patchEdgeInt2(u0);
```

Draw solution profile Separate patches with NaNs, then reshape arrays to suit 2D space surface plots.

```
138 figure(1), clf, colormap(0.8*hsv)
139 patches.x(end+1,:,:)=nan;  u0(end+1,:,:)=nan;
140 patches.y(:,end+1,:)=nan;  u0(:,end+1,:)=nan;
141 u = reshape(permute(squeeze(u0),[1 3 2 4]) ...
142     , [numel(patches.x) numel(patches.y)]);
```

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

```
mesh(patches.x(:),patches.y(:),u'); view(60,55)
xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')
ifOurCf2eps(mfilename) %optionally save plot
```

8.1 `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
  i = 2:size(u,1)-1; % x interior points in a patch
  j = 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(i,j,:) = diff(patches.cs(:,j).*diff(u(:,j,:)))/dx^2 ...
      + diff(patches.cs(i,:).*diff(u(i,:,:),1,2),1,2)/dy^2 ...
      + 10;
end%function abdulleDiffForce2
```

8.2 `theRes2()`: 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=theRes2(u)
  global patches i
  v=nan(size(patches.x+patches.y));
  v(patches.i)=u;
  f=patchSys2(0,v(:),patches);
  f=f(patches.i);
end%function theRes2
```

9 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.[7] §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 boundary conditions, for microscale random pseudo-diffusion and pseudo-advection coefficients, $\mu_1(x, y) \in [0.01, 0.1]$[8] and $\mu_2(x, y) \in [0, 2\pi)$, and for forcing

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

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 $\ell < 0.01 = \min \mu_1$. Due to the randomness, we get different solutions each execution of this code. Figure 11 plots one example. A physical interpretation of the solution field is confounded because the problem is pseudo-advection-diffusion due to the varying coefficients being outside the $\vec{\nabla}$ operator.

Clear, and initiate globals.

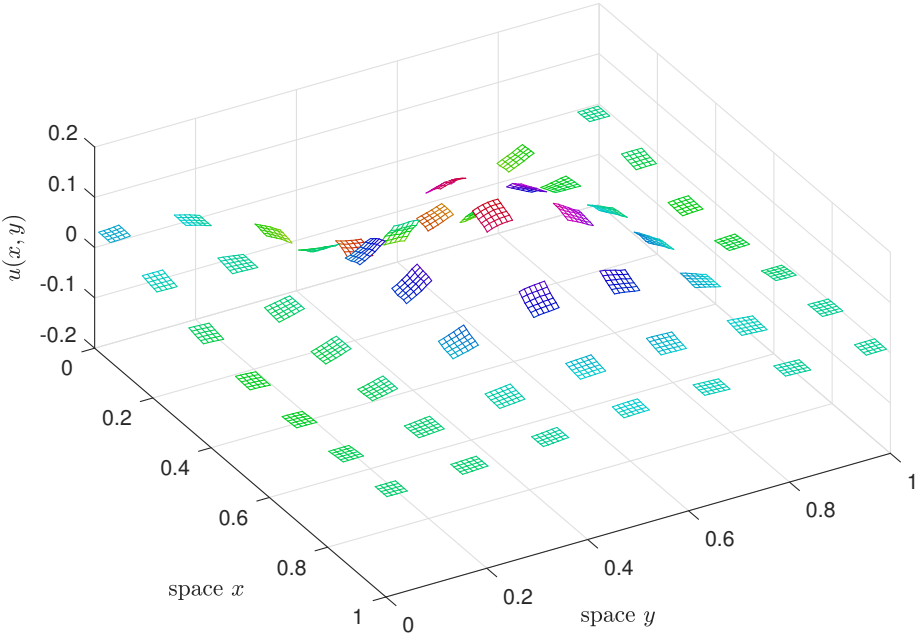
```
clear all
global patches
%global OurCf2eps, OurCf2eps=true %option to save plot
```

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

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

[8] More interesting microscale structure arises here for μ_1 a factor of three smaller.

Figure 11: Equilibrium of the macroscale diffusion problem of Bonizzoni et al. with Neumann boundary conditions of zero (Section 9). Here the patches have a equispaced spatial distribution. The microscale periodicity, and hence the patch size, is chosen large enough to see within.




```
mPeriod = 4
mu1 = 0.01*10.^rand(mPeriod)
mu2 = 2*pi*rand(mPeriod)
cs = cat(3,mu1,cos(mu2),sin(mu2));
meanDiffAdvec=squeeze(mean(mean(cs)))
```

Set the periodicity ϵ, here big enough so we can see the patches, and other microscale parameters.

```
epsilon = 0.04
dx = epsilon/mPeriod
nPeriodsPatch = 1 % any integer
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 9.1.

```
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 `theRes2`, 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(@theRes2,u0(patches.i) ...
        ,optimoptions('fsolve','Display','off'));
solnTime = toc
normResidual = norm(theRes2(uSoln))
normSoln = norm(uSoln)
```

Store the solution vector into the patches, and interpolate, but have not bothered to set boundary values so they stay NaN from the interpolation.

```
u0(patches.i) = uSoln;
u0 = patchEdgeInt2(u0);
```

Draw solution profile Separate patches with NaNs, then reshape arrays to suit 2D space surface plots.

```
figure(1), clf, colormap(0.8*hsv)
patches.x(end+1,:,:)=nan;  u0(end+1,:,:)=nan;
patches.y(:,end+1,:)=nan;  u0(:,end+1,:)=nan;
u = reshape(permute(squeeze(u0),[1 3 2 4]) ...
    , [numel(patches.x) numel(patches.y)]);
```

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

```
mesh(patches.x(:),patches.y(:),u'); view(60,55)
xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')
ifOurCf2eps(mfilename) %optionally save plot
```

9.1 `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
  i = 2:size(u,1)-1; % x interior points in a patch
  j = 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: that is, the edge value equals the next-to-edge value.

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

9.2 theRes2(): 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=theRes2(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 theRes2
```

10 homoDiffBdryEquil3: equilibrium via computational homogenisation of a 3D heterogeneous diffusion on small patches

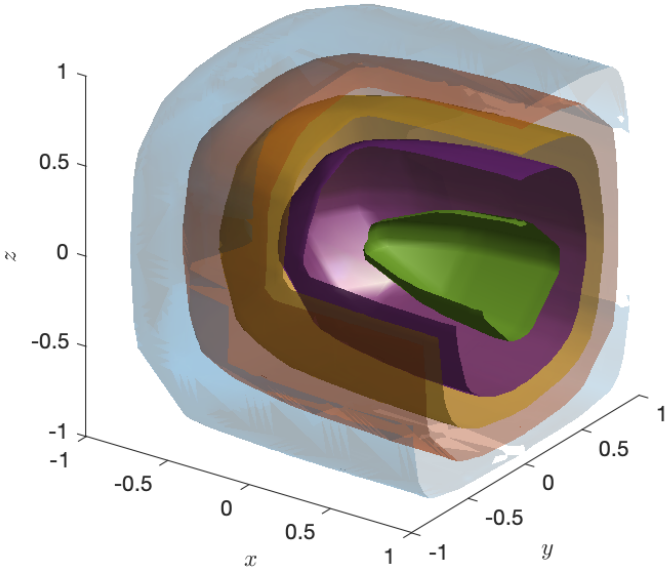
Find the equilibrium of a forced heterogeneous diffusion in 3D space on 3D patches as an example application. Boundary conditions are Neumann on the right face of the cube, and Dirichlet on the other faces. Figure 12 shows five isosurfaces of the 3D solution field.

Clear variables, and establish globals.

```
clear all
global patches
%global OurCf2eps, OurCf2eps=true %option to save plots
```

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.

Figure 12: macroscale of the random heterogeneous diffusion in 3D with boundary conditions of zero on all faces except for the Neumann condition on $x = 1$ (Section 10). The small patches are equispaced in space.




```
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 cubic 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 1], Dom ...
```

```
64      , nPatch, 0, 0.05, nSubP, 'EdgyInt',true  ...
65      ,'hetCoeffs',cDiff );
```

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

```
73  patches.fu = 10*exp(-patches.x.^2-patches.y.^2-patches.z.^2);
74  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.

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

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

```
100 tic;
101 uSoln = fsolve(@theRes3,u0(patches.i) ...
102         ,optimoptions('fsolve','Display','off'));
103 solveTime = toc
104 normResidual = norm(theRes3(uSoln))
105 normSoln = norm(uSoln)
```

Store the solution into the patches, and give magnitudes.

```
111 u0(patches.i) = uSoln;
112 u0 = patchEdgeInt3(u0);
```

Plot isosurfaces of the solution

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

Reshape spatial coordinates of patches.

```
128 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 = reshape( permute(squeeze(u0),[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 some isosurfaces.

```
clf;
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
hold off
axis equal, axis([-1 1 -1 1 -1 1]), view(35,25)
xlabel('$x$'), ylabel('$y$'), zlabel('$z$')
camlight, lighting gouraud
ifOurCf2eps(mfilename) %optionally save plot
if exist('OurCf2eps') && OurCf2eps,  print('-dpng',['Figs/' mfilename
```

10.1 `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.

```
197   dx = diff(patches.x(2:3));  % x micro-scale step
198   dy = diff(patches.y(2:3));  % y micro-scale step
199   dz = diff(patches.z(2:3));  % z micro-scale step
200   i = 2:size(u,1)-1; % x interior points in a patch
201   j = 2:size(u,2)-1; % y interior points in a patch
202   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).

```
210   u( 1 ,:,:,:,:, 1 ,:,:)=0; %left face of leftmost patch
211   u(end,:,:,:,:,end,:,:)=u(end-1,:,:,:,:,end,:,:); %right face of rig
212   u(:, 1 ,:,:,:,:, 1 ,:)=0; %bottom face of bottommost
213   u(:,end,:,:,:,:,end,:)=0; %top face of topmost
214   u(:,:, 1 ,:,:,:,:, 1 )=0; %front face of frontmost
215   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`

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

10.2 `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.

```
243 function f=theRes3(u)
244   global patches
245   v=nan(size(patches.x+patches.y+patches.z));
246   v(patches.i)=u;
247   f=patchSys3(0,v(:),patches);
248   f=f(patches.i);
249 end%function theRes3
```

11 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). [9]

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

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

Input

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

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

- `.stag` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation, and zero for ordinary grid.
- `.Cwtsr` and `.Cwtsl` are the coupling coefficients for finite width interpolation—when invoking a periodic domain.
- `.EdgyInt`, true/false, for determining patch-edge values by interpolation: true, from opposite-edge next-to-edge values (often preserves symmetry); false, from centre-patch values (original scheme).
- `.nEnsem` the number of realisations in the ensemble.
- `.parallel` whether serial or parallel.
- `.nCore` [10]
- _ [11]

Output

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

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

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

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

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

[10] **ToDo:** introduced sometime but not fully implemented yet, because prefer ensemble
[11] **ToDo:** additional macros bdry info

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.

[12] 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);
```

11.1 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];
```

[12] **ToDo:** Revise??

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
311       stagShift = 0;
312       iV = 1:nVars;
313     end%if patches.stag
```

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

```
320     kMax = floor((Nx-1)/2);
321     ks = shiftdim( ...
322         2*pi/Nx*(mod((0:Nx-1)+kMax,Nx)-kMax) ...
323         ,-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`.

```
336 if ~patches.EdgyInt
337     Cleft = fft(u(i0  ,:,:,:),[],4);
338     Cright = Cleft;
339 else
340     Cleft = fft(u(2   ,:,:,:),[],4);
341     Cright= fft(u(nx-1,:,:,:),[],4);
342 end
```

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

```
349     u(nx,iV,patches.ri,:) = uclean( ifft( ...
350         Cleft.*exp(1i*ks.*(stagShift+r)) ,[],4));
351     u(1 ,iV,patches.le,:) = uclean( ifft( ...
352         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?

```
360 if patches.stag
361     nVars = nVars/2;
362     u=reshape(u,nx,nVars,2,nEnsem,Nx);
363     Nx = 2*Nx;
364     v(:,:,:,1:2:Nx) = u(:,:,1,:,:);
365     v(:,:,:,2:2:Nx) = u(:,:,2,:,:);
```

```
  u = v;
end%if patches.stag
end%if patches.ordCC
```

11.2 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.[13] First alternative: the case of order p interpolation across the domain, asymmetric near the boundary. Use this first alternative for now.

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

Second alternative: lower the degree of interpolation near the boundary to maintain the band-width of the interpolation. Such symmetry might be essential for multi-D. [14]

```
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
```

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

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

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.

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

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys1()`. Section 12.1 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 12.1 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.[15]

```
180 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.
- `.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

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

  - 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.

12.1 If no arguments, then execute an example

```
252 if nargin==0
253 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.

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

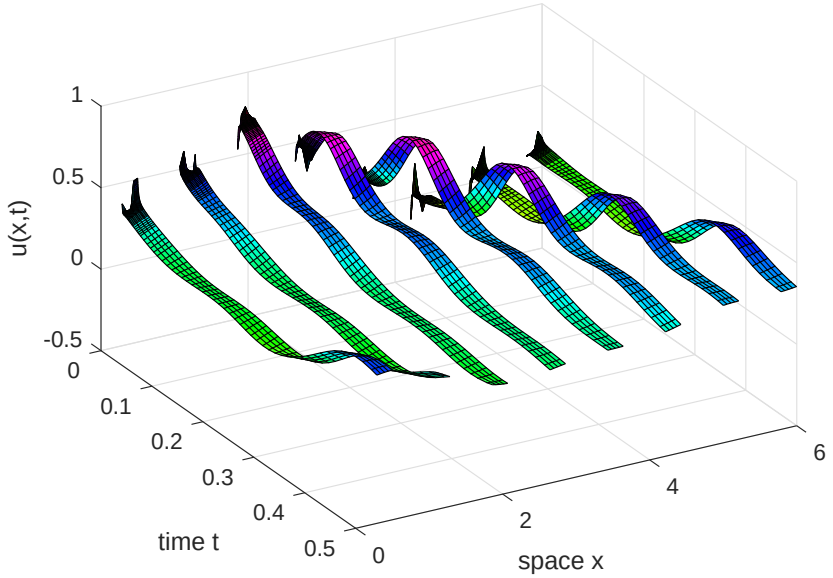
Set some initial condition, with some microscale randomness.

```
280 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()` (**??**).

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

Figure 13: field $u(x,t)$ of the patch scheme applied to Burgers' PDE.



Plot the simulation using only the microscale values interior to the patches: either set x-edges to `nan` to leave the gaps; or use `patchEdgyInt1` to re-interpolate correct patch edge values and thereby join the patches. Figure 13 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
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 13, then exit this function.

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

12.2 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
```

12.3 The code to make patches and interpolation

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

```
453 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.

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

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

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

Check for staggered grid and periodic case.

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

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

```
489 switch Dom.type
```

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

```
497 case 'periodic'
498   X=linspace(Xlim(1),Xlim(2),nPatch+1);
499   DX=X(2)-X(1);
500   X=X(1:nPatch)+diff(X)/2;
501   pEI=patches.EdgyInt;% abbreviation
502   if pre2023, dx = ratio*DX/(nSubP-1-pEI)*(2-pEI);
503   else        ratio = dx/DX*(nSubP-1-pEI)/(2-pEI);  end
504   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'.[16]

```
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
```

[16] 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??

```
  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.

```
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);
```

12.4 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??

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

End the two if-statements.

```
698   end%if-else nEnsem>1
699 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.*[17]

```
718 if patches.parallel
719 %  theparpool=gcp()
720   spmd
```

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

```
727   pari = 1;
728   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.

```
738   switch pari
739     case 1, patches.x=codistributed(patches.x,patches.codist);
740   otherwise
741     error('should never have bad index for parallel distribution')
742   end%switch
743   end%spmd
```

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

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

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

Fin

```
762 end% function
```

13 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 (**Bunder2019c**; Roberts, MacKenzie, and Bunder 2014) indicates the patch centre-values are sensible macroscale variables, and macroscale interpolation of these determine patch-edge values. However, for computational homogenisation in multi-D, interpolating patch next-to-edge values appears better (Bunder, Kevrekidis, and Roberts 2021). This function is primarily used by `patchSys2()` but is also useful for user graphics. [18]

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

```
29 function u = patchEdgeInt2(u,patches)
30 if nargin<2, global patches, end
31 %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

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

equi-spaced lattice on the microscale index i, but may be variable spaced in macroscale index I.

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

```
120   if max(abs(imag(u(:))))<1e-9*max(abs(u(:)))
121        uclean=@(u) real(u);
122   else uclean=@(u) u;
123   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')
```

13.1 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.

```
194   if patches.EdgyInt % interpolate next-to-edge values
195     Ux = u([2 nx-1],2:(ny-1),:,:,I,J);
196     Uy = u(2:(nx-1),[2 ny-1],:,:,I,J);
197   else % interpolate centre-cross values
198     Ux = u(i0,2:(ny-1),:,:,I,J);
199     Uy = u(2:(nx-1),j0,:,:,I,J);
200   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.

```
208 szUx0=size(Ux); szUx0=[szUx0 ones(1,6-length(szUx0)) ordCC];
209 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.

```
219   if patches.parallel
220     dmux = zeros(szUx0,patches.codist); % 7D
221     dmuy = zeros(szUy0,patches.codist); % 7D
222   else
223     dmux = zeros(szUx0); % 7D
224     dmuy = zeros(szUy0); % 7D
225   end%if patches.parallel
```

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

```
232   if patches.stag % use only odd numbered neighbours
233     error('polynomial interpolation not yet for staggered patch coupl
234     dmux(:,:,:,:,I,:,1) = (Ux(:,:,:,:,Ip,:)+Ux(:,:,:,:,Im,:))/2; % \m
235     dmux(:,:,:,:,I,:,2) = (Ux(:,:,:,:,Ip,:)-Ux(:,:,:,:,Im,:)); % \del
236     Ip = Ip(Ip); Im = Im(Im); % increase shifts to \pm2
237     dmuy(:,:,:,:,:,J,1) = (Ux(:,:,:,:,:,Jp)+Ux(:,:,:,:,:,Jm))/2; % \m
238     dmuy(:,:,:,:,:,J,2) = (Ux(:,:,:,:,:,Jp)-Ux(:,:,:,:,:,Jm)); % \del
239     Jp = Jp(Jp); Jm = Jm(Jm); % increase shifts to \pm2
240   else %disp('starting standard interpolation')
241     dmux(:,:,:,:,I,:,1) = (Ux(:,:,:,:,Ip,:) ...
242                            -Ux(:,:,:,:,Im,:))/2; %\mu\delta
243     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 `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
```

13.2 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)
```

13.2.1 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,:,:,:,:,:);
```

13.2.2 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,:,:,:,:);
```

13.2.3 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
```

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

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys2()`. Section 14.1 lists an example of its use.

```
19 function patches = configPatches2(fun,Xlim,Dom ...
20     ,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 14.1 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)`$] \times [$`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.[19]

```
205 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`, 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

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

  - `[]` 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.

14.1 If no arguments, then execute an example

```
288 if nargin==0
289 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). Roberts, MacKenzie, and Bunder (2014) established that this scheme is consistent with the PDE (as the patch spacing decreases).

```
312 global patches
313 patches = configPatches2(@nonDiffPDE,[-3 3 -2 2], [] ...
314     , [9 7], 0, 0.12, 5 ,'EdgyInt',false);
```

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

```
321 u0 = exp(-patches.x.^2-patches.y.^2);
322 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 14 while the simulation computes.

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

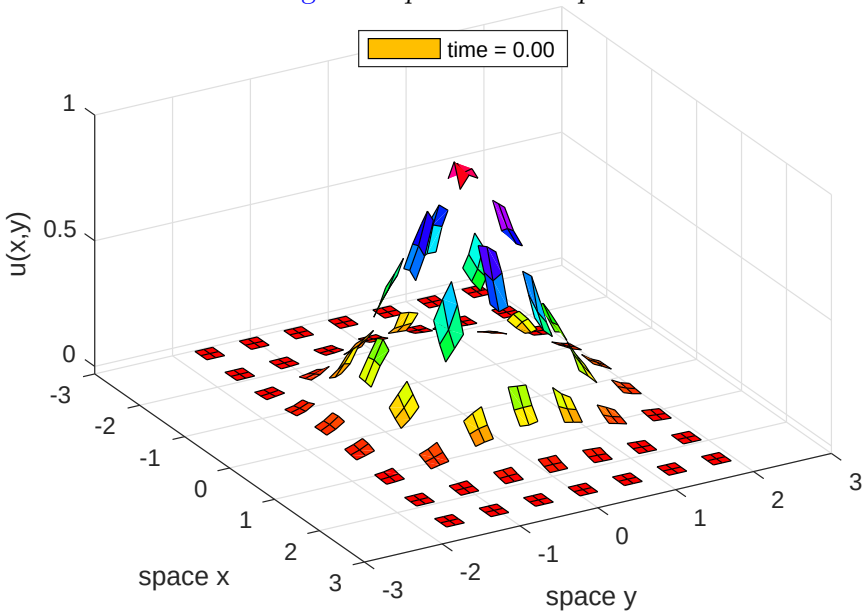
Integrate in time to $t = 4$ using standard functions. In MATLAB `ode15s` would be natural as the patch scheme is naturally stiff, but `ode23` is quicker (Maclean, Bunder, and Roberts 2021, Fig. 4). Ask for output at non-uniform times because the diffusion slows.

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

Animate the computed simulation to end with Figure 15. 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)]);
```

Figure 14: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to a nonlinear 'diffusion' PDE: Figure 15 plots the computed field at time $t = 3$.




```
384   set(hsurf,'ZData', u');
385   legend(['time = ' num2str(ts(i),'%4.2f')])
386   pause(0.1)
387 end
388 ifOurCf2eps([mfilename 't3'])
```

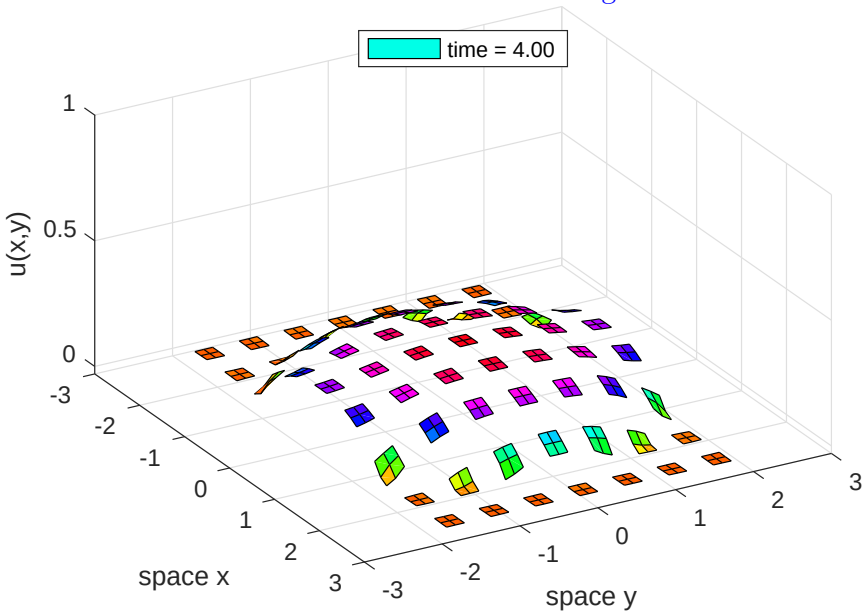
Upon finishing execution of the example, exit this function.

```
403 return
404 end%if no arguments
```

14.2 Parse input arguments and defaults

```
418 p = inputParser;
419 fnValidation = @(f) isa(f, 'function_handle');%test for fn name
420 addRequired(p,'fun',fnValidation);
421 addRequired(p,'Xlim',@isnumeric);
422 %addRequired(p,'Dom'); % nothing yet decided
423 addRequired(p,'nPatch',@isnumeric);
```

Figure 15: field $u(x, y, t)$ at time $t = 3$ of the patch scheme applied to a nonlinear 'diffusion' PDE with initial condition in Figure 14.




```
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.

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

Check parameters.

```
462 assert(Xlim(1)<Xlim(2) ...
463       ,'first pair of Xlim must be ordered increasing')
464 assert(Xlim(3)<Xlim(4) ...
465       ,'second pair of Xlim must be ordered increasing')
466 assert(patches.nEdge==1 ...
467       ,'multi-edge-value interp not yet implemented')
468 assert(all(2*patches.nEdge<nSubP) ...
469       ,'too many edge values requested')
470 %if patches.nCore>1
471 %    warning('nCore>1 not yet tested in this version')
472 %    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.

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

Default macroscale conditions are periodic with evenly spaced patches.

```
493 if isempty(Dom), Dom=struct('type','periodic'); end
494 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.

```
502 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.

```
515 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.

```
523 patches.periodic=false;
524 for p=1:2
525 switch Dom.type(p,:)
526 case 'periodic'
527     patches.periodic=true;
528     if isfield(Dom,'bcOffset')
529     warning('bcOffset not available for Dom.type = periodic'), end
530     msg=' not available for Dom.type = periodic';
531     if isfield(Dom,'X'), warning(['X' msg]), end
532     if isfield(Dom,'Y'), warning(['Y' msg]), end
533 case {'equispace','chebyshev'}
534     if ~isfield(Dom,'bcOffset'), Dom.bcOffset=zeros(2,2); end
535     % for mixed with usergiven, following should still work
536     if numel(Dom.bcOffset)==1
537         Dom.bcOffset=repmat(Dom.bcOffset,2,2); end
538     if numel(Dom.bcOffset)==2
539         Dom.bcOffset=repmat(Dom.bcOffset(:)',2,1); end
540     msg=' not available for Dom.type = equispace or chebyshev';
541     if (p==1)& isfield(Dom,'X'), warning(['X' msg]), end
542     if (p==2)& isfield(Dom,'Y'), warning(['Y' msg]), end
543 case 'usergiven'
544 %    if isfield(Dom,'bcOffset')
545 %    warning('bcOffset not available for usergiven Dom.type'), end
546     msg=' required for Dom.type = usergiven';
547     if p==1, assert(isfield(Dom,'X'),['X' msg]), end
548     if p==2, assert(isfield(Dom,'Y'),['Y' msg]), end
549 otherwise
550     error([Dom.type 'is unknown Dom.type'])
551 end%switch Dom.type
552 end%for p
```

14.3 The code to make patches

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

```
565 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);
```

```
628   else         ratio(q) = dx(q)/DQ*(nSubP(q)-1-pEI)/(2-pEI);
629   end
630   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.

```
639 case 'equispace'
640   Q=linspace(Xlim(qq)+((nSubP(q)-1)/2-Dom.bcOffset(qq))*dx(q) ...
641           ,Xlim(qq+1)-((nSubP(q)-1)/2-Dom.bcOffset(qq+1))*dx(q) ...
642           ,nPatch(q));
643   DQ=diff(Q(1:2));
644   width=(1+patches.EdgyInt)/2*(nSubP(q)-1-patches.EdgyInt)*dx;
645   if DQ<width*0.999999
646      warning('too many equispace patches (double overlapping)')
647      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'.[20]

```
664 case 'chebyshev'
665   halfWidth=dx(q)*(nSubP(q)-1)/2;
666   Q1 = Xlim(1)+halfWidth-Dom.bcOffset(qq)*dx(q);
667   Q2 = Xlim(2)-halfWidth+Dom.bcOffset(qq+1)*dx(q);
668 %  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`.

```
677   width=(1+patches.EdgyInt)/2*(nSubP(q)-1-patches.EdgyInt)*dx(q);
678   for b=0:2:nPatch(q)-2
679     DQmin=(Q2-Q1-b*width)/2*( 1-cos(pi/(nPatch(q)-b-1)) );
680     if DQmin>width, break, end
681   end
```

[20] 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??

```
  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 ];
```

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.)

```
753 if patches.periodic
754   ratio = reshape(ratio,1,2); % force to be row vector
755   patches.ratio=ratio;
756   if ordCC>0
757     [Cwtsr,Cwtsl] = patchCwts(ratio,ordCC,patches.stag);
758     patches.Cwtsr = Cwtsr;  patches.Cwtsl = Cwtsl;
759   end%if
760 end%if patches.periodic
```

14.4 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()`.

```
787 nE = patches.nEnsem;
788 patches.le = 1:nE;  patches.ri = 1:nE;
789 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.

```
801 if ~isempty(cs)
802   [mx,my,nc] = size(cs);
803   nx = nSubP(1); ny = nSubP(2);
804   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.

```
812   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.

```
821     patches.nEnsem = mx*my;
822     patches.cs = nan(nx-1,ny-1,nc,mx,my);
823     for j = 1:my
824         js = (j:j+ny-2);
825         for i = 1:mx
826             is = (i:i+nx-2);
827             patches.cs(:,:,:,i,j) = cs(is,js,:);
828         end
829     end
830     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.

```
840     le = mod((0:mx-1)+mod(nx-2,mx),mx)+1;
841     patches.le = reshape(  le'+mx*(0:my-1)  ,[],1);
842     ri = mod((0:mx-1)-mod(nx-2,mx),mx)+1;
843     patches.ri = reshape(  ri'+mx*(0:my-1)  ,[],1);
844     bo = mod((0:my-1)+mod(ny-2,my),my)+1;
845     patches.bo = reshape( (1:mx)'+mx*(bo-1) ,[],1);
846     to = mod((0:my-1)-mod(ny-2,my),my)+1;
847     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??

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

End the two if-statements.

```
864   end%if-else nEnsem>1
865 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.*[21]

```
884 if patches.parallel
885 %  theparpool=gcp()
886   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.

```
895   [~,pari]=max(nPatch+0.01*(1:2));
896   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.

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

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

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

Fin

```
931 end% function
```

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

15 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 (**Bunder2019c**; Roberts, MacKenzie, and Bunder 2014), 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. [22]

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

```
27 function u = patchEdgeInt3(u,patches)
28 if nargin<2, global patches, end
29 %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.

[22] 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.

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

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

```
140 [~,~,nz,~,~,~,~,Nz] = size(patches.z);
141 [~,ny,~,~,~,~,Ny,~] = size(patches.y);
142 [nx,~,~,~,~,Nx,~,~] = size(patches.x);
143 nEnsem = patches.nEnsem;
144 nVars = round( numel(u)/numel(patches.x) ...
```

```
    /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]);
%usize1=size(u)%%%%%
```

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')
```

15.1 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.

```
210   if patches.EdgyInt % interpolate next-to-face values
211     Ux = u([2 nx-1],2:(ny-1),2:(nz-1),:,:,I,J,K);
212     Uy = u(2:(nx-1),[2 ny-1],2:(nz-1),:,:,I,J,K);
213     Uz = u(2:(nx-1),2:(ny-1),[2 nz-1],:,:,I,J,K);
214   else % interpolate centre-cross values
215     Ux = u(i0,2:(ny-1),2:(nz-1),:,:,I,J,K);
216     Uy = u(2:(nx-1),j0,2:(nz-1),:,:,I,J,K);
217     Uz = u(2:(nx-1),2:(ny-1),k0,:,:,I,J,K);
218   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.

```
226 szUx0=size(Ux); szUx0=[szUx0 ones(1,8-length(szUx0)) ordCC];
227 szUy0=size(Uy); szUy0=[szUy0 ones(1,8-length(szUy0)) ordCC];
228 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.

```
238   if patches.parallel
239     dmux = zeros(szUx0,patches.codist); % 9D
240     dmuy = zeros(szUy0,patches.codist); % 9D
241     dmuz = zeros(szUz0,patches.codist); % 9D
242   else
243     dmux = zeros(szUx0); % 9D
244     dmuy = zeros(szUy0); % 9D
245     dmuz = zeros(szUz0); % 9D
246   end%if patches.parallel
```

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

```
253   if patches.stag % use only odd numbered neighbours
254     error('polynomial interpolation not yet for staggered patch coupl
255     dmux(:,:,:,:,:,:,I,:,:,1) = (Ux(:,:,:,:,:,:,Ip,:,:) +Ux(:,:,:,:,:,:,Im,
256     dmux(:,:,:,:,:,:,I,:,:,2) = (Ux(:,:,:,:,:,:,Ip,:,:) -Ux(:,:,:,:,:,:,Im,
257     Ip = Ip(Ip); Im = Im(Im); % increase shifts to \pm2
258     dmuy(:,:,:,:,:,:,:,J,:,1) = (Ux(:,:,:,:,:,:,:,Jp,:)+Ux(:,:,:,:,:,:,:,Jm
259     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,:,:,:) ...
```

```
414           ,[],6),[],7),[],8);
415       Cbo = fft(fft(fft( u(ix,2   ,iz ,:,patches.bo,:,:,:) ...
416           ,[],6),[],7),[],8);
417       Cto = fft(fft(fft( u(ix,ny-1,iz ,:,patches.to,:,:,:) ...
418           ,[],6),[],7),[],8);
419       Cba = fft(fft(fft( u(ix,iy,2    ,:,patches.ba,:,:,:) ...
420           ,[],6),[],7),[],8);
421       Cfr = fft(fft(fft( u(ix,iy,nz-1 ,:,patches.fr,:,:,:) ...
422           ,[],6),[],7),[],8);
423   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.

```
431   u(nx,iy,iz,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
432       Cle.*exp(1i*(stagShift+krx))  ,[],6),[],7),[],8) );
433   u( 1,iy,iz,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
434       Cri.*exp(1i*(stagShift-krx))  ,[],6),[],7),[],8) );
435   u(ix,ny,iz,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
436       Cbo.*exp(1i*(stagShift+kry))  ,[],6),[],7),[],8) );
437   u(ix, 1,iz,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
438       Cto.*exp(1i*(stagShift-kry))  ,[],6),[],7),[],8) );
439   u(ix,iy,nz,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
440       Cba.*exp(1i*(stagShift+krz))  ,[],6),[],7),[],8) );
441   u(ix,iy, 1,:,:,:,:,:) = uclean( ifft(ifft(ifft( ...
442       Cfr.*exp(1i*(stagShift-krz))  ,[],6),[],7),[],8) );
443   end% if ordCC>0 else, so spectral
```

15.2 Non-periodic macroscale interpolation

```
454   else% patches.periodic false
455   %disp('executing new non-periodic code')
456   assert(~patches.stag, ...
457   '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;
```

15.2.1 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`.

```
518 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.

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

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

```
541 u(1 ,iy,iz,:,patches.le,:,:,:) = Uface(1,:,:,:,:,:,:,:);
542 u(nx,iy,iz,:,patches.ri,:,:,:) = Uface(2,:,:,:,:,:,:,:);
543 %usize2=size(u)%%%%%
```

15.2.2 y-direction values

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

```
553   F = nan(nx,patches.EdgyInt+1,nz-2,nVars,nEnsem,Nx,Ny,Nz,py+1);
554   if patches.EdgyInt % interpolate next-to-face values
555     F(:,:,:,:,:,:,:,:,1) = u(:,[ny-1 2],iz,:,:,:,:,:);
556     Y = patches.y(:,[ny-1 2],:,:,:,:,:,:);
557   else % interpolate mid-patch cross-patch values
558     F(:,:,:,:,:,:,:,:,1) = u(:,j0,iz,:,:,:,:,:);
559     Y = patches.y(:,j0,:,:,:,:,:,:);
560   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,:,:,:,:,:,:);
%usize3=size(u)%%%%%
```

15.2.3 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,:,:,:,:,:);
%usize4=size(u)%%%%%
```

15.2.4 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(:,:,nz,:,:,:,:,Nz) = nan;
%usize5=size(u)%%%%%
```

End of the non-periodic interpolation code.

```
684 %disp('finished new non-periodic code')
685 end%if patches.periodic else
```

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

```
696 end% function patchEdgeInt3
```

16 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 16.1 and **??** list examples of its use.

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

Input If invoked with no input arguments, then executes an example of simulating a heterogeneous wave PDE—see Section 16.1 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)`$] \times [$`Xlim(3)`$,$`Xlim(4)` $\times$ $[$`Xlim(5)`$,$`Xlim(6)`$]$. If `Xlim` has two elements, then the domain is the cubic domain of the same interval in all three directions.
- `Dom` sets the type of macroscale conditions for the patches, and reflects the type of microscale boundary conditions of the problem. If `Dom` is `NaN` or `[]`, then the field `u` is triply macro-periodic in the 3D spatial domain, and resolved on equi-spaced patches. If `Dom` is a character string, then that specifies the `.type` of the following structure, with `.bcOffset` set to the default zero. Otherwise `Dom` is a structure with the following components.
  - `.type`, string, of either `'periodic'` (the default), `'equispace'`, `'chebyshev'`, `'usergiven'`. For all cases except `'periodic'`, users

must code into `fun` the micro-grid boundary conditions that apply at the left/right/bottom/top/back/front faces of the leftmost/rightmost/bottommost/topmost/backmost/frontmost patches, respectively.

- `.bcOffset`, optional one, three or six element vector/array, in the cases of `'equispace'` or `'chebyshev'` the patches are placed so the left/right macroscale boundaries are aligned to the left/right faces of the corresponding extreme patches, but offset by `bcOffset` of the sub-patch micro-grid spacing. For example, use `bcOffset=0` when the micro-code applies Dirichlet boundary values on the extreme face micro-grid points, whereas use `bcOffset=0.5` when the microcode applies Neumann boundary conditions halfway between the extreme face micro-grid points. Similarly for the top, bottom, back, and front faces.

  If 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.[23]

```
217 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.

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

- `.Cwtsr` and `.Cwtsl` are the `ordCC` $\times$ 3-array of weights for the inter-patch interpolation onto the right/top/front and left/bottom/back faces (respectively) with patch:macroscale ratio as specified or as derived from `dx`.

- `.x` (8D) is `nSubP(1)` $\times 1 \times 1 \times 1 \times$ `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)` $\times 1 \times 1 \times 1 \times$ `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)` $\times 1 \times 1 \times 1 \times$ `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.

16.1 If no arguments, then execute an example

```
306 if nargin==0
307 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.

```
325 mPeriod = [2 2 2];
326 cHetr = exp(0.9*randn([mPeriod 3]));
327 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.

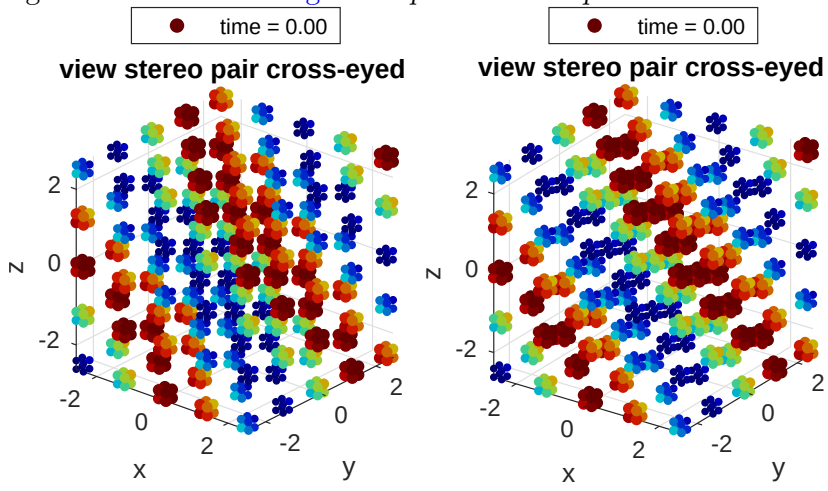
```
340 global patches
341 patches = configPatches3(@heteroWave3,[-pi pi], 'periodic' ...
342     , 5, 0, 0.22, mPeriod+2 ,'EdgyInt',true ...
343     ,'hetCoeffs',cHetr);
```

Set a wave initial state using auto-replication of the spatial grid, and as Figure 16 shows. This wave propagates diagonally across space. Concatenate the two u, v-fields to be the two components of the fourth dimension.

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

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

Figure 16: initial field $u(x, y, z, t)$ at time $t = 0$ of the patch scheme applied to a heterogeneous wave PDE*: Figure 17 plots the computed field at time $t = 6$.*




```
disp('Simulate heterogeneous wave u_tt=div[C*grad(u)]')
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@patchSys3,linspace(0,6),uv0(:));
else %disp('octave version is very slow for me')
    lsode_options('absolute tolerance',1e-4);
    lsode_options('relative tolerance',1e-4);
    [ts,us] = odeOcts(@patchSys3,[0 1 2],uv0(:));
end
```

Animate the computed simulation to end with Figure 17. 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
```

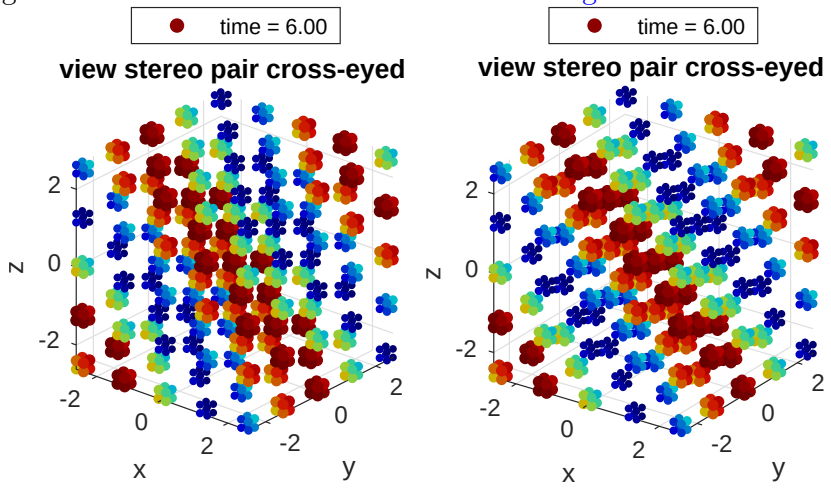
Optionally save the initial condition to graphic file for Figure 14, and optionally save the last plot.

```
  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
```

Figure 17: 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 16.*



16.2 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 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.

```
566 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.

```
579 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.

```
587 patches.periodic=false;
588 for p=1:3
589 switch Dom.type(p,:)
590 case 'periodic'
591     patches.periodic=true;
592     if isfield(Dom,'bcOffset')
593     warning('bcOffset not available for Dom.type = periodic'), end
594     msg=' not available for Dom.type = periodic';
595     if isfield(Dom,'X'), warning(['X' msg]), end
596     if isfield(Dom,'Y'), warning(['Y' msg]), end
597     if isfield(Dom,'Z'), warning(['Z' msg]), end
598 case {'equispace','chebyshev'}
599     if ~isfield(Dom,'bcOffset'), Dom.bcOffset=zeros(2,2); end
600     % for mixed with usergiven, following should still work
601     if numel(Dom.bcOffset)==1
602         Dom.bcOffset=repmat(Dom.bcOffset,2,3); end
603     if numel(Dom.bcOffset)==3
604         Dom.bcOffset=repmat(Dom.bcOffset(:)',2,1); end
605     msg=' not available for Dom.type = equispace or chebyshev';
606     if (p==1)& isfield(Dom,'X'), warning(['X' msg]), end
607     if (p==2)& isfield(Dom,'Y'), warning(['Y' msg]), end
608     if (p==3)& isfield(Dom,'Z'), warning(['Z' msg]), end
609 case 'usergiven'
610 %    if isfield(Dom,'bcOffset')
611 %    warning('bcOffset not available for usergiven Dom.type'), end
612     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
```

16.3 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`:

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

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

```
691 case 'periodic'
692   Q=linspace(Xlim(qq),Xlim(qq+1),nPatch(q)+1);
693   DQ=Q(2)-Q(1);
694   Q=Q(1:nPatch(q))+diff(Q)/2;
695   pEI=patches.EdgyInt;% abbreviation
696 %  sizedx=size(dx), sizenSubP=size(nSubP)
697   if pre2023, dx(q) = ratio(q)*DQ/(nSubP(q)-1-pEI)*(2-pEI);
698   else        ratio(q) = dx(q)/DQ*(nSubP(q)-1-pEI)/(2-pEI);
699   end
700   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.

```
709 case 'equispace'
710   Q=linspace(Xlim(qq)+((nSubP(q)-1)/2-Dom.bcOffset(qq))*dx(q) ...
711           ,Xlim(qq+1)-((nSubP(q)-1)/2-Dom.bcOffset(qq+1))*dx(q) ...
712           ,nPatch(q));
713   DQ=diff(Q(1:2));
714   width=(1+patches.EdgyInt)/2*(nSubP(q)-1-patches.EdgyInt)*dx;
715   if DQ<width*0.999999
716      warning('too many equispace patches (double overlapping)')
717      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'.[24]

```
734 case 'chebyshev'
735   halfWidth=dx(q)*(nSubP(q)-1)/2;
```

[24] 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??

```
736   Q1 = Xlim(1)+halfWidth-Dom.bcOffset(qq)*dx(q);
737   Q2 = Xlim(2)-halfWidth+Dom.bcOffset(qq+1)*dx(q);
738 %  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`.

```
747   width=(1+patches.EdgyInt)/2*(nSubP(q)-1-patches.EdgyInt)*dx(q);
748   for b=0:2:nPatch(q)-2
749     DQmin=(Q2-Q1-b*width)/2*( 1-cos(pi/(nPatch(q)-b-1)) );
750     if DQmin>width, break, end
751   end%for
752   if DQmin<width*0.999999
753     warning('too many Chebyshev patches (mid-domain overlap)')
754   end%if
```

Assign the centre-patch coordinates.

```
760   Q =[ Q1+(0:b/2-1)*width ...
761        (Q1+Q2)/2-(Q2-Q1-b*width)/2*cos(linspace(0,pi,nPatch(q)-b)) ..
762        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.

```
771 case 'usergiven'
772   if q==1, Q = reshape(Dom.X,1,[]); end
773   if q==2, Q = reshape(Dom.Y,1,[]); end
774   if q==3, Q = reshape(Dom.Z,1,[]); end
775 end%switch Dom.type
```

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

```
782 if q==1, X=Q; end
783 if q==2, Y=Q; end
784 if q==3, Z=Q; end
785 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
```

16.4 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.*[25]

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

```
966 if patches.parallel
967   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.

```
976   [~,pari]=max(nPatch+0.01*(1:3));
977   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.

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

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

```
1002 else% not parallel
1003   if isfield(patches,'codist'), rmfield(patches,'codist'); end
1004 end%if-parallel
```

Fin

```
1013 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. 37).

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. 50, 71, 97).

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. 50, 53, 75, 101).

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.

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. 4, 9, 17).

Maclean, John, J. E. Bunder, and A. J. Roberts (2021). "A toolbox of Equation-Free functions in Matlab/Octave for efficient system level simulation". In: *Numerical Algorithms* 87, pp. 1729–1748. DOI: 10.1007/s11075-020-01027-z (cit. on pp. 86, 114).

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. 50).

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. 50, 53, 75, 101).

Roberts, A. J., Tony MacKenzie, and Judith Bunder (2014). "A dynamical systems approach to simulating macroscale spatial dynamics in multiple dimensions". In: *J. Engineering Mathematics* 86.1, pp. 175–207. DOI: 10.1007/s10665-013-9653-6 (cit. on pp. 71, 85, 97).

Wikipedia (2022). *Divided differences*. https://en.wikipedia.org/wiki/Divided_differences (visited on 12/28/2022) (cit. on pp. 56, 78, 105).