

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

A. J. Roberts*

February 1, 2023

Contents

Examples	3
1 Eckhardt2210eg2: example of a 1D heterogeneous diffusion by simulation on small patches	3
1.1 Simulate heterogeneous diffusion systems	5
1.2 heteroDiffF(): forced heterogeneous diffusion	7
2 EckhardtEquilib: find an equilibrium of a 1D heterogeneous diffusion via small patches	8
3 EckhardtEquilibErrs: explore errors in equilibria of a 1D heterogeneous diffusion on small patches	11
4 Eckhardt2210eg1: example of 1D space-time heterogeneous diffusion via computational homogenisation with projective integration and small patches	16
4.1 Simulate heterogeneous diffusion systems	17
4.2 heteroBurstF(): a burst of heterogeneous diffusion	20
5 monoscaleDiffEquil2: equilibrium of a 2D monoscale heterogeneous diffusion via small patches	20
5.1 monoscaleDiffForce2(): microscale discretisation inside patches of forced diffusion PDE	23

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

6	twoscaleDiffEquil2: equilibrium of a 2D twoscale heterogeneous diffusion via small patches	24
6.1	twoscaleDiffForce2(): microscale discretisation inside patches of forced diffusion PDE	26
7	twoscaleDiffEquil2Errs: errors in equilibria of a 2D twoscale heterogeneous diffusion via small patches	27
7.1	twoscaleDiffForce2(): microscale discretisation inside patches of forced diffusion PDE	34
8	abdulleDiffEquil2: equilibrium of a 2D multiscale heterogeneous diffusion via small patches	35
8.1	abdulleDiffForce2(): microscale discretisation inside patches of forced diffusion PDE	37
9	randAdvecDiffEquil2: equilibrium of a 2D random heterogeneous advection-diffusion via small patches	38
9.1	randAdvecDiffForce2(): microscale discretisation inside patches of forced diffusion PDE	41
10	homoDiffBdryEquil3: equilibrium via computational homogenisation of a 3D heterogeneous diffusion on small patches	42
10.1	microDiffBdry3(): 3D forced heterogeneous diffusion with boundaries	45
11	theRes(): wrapper function to zero	46
New configuration and interpolation		47
12	patchEdgeInt1(): sets patch-edge values from interpolation over the 1D macroscale	47
12.1	Periodic macroscale interpolation schemes	49
12.2	Non-periodic macroscale interpolation	53
13	configPatches1(): configures spatial patches in 1D	55
13.1	If no arguments, then execute an example	59
13.2	Parse input arguments and defaults	60
13.3	The code to make patches and interpolation	63
13.4	Set ensemble inter-patch communication	66

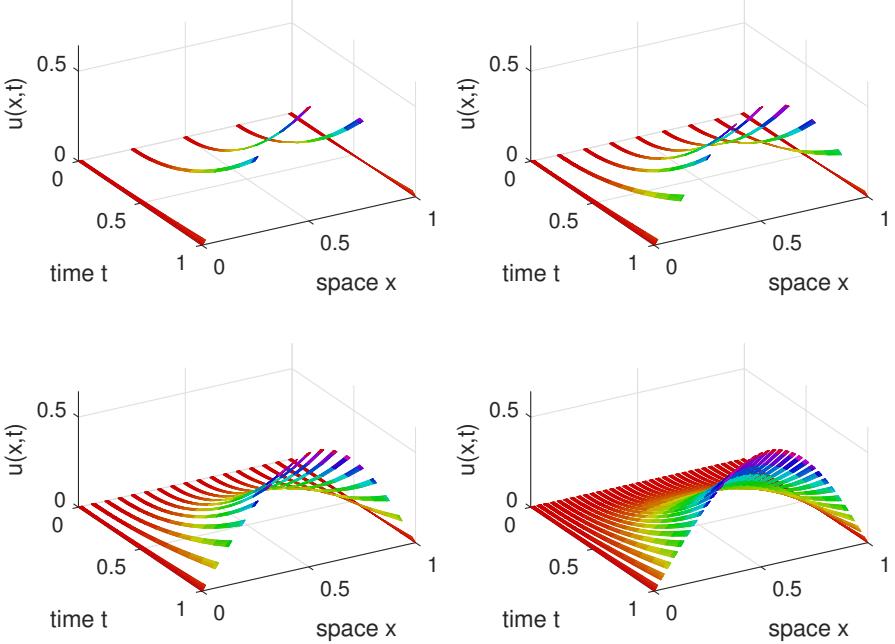
14 patchEdgeInt2(): sets 2D patch edge values from 2D macroscale interpolation	68
14.1 Periodic macroscale interpolation schemes	70
14.2 Non-periodic macroscale interpolation	75
14.2.1 x -direction values	75
14.2.2 y -direction values	76
14.2.3 Optional NaNs for safety	77
15 configPatches2(): configures spatial patches in 2D	78
15.1 If no arguments, then execute an example	82
15.2 Parse input arguments and defaults	85
15.3 The code to make patches	88
15.4 Set ensemble inter-patch communication	91
16 patchEdgeInt3(): sets 3D patch face values from 3D macroscale interpolation	94
16.1 Periodic macroscale interpolation schemes	96
16.2 Non-periodic macroscale interpolation	102
16.2.1 x -direction values	102
16.2.2 y -direction values	103
16.2.3 z -direction values	104
16.2.4 Optional NaNs for safety	105
17 configPatches3(): configures spatial patches in 3D	106
17.1 If no arguments, then execute an example	111
17.2 Parse input arguments and defaults	114
17.3 The code to make patches	117
17.4 Set ensemble inter-patch communication	120

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].

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).



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

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

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.

```
78 clear all
79 %global OurCf2eps, OurCf2eps=true %option to save plots
80 mPeriod = 6
81 y = linspace(0,1,mPeriod+1)';
82 a = 1./(2-cos(2*pi*y(1:mPeriod)));
83 global microTimePeriod; microTimePeriod=0;
```

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

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

Loop to explore errors on various sized patches.

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

Determine indices of patches that are common in various resolutions

```
112 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.EdgeyInt` true means the edge-values come from interpolating the opposite next-to-edge values of the patches (not the mid-patch values).

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

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

```
140 if 0 % given forcing is exact
141     patches.f1=2*( patches.x-patches.x.^2 );
142     patches.f2=2*0.5+0*patches.x;
143 else% simple exp.sine forcing
144     patches.f1=sin(pi*patches.x).*exp(patches.x);
145     patches.f2=pi/2*sin(pi*patches.x).*exp(patches.x);
146 end%if
```

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

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

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

```
173 xs = squeeze(patches.x);
174 us = patchEdgeInt1( permute( reshape(us ...
175     ,length(ts),nSubP,1,nP) ,[2 1 3 4]) );
176 us = squeeze(us);
177 xs(end+1,:) = nan; us(end+1,:,:,:) = nan;
178 uss=reshape(permute(us,[1 3 2]),[],length(ts));
```

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

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

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

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

Assess errors by comparing to the full-domain solution

```
217 DXs=DXs
218 Uerr=squeeze(max(max(abs(Us-Us(:,:,end)))))
219 figure(2),clf,
220 loglog(DXs,Uerr,'o:')
221 xlabel('H'),ylabel('error')
222 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.

```
17 function ut = heteroDiffF(t,u,patches)
```

Cater for the two cases: one of a non-autonomous forcing oscillating in time when `microTimePeriod > 0`, or otherwise the case of an autonomous diffusion constant in time.

```
26 global microTimePeriod
27 if microTimePeriod>0 % optional time fluctuations
28     at = cos(2*pi*t/microTimePeriod)/30;
29 else at=0; end
```

Two basic parameters, and initialise result array to NaNs.

```
35 dx = diff(patches.x(2:3)); % space step
36 i = 2:size(u,1)-1; % interior points in a patch
37 ut = nan+u; % preallocate output array
```

The macroscale Dirichlet boundary conditions are zero at the extreme edges of the two extreme patches.

```
44 u( 1 ,:,:, 1 )=0; % left-edge of leftmost is zero
45 u(end,:,:,:)=0; % right-edge of rightmost is zero
```

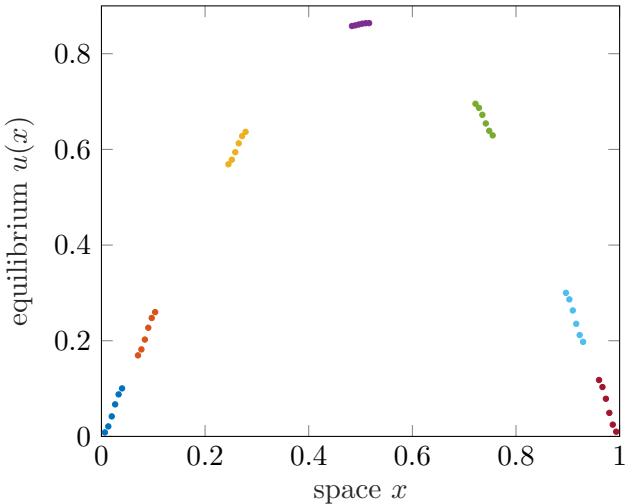
Code the microscale forced diffusion.

```
51 ut(i,:,:,:) = diff((patches.cs(:,1,:)+at).*diff(u))/dx^2 ...
52     +patches.f2(i,:,:,:)*t^2+patches.f1(i,:,:,:)*t;
53 end% function
```

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

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

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.



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].

```

46 clear all
47 global patches
48 %global OurCf2eps, OurCf2eps=true %option to save plots
49 mPeriod = 6
50 y = linspace(0,1,mPeriod+1)';
51 a = 1./(2-cos(2*pi*y(1:mPeriod)))
52 global microTimePeriod; microTimePeriod=0;

```

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

```

61 nPatch = 7
62 nPeriodsPatch = 1 % any integer
63 rEpsilon = 25 % 25 for graphic, up to 2000 say
64 dx = 1/(mPeriod*rEpsilon+1)
65 nSubP = nPeriodsPatch*mPeriod+2

```

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

```

77 ordCC = 4
78 configPatches1(@heteroDiffF,[0 1], 'chebyshev', nPatch ...
79 ,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.

```

88 if 0 % given forcing
89   patches.f1 = 2*( patches.x-patches.x.^2 );
90   patches.f2 = 2*0.5+0*patches.x;
91 else% simple exp-sine forcing
92   patches.f1 = sin(pi*patches.x).*exp(patches.x);
93   patches.f2 = pi/2*sin(pi*patches.x).*exp(patches.x);
94 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). For this linear problem, it is computationally quicker using a linear solver, but **fsolve** is quicker in human time, Start the search from a zero field.

```

107 u = 0*patches.x;

```

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

```

115 u([1 end],:,:, :) = nan;
116 patches.i = find(~isnan(u));

```

Seek the equilibrium, and report the norm of the residual, via the generic patch system wrapper **theRes** ([Section 11](#)).

```

124 [u(patches.i),res] = fsolve(@theRes,u(patches.i));
125 normRes = norm(res)

```

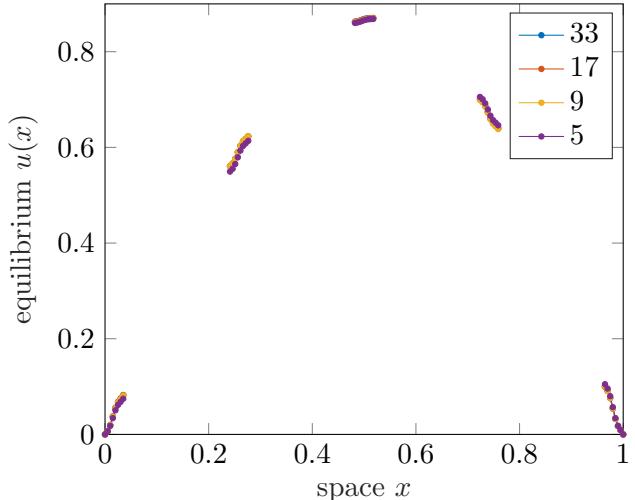
Plot the equilibrium see [Figure 2](#).

```

132 clf, plot(squeeze(patches.x),squeeze(u),'.')
133 xlabel('space $x$'), ylabel('equilibrium $u(x)$')
134 ifOurCf2tex(mfilename)%optionally save

```

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.



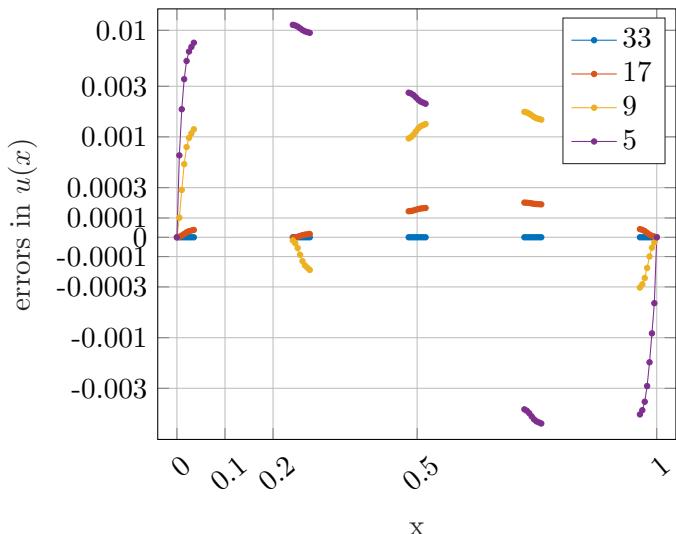
3 EckhardtEquilibErrs: explore errors in equilibria of a 1D heterogeneous diffusion on small patches

Section 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, see Figures 3 and 4. 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

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.



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 appears 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 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).

```

136 clear all
137 global patches
138 %global OurCf2eps, OurCf2eps=true %option to save plots
139 switch 1
140   case 1, Dom.type = 'equispace'
```

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

```

141     case 2, Dom.type = 'chebyshev'
142     case 3, Dom.type = 'usergiven'
143     end% switch
144 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.

```

155 mPeriod = 6
156 z = (0.5:mPeriod)'/mPeriod;
157 a = 1./(2-cos(2*pi*z))
158 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 $\log_2 N_{\max} = 129$ ($\epsilon = 0.008$) in a few seconds:

```

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

```

Set the periodicity ϵ , and other microscale parameters.

```

177 nPeriodsPatch = 1 % any integer
178 nSubP = nPeriodsPatch*mPeriod+2 % for edgy int
179 epsilon = 1/(nPatchMax*nPeriodsPatch+1/mPeriod)
180 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.

```

193 us=[]; xs=[]; nPs=[];
194 for log2N=log2Nmax:-1:2
195 if log2N==log2Nmax
196 Ps=linspace(1,nPatchMax ...
197 ,5-2*all(Dom.type=='chebyshev'))
198 else Ps=(Ps+1)/2
199 end

```

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

```
205 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).

```
216 if all(Dom.type=='usergiven')
217     halfWidth=dx*(nSubP-1)/2;
218     X1 = 0+halfWidth; X2 = 1-halfWidth;
219     Dom.X = (X1+X2)/2-(X2-X1)/2*cos(linspace(0,pi,nPatch));
220 end
```

Configure the patches:

```
226 configPatches1(@heteroDiffF,[0 1],Dom,nPatch ...
227 ,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.

```
236 if 0 %given forcing gives exact answers for ordInt=4 !
237     patches.f1 = 2*( patches.x-patches.x.^2 );
238     patches.f2 = 2*0.5+0*patches.x;
239 else% simple exp-sine forcing
240     patches.f1 = sin(pi*patches.x).*exp(patches.x);
241     patches.f2 = pi/2*sin(pi*patches.x).*exp(patches.x);
242 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.

```
254 if log2N==log2Nmax
255     u0 = zeros(nSubP,1,1,nPatch);
256 else u0 = u0(:, :, :, 1:2:end);
257 end
258 u0([1 end], :) = nan;
259 patches.i = find(~isnan(u0));
260 nVariables = numel(patches.i)
```

Solve via `fsolve` for simplicity and robustness (and using `optimoptions` to omit trace information), via the generic patch system wrapper `theRes` ([Section 11](#)).

```
269 tic;
270 uSoln = fsolve(@theRes,u0(patches.i) ...
271     ,optimoptions('fsolve','Display','off'));
272 fsolveTime = toc
```

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

```
279 normSoln = norm(uSoln,Inf)
280 normResidual = norm(theRes(uSoln),Inf)
281 u0(patches.i) = uSoln;
282 u0 = patchEdgeInt1(u0);
283 u0( 1 ,:,:, 1 ) = 0;
284 u0(end,:,:,:end) = 0;
```

Concatenate the solution on common patches into stores.

```
290 us=cat(3,us,squeeze(u0(:,:,:,Ps)));
291 xs=cat(3,xs,squeeze(patches.x(:,:,:,Ps)));
292 nPs = [nP;nP];
```

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

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

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

```
314 x=xs(:,:,1); u=us;
315 x(end+1,:)=nan; u(end+1,:)=nan;
316 u=reshape(u,numel(x),[]);
```

Reshape solution field.

```
322 figure(1),clf
323 plot(x(:,u,'.-'), legend(num2str(nP));
324 xlabel('space $x$'), ylabel('equilibrium $u(x)$')
325 ifOurCf2tex([mfilename 'us'])%optionally save
```

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

```
333 err = u(:,1)-u;
334 figure(2), clf
335 h=plot(x(:,1),err,'.-'); legend(num2str(nPs))
336 quasiLogAxes(h,10,sqrt(prod(meanAbsErrs(2:3))))
337 xlabel('space $x$'), ylabel('errors in $u(x)$')
338 ifOurCf2tex(mfilename)%optionally save
```

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

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

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.

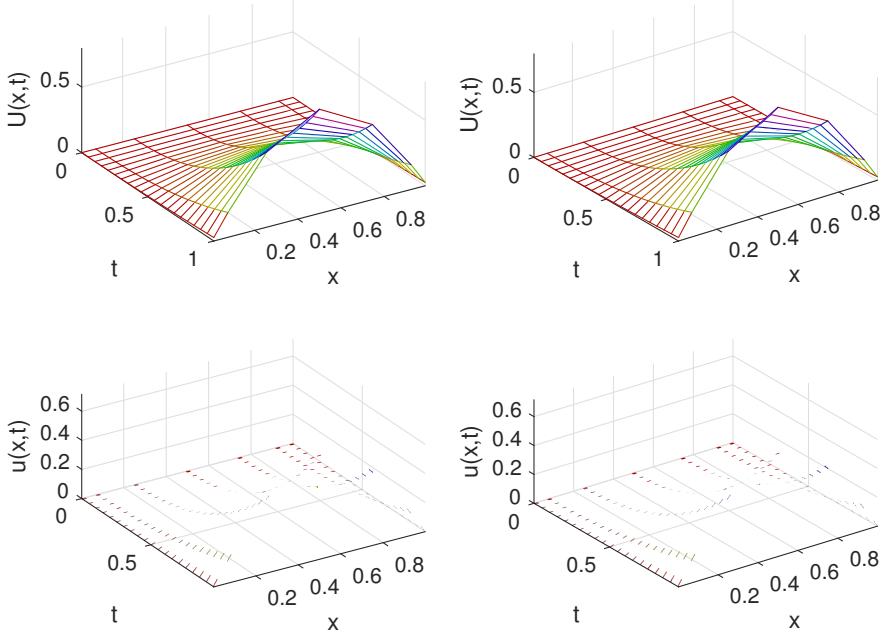
The approximate homogenised PDE is $U_t = A_0 U_{xx} + F$ with $U = 0$ at $x = 0, 1$. Its slowest mode is then $U = \sin(\pi x)e^{-A_0\pi^2 t}$. When $A_0 = 3.3524$ as in Eckhardt then the rate of evolution is about 33 which is relatively fast on the simulation time-scale of $T = 1$. Let's slow down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

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

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

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

Figure 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.



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.

```
98 clear all
99 %global OurCf2eps, OurCf2eps=true %option to save plots
```

```

100 mPeriod = 6
101 y = linspace(0,1,mPeriod+1)';
102 a = ( 3+cos(2*pi*y(1:mPeriod)) )/30
103 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.

```

116 nPeriodsPatch = 2 % any integer
117 rEpsilon = 100
118 dx = 1/(mPeriod*rEpsilon+1)
119 nSubP = nPeriodsPatch*mPeriod+2
120 tol=1e-9;

```

Set the time periodicity (global).

```

126 global microTimePeriod
127 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.EdgeInt` true means the edge-values come from interpolating the opposite next-to-edge values of the patches (not the mid-patch values).

```

144 nPatch = 7
145 ordCC = 4
146 Dom = 'chebyshev'
147 global patches
148 configPatches1(@heteroDiffF,[0 1],Dom,nPatch ...
149 ,ordCC,dx,nSubP,'EdgeInt',true,'hetCoeffs',a);
150 DX = mean(diff(squeeze(patches.x(1,1,1,:))))

```

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

```

158 if 0 % given forcing
159 patches.f1=2*( patches.x-patches.x.^2 );
160 patches.f2=2*0.5+0*patches.x;

```

```

161 else% simple sine forcing
162 patches.f1=sin(pi*patches.x);
163 patches.f2=pi/2*sin(pi*patches.x);
164 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.

```

175 u0 = 0*patches.x;
176 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.

```

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

```

Time the projective integration simulation.

```

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

```

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

```

216 xs=squeeze(patches.x);
217 Xs=mean(xs);
218 Us=squeeze(mean( reshape(us,length(ts),[],nPatch), 2,'omitnan'));
219 figure(1),clf

```

```

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

```

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

```

232 xs([1 end],:) = nan;
233 for k = 1:2, subplot(2,2,2+k)
234   surf(tss,xs(:,),uss', 'EdgeColor','none')
235   ylabel('x'), xlabel('t'), zlabel('u(x,t)')
236   colormap(0.7*hsv), axis tight, view(62-4*k,45)
237 end
238 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).

```

17 function [ts, ucts] = heteroBurstF(ti, ui, bT)
18   global microTimePeriod
19   [ts,ucts] = ode45( @patchSys1,ti+(0:microTimePeriod:bT),ui(:));
20 end

```

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)

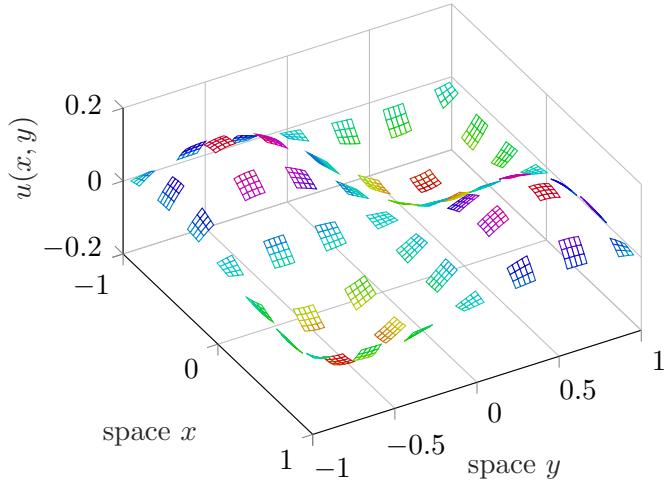
$$u_t = A(x, y) \vec{\nabla} \vec{\nabla} u - f,$$

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

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

² <http://arxiv.org/abs/2211.13731>

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.



and for forcing $f(x, y)$ such that the exact equilibrium is $u = x(1 - e^{1-|x|})y(1 - e^{1-|y|})$. 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.

```
57 clear all
58 global patches
59 %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.

```
70 nPatch = 7
71 nSubP = 5
72 dx = 0.03
```

Specify some order of interpolation.

```
78 configPatches2(@monoscaleDiffForce2, [-1 1 -1 1], 'equispace' ...
79 ,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](#).

```
87 x=patches.x;  y=patches.y;
88 patches.A = sin(pi*x).*sin(pi*y);
89 patches.fu = ...
90     +2*patches.A.*((9*x.^2.*y.^2-3*x.^2-3*y.^2+1) ...
91     +12*x.*y.*((x.^2+y.^2-2));
```

By construction, the PDE has analytic solution

```
97 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.

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

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

```
121 tic;
122 uSoln = fsolve(@theRes,u0(patches.i) ...
123     ,optimoptions('fsolve','Display','off'));
124 solnTime = toc
125 normResidual = norm(theRes(uSoln))
126 normSoln = norm(uSoln)
127 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.

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

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

```
147 figure(1), clf, colormap(0.8* hsv)
148 x(end+1,:,:)=nan; u0(end+1,:,:)=nan;
149 y(:,end+1,:)=nan; u0(:,end+1,:)=nan;
150 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.

```
157 mesh(x(:),y(:),u'); view(60,55)
158 xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')
159 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 .

```
176 function ut = monoscaleDiffForce2(t,u,patches)
177     dx = diff(patches.x(2:3)); % x space step
178     dy = diff(patches.y(2:3)); % y space step
179     i = 2:size(u,1)-1; % x interior points in a patch
180     j = 2:size(u,2)-1; % y interior points in a patch
181     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 , \cdot .

```

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

```

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.

```

41 clear all
42 global patches
43 %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.

```

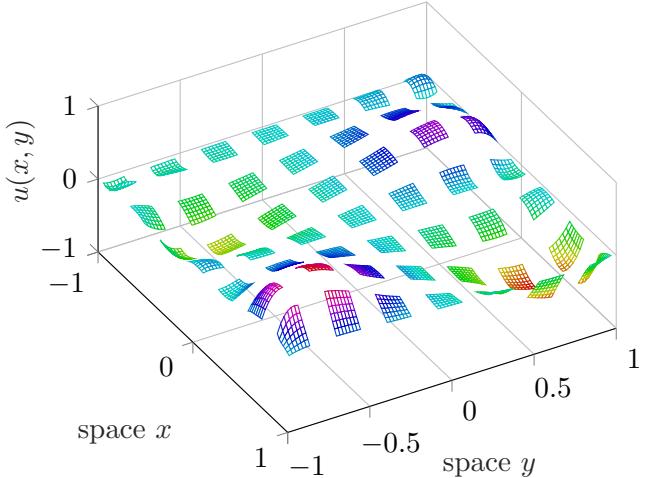
56 mPeriod = 6
57 z = (0.5:mPeriod)'/mPeriod;
58 A = sin(2*pi*z).*sin(2*pi*z');

```

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

³ <http://arxiv.org/abs/2211.13731>

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.



```

65 nPeriodsPatch = 1 % any integer
66 epsilon = 2^(-6) % 4 or 5 to see the patches
67 dx = (2*epsilon)/mPeriod
68 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’:

```

79 nPatch = 7
80 configPatches2(@twoscaleDiffForce2, [-1 1], 'equispace' ...
81 ,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.

```

89 x = patches.x; y = patches.y;
90 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.

```

104 u0 = zeros(nSubP,nSubP,1,1,nPatch,nPatch);
105 u0([1 end],:,:) = nan; u0(:,[1 end],:) = nan;

```

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

Solve by iteration. Use `fsolve` for simplicity and robustness (and using `optimoptions` to omit trace information), via the generic patch system wrapper `theRes` (Section 11), and give magnitudes.

```
116 tic;  
117 uSoln = fsolve(@theRes,u0(patches.i) ...  
118 ,optimoptions('fsolve','Display','off'));  
119 solveTime = toc  
120 normResidual = norm(theRes(uSoln))  
121 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.

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

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

```
141 figure(1), clf, colormap(0.8*hsv)  
142 x(end+1,:,:)=nan; u0(end+1,:,:)=nan;  
143 y(:,end+1,:)=nan; u0(:,end+1,:)=nan;  
144 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.

```
151 mesh(x(:,y(:,u')); view(60,55)  
152 xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')  
153 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`.

```

172 function ut = twoscaleDiffForce2(t,u,patches)
173 dx = diff(patches.x(2:3)); % x space step
174 dy = diff(patches.y(2:3)); % y space step
175 i = 2:size(u,1)-1; % x interior points in a patch
176 j = 2:size(u,2)-1; % y interior points in a patch
177 ut = nan+u; % preallocate output array

```

Set Dirichlet boundary value of zero around the square domain.

```

184 u( 1 ,:,:, :, 1 ,:) = 0; % left edge of left patches
185 u(end,:,:,:,end,:) = 0; % right edge of right patches
186 u(:, 1 ,:,:, :, 1 ) = 0; % bottom edge of bottom patches
187 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 ,:.

```

195 ut(i,j,:)
196 = 2*diff(u(:,j,:),2,1)/dx^2 + 2*diff(u(i,:,:),2,2)/dy^2 ...
197 + 2*patches.cs(i,j).*( u(i+1,j+1,:) - u(i-1,j+1,:) ...
198 - u(i+1,j-1,:) + u(i-1,j-1,:) )/(4*dx*dy) ...
199 - patches.fu(i,j,:);
200 end%function twoscaleDiffForce2

```

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

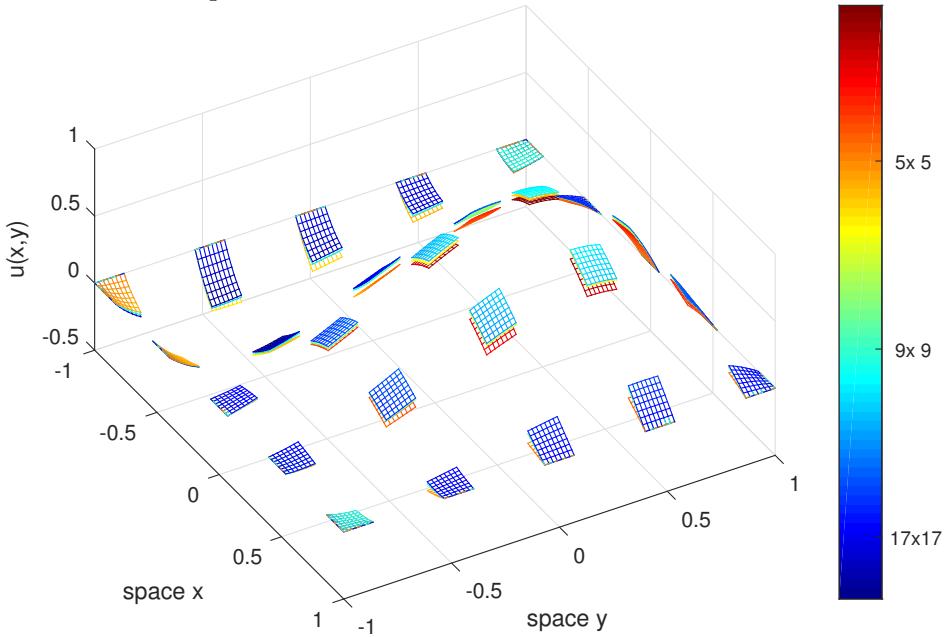
$$A := \begin{bmatrix} 2 & a \\ a & 2 \end{bmatrix} \quad \text{with } a := \sin(\pi x/\epsilon) \sin(\pi y/\epsilon),$$

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 invoke their smaller $\epsilon \approx 0.01$.

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.

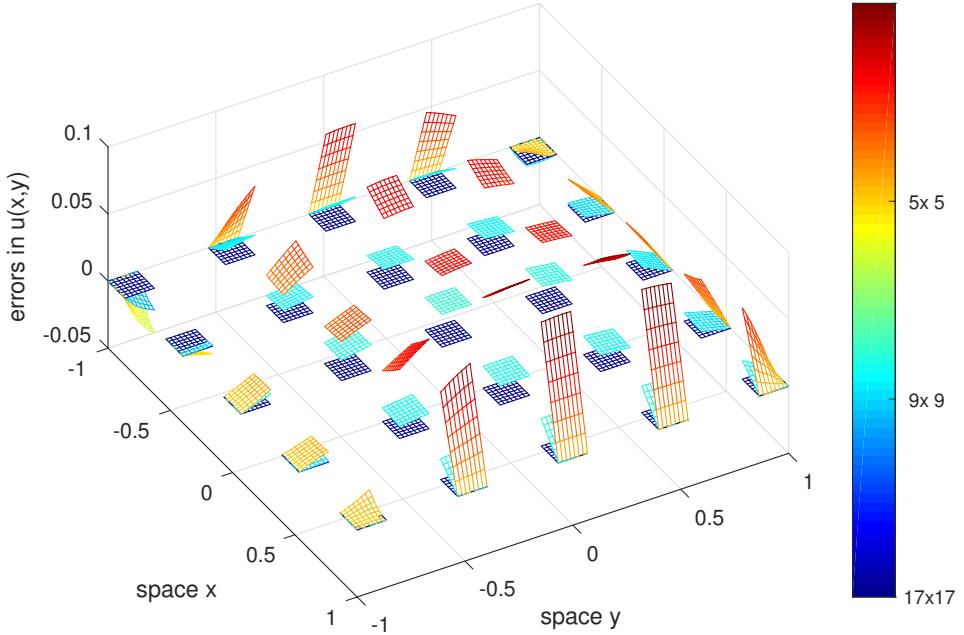


Here we explore the errors for increasing number N of patches (in both directions). Find mean-abs errors to be the following (for different orders of interpolation and patch distribution):

	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).

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.



```

81 clear all
82 global patches
83 %global OurCf2eps, OurCf2eps=true %option to save plot
84 switch 1
85     case 1, Dom.type = 'equispace'
86     case 2, Dom.type = 'chebyshev'
87     case 3, Dom.type = 'usergiven'
88 end% switch
89 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.

```

100 mPeriod = 6
101 z = (0.5:mPeriod)'/mPeriod;

```

```
102 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 $\log_2 N_{\max} = 5$ ($\epsilon = 0.06$) within minutes:

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

Set the periodicity ϵ , and other microscale parameters.

```
120 nPeriodsPatch = 1 % any integer  
121 nSubP = nPeriodsPatch*mPeriod+2 % for edgy int  
122 epsilon = 2/(nPatchMax*nPeriodsPatch+1/mPeriod)  
123 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.

```
136 us=[]; xs=[]; ys=[]; nPs=[];  
137 for log2N=log2Nmax:-1:2  
138     if log2N==log2Nmax  
139         Ps=linspace(1,nPatchMax ...  
140             ,5-2*all(Dom.type=='chebyshev'))  
141     else Ps=(Ps+1)/2  
142 end
```

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

```
148 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).

```
159 if all(Dom.type=='usergiven')  
160     halfWidth = dx*(nSubP-1)/2;  
161     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`, via the generic patch system wrapper `theRes` ([Section 11](#)).

```

207 tic;
208 maxIt = ceil(nVariables/10);
209 rhsb = theRes( zeros(size(patches.i)) );
210 [uSoln,flag] = bicgstab(@(u) rhsb-theRes(u),rhsb ...
211 ,1e-9,maxIt,[],[],u0(patches.i));
212 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.

```

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

```

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

```

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

```

Store the solution into the patches, and give magnitudes— Inf norm is $\max(\text{abs}())$.

```

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

```

Concatenate the solution on common patches into stores.

```

265 us=cat(5,us,squeeze(u0(:,:,(:,:,Ps,Ps))));
266 xs=cat(3,xs,squeeze(patches.x(:,:,(:,:,Ps,:))));
267 ys=cat(3,ys,squeeze(patches.y(:,:,(:,:,Ps,:))));
268 nPs = [nP; 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’).

```

277 end%for log2N
278 assert(max(abs(reshape(diff(xs,1,3),[],1)))<1e-12,'x-coord failure')
279 assert(max(abs(reshape(diff(ys,1,3),[],1)))<1e-12,'y-coord failure')
280 errs = us-us(:,:,(:,:,1));
281 meanAbsErrs = mean(abs(reshape(errs,[],size(us,5))));
282 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.

```

294 x = xs(:,:,1); y = ys(:,:,1); u=us;
295 x(end+1,:)=nan; y(end+1,:)=nan;
296 u(end+1,:,:)=nan; u(:,end+1,:)=nan;
297 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.

```

305 figure(1), clf, colormap(jet)
306 for p=1:size(u,3)
307     mesh(x(:,y(:,u(:,:,p)',p+u(:,:,p)'));
308     hold on;
309 end, hold off
310 view(60,55)
311 colorbar('Ticks',1:size(u,3) ...
312     , 'TickLabels',[num2str(nPs) ['x';'x';'x'] num2str(nPs)]);
313 xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
314 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.

```

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

```

```

329 for p=1:size(u,3)
330     mesh(x(:,y(:,err(:,:,p)',p+err(:,:,p)')/maxAbsErr);
331     hold on;
332 end, hold off
333 view(60,55)
334 colorbar('Ticks',1:size(u,3) ...
335     , 'TickLabels',[num2str(nPs) ['x';'x';'x'] num2str(nPs)]);
336 xlabel('space x'), ylabel('space y')
337 zlabel('errors in u(x,y)')
338 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 .

```

357 function ut = twoscaleDiffForce2(t,u,patches)
358     dx = diff(patches.x(2:3)); % x space step
359     dy = diff(patches.y(2:3)); % y space step
360     i = 2:size(u,1)-1; % x interior points in a patch
361     j = 2:size(u,2)-1; % y interior points in a patch
362     ut = nan+u;           % preallocate output array

```

Set Dirichlet boundary value of zero around the square domain.

```

369 u( 1 ,:,:, :, 1 ,:) = 0; % left edge of left patches
370 u(end,:,:,:,end,:) = 0; % right edge of right patches
371 u(:, 1 ,:,:, :, 1 ) = 0; % bottom edge of bottom patches
372 u(:,end,:,:, :,end) = 0; % top edge of top patches

```

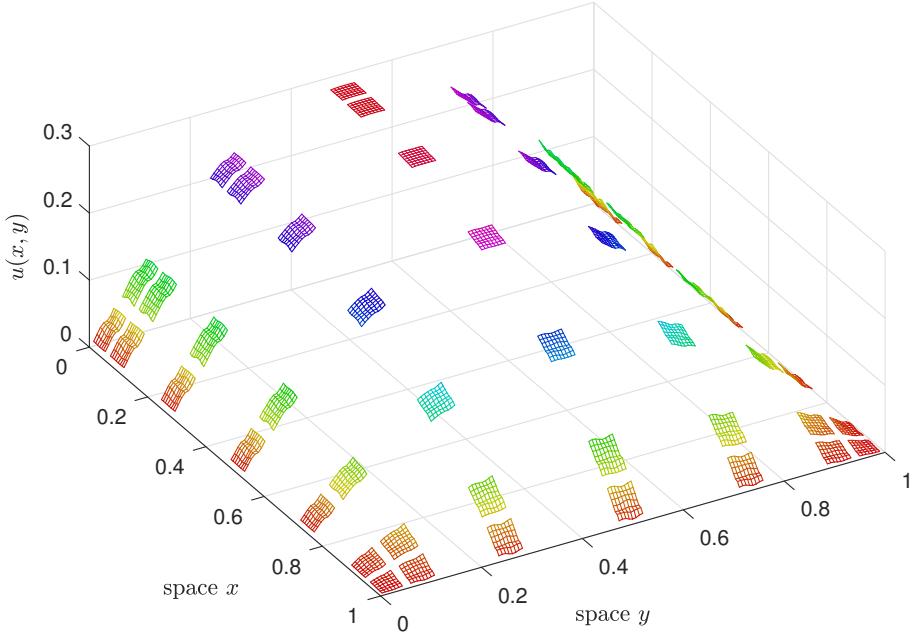
Compute the time derivatives via stored forcing and coefficients. Easier to code by conflating the last four dimensions into the one $,:$.

```

380 ut(i,j,:) ...
381 = 2*diff(u(:,j,:),2,1)/dx^2 +2*diff(u(i,:,:),2,2)/dy^2 ...
382 +2*patches.cs(i,j).*( u(i+1,j+1,:) -u(i-1,j+1,:) ...
383 -u(i+1,j-1,:) +u(i-1,j-1,:) )/(4*dx*dy) ...
384 +patches.fu(i,j,:);
385 end%function twoscaleDiffForce2

```

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.



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.

```

38 clear all
39 global patches
40 %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.)

```

53 mPeriod = 6
54 x = (0.5:mPeriod)'/mPeriod; y=x';
55 a = (2+1.8*sin(2*pi*x))./(2+1.8*sin(2*pi*y)) ...
56 +(2+ sin(2*pi*y))./(2+1.8*sin(2*pi*x));
57 diffusivityRange = [min(a(:)) max(a(:))]

```

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

```

64 epsilon = 0.04
65 dx = epsilon/mPeriod
66 nPeriodsPatch = 1 % any integer
67 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), via the generic patch system wrapper `theRes` (Section 11), and give magnitudes.

```

114 tic;
115 uSoln = fsolve(@theRes,u0(patches.i) ...
116     ,optimoptions('fsolve','Display','off'));
117 solnTime = toc
118 normResidual = norm(theRes(uSoln))
119 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.

```

127 u0(patches.i) = uSoln;
128 u0 = patchEdgeInt2(u0);

```

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

```

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

```

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

```

150 mesh(patches.x(:,patched.y(:,u')); view(60,55)
151 xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')
152 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 .

```

171 function ut = abdulleDiffForce2(t,u,patches)
172 dx = diff(patches.x(2:3)); % x space step
173 dy = diff(patches.y(2:3)); % y space step
174 i = 2:size(u,1)-1; % x interior points in a patch
175 j = 2:size(u,2)-1; % y interior points in a patch
176 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.

```

184 u( 1 ,:,:, :, 1 ,:) = 0; % left edge of left patches
185 u(end,:,:,:,end,:)=0; % right edge of right patches
186 u(:, 1 ,:,:, :, 1 )=0; % bottom edge of bottom patches
187 u(:,end,:,:,:,end)=0; % top edge of top patches
188 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 , :.

```

196 ut(i,j,:) = diff(patches.cs(:,j).*diff(u(:,j,:)))/dx^2 ...
197 + diff(patches.cs(i,:).*diff(u(i,:,:),1,2),1,2)/dy^2 ...
198 + 10;
199 end%function abdulleDiffForce2

```

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)

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

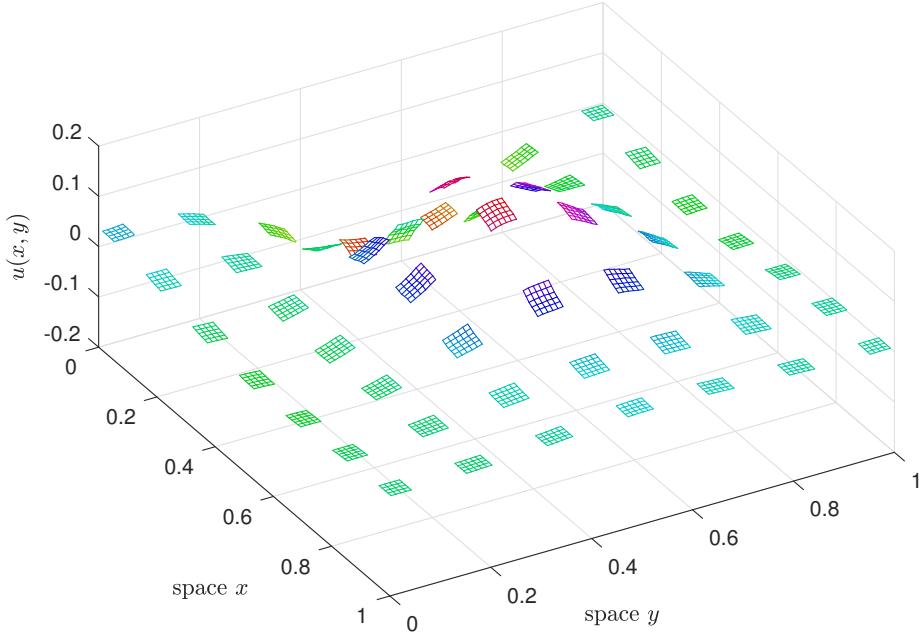
on domain $[0, 1]^2$ with Neumann boundary conditions, for microscale random pseudo-diffusion and pseudo-advection coefficients, $\mu_1(x, y) \in [0.01, 0.1]^8$ and $\mu_2(x, y) \in [0, 2\pi)$, and for forcing

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

⁷ <http://arxiv.org/abs/2211.15221>

⁸More 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.



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.

```
50 clear all
51 global patches
52 %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.

```
63 mPeriod = 4
```

```

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

```

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

```

74 epsilon = 0.04
75 dx = epsilon/mPeriod
76 nPeriodsPatch = 1 % any integer
77 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:

```

89 nPatch = 7
90 Dom.type= 'equispace';
91 Dom.bcOffset = 0.5;
92 configPatches2(@randAdvecDiffForce2,[0 1],Dom ...
93 ,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](#).

```

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

```

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

```

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

```

Solve by iteration. Use `fsolve` for simplicity and robustness (and using `optimoptions` to omit trace information), via the generic patch system wrapper `theRes` ([Section 11](#)).

```

130 tic;
131 uSoln = fsolve(@theRes,u0(patches.i) ...
132     ,optimoptions('fsolve','Display','off'));
133 solnTime = toc
134 normResidual = norm(theRes(uSoln))
135 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.

```

143 u0(patches.i) = uSoln;
144 u0 = patchEdgeInt2(u0);

```

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

```

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

```

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

```

166 mesh(patches.x(:),patches.y(:),u'); view(60,55)
167 xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')
168 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 .

```

187 function ut = randAdvecDiffForce2(t,u,patches)
188 dx = diff(patches.x(2:3)); % x space step
189 dy = diff(patches.y(2:3)); % y space step
190 i = 2:size(u,1)-1; % x interior points in a patch
191 j = 2:size(u,2)-1; % y interior points in a patch
192 ut = nant+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.

```

200 u( 1 ,:,:, :, 1 ,:) = u( 2 ,:,:, :, 1 ,:); % left edge of left patche
201 u(end,:,:, :,end,:) = u(end-1,:,:, :,end,:); % right edge of right patc
202 u(:, 1 ,:,:, :, 1 ) = u(:, 2 ,:,:, :, 1 ); % bottom edge of bottom pa
203 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 ,:.

```

211 ut(i,j,:)
212 = patches.cs(i,j,1).*(diff(u(:,j,:),2,1)/dx^2 ...
213 +diff(u(:, :, :),2,2)/dy^2)...
214 -patches.cs(i,j,2).*(u(i+1,j,:)-u(i-1,j,:))/(2*dx) ...
215 -patches.cs(i,j,3).*(u(i,j+1,:)-u(i,j-1,:))/(2*dy) ...
216 -u(i,j,:)+patches.fu(i,j,:);
217 end%function randAdvecDiffForce2

```

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.

```

33 clear all
34 global patches
35 %global OurCf2eps, OurCf2eps=true %option to save plots

```

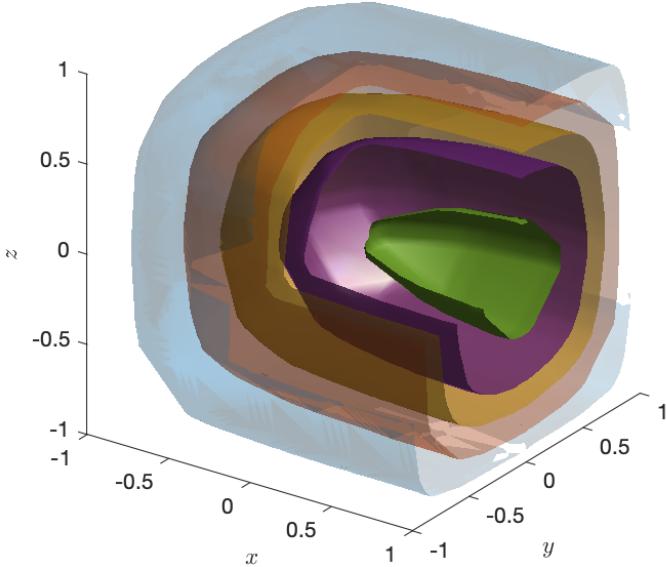
Set random heterogeneous diffusivities of random (small) period in each of the three directions. Crudely normalise by the harmonic mean so the decay time scale is roughly one.

```

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

```

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.



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.

```

59 nSubP = mPeriod+2;
60 nPatch = 5;
61 Dom.type = 'equispace';
62 Dom.bcOffset = zeros(2,3); Dom.bcOffset(2) = 0.5;
63 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 `theRes`, 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,[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), via the generic patch system wrapper `theRes` ([Section 11](#)).

```
101 disp('Solving system, takes 10--40 secs'),tic  
102 uSoln = fsolve(@theRes,u0(patches.i) ...  
103     ,optimoptions('fsolve','Display','off'));  
104 solveTime = toc  
105 normResidual = norm(theRes(uSoln))  
106 normSoln = norm(uSoln)
```

Store the solution into the patches, and give magnitudes.

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

Plot isosurfaces of the solution

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

Reshape spatial coordinates of patches.

```
129 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.

```

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

```

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

```

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

```

Draw some isosurfaces.

```

151 clf;
152 for iso=5:-1:1
153     isov=(iso-0.5)/5*(maxu-minu)+minu;
154     hsurf(iso) = patch(isosurface(x,y,z,u, isov));
155     isonormals(x,y,z,u,hsurf(iso))
156     set(hsurf(iso) , 'FaceColor',rgb(iso,:)... ...
157         , 'EdgeColor', 'none' , 'FaceAlpha',iso/5);
158     hold on
159 end
160 hold off
161 axis equal, axis([-1 1 -1 1 -1 1]), view(35,25)
162 xlabel('$x$'), ylabel('$y$'), zlabel('$z$')
163 camlight, lighting gouraud
164 ifOurCf2eps(mfilename) %optionally save plot
165 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_{ijk}^x , c_{ijk}^y and c_{ijk}^z , 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.

```

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

```

Microscale space-steps.

```

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

```

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

```

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

```

11 theRes(): wrapper 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 at time $t = 1$, and returns the vector of patch-interior time derivatives.

```

15 function f=theRes(u)
16 global patches
17 switch numel(size(patches.x))
18 case 4, pSys = @patchSys1;
19 v=nan(size(patches.x));
20 case 5, pSys = @patchSys2;

```

```

21         v=nan(size(patches.x+patches.y));
22 case 6, pSys = @patchSys3;
23         v=nan(size(patches.x+patches.y+patches.z));
24 otherwise error('number of dimensions is somehow wrong')
25 end%switch
26 v(patches.i) = u;
27 f = pSys(1,v(:,),patches);
28 f = f(patches.i);
29 end%function theRes

```

New configuration and interpolation

12 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 $n_{SubP} \cdot n_{Vars} \cdot n_{Ensem} \cdot n_{Patch}$ where there are $n_{Vars} \cdot n_{Ensem}$ field values at each of the points in the $n_{SubP} \times n_{Patch}$ multiscale spatial grid.

⁹Script patchEdgeInt1test.m verifies this code.

- patches a struct largely set by `configPatches1()`, and which includes the following.

- `.x` is $n_{SubP} \times 1 \times 1 \times n_{Patch}$ array of the spatial locations x_{iI} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index i , but may be variable spaced in macroscale index I .
- `.ordCC` is order of interpolation, integer ≥ -1 .
- `.periodic` indicates whether macroscale is periodic domain, or alternatively that the macroscale has left and right boundaries so interpolation is via divided differences.
- `.stag` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation, and zero for ordinary grid.
- `.Cwtsr` and `.Cwtsl` are the coupling coefficients for finite width interpolation—when invoking a periodic domain.
- `.EdgyInt`, true/false, for determining patch-edge values by interpolation: true, from opposite-edge next-to-edge values (often preserves symmetry); false, from centre-patch values (original scheme).
- `.nEnsem` the number of realisations in the ensemble.
- `.parallel` whether serial or parallel.
- `.nCore`¹⁰
- ¹¹

Output

- `u` is 4D array, $n_{SubP} \times n_{Vars} \times n_{Ensem} \times n_{Patch}$, 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
```

¹⁰**ToDo:** introduced sometime but not fully implemented yet, because prefer ensemble

¹¹**ToDo:** additional macros bdry info

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);
```

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.

```
144 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).

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

12.1 Periodic macroscale interpolation schemes

```
161 if patches.periodic
```

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

```
168 r = patches.ratio(1);  
169 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.

¹²**ToDo:** Revise??

```

179 if patches.EdgyInt % interpolate next-to-edge values
180   Ux = u([2 nx-1],:,:,I);
181 else % interpolate mid-patch values/sums
182   Ux = sum( u((i0-c):(i0+c),:,:,I) ,1);
183 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.

```

191 szUx0=size(Ux);
192 szUx0=[szUx0 ones(1,4-length(szUx0)) patches.ordCC];

```

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

```

201 if patches.parallel
202   dmu = zeros(szUx0,patches.codist); % 5D
203 else
204   dmu = zeros(szUx0); % 5D
205 end

```

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

```

212 if patches.stag % use only odd numbered neighbours
213   dmu(:,:,:,:,I,1) = (Ux(:,:,:,:,Ip)+Ux(:,:,:,:,Im))/2; % \mu
214   dmu(:,:,:,:,I,2) = (Ux(:,:,:,:,Ip)-Ux(:,:,:,:,Im)); % \delta
215   Ip = Ip(Ip); Im = Im(Im); % increase shifts to \pm2
216 else % standard
217   dmu(:,:,:,:,I,1) = (Ux(:,:,:,:,Ip)-Ux(:,:,:,:,Im))/2; % \mu\delta
218   dmu(:,:,:,:,I,2) = (Ux(:,:,:,:,Ip)-2*Ux(:,:,:,:,I) ...
219                           +Ux(:,:,:,:,Im)); % \delta^2
220 end%if patches.stag

```

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

```

227 for k = 3:patches.ordCC
228   dmu(:,:,:,:,k) =      dmu(:,:,:,:,Ip,k-2) ...
229     -2*dmu(:,:,:,:,I,k-2) +dmu(:,:,:,:,Im,k-2);
230 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`.

```

245 if patches.nCore==1
246   k=1+patches.EdgyInt; % use centre/core or two edges
247   u(nx,:,:,:,I) = Ux(1,:,:,:)*(1-patches.stag) ...
248     +sum( shiftdim(patches.Cwtsr,-4).*dmu(1,:,:,:,:) ,5);
249   u(1,:,:,:,I) = Ux(k,:,:,:)*(1-patches.stag) ...
250     +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.

```

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

```

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

```

278 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’ $|k| < \text{Nx}/2$, so set row vector $\mathbf{ks} = k2\pi/N$ for ‘wavenumbers’ $k = (0, 1, \dots, k_{\max}, -k_{\max}, \dots, -1)$ for odd N , and $k = (0, 1, \dots, k_{\max}, (k_{\max}+1), -k_{\max}, \dots, -1)$ 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??

```

302 if patches.stag % transform by doubling the number of fields
303   v = nan(size(u)); % currently to restore the shape of u
304   u = [u,:,:,:,1:2:Nx) u,:,:,:,2:2:Nx];
305   stagShift = 0.5*[ones(1,nVars) -ones(1,nVars)];
306   iV = [nVars+1:2*nVars 1:nVars]; % scatter interp to alternate fie
307   r = r/2; % ratio effectively halved
308   Nx = Nx/2; % halve the number of patches
309   nVars = nVars*2; % double the number of fields
310 else % the values for standard spectral
311   stagShift = 0;
312   iV = 1:nVars;
313 end%if patches.stag

```

Now set wavenumbers (when N_x 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,:,:);
366 u = v;
367 end%if patches.stag
368 end%if patches.ordCC

```

12.2 Non-periodic macroscale interpolation

```

376 else% patches.periodic false
377 assert(~patches.stag, ...
378 '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.

```

385 if patches.ordCC<1, patches.ordCC = Nx-1; end
386 p = min(patches.ordCC,Nx-1);
387 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.

```

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

```

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

```

411 for q = 1:p
412   i = 1:Nx-q;
413   F(:,:,:,:,i,q+1) = (F(:,:,:,:,i+1,q)-F(:,:,:,:,i,q)) ...
414           ./ (X(:,:,:,:,i+q) - X(:,:,:,:,i));
415 end

```

Now interpolate to the edge-values at locations X_{edge} .

```

421 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.

```

437 if true
438   i = max(1,min(1:Nx,Nx-ceil(p/2))-floor(p/2));
439   Uedge = F(:,:,:,:,i,p+1);
440   for q = p:-1:1
441     Uedge = F(:,:,:,:,i,q)+(Xedge-X(:,:,:,:,i+q-1)).*Uedge;
442   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

```

453 else%if false
454   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.

```

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

```

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??

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

Initialise to highest divide difference, surrounded by zeros.

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

Complete Horner evaluation of the relevant polynomials.

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

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

```
490 u(1 ,:,patches.le,I) = Uedge(1,:,:,:,I);  
491 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.

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

End of the non-periodic interpolation code.

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

Fin, returning the 4D array of field values.

13 configPatches1(): configures spatial patches in 1D

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys1()`. [Section 13.1](#) lists an example of its use.

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

Input If invoked with no input arguments, then executes an example of simulating Burgers' PDE—see [Section 13.1](#) 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 $\text{ratio} = \frac{1}{2}$ 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`.

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

- `.x` (4D) is $\text{nSubP} \times 1 \times 1 \times \text{nPatch}$ array of the regular spatial locations x_{iI} of the i th microscale grid point in the I th 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, $(\text{nSubP}(1) - 1) \times n_c$ 2D array of microscale heterogeneous coefficients, or
 - if `nEnsem` > 1, $(\text{nSubP}(1) - 1) \times n_c \times m_x$ 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.

13.1 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] = ode0cts(@patchSys1,[0 0.5],u0(:));
292 end

```

Plot the simulation using only the microscale values interior to the patches: either set x -edges to `nan` to leave the gaps; or use `patchEdgyInt1` to re-interpolate correct patch edge values and thereby join the patches. [Figure 13](#) illustrates an example simulation in time generated by the patch scheme applied to Burgers' PDE.

```

304 figure(1),clf
305 if 1, patches.x([1 end],:,:, :)=nan; us=us.';
306 else us=reshape(patchEdgyInt1(us.'),[],length(ts));
307 end
308 surf(ts,patches.x(:,us)
309 view(60,40), colormap(0.8*hsv)
310 title('Burgers PDE: patches in space, continuous time')
311 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.

```

325 ifOurCf2eps(mfilename)
326 return
327 end%if nargin==0

```

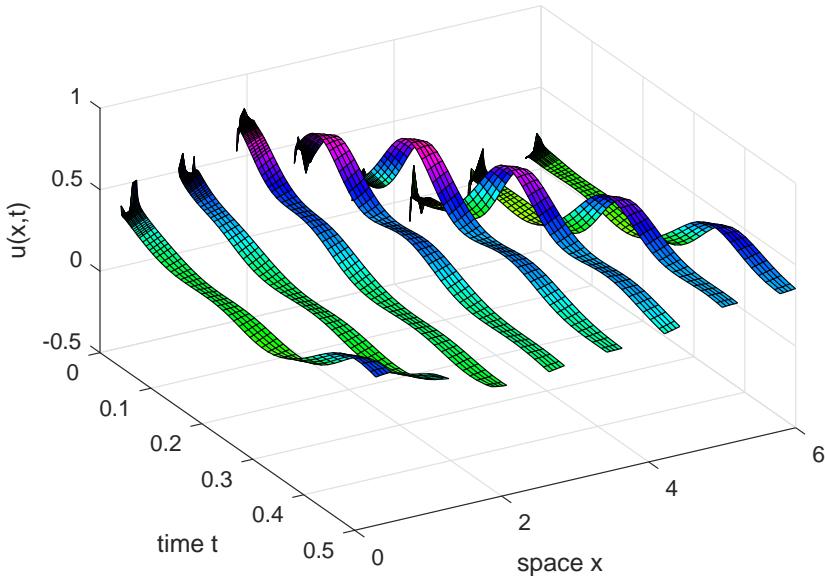
13.2 Parse input arguments and defaults

```

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

```

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



```

344 addRequired(p,'fun',fnValidation);
345 addRequired(p,'Xlim',@isnumeric);
346 %addRequired(p,'Dom'); % nothing yet decided
347 addRequired(p,'nPatch',@isnumeric);
348 addRequired(p,'ordCC',@isnumeric);
349 addRequired(p,'dx',@isnumeric);
350 addRequired(p,'nSubP',@isnumeric);
351 addParameter(p,'nEdge',1,@isnumeric);
352 addParameter(p,'EdgyInt',false,@islogical);
353 addParameter(p,'nEnsem',1,@isnumeric);
354 addParameter(p,'hetCoeffs',[],@isnumeric);
355 addParameter(p,'parallel',false,@islogical);
356 addParameter(p,'nCore',1,@isnumeric);
357 parse(p,fun,Xlim,nPatch,ordCC,dx,nSubP,varargin{:});
```

Set the optional parameters.

```

363 patches.nEdge = p.Results.nEdge;
364 patches.EdgyInt = p.Results.EdgyInt;
365 patches.nEnsem = p.Results.nEnsem;
```

```
366 cs = p.Results.hetCoeffs;  
367 patches.parallel = p.Results.parallel;  
368 patches.nCore = p.Results.nCore;
```

Check parameters.

```
375 assert(Xlim(1)<Xlim(2) ...  
376     , 'two entries of Xlim must be ordered increasing')  
377 assert(patches.nEdge==1 ...  
378     , 'multi-edge-value interp not yet implemented')  
379 assert(2*patches.nEdge+1<=nSubP ...  
380     , 'too many edge values requested')  
381 if patches.nCore>1  
382     warning('nCore>1 not yet tested in this version')  
383 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.

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

Default macroscale conditions are periodic with evenly spaced patches.

```
403 if isempty(Dom), Dom=struct('type','periodic'); end  
404 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.

```
412 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.

```
420 patches.periodic=false;  
421 switch Dom.type  
422 case 'periodic'  
423     patches.periodic=true;  
424     if isfield(Dom,'bcOffset')  
425         warning('bcOffset not available for Dom.type = periodic'), end  
426     if isfield(Dom,'X')
```

```

427     warning('X not available for Dom.type = periodic'), end
428 case {'equispace','chebyshev'}
429     if ~isfield(Dom,'bcOffset'), Dom.bcOffset=[0;0]; end
430     if length(Dom.bcOffset)==1
431         Dom.bcOffset=repmat(Dom.bcOffset,2,1); end
432     if isfield(Dom,'X')
433         warning('X not available for Dom.type = equispace or chebyshev')
434     end
435 case 'usergiven'
436     if isfield(Dom,'bcOffset')
437         warning('bcOffset not available for usergiven Dom.type'), end
438         assert(isfield(Dom,'X'),'X required for Dom.type = usergiven')
439 otherwise
440     error(['Dom.type 'is unknown Dom.type'])
441 end%switch Dom.type

```

13.3 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.)

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

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

```
524 case 'equispace'
525     X=linspace(Xlim(1)+((nSubP-1)/2-Dom.bcOffset(1))*dx ...
526                     ,Xlim(2)-((nSubP-1)/2-Dom.bcOffset(2))*dx ,nPatch);
527     DX=diff(X(1:2));
528     width=(1+patches.EdgyInt)/2*(nSubP-1-patches.EdgyInt)*dx;
529     if DX<width*0.999999
530         warning('too many equispace patches (double overlapping)')
531     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

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??

```

548 case 'chebyshev'
549 halfWidth=dx*(nSubP-1)/2;
550 X1 = Xlim(1)+halfWidth-Dom.bcOffset(1)*dx;
551 X2 = Xlim(2)-halfWidth+Dom.bcOffset(2)*dx;
552 % 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`.

```

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

```

Assign the centre-patch coordinates.

```

574 X = [ X1+(0:b/2-1)*width ...
575         (X1+X2)/2-(X2-X1-b*width)/2*cos(linspace(0,pi,nPatch-b)) ...
576         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.

```

585 case 'usergiven'
586     X = reshape(Dom.X,1,[]);
587 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).

```

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

```

13.4 Set ensemble inter-patch communication

For EdgyInt or centre interpolation respectively,

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

`re` and `li` are ‘transposes’ of each other as `re(li)=le(ri)` are both `1:nEnsem`.

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()`.

```
630 nE = patches.nEnsem;
631 patches.le = 1:nE;
632 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.

```
644 if ~isempty(cs)
645 [mx,nc] = size(cs);
646 nx = nSubP(1);
647 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.

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

But for $n_{\text{Ensem}} > 1$ 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.

```
664 patches.nEnsem = mx;
665 patches.cs = nan(nx-1,nc,mx);
```

```

666     for i = 1:mx
667         is = (i:i+nx-2);
668         patches.cs(:,:,i) = cs(is,:);
669     end
670     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.

```

680     patches.le = mod((0:mx-1)' + mod(nx-2,mx),mx)+1;
681     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);

```

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

`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.

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

Fin

```
762 end% function
```

14 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')
```

¹⁸Script `patchEdgeInt2test.m` verifies this code.

Input

- \mathbf{u} is a vector/array of length $\text{prod}(\text{nSubP}) \cdot \text{nVars} \cdot \text{nEnsem} \cdot \text{prod}(\text{nPatch})$ where there are $\text{nVars} \cdot \text{nEnsem}$ field values at each of the points in the $\text{nSubP1} \cdot \text{nSubP2} \cdot \text{nPatch1} \cdot \text{nPatch2}$ multiscale spatial grid on the $\text{nPatch1} \cdot \text{nPatch2}$ array of patches.
- **patches** a struct set by `configPatches2()` which includes the following information.
 - $.x$ is $\text{nSubP1} \times 1 \times 1 \times 1 \times \text{nPatch1} \times 1$ array of the spatial locations x_{iI} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index i , but may be variable spaced in macroscale index I .
 - $.y$ is similarly $1 \times \text{nSubP2} \times 1 \times 1 \times 1 \times \text{nPatch2}$ 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 $\{0, 2, 4, \dots\}$
 - $.periodic$ 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 $.Cwts1$ 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

- \mathbf{u} is 6D array, $\text{nSubP1} \cdot \text{nSubP2} \cdot \text{nVars} \cdot \text{nEnsem} \cdot \text{nPatch1} \cdot \text{nPatch2}$, of the fields with edge values set by interpolation (and corner values 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.

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

```
147 I=1:Nx; Ip=mod(I,Nx)+1; Im=mod(I-2,Nx)+1;
148 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

```
155 i0 = round((nx+1)/2);
156 j0 = round((ny+1)/2);
157 %disp('finished common preamble')
```

14.1 Periodic macroscale interpolation schemes

```
166 if patches.periodic
```

Get the size ratios of the patches.

```
172 rx = patches.ratio(1);
173 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.

```
183 ordCC = patches.ordCC;
184 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 ($\mu\delta, \delta^2, \mu\delta^3, \delta^4, \dots$) 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; % \mu\delta
235 dmux(:,:,,:,I,:,2) = (Ux(:,:,,:,Ip,:)-Ux(:,:,,:,Im,:)); % \delta\mu
236 Ip = Ip(Ip); Im = Im(Im); % increase shifts to \pm2
237 dmuy(:,:,,:,J,1) = (Ux(:,:,,:,Jp)+Ux(:,:,,:,Jm))/2; % \mu\delta
238 dmuy(:,:,,:,J,2) = (Ux(:,:,,:,Jp)-Ux(:,:,,:,Jm)); % \delta\mu
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,:)
244 % -2*Ux(:,:,,:,I,:)+Ux(:,:,,:,Im,:); % \delta^2
245 dmuy(:,:,,:,J,1) = (Uy(:,:,,:,Jp))
246 % -Uy(:,:,,:,Jm))/2; %\mu\delta
247 dmuy(:,:,,:,J,2) = Uy(:,:,,:,Jp)
248 % -2*Uy(:,:,,:,J)+Uy(:,:,,:,Jm); % \delta^2
249 end% if patches.stag

```

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

```

256 for k = 3:ordCC
257 dmux(:,:,,:,I,:,k) = dmux(:,:,,:,Ip,:,k-2)
258 % -2*dmux(:,:,,:,I,:,k-2)+dmux(:,:,,:,Im,:,k-2);
259 dmuy(:,:,,:,J,k) = dmuy(:,:,,:,Jp,k-2)
260 % -2*dmuy(:,:,,:,J,k-2)+dmuy(:,:,,:,Jm,k-2);
261 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`.

```

276 k=1+patches.EdgyInt; % use centre or two edges
277 u(nx,2:(ny-1),:,patches.ri,I,:) ...
278 = Ux(1,:,:,:,:,:)*(1-patches.stag) ...
279 +sum( shiftdim(patches.Cwtsr(:,1),-6).*dmux(1,:,:,:,:,:),7);
280 u(1,2:(ny-1),:,patches.le,I,:) ...
281 = Ux(k,:,:,:,:,:)*(1-patches.stag) ...

```

```

282     +sum( shiftdim(patches.Cwtsl(:,1),-6).*dmux(k,:,:,:, :, :) ,7);
283 u(2:(nx-1),ny,:,:atches.to,:,:J) ...
284 = Uy(:,1,:,:,:, :)*(1-patches.stag) ...
285 +sum( shiftdim(patches.Cwtsr(:,2),-6).*dmuy(:,1,:,:,:, :, :) ,7);
286 u(2:(nx-1),1 ,:,atches.bo,:,:J) ...
287 = Uy(:,k,:,:,:, :)*(1-patches.stag) ...
288 +sum( shiftdim(patches.Cwtsl(:,2),-6).*dmuy(:,k,:,:,:, :, :) ,7);
289 u([1 nx],[1 ny],:,:,:,:)=nan; % remove corner values

```

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

```

299 else% patches.ordCC<=0, spectral interpolation
300 %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 $\mathbf{ks} = k2\pi/N$ for ‘wavenumbers’ $k = (0, 1, \dots, k_{\max}, -k_{\max}, \dots, -1)$ for odd N , and $k = (0, 1, \dots, k_{\max}, \pm(k_{\max} + 1) - k_{\max}, \dots, -1)$ 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.

```

323 if patches.stag % transform by doubling the number of fields
324 error('staggered grid not yet implemented??')
325 v=nan(size(u)); % currently to restore the shape of u
326 u=cat(3,u(:,1:2:nPatch,:),u(:,2:2:nPatch,:));
327 stagShift=reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
328 iV=[nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
329 r=r/2; % ratio effectively halved
330 nPatch=nPatch/2; % halve the number of patches
331 nVars=nVars*2; % double the number of fields
332 else % the values for standard spectral
333     stagShift = 0;
334     iV = 1:nVars;
335 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, $k = (0, 1, \dots, k_{\max}, +k_{\max} + 1) - k_{\max}, \dots, -1)$.

```
345 kMax = floor((Nx-1)/2);
346 krx = shiftdim( rx*2*pi/Nx*(mod((0:Nx-1)+kMax,Nx)-kMax) , -3);
347 kMay = floor((Ny-1)/2);
348 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`.

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

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

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

14.2 Non-periodic macroscale interpolation

```
399 else% patches.periodic false
400 %disp('executing new non-periodic code')
401 assert(~patches.stag, ...
402 '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.

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

14.2.1 x-direction values

Set function values in first ‘column’ of the tables for every variable and across ensemble. For EdgyInt, the ‘reversal’ of the next-to-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.)

```
433 F = nan(patches.EdgyInt+1,ny-2,nVars,nEnsem,Nx,Ny,px+1);
434 if patches.EdgyInt % interpolate next-to-edge values
435     F(:,:,(:,:,1,:,:,1)) = u([nx-1 2],iy,:,:,:,:,:);
436     X = patches.x([nx-1 2],(:,:,1,:,:,1));
437 else % interpolate mid-patch cross-patch values
438     F(:,:,(:,:,1,:,:,1)) = u(i0,iy,:,:,:,:,:);
439     X = patches.x(i0,:,:,:,:,:);
440 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.

```

451 for q = 1:px
452     i = 1:Nx-q;
453     F(:,:,i,:,:,q+1) ...
454     = (F(:,:,i+1,:,:,q)-F(:,:,i,:,:,q)) ...
455     ./ (X(:,:,i+q,:,:) - X(:,:,i,:));
456 end

```

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

```

465 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.

```

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

```

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

```

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

```

14.2.2 y -direction values

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

```

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

```

Form tables of divided differences.

```
511 for q = 1:py  
512     j = 1:Ny-q;  
513     F(:,:,(:,:,j,q+1) ...  
514     = (F(:,:,(:,:,j+1 ,q)-F(:,:,(:,:,j,q)) ...  
515     ./ (Y(:,:,(:,:,j+q) -Y(:,:,(:,:,j));  
516 end
```

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

```
523 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.

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

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

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

14.2.3 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`.

```
557 u( 1,:,:,:, 1,:)=nan;  
558 u(nx,:,:,:,Nx,:)=nan;  
559 u(:, 1,:,:,:, 1)=nan;  
560 u(:,ny,:,:,:,Ny)=nan;
```

End of the non-periodic interpolation code.

```
567 %disp('finished new non-periodic code')  
568 end%if patches.periodic else
```

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

577 end% function patchEdgeInt2

15 configPatches2(): configures spatial patches in 2D

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys2()`. [Section 15.1](#) 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 15.1](#) for an example code.

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)`, that computes time-derivatives (or time-steps) of quantities on the 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, . . . ; 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 = 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 `.Cwts1`, only for macro-periodic conditions, are the `ordCC × 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) × 1 × 1 × 1 × nPatch(1) × 1` array of the regular spatial locations x_{iI} of the microscale grid points in every patch.
- `.y` (6D) is `1 × nSubP(2) × 1 × 1 × 1 × nPatch(2)` array of the regular spatial locations y_{jJ} of the microscale grid points in every patch.

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

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

15.1 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.

```

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

```

Start by showing the initial conditions of [Figure 14](#) while the simulation computes.

```

339 u = reshape(permute(squeeze(u0) ...
340     ,[1 3 2 4]), [numel(x) numel(y)]);
341 hsurf = surf(x(:,y(:,u'));
342 axis([-3 3 -3 3 -0.03 1]), view(60,40)
343 legend('time = 0.00','Location','north')
344 xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
345 colormap(hsv)
346 ifOurCf2eps([mfilename 'ic'])

```

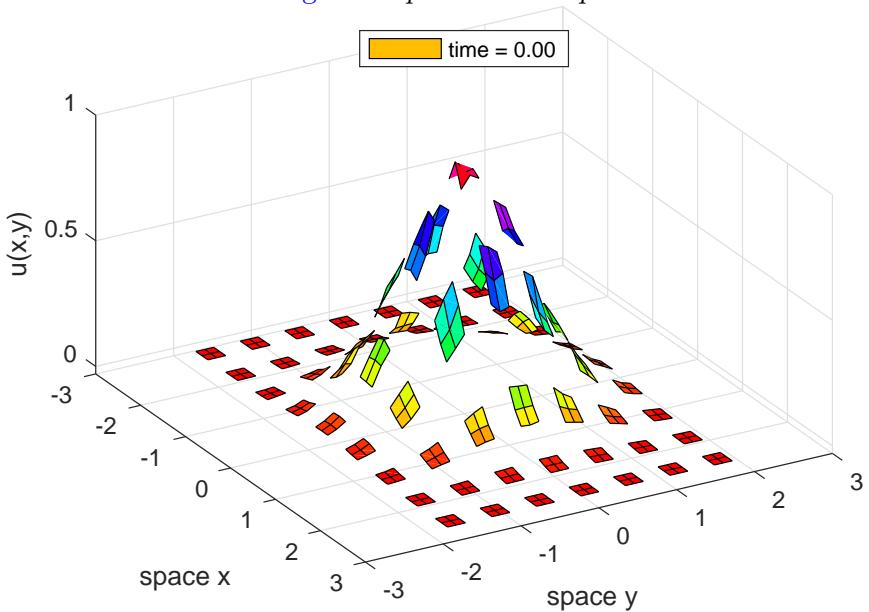
Integrate in time to $t = 4$ using standard functions. In MATLAB `ode15s` would be natural as the patch scheme is naturally stiff, but `ode23` is quicker (Maclean, Bunder, and Roberts [2021](#), Fig. 4). Ask for output at non-uniform times because the diffusion slows.

```

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

```

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 $t = 3$.



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).

```

380 for i = 1:length(ts)
381     u = patchEdgeInt2(us(i,:));
382     u = reshape(permute(squeeze(u) ...
383         ,[1 3 2 4]), [numel(x) numel(y)]);
384     set(hsurf,'ZData', u');
385     legend(['time = ' num2str(ts(i),'%4.2f')])
386     pause(0.1)
387 end
388 ifOurCf2eps([mfilename 't3'])

```

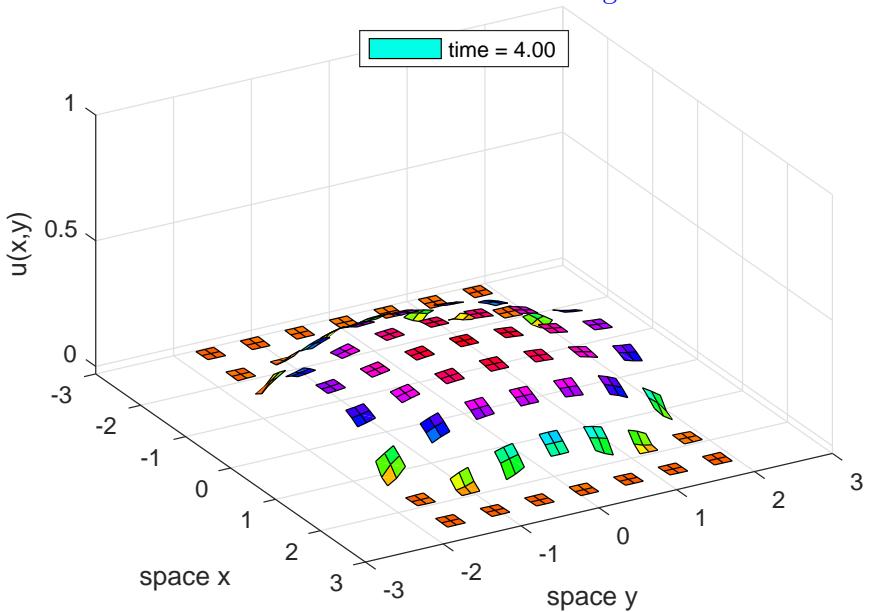
Upon finishing execution of the example, exit this function.

```

403 return
404 end%if no arguments

```

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.



15.2 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);
424 addRequired(p,'ordCC',@isnumeric);
425 addRequired(p,'dx',@isnumeric);
426 addRequired(p,'nSubP',@isnumeric);
427 addParameter(p,'nEdge',1,@isnumeric);
428 addParameter(p,'EdgyInt',false,@islogical);
429 addParameter(p,'nEnsem',1,@isnumeric);
430 addParameter(p,'hetCoeffs',[],@isnumeric);
431 addParameter(p,'parallel',false,@islogical);
432 %addParameter(p,'nCore',1,@isnumeric); % not yet implemented
433 parse(p,fun,Xlim,nPatch,ordCC,dx,nSubP,varargin{:});
```

Set the optional parameters.

```
439 patches.nEdge = p.Results.nEdge;
440 patches.EdgyInt = p.Results.EdgyInt;
441 patches.nEnsem = p.Results.nEnsem;
442 cs = p.Results.hetCoeffs;
443 patches.parallel = p.Results.parallel;
444 %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
```

15.3 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.

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

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

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

Check for staggered grid and periodic case.

```
591 if patches.stag, assert(all(mod(nPatch,2)==0), ...
592     'Require an even number of patches for staggered grid')
593 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.

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

Distribution depends upon `Dom.type`:

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

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

```

621 case 'periodic'
622 Q=linspace(Xlim(qq),Xlim(qq+1),nPatch(q)+1);
623 DQ=Q(2)-Q(1);
624 Q=Q(1:nPatch(q))+diff(Q)/2;
625 pEI=patches.EdgyInt;% abbreviation
626 % sizedx=size(dx), sizenSubP=size(nSubP)
627 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));
```

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??

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
682 if DQmin<width*0.999999
683   warning('too many Chebyshev patches (mid-domain overlap)')
684 end

```

Assign the centre-patch coordinates.

```

690 Q =[ Q1+(0:b/2-1)*width ...
691      (Q1+Q2)/2-(Q2-Q1-b*width)/2*cos(linspace(0,pi,nPatch(q)-b)) ...
692      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??.

```

701 case 'usergiven'
702   if q==1, Q = reshape(Dom.X,1,[]);
703   else     Q = reshape(Dom.Y,1,[]);
704   end%if
705 end%switch Dom.type

```

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

```

712 if q==1, X=Q; end
713 if q==2, Y=Q; end
714 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).

```

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

```

Next the y -direction.

```
740 i0 = (nSubP(2)+1)/2;
741 patches.y = reshape(dx(2)*(-i0+1:i0-1)'+Y ...
742 ,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
```

15.4 Set ensemble inter-patch communication

For EdgyInt or centre interpolation respectively,

- the right-edge/centre realisations $1:nE_{ensem}$ are to interpolate to left-edge le , and
- the left-edge/centre realisations $1:nE_{ensem}$ are to interpolate to re .

re and li are ‘transposes’ of each other as $re(li)=le(ri)$ are both $1:nE_{ensem}$. 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 than 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

```

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

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

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

Fin

931 end% function
```

16 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) · nVars · nEnsem · prod(nPatch)` where there are `nVars · nEnsem` field values at each of the points in the `nSubP1 · nSubP2 · nSubP3 · nPatch1 · nPatch2 · nPatch3` multiscale spatial grid on the `nPatch1 · nPatch2 · nPatch3` array of patches.
- `patches` a struct set by `configPatches3()` which includes the following information.

²²Script `patchEdgeInt3test.m` verifies this code.

- `.x` is $n_{SubP1} \times 1 \times 1 \times 1 \times 1 \times n_{Patch1} \times 1 \times 1$ 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 $1 \times n_{SubP2} \times 1 \times 1 \times 1 \times 1 \times n_{Patch2} \times 1$ array of the spatial locations y_{jJ} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index j , but may be variable spaced in macroscale index J .
- `.z` is similarly $1 \times 1 \times n_{SubP3} \times 1 \times 1 \times 1 \times n_{Patch3}$ array of the spatial locations z_{kK} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index k , but may be variable spaced in macroscale index K .
- `.ordCC` is order of interpolation, currently only $\{0, 2, 4, \dots\}$
- `.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, $n_{SubP1} \cdot n_{SubP2} \cdot n_{SubP3} \cdot n_{Vars} \cdot n_{Ensem} \cdot n_{Patch1} \cdot n_{Patch2} \cdot n_{Patch3}$, of the fields with face values set by interpolation (edge and corner values 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) ...
145     /numel(patches.y)/numel(patches.z)/nEnsem );
146 assert(numel(u) == nx*ny*nz*Nx*Ny*Nz*nVars*nEnsem ...
147     , 'patchEdgeInt3: input u has wrong size for parameters')
148 u = reshape(u,[nx ny nz nVars nEnsem Nx Ny Nz]);

```

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

```

157 I=1:Nx; Ip=mod(I,Nx)+1; Im=mod(I-2,Nx)+1;
158 J=1:Ny; Jp=mod(J,Ny)+1; Jm=mod(J-2,Ny)+1;
159 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

```

167 i0 = round((nx+1)/2);
168 j0 = round((ny+1)/2);
169 k0 = round((nz+1)/2);
170 %disp('finished common preamble')

```

16.1 Periodic macroscale interpolation schemes

```

179 if patches.periodic

```

Get the size ratios of the patches in each direction.

```

185 rx = patches.ratio(1);
186 ry = patches.ratio(2);
187 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.

```
198 ordCC = patches.ordCC;
199 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.

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

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

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

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

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

```

252 if patches.stag % use only odd numbered neighbours
253 error('polynomial interpolation not yet for staggered patch couple')
254 dmux(:,:,,:, :,I,:,:,1) = (Ux(:,:,,:, :,Ip,:,:) + Ux(:,:,,:, :,Im,
255 dmux(:,:,,:, :,I,:,:,2) = (Ux(:,:,,:, :,Ip,:,:) - Ux(:,:,,:, :,Im,
256 Ip = Ip(Ip); Im = Im(Im); % increase shifts to \pm2
257 dmuy(:,:,,:, :,J,:,:,1) = (Ux(:,:,,:, :,Jp,:)+Ux(:,:,,:, :,Jm,
258 dmuy(:,:,,:, :,J,:,:,2) = (Ux(:,:,,:, :,Jp,:)-Ux(:,:,,:, :,Jm,
259 Jp = Jp(Jp); Jm = Jm(Jm); % increase shifts to \pm2
260 dmuz(:,:,,:, :,K,1) = (Ux(:,:,,:, :,Kp)+Ux(:,:,,:, :,Kp),
261 dmuz(:,:,,:, :,K,2) = (Ux(:,:,,:, :,Kp)-Ux(:,:,,:, :,Kp),
262 Kp = Kp(Kp); Km = Km(Km); % increase shifts to \pm2
263 else %disp('starting standard interpolation')
264 dmux(:,:,,:, :,I,:,:,1) = (Ux(:,:,,:, :,Ip,:,:) ...
265 -Ux(:,:,,:, :,Im,:,:))/2; %\mu\delta
266 dmux(:,:,,:, :,I,:,:,2) = (Ux(:,:,,:, :,Ip,:,:) ...
267 -2*Ux(:,:,,:, :,I,:,:)) +Ux(:,:,,:, :,Im,:,:)); %\delta^2
268 dmuy(:,:,,:, :,J,:,:,1) = (Uy(:,:,,:, :,Jp,:)) ...
269 -Uy(:,:,,:, :,Jm,:))/2; %\mu\delta
270 dmuy(:,:,,:, :,J,:,:,2) = (Uy(:,:,,:, :,Jp,:)) ...
271 -2*Uy(:,:,,:, :,J,:)) +Uy(:,:,,:, :,Jm,:)); %\delta^2
272 dmuz(:,:,,:, :,K,1) = (Uz(:,:,,:, :,Kp)) ...
273 -Uz(:,:,,:, :,Km))/2; %\mu\delta
274 dmuz(:,:,,:, :,K,2) = (Uz(:,:,,:, :,Kp)) ...
275 -2*Uz(:,:,,:, :,K)) +Uz(:,:,,:, :,Km)); %\delta^2
276 end% if stag

```

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

```

283 for k = 3:ordCC
284 dmux(:,:,,:, :,I,:,:,k) = dmux(:,:,,:, :,Ip,:,:,k-2) ...
285 -2*dmux(:,:,,:, :,I,:,:,k-2) +dmux(:,:,,:, :,Im,:,:,k-2);
286 dmuy(:,:,,:, :,J,:,:,k) = dmuy(:,:,,:, :,Jp,:,:,k-2) ...
287 -2*dmuy(:,:,,:, :,J,:,:,k-2) +dmuy(:,:,,:, :,Jm,:,:,k-2);
288 dmuz(:,:,,:, :,K,k) = dmuz(:,:,,:, :,Kp,k-2) ...
289 -2*dmuz(:,:,,:, :,K,k-2) +dmuz(:,:,,:, :,Km,k-2);
290 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`.

```

306 k=1+patches.EdgyInt; % use centre or two faces
307 u(nx,2:(ny-1),2:(nz-1),:,: patches.ri,I,:,:,:) ...
308 = Ux(1,:,:,:,:,:,:,:)*(1-patches.stag) ...
309 +sum( shiftdim(patches.Cwtsr(:,1),-8).*dmux(1,:,:,:,:,:,:,:,:) ,9);
310 u(1 ,2:(ny-1),2:(nz-1),:,: patches.le,I,:,:,:) ...
311 = Ux(k,:,:,:,:,:,:,:)*(1-patches.stag) ...
312 +sum( shiftdim(patches.Cwtsl(:,1),-8).*dmux(k,:,:,:,:,:,:,:,:) ,9);
313 u(2:(nx-1),ny,2:(nz-1),:,: patches.to,:,J,:) ...
314 = Uy(:,1,:,:,:,:,:)*(1-patches.stag) ...
315 +sum( shiftdim(patches.Cwtsr(:,2),-8).*dmuy(:,1,:,:,:,:,:,:) ,9);
316 u(2:(nx-1),1 ,2:(nz-1),:,: patches.bo,:,J,:) ...
317 = Uy(:,k,:,:,:,:,:)*(1-patches.stag) ...
318 +sum( shiftdim(patches.Cwtsl(:,2),-8).*dmuy(:,k,:,:,:,:,:,:) ,9);
319 u(2:(nx-1),2:(ny-1),nz,:,: patches.fr,:,:,:,K) ...
320 = Uz(:, :,1,:,:,:,:,:)*(1-patches.stag) ...
321 +sum( shiftdim(patches.Cwtsr(:,3),-8).*dmuz(:, :,1,:,:,:,:,:,:) ,9);
322 u(2:(nx-1),2:(ny-1),1 ,:,: patches.ba,:,:,:,K) ...
323 = Uz(:, :,k,:,:,:,:,:)*(1-patches.stag) ...
324 +sum( shiftdim(patches.Cwtsl(:,3),-8).*dmuz(:, :,k,:,:,:,:,:,:) ,9);
```

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

```
334 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 $\mathbf{ks} = k2\pi/N$ for ‘wavenumbers’ $k = (0, 1, \dots, k_{\max}, -k_{\max}, \dots, -1)$ for odd N , and $k = (0, 1, \dots, k_{\max}, \pm(k_{\max} + 1) - k_{\max}, \dots, -1)$ 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.

```

357 if patches.stag % transform by doubling the number of fields
358 error('staggered grid not yet implemented??')
359 v=nan(size(u)); % currently to restore the shape of u
360 u=cat(3,u(:,1:2:nPatch,:),u(:,2:2:nPatch,:));
361 stagShift=reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
362 iV=[nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
363 r=r/2; % ratio effectively halved
364 nPatch=nPatch/2; % halve the number of patches
365 nVars=nVars*2; % double the number of fields
366 else % the values for standard spectral
367   stagShift = 0;
368   iV = 1:nVars;
369 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, $k = (0, 1, \dots, k_{\max}, +(k_{\max} + 1) - k_{\max}, \dots, -1)$.

```

379 kMax = floor((Nx-1)/2);
380 krx = shiftdim( rx*2*pi/Nx*(mod((0:Nx-1)+kMax,Nx)-kMax) ,-4);
381 kMay = floor((Ny-1)/2);
382 kry = shiftdim( ry*2*pi/Ny*(mod((0:Ny-1)+kMay,Ny)-kMay) ,-5);
383 kMaz = floor((Nz-1)/2);
384 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`.

```

399 % indices of interior
400 ix=(2:nx-1)'; iy=2:ny-1; iz=shiftdim(2:nz-1,-1);
401 if ~patches.EdgyInt

```

```

402     Cle = fft(fft(fft( u(i0,iy,iz,:,:,:,:,:) ...
403         ,[],6),[],7),[],8);
404     Cbo = fft(fft(fft( u(ix,j0,iz,:,:,:,:,:) ...
405         ,[],6),[],7),[],8);
406     Cba = fft(fft(fft( u(ix,iy,k0,:,:,:,:,:) ...
407         ,[],6),[],7),[],8);
408     Cri = Cle;      Cto = Cbo;      Cfr = Cba;
409 else
410     Cle = fft(fft(fft( u(    2,iy,iz ,:,patches.le,:,:,:) ...
411         ,[],6),[],7),[],8);
412     Cri = fft(fft(fft( u(nx-1,iy,iz ,:,patches.ri,:,:,:) ...
413         ,[],6),[],7),[],8);
414     Cbo = fft(fft(fft( u(ix,2    ,iz ,:,patches.bo,:,:,:) ...
415         ,[],6),[],7),[],8);
416     Cto = fft(fft(fft( u(ix,ny-1,iz ,:,patches.to,:,:,:) ...
417         ,[],6),[],7),[],8);
418     Cba = fft(fft(fft( u(ix,iy,2    ,:,patches.ba,:,:,:) ...
419         ,[],6),[],7),[],8);
420     Cfr = fft(fft(fft( u(ix,iy,nz-1 ,:,patches.fr,:,:,:) ...
421         ,[],6),[],7),[],8);
422 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.

```

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

```

16.2 Non-periodic macroscale interpolation

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

```
465 if patches.ordCC<1  
466     px = Nx-1;  py = Ny-1;  pz = Nz-1;  
467 else px = min(patches.ordCC,Nx-1);  
468     py = min(patches.ordCC,Ny-1);  
469     pz = min(patches.ordCC,Nz-1);  
470 end  
471 % interior indices of faces  (ix n/a)  
472 ix=2:nx-1;  iy=2:ny-1;  iz=2:nz-1;
```

16.2.1 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.)

```
485 F = nan(patches.EdgyInt+1,ny-2,nz-2,nVars,nEnsem,Nx,Ny,Nz,px+1);  
486 if patches.EdgyInt % interpolate next-to-face values  
487     F(:,:,(:,:,1,:,:,1)) = u([nx-1 2],iy,iz,:,:,:,:,:);  
488     X = patches.x([nx-1 2],(:,:,1,:,:,1,:));  
489 else % interpolate mid-patch cross-patch values  
490     F(:,:,(:,:,1,:,:,1)) = u(i0,iy,iz,:,:,:,:,:);  
491     X = patches.x(i0,:,:,:,:,:,:,:);  
492 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.

```
504 for q = 1:px
505   i = 1:Nx-q;
506   F(:,:,(:,:,i,:,:,:q+1) ...
507     = ( F(:,:,(:,:,i+1,:,:,:q)-F(:,:,(:,:,i,:,:,:q)) ...
508       ./ (X(:,:,(:,:,i+q,:,:))-X(:,:,(:,:,i,:,:))); 
509 end
```

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

```
517 Xface = patches.x([1 nx],(:,:,(:,:,i,:,:,:));
```

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.

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

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

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

16.2.2 y -direction values

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

```

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

```

Form tables of divided differences.

```

564 for q = 1:py
565   j = 1:Ny-q;
566   F(:,:,,:, :, :, j, :, q+1) ...
567     = ( F(:,:,,:, :, :, j+1, :, q)-F(:,:,,:, :, :, j, :, q)) ...
568     ./ (Y(:,:,,:, :, :, j+q, :) -Y(:,:,,:, :, :, j, :));
569 end

```

Interpolate to find the top/bottom faces Y_{face} for every x and interior z .

```

575 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.

```

584 j = max(1,min(1:Ny,Ny-ceil(py/2))-floor(py/2));
585 Uface = F(:,:,,:, :, :, j, :, py+1);
586 for q = py:-1:1
587   Uface = F(:,:,,:, :, :, j, :, q) ...
588   +(Yface-Y(:,:,,:, :, :, j+q-1, :)).*Uface;
589 end

```

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

```

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

```

16.2.3 z -direction values

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

```

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

```

Form tables of divided differences.

```

621 for q = 1:pz
622   k = 1:Nz-q;
623   F(:,:,,:, :, :, :, k, q+1) ...
624   = ( F(:,:,,:, :, :, :, k+1, q)-F(:,:,,:, :, :, k, q)) ...
625     ./ (Z(:,:,,:, :, :, k+q) - Z(:,:,,:, :, :, k));
626 end

```

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

```

632 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.

```

641 k = max(1,min(1:Nz,Nz-ceil(pz/2))-floor(pz/2));
642 Uface = F(:,:,,:, :, :, k, pz+1);
643 for q = pz:-1:1
644   Uface = F(:,:,,:, :, :, k, q) ...
645   +(Zface-Z(:,:,,:, :, :, k+q-1)).*Uface;
646 end

```

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

```

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

```

16.2.4 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.

```

667 u( 1,:,:,:,:,:, 1,:,:,:) = nan;
668 u(nx,:,:,:,:,:,Nx,:,:) = nan;
669 u(:, 1,:,:,:,:,:, 1,:) = nan;
670 u(:,ny,:,:,:,:,:,Ny,:) = nan;
671 u(:, :, 1,:,:,:,:,:, 1) = nan;
672 u(:, :, nz,:,:,:,:,:, Nz) = nan;

End of the non-periodic interpolation code.

```

```

679 %disp('finished new non-periodic code')
680 end%if patches.periodic else

```

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

```

691 end% function patchEdgeInt3

```

17 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 17.1](#) 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 17.1](#) 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, . . . ; 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 $\text{ratio} = \frac{1}{2}$ 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.

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

- **.periodic**: either true, for interpolation on the macro-periodic domain; or false, for general interpolation by divided differences over non-periodic domain or unevenly distributed patches.
- **.stag** is true for interpolation using only odd neighbouring patches as for staggered grids, and false for the usual case of all neighbour coupling—not yet implemented.
- **.Cwtsr** and **.Cwtsl** are the `ordCC` × 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)` × 1 × 1 × 1 × 1 × `nPatch(1)` × 1 × 1 array of the regular spatial locations x_{iI} of the microscale grid points in every patch.
- **.y** (8D) is 1 × `nSubP(2)` × 1 × 1 × 1 × 1 × `nPatch(2)` × 1 array of the regular spatial locations y_{jJ} of the microscale grid points in every patch.
- **.z** (8D) is 1 × 1 × `nSubP(3)` × 1 × 1 × 1 × 1 × `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, $(n_{\text{SubP}}(1) - 1) \times (n_{\text{SubP}}(2) - 1) \times (n_{\text{SubP}}(3) - 1) \times n_c$ 4D array of microscale heterogeneous coefficients, or
 - if `nEnsem` > 1, $(n_{\text{SubP}}(1) - 1) \times (n_{\text{SubP}}(2) - 1) \times (n_{\text{SubP}}(3) - 1) \times n_c \times m_x m_y m_z$ 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.

17.1 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(:))
```

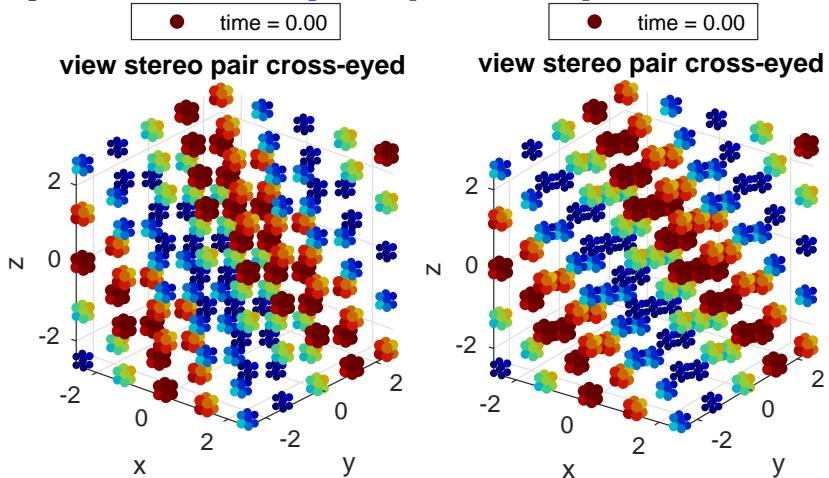
Establish global patch data struct to interface with a function coding a nonlinear 'diffusion' PDE: to be solved on $[-\pi, \pi]^3$ -periodic domain, with 5^3 patches, spectral interpolation (0) couples the patches, each patch with micro-grid spacing 0.22 (relatively large for visualisation), and with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

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 $t = 6$.



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).

```

372 disp('Simulate heterogeneous wave u_tt=div[C*grad(u)]')
373 if ~exist('OCTAVE_VERSION','builtin')
374     [ts,us] = ode23(@patchSys3,linspace(0,6),uv0(:));
375 else %disp('octave version is very slow for me')
376     lsode_options('absolute tolerance',1e-4);
377     lsode_options('relative tolerance',1e-4);
378     [ts,us] = odeOcts(@patchSys3,[0 1 2],uv0(:));
379 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.

```

394 figure(1), clf, colormap(0.8*jet)
395 xs = patches.x+0*patches.y+0*patches.z;
396 ys = patches.y+0*patches.x+0*patches.z;

```

```

397 zs = patches.z+0*patches.y+0*patches.x;
398 if 1, xs([1 end],:,:,:) = nan;
399     xs(:,[1 end],:,:,:) = nan;
400     xs(:,:,1,[1 end],:) = nan;
401 end;%option
402 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.

```

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

```

Loop to plot at each and every time step.

```

417 for i = 1:length(ts)
418     uv = patchEdgeInt3(us(i,:,:));
419     u = uv(:,:,1,:);
420     for p=1:2
421         subplot(1,2,p)
422         if (i==1) | exist('OCTAVE_VERSION','builtin')
423             scat(p) = scatter3(xs(j),ys(j),zs(j),'filled');
424             axis equal, caxis(col([0 1])), view(45-5*p,25)
425             xlabel('x'), ylabel('y'), zlabel('z')
426             title('view stereo pair cross-eyed')
427         end % in matlab just update values
428         set(scat(p),'CData',col(u(j)) ...
429             , 'SizeData',pix((8+xs(j)-ys(j)+zs(j))/6+0*u(j)));
430         legend(['time = ' num2str(ts(i),'%4.2f')],'Location','north')
431     end

```

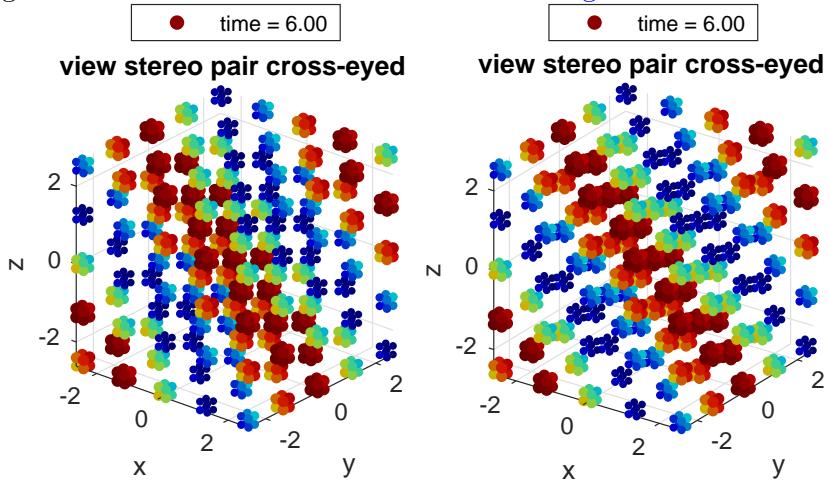
Optionally save the initial condition to graphic file for [Figure 14](#), and optionally save the last plot.

```

439 if i==1,
440     ifOurCf2eps([mfilename 'ic'])
441         disp('Type space character to animate simulation')
442         pause
443     else pause(0.05)
444     end
445 end% i-loop over all times
446 ifOurCf2eps([mfilename 'fin'])

```

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.



Upon finishing execution of the example, exit this function.

```
461 return
462 end%if no arguments
```

17.2 Parse input arguments and defaults

```
479 p = inputParser;
480 fnValidation = @(f) isa(f, 'function_handle'); %test for fn name
481 addRequired(p, 'fun', fnValidation);
482 addRequired(p, 'Xlim', @isnumeric);
483 %addRequired(p, 'Dom'); % nothing yet decided
484 addRequired(p, 'nPatch', @isnumeric);
485 addRequired(p, 'ordCC', @isnumeric);
486 addRequired(p, 'dx', @isnumeric);
487 addRequired(p, 'nSubP', @isnumeric);
488 addParameter(p, 'nEdge', 1, @isnumeric);
489 addParameter(p, 'EdgyInt', false, @islogical);
490 addParameter(p, 'nEnsem', 1, @isnumeric);
491 addParameter(p, 'hetCoeffs', [], @isnumeric);
492 addParameter(p, 'parallel', false, @islogical);
493 %addParameter(p, 'nCore', 1, @isnumeric); % not yet implemented
494 parse(p, fun, Xlim, nPatch, ordCC, dx, nSubP, varargin{:});
```

Set the optional parameters.

```
500 patches.nEdge = p.Results.nEdge;
501 patches.EdgyInt = p.Results.EdgyInt;
502 patches.nEnsem = p.Results.nEnsem;
503 cs = p.Results.hetCoeffs;
504 patches.parallel = p.Results.parallel;
505 %patches.nCore = p.Results.nCore;
```

Initially duplicate parameters for three space dimensions as needed.

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

Check parameters.

```
523 assert(Xlim(1)<Xlim(2) ...
524     , 'first pair of Xlim must be ordered increasing')
525 assert(Xlim(3)<Xlim(4) ...
526     , 'second pair of Xlim must be ordered increasing')
527 assert(Xlim(5)<Xlim(6) ...
528     , 'third pair of Xlim must be ordered increasing')
529 assert(patches.nEdge==1 ...
530     , 'multi-edge-value interp not yet implemented')
531 assert(all(2*patches.nEdge<nSubP) ...
532     , 'too many edge values requested')
533 %if patches.nCore>1
534 %    warning('nCore>1 not yet tested in this version')
535 %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.

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

Default macroscale conditions are periodic with evenly spaced patches.

```
557 if isempty(Dom), Dom=struct('type','periodic'); end
558 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';
613 if p==1, assert(isfield(Dom,'X'),['X' msg]), end
614 if p==2, assert(isfield(Dom,'Y'),['Y' msg]), end
615 if p==3, assert(isfield(Dom,'Z'),['Z' msg]), end
616 otherwise
617     error([Dom.type 'is unknown Dom.type'])
618 end%switch Dom.type
619 end%for p

```

17.3 The code to make patches

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

```

633 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.

```

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

```

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

```

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

```

Check for staggered grid and periodic case.

```

659 if patches.stag, assert(all(mod(nPatch,2)==0), ...
660     'Require an even number of patches for staggered grid')
661 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.

```

676 for q=1:3
677 qq=2*q-1;
683 switch Dom.type(q,:)

```

Distribution depends upon Dom.type:

```

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

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??

```

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

```

800 nSubP = reshape(nSubP,1,3); % force to be row vector
801 assert(patches.EdgyInt | all(mod(nSubP,2)==1), ...
802     'configPatches3: nSubP must be odd')
803 i0 = (nSubP(1)+1)/2;
804 patches.x = reshape( dx(1)*(-i0+1:i0-1)'+X ...
805     ,nSubP(1),1,1,1,1,nPatch(1),1,1);
806 i0 = (nSubP(2)+1)/2;
807 patches.y = reshape( dx(2)*(-i0+1:i0-1)'+Y ...
808     ,1,nSubP(2),1,1,1,1,nPatch(2),1);
809 i0 = (nSubP(3)+1)/2;
810 patches.z = reshape( dx(3)*(-i0+1:i0-1)'+Z ...
811     ,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.)

```

823 if patches.periodic
824     ratio = reshape(ratio,1,3); % force to be row vector
825     patches.ratio = ratio;
826     if ordCC>0
827         [Cwtsr,Cwtsl] = patchCwts(ratio,ordCC,patches.stag);
828         patches.Cwtsr = Cwtsr; patches.Cwtsl = Cwtsl;
829     end%if
830 end%if patches.periodic

```

17.4 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()`.

```
859 nE = patches.nEnsem;
860 patches.le = 1:nE; patches.ri = 1:nE;
861 patches.bo = 1:nE; patches.to = 1:nE;
862 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 than 4D, then the higher-dimensions are reshaped into the 4th dimension.

```
874 if ~isempty(cs)
875     [mx,my,mz,nc] = size(cs);
876     nx = nSubP(1); ny = nSubP(2); nz = nSubP(3);
877     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.

```
885 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.

```
895 patches.nEnsem = mx*my*mz;
896 patches.cs = nan(nx-1,ny-1,nz-1,nc,mx,my,mz);
897 for k = 1:mz
898     ks = (k:k+nz-2);
899     for j = 1:my
900         js = (j:j+ny-2);
901         for i = 1:mx
902             is = (i:i+nx-2);
903             patches.cs(:,:, :, i, j, k) = cs(is,js,ks,:);
904         end
905     end
906 end
907 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.

```

917 mmx=(0:mx-1)'; mmy=0:my-1; mmz=shiftdim(0:mz-1,-1);
918 le = mod(mmx+mod(nx-2,mx),mx)+1;
919 patches.le = reshape( le+mx*(mmy+my*mmz) ,[],1);
920 ri = mod(mmx-mod(nx-2,mx),mx)+1;
921 patches.ri = reshape( ri+mx*(mmy+my*mmz) ,[],1);
922 bo = mod(mmy+mod(ny-2,my),my)+1;
923 patches.bo = reshape( 1+mmx+mx*(bo-1+my*mmz) ,[],1);
924 to = mod(mmy-mod(ny-2,my),my)+1;
925 patches.to = reshape( 1+mmx+mx*(to-1+my*mmz) ,[],1);
926 ba = mod(mmz+mod(nz-2,mz),mz)+1;
927 patches.ba = reshape( 1+mmx+mx*(mmy+my*(ba-1)) ,[],1);
928 fr = mod(mmz-mod(nz-2,mz),mz)+1;
929 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??

```

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

```

End the two if-statements.

```

946 end%if-else nEnsem>1
947 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

²⁵If 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. 35).

- 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](https://doi.org/10.1007/s00211-021-01232-5) (cit. on pp. 47, 68, 94).
- 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](https://doi.org/10.1016/j.jcp.2017.02.004) (cit. on pp. 47, 51, 72, 99).
- 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](https://doi.org/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](https://arxiv.org/abs/2210.04536) (cit. on pp. 3, 5, 9, 16).
- Maclean, John, J. E. Bunder, and A. J. Roberts (2021). “A toolbox of Equation-Free functions in Matlab/Octave for efficient system level simulation”. In: *Numerical Algorithms* 87, pp. 1729–1748. DOI: [10.1007/s11075-020-01027-z](https://doi.org/10.1007/s11075-020-01027-z) (cit. on pp. 83, 112).
- 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](https://doi.org/10.1090/S0025-5718-02-01448-5). (Cit. on p. 47).
- 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](https://doi.org/10.1137/060654554) (cit. on pp. 47, 51, 72, 99).
- 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](https://doi.org/10.1007/s10665-013-9653-6) (cit. on pp. 68, 82, 94).
- Wikipedia (2022). *Divided differences*. https://en.wikipedia.org/wiki/Divided_differences (visited on 12/28/2022) (cit. on pp. 54, 75, 103).