

Equation-free computational homogenisation with thicker edge coupling

A. J. Roberts*

April 13, 2023

Contents

Examples	3
1 hyperDiffHetero: simulate a heterogeneous hyper-diffusion PDE in 1D on patches	3
1.1 Heterogeneous hyper-diffusion PDE inside patches	6
2 SwiftHohenbergPattern: patterns of the Swift–Hohenberg PDE in 1D on patches	7
2.0.1 Find equilibrium with fsolve	9
2.0.2 Simulate in time	10
2.1 The Swift–Hohenberg PDE and BCs inside patches	10
2.2 theRes(): wrapper function to zero for equilibria	11
3 SwiftHohenbergHetero: patterns of a heterogeneous Swift–Hohenberg PDE in 1D on patches	11
3.0.1 Explore the Jacobian	14
3.0.2 Find an equilibrium with fsolve	17
3.0.3 Simulate in time	18
3.1 Heterogeneous SwiftHohenberg PDE+BCs inside patches	19
3.2 theRes(): a wrapper function	20
New configuration and interpolation	20

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

4	patchEdgeInt1(): sets patch-edge values from interpolation over the 1D macroscale	20
4.1	Periodic macroscale interpolation schemes	23
4.2	Non-periodic macroscale interpolation	27
5	configPatches1(): configure spatial patches in 1D	29
5.1	If no arguments, then execute an example	33
5.2	Parse input arguments and defaults	34
5.3	The code to make patches and interpolation	37
5.4	Set ensemble inter-patch communication	40
6	patchSys1(): interface 1D space to time integrators	42
7	patchEdgeInt2(): sets 2D patch edge values from 2D macroscale interpolation	44
7.1	Periodic macroscale interpolation schemes	47
7.1.1	Lagrange interpolation gives patch-edge values	47
7.1.2	Case of spectral interpolation	50
7.2	Non-periodic macroscale interpolation	52
7.2.1	x -direction values	53
7.2.2	y -direction values	54
7.2.3	Optional NaNs for safety	55
8	configPatches2(): configures spatial patches in 2D	56
8.1	If no arguments, then execute an example	61
8.2	Parse input arguments and defaults	64
8.3	The code to make patches	66
8.4	Set ensemble inter-patch communication	70
9	patchSys2(): interface 2D space to time integrators	73
10	patchEdgeInt3(): sets 3D patch face values from 3D macroscale interpolation	74
10.1	Periodic macroscale interpolation schemes	78
10.1.1	Lagrange interpolation gives patch-face values	78
10.1.2	Case of spectral interpolation	82
10.2	Non-periodic macroscale interpolation	85
10.2.1	x -direction values	86
10.2.2	y -direction values	87
10.2.3	z -direction values	88

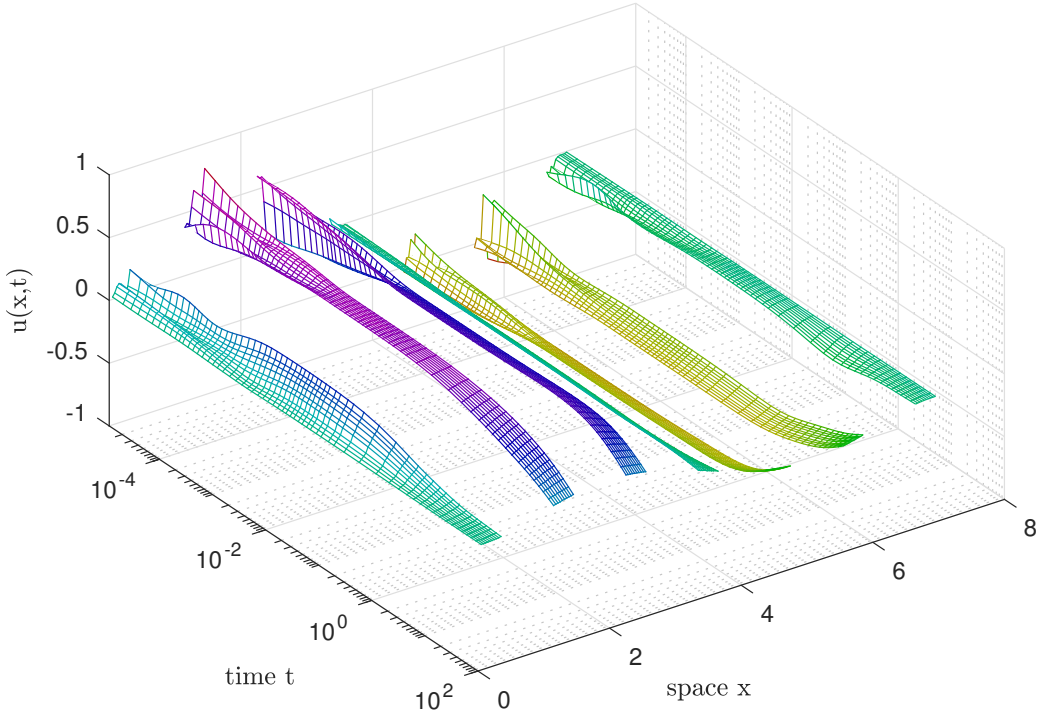
10.2.4	Optional NaNs for safety	89
11	configPatches3(): configures spatial patches in 3D	90
11.1	If no arguments, then execute an example	95
11.2	Parse input arguments and defaults	98
11.3	The code to make patches	102
11.4	Set ensemble inter-patch communication	105
12	patchSys3(): interface 3D space to time integrators	108
13	New interpolation tests	110
13.1	patchEdgeInt1test: test the 1D patch coupling	110
13.1.1	Check divided difference interpolation	110
13.1.2	Test standard spectral interpolation	112
13.1.3	Now test spectral interpolation on staggered grid	113
13.1.4	Check standard finite width interpolation	115
13.1.5	Now test finite width interpolation on staggered grid	117
13.1.6	Finish	118
13.2	patchEdgeInt2test: tests 2D patch coupling	118
13.2.1	Check divided difference interpolation	119
13.2.2	Test standard spectral interpolation	121
13.2.3	Check polynomial finite width interpolation	122
13.2.4	Finished	123
13.3	patchEdgeInt3test: tests 3D patch coupling	123
13.3.1	Check divided difference interpolation	124
13.3.2	Test standard spectral interpolation	126
13.3.3	Check polynomial finite width interpolation	127
13.3.4	Finished	128

Examples

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

Figure 1 shows an example simulation in time generated by the patch scheme applied to a heterogeneous version of the hyper-diffusion PDE. That such simulations makes valid predictions was established by Bunder, Roberts, and

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



Kevrekidis (2017) who proved that the scheme is accurate when the number of points in a patch is tied to a multiple of the periodicity of the pattern.

We aim to simulate the heterogeneous hyper-diffusion PDE

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

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

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

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

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

```

57 clear all
58 basename = mfilename
59 %global OurCf2eps, OurCf2eps=true %optional to save plots
60 nGap = 3 % controls size of gap between patches
61 nPtsPeriod = 5
62 dx = 0.5/nGap/nPtsPeriod

```

Create some random heterogeneous coefficients, log-uniform.

```

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

```

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

```

80 nPatch = 7
81 nSubP = 2*nPtsPeriod+4 % or +2 for not-edgyInt
82 Len = nPatch;
83 ordCC = 0;
84 dom.type = 'equispace';
85 dom.bcOffset = 0.5 % for BC type
86 patches = configPatches1(@hyperDiffPDE,[0 Len],dom ...
87     ,nPatch,ordCC,dx,nSubP,'EdgyInt',true,'nEdge',2 ...
88     , 'hetCoeffs',cs);
89 xs=squeeze(patches.x);

```

Simulate in time Set an initial condition, and here integrate forward in time using a standard method for stiff systems—because of the simplicity of linear problems this method works quite efficiently here. Integrate the interface `patchSys1` ([Section 6](#)) to the microscale differential equations.

```

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

```

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

```

116 figure(1),clf
117 xs([1:2 end-1:end],:) = nan;
118 t0=min(find(ts>1e-5));
119 mesh(ts(t0:3:end),xs(:),us(t0:3:end,:')), view(55,50)
120 colormap(0.7*hsv)
121 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
122 ca=gca; ca.XScale='log'; ca.XLim=ts([t0 end]);
123 if0urCf2eps([basename 'Uxt'])

```

Fin.

1.1 Heterogeneous hyper-diffusion PDE inside patches

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

```

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

```

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

```

148     u = squeeze(u);
149     if ~patches.periodic % discretise BC u=u_x=0
150         u(1:2,1)=0;
151         u(end-1:end,end)=0;
152     end%if

```

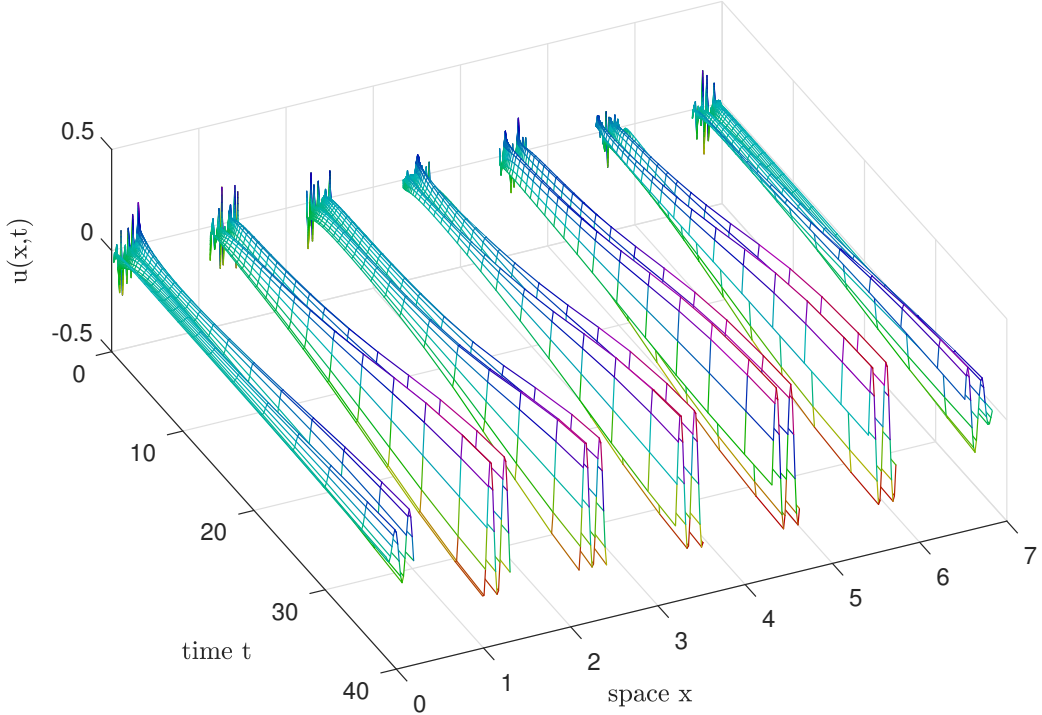
Here code straightforward centred discretisation in space.

```

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

```

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



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

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

Consider a lattice of values $u_i(t)$, with lattice spacing dx , and governed by

a microscale centred discretisation of the Swift–Hohenberg PDE

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

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

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

```

54 clear all, close all
55 %global OurCf2eps, OurCf2eps=true %optional to save plots
56 Ra = 0.1 % Ra>0 leads to patterns
57 nGap = 3
58 %waveLength = 0.496688741721854 /nGap %for nPatch==5
59 waveLength = 0.497630331753555 /nGap %for nPatch==7
60 %waveLength = 0.5 /nGap %for periodic case
61 nPtsPeriod = 10
62 dx = waveLength/nPtsPeriod
63 k0 = 2*pi/waveLength

```

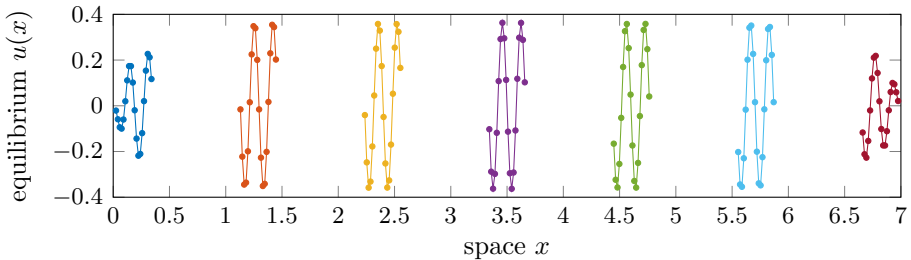
Establish global data struct `patches` for heterogeneous diffusion on 2π -periodic domain. Use seven patches. Quartic (fourth-order) interpolation `ordCC = 4` provides values for the inter-patch coupling conditions.

```

74 nPatch = 7
75 nSubP = 2*nPtsPeriod+4
76 %nSubP = 2*nGap*nPtsPeriod+4 % full-domain
77 Len = nPatch;
78 ordCC = 4;
79 dom.type='equispace';
80 dom.bcOffset=0.5
81 patches = configPatches1(@SwiftHohenbergPDE,[0 Len],dom ...
82     ,nPatch,ordCC,dx,nSubP,'EdgyInt',true,'nEdge',2);
83 xs=squeeze(patches.x);

```


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



2.0.1 Find equilibrium with fsolve

Start the search from some guess.

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

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

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

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

```
123 tic
124 [u(patches.i),res] = fsolve(@(v) theRes(v,patches,k0,Ra) ...
125     ,u(patches.i) ,optimoptions('fsolve','Display','off'));
126 solveTime = toc
127 normRes = norm(res)
128 assert(normRes<1e-6,'**** fsolve solution not accurate')
```

Plot the equilibrium see [Figure 3](#).

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

2.0.2 Simulate in time

Set an initial condition, and here integrate forward in time using a standard method for stiff systems—because of the simplicity of linear problems this method works quite efficiently here. Integrate the interface `patchSys1` (Section 6) to the microscale differential equations.

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

Plot the simulation in [Figure 2](#).

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

Fin.

2.1 The Swift–Hohenberg PDE and BCs inside patches

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

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

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

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

Here code straightforward centred discretisation in space.

```

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

```

2.2 theRes(): wrapper function to zero for equilibria

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

```

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

```

3 SwiftHohenbergHetero: patterns of a heterogeneous Swift–Hohenberg PDE in 1D on patches

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

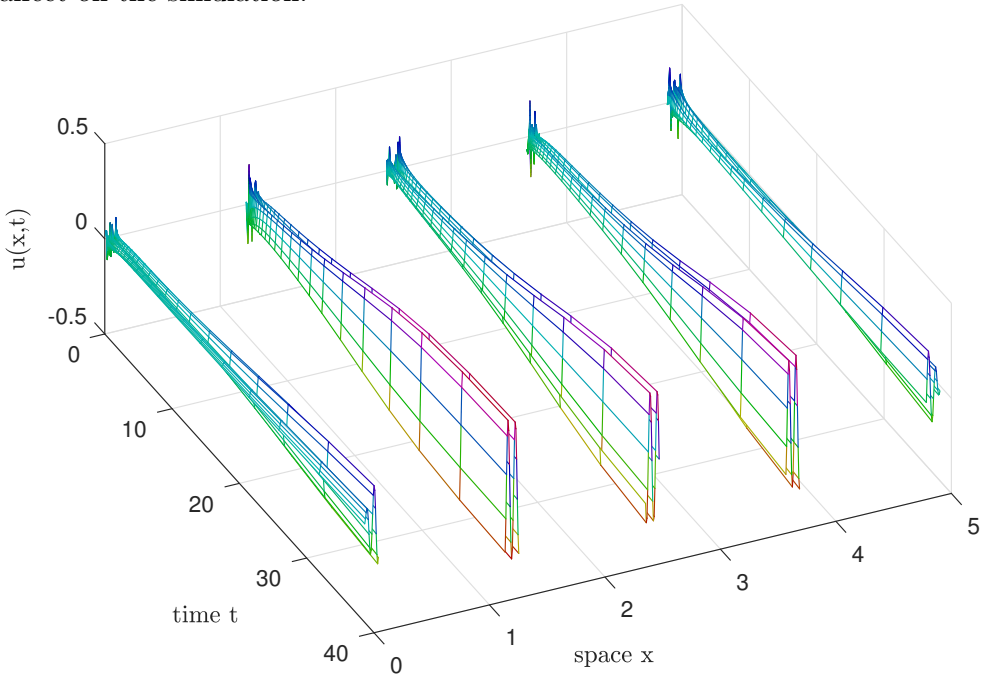
Consider a lattice of values $u_i(t)$, with lattice spacing dx , arising from a microscale discretisation of the pattern forming, heterogeneous, Swift–Hohenberg PDE

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

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

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

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



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

- for homogeneous, the $2N$ modes are local oscillations in each patch, with two modes each corresponding to phase shifts of the possible oscillations;
- for heterogeneous
 - N eigenmodes appear to be one phase ‘locking’ to the heterogeneity; and

- N eigenmodes appear to be other phase ‘locking’ to the heterogeneity. Unless it is something to do with the coupling, but then it only appears with heterogeneity.

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

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

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

```

102 clear all
103 %global OurCf2eps, OurCf2eps=true %optional to save plots
104 basename = ['r' num2str(floor(1e5*rem(now,1))) mfilename]
105 Ra = 0.1 % Ra>0 leads to patterns
106 nGap = 8 % controls size of gap between patches
107 waveLength = 0.496688741721854 /nGap %for nPatch==5
108 %waveLength = 0.497630331753555 /nGap %for nPatch==7
109 %waveLength = 0.5 /nGap %for periodic case
110 nPtsPeriod = 10
111 dx = waveLength/nPtsPeriod
112 k0 = 2*pi/waveLength

```

Create some random heterogeneous coefficients.

```

119 heteroVar = 0.99*[1 1] % must be <2
120 c1 = 1./(1-heteroVar/2+heteroVar.*rand(nPtsPeriod,2));
121 cRange = quantile(c1,0:0.5:1)

```

Establish global data struct **patches** for heterogeneous Swift–Hohenberg PDE with, on average, one patch per units length. Use seven patches to start with. Quartic (fourth-order) interpolation **ordCC** = 4 provides values

for the inter-patch coupling conditions. Or use as high-order as possible with `ordCC = 0`.

```

133 nPatch = 5
134 nSubP = 2*nPtsPeriod+4 % +2 for not-edgyInt
135 %nSubP = 2*nGap*nPtsPeriod+4 % approx full-domain
136 Len = nPatch;
137 ordCC = 0;
138 dom.type='equispace';
139 dom.bcOffset=0.5
140 patches = configPatches1(@heteroSwiftHohenbergPDE,[0 Len],dom ...
141     ,nPatch,ordCC,dx,nSubP,'EdgyInt',true,'nEdge',2 ...
142     , 'hetCoeffs',cl);
143 xs=squeeze(patches.x);

```

3.0.1 Explore the Jacobian

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

```

155 fprintf('\n**** Explore the Jacobian\n')
156 u0 = 0*patches.x;
157 u0([1:2 end-1:end],:) = nan;
158 patches.i = find(~isnan(u0));
159 nVars = numel(patches.i)
160 Jac = nan(nVars);
161 for j=1:nVars
162     Jac(:,j)=theRes((1:nVars)==j,patches,k0,0,0);
163 end

```

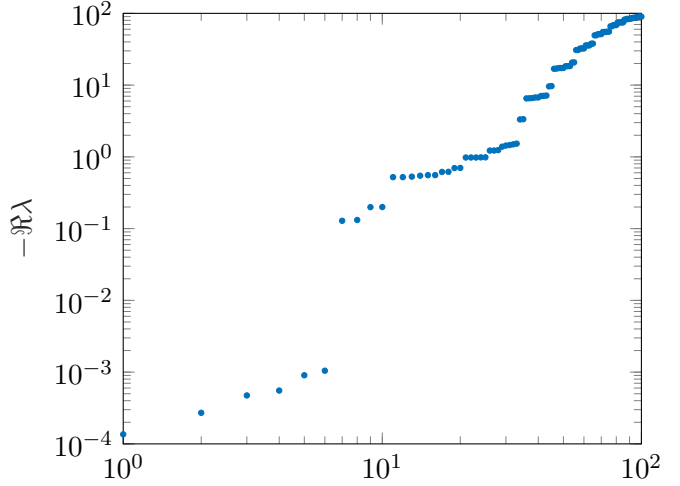
Check on the symmetry of the Jacobian

```

169 nonSymmetric = norm(Jac-Jac')
170 Jac(abs(Jac)<1e-12)=0;
171 antiJac = Jac-Jac';
172 antiJac(abs(antiJac)<1e-12)=0;
173 figure(6),clf
174 spy(Jac,','),hold on, spy(antiJac,'rx'),hold off
175 if nonSymmetric>5e-9, warning('failed symmetry'),
176 else Jac = (Jac+Jac')/2; %tweak to symmetry
177 end

```

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



Compute eigenvalues and eigenvectors.

```

183 figure(5),clf
184 [evec,mEval] = eig(-Jac , 'vector');
185 [~,j]=sort(real(mEval));
186 mEval=mEval(j); evec=evec(:,j);
187 loglog(real(mEval),'.')
188 ylabel('$-\text{Re}\lambda$')
189 ifOurCf2tex([basename 'Eval'])%optionally save

```

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

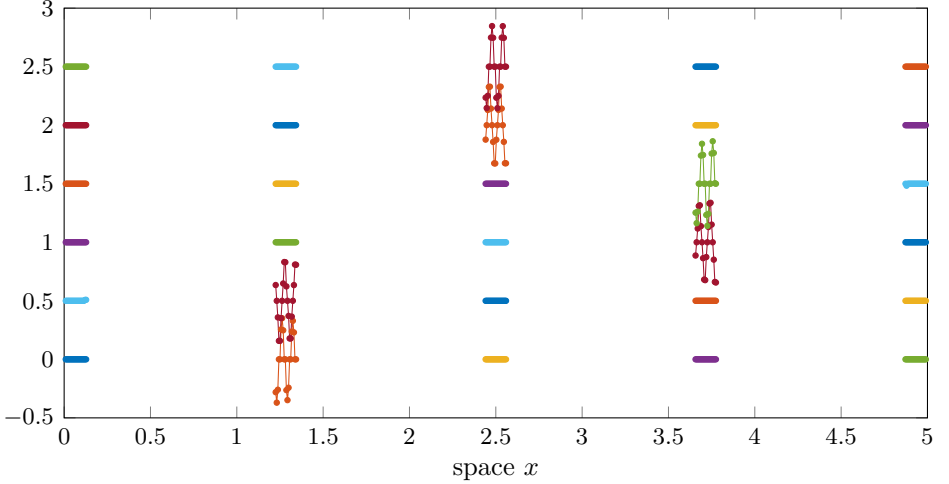
```

210 logGaps=diff(log10(real(mEval)));
211 [~,j]=sort(-logGaps);
212 %someLogGaps=[logGaps(j(1:5)) j(1:5)]
213 if logGaps(j(2))<0.4*logGaps(j(1)), nSlow=j(1)
214 else nSlow=min( sort(j(1:2)) , 3*nPatch)
215 end
216 log10Gap=logGaps(nSlow)
217 smallEvals=-mEval(1:nSlow(end)+2)

```

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

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



```

227 js=find(imag(mEval)>0);
228 evec(:,js)=imag(evec(:,js));
229 evec=real(evec);
230 if numel(nSlow)==1, S = spcart(evec(:,1:nSlow));
231 else S = spcart(evec(:,1:nSlow(1)));
232     S = [S spcart(evec(:,nSlow(1)+1:nSlow(2)))] ;
233 end;
234 figure(3),clf
235 vStep=ceil(max(abs(S(:)))*10+1)/10
236 for j=1:nSlow(end)
237     u0(patches.i)=S(:,j);
238     plot(xs,vStep*(j-1)+squeeze(u0),'.-'),hold on
239 end
240 hold off, xlabel('space $x$')
241 ifOurCf2tex([basename 'Evec'])%optionally save

```

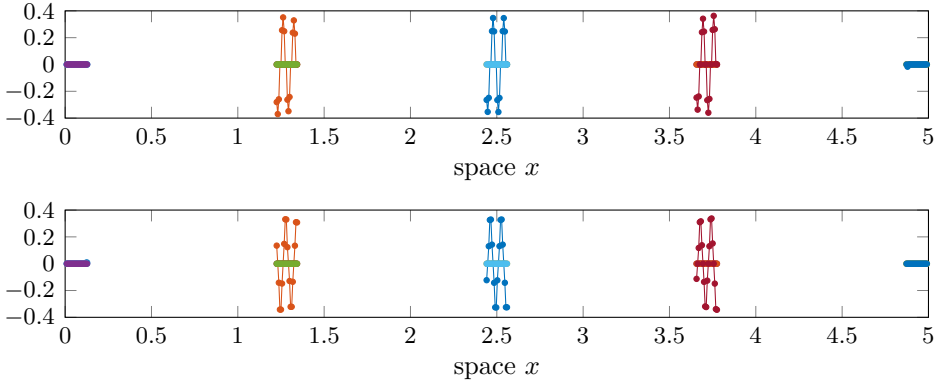
Reorganise the eigenvectors to maybe clarify.

```

262 [i,j]=find(abs(S)>vStep/2);

```


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



```

263 j=find([1;diff(j)]);
264 [i,k]=sort(i(j));
265 figure(4)
266 for p=1:2
267     clf,subplot(2,1,1)
268     for j=p:2:numel(k)
269         u0(patches.i)=S(:,k(j));
270         plot(xs,squeeze(u0),'-'),hold on
271     end% for j
272     hold off, xlabel('space $x$')
273     ifOurCf2tex([basename 'Evec' num2str(p)])%optionally save
274 end%for p

```

3.0.2 Find an equilibrium with fsolve

Start the search from some guess.

```

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

```

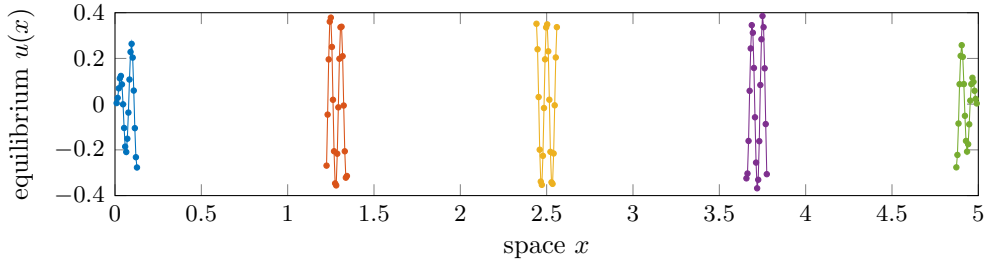
But set the pairs of patch-edge values to Nan in order to use patches.i to index the interior sub-patch points as they are the variables.

```

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

```

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



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

```

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

```

Plot the equilibrium see Figure 8.

```

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

```

3.0.3 Simulate in time

Set an initial condition, and here integrate forward in time using a standard method for stiff systems—because of the simplicity of linear problems this method works quite efficiently here. Integrate the interface `patchSys1` (Section 6) to the microscale differential equations.

```

357 fprintf('\n**** Simulate in time\n')
358 u0 = 0*sin(2*pi/waveLength*patches.x)+0.1*randn(nSubP,1,1,nPatch);
359 tic

```

```

360 [ts,us] = ode15s(@patchSys1, [0 40], u0(:) ,[],patches,k0,Ra,1);
361 simulateTime = toc
362 us = reshape(us,length(ts),numel(patches.x(:)),[]);

```

Plot the simulation in [Figure 4](#).

```

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

```

Fin.

3.1 Heterogeneous SwiftHohenberg PDE+BCs inside patches

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

```

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

```

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

```

399     u = squeeze(u);
400     if ~patches.periodic
401         switch 1
402             case 1 % these are u=u_x=0
403                 u(1:2,1)=0;
404                 u(end-1:end,end)=0;
405             case 2 % these are u=u_{xx}=0
406                 u(1:2,1) = [-u(3,1); 0];
407                 u(end-1:end,end) = [0; -u(end-2,end)];
408             end% case
409         end%if

```

Here code straightforward centred discretisation in space.

```

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

```

3.2 theRes(): a wrapper function

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

```

435 function f=theRes(u,patches,k0,Ra,cubic)
436     v=nan(size(patches.x));
437     v(patches.i) = u;
438     f = patchSys1(0,v(:),patches,k0,Ra,cubic);
439     f = f(patches.i);
440 end%function theRes

```

New configuration and interpolation

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

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

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

¹Script `patchEdgeInt1test.m` verifies this code.

```

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

```

Input

- **u** is a vector/array of length $\text{nSubP} \cdot \text{nVars} \cdot \text{nEnsem} \cdot \text{nPatch}$ where there are $\text{nVars} \cdot \text{nEnsem}$ field values at each of the points in the $\text{nSubP} \times \text{nPatch}$ multiscale spatial grid.
- **patches** a struct largely set by `configPatches1()`, and which includes the following.
 - **.x** is $\text{nSubP} \times 1 \times 1 \times \text{nPatch}$ array of the spatial locations x_{iI} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index i , but may be variable spaced in macroscale index I .
 - **.ordCC** is order of interpolation, integer ≥ -1 .
 - **.periodic** indicates whether macroscale is periodic domain, or alternatively that the macroscale has left and right boundaries so interpolation is via divided differences.
 - **.stag** in $\{0, 1\}$ is one for staggered grid (alternating) interpolation, and zero for ordinary grid.
 - **.Cwtsr** and **.Cwtsl** are the coupling coefficients for finite width interpolation—when invoking a periodic domain.
 - **.EdgyInt**, true/false, for determining patch-edge values by interpolation: true, from opposite-edge next-to-edge values (often preserves symmetry); false, from centre-patch values (original scheme).
 - **.nEdge**, for each patch, the number of edge values set by interpolation at the edge regions of each patch (default is one).
 - **.nEnsem** the number of realisations in the ensemble.
 - **.parallel** whether serial or parallel.
 - **.nCore** ²

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

Output

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

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

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

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

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

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

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

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

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

```
153 x = patches.x;
154 if patches.nEdge>1
155     nEdge = patches.nEdge;
156     x = x(1:nEdge:nx,:,:,:);
157     nx = nx/nEdge;
158     u = reshape(u,nEdge,nx,nVars,nEnsem,Nx);
```

```

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

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

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

```

4.1 Periodic macroscale interpolation schemes

```

180 if patches.periodic

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

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

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

```

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

```

279   else% patches.nCore>1
280       error('not yet considered, july--dec 2020 ??')
281       u(nx,:,I) = Ux(:,I)*(1-patches.stag) ...

```



```

282         + reshape(-sum(u((nx-patches.nCore+1):(nx-1),:,:,I),1) ...
283         + sum( patches.Cwtsr.*dmu ),Nx,nVars);
284     u(1,:,:,I) = Ux(:,:,I)*(1-patches.stag) ...
285         + reshape(-sum(u(2:patches.nCore,:,:,I),1) ...
286         + sum( patches.Cwtsl.*dmu ),Nx,nVars);
287 end%if patches.nCore

```

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

```

297 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 N_x patches we resolve ‘wavenumbers’ $|k| < N_x/2$, so set row vector $\mathbf{k}s = k2\pi/N$ 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. ^{3 4}

```

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

```

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

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

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

4.2 Non-periodic macroscale interpolation

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

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

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

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

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

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

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

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

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

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

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

```

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

```

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

```

474 elseif false
475     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.

```

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

```

Initialise to highest divide difference, surrounded by zeros.

```

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

```

Complete Horner evaluation of the relevant polynomials.

```

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

```

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

```

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

```

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

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

```

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

```

End of the non-periodic interpolation code.

```

532 end%if patches.periodic

```

Unfold multiple edges No need to restore x .

```

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

```

Fin, returning the 4D array of field values.

5 configPatches1(): configure spatial patches in 1D

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

```

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

```

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

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)` or `fun(t,u,patches,...)`, that computes time derivatives (or time-steps) of quantities on the 1D micro-grid within all the 1D patches.

- `Xlim` give the macro-space spatial domain of the computation, namely the interval `[Xlim(1), Xlim(2)]`.
- `Dom` sets the type of macroscale conditions for the patches, and reflects the type of microscale boundary conditions of the problem. If `Dom` is `NaN` or `[]`, then the field `u` is macro-periodic in the 1D spatial domain, and resolved on equi-spaced patches. If `Dom` is a character string, then that specifies the `.type` of the following structure, with `.bcOffset` set to the default zero. Otherwise `Dom` is a structure with the following components.
 - `.type`, string, of either `'periodic'` (the default), `'equispace'`, `'chebyshev'`, `'usergiven'`. For all cases except `'periodic'`, users *must* code into `fun` the micro-grid boundary conditions that apply at the left(right) edge of the leftmost(rightmost) patches.
 - `.bcOffset`, optional one or two element array, in the cases of `'equispace'` or `'chebyshev'` the patches are placed so the left/right macroscale boundaries are aligned to the left/right edges of the corresponding extreme patches, but offset by `bcOffset` of the sub-patch micro-grid spacing. For example, use `bcOffset=0` when applying Dirichlet boundary values on the extreme edge micro-grid points, whereas use `bcOffset=0.5` when applying Neumann boundary conditions halfway between the extreme edge micro-grid points.
 - `.X`, optional array, in the case `'usergiven'` it specifies the locations of the centres of the `nPatch` patches—the user is responsible it makes sense.
- `nPatch` is the number of equi-spaced spatial patches.
- `ordCC`, must be ≥ -1 , is the ‘order’ of interpolation across empty space of the macroscale patch values to the edge of the patches for inter-patch coupling: where `ordCC` of 0 or -1 gives spectral interpolation; and `ordCC` being odd specifies staggered spatial grids.
- `dx` (real) is usually the sub-patch micro-grid spacing in x .

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

ratio = 1 means the patches abut. Small **ratio** should greatly reduce computational time.

- **nSubP** is the number of equi-spaced microscale lattice points in each patch. If not using **EdgyInt**, then **nSubP/nEdge** must be odd integer so that there is/are centre-patch lattice point(s). So for the defaults of **nEdge** = 1 and not **EdgyInt**, then **nSubP** must be odd.
- '**nEdge**', *optional*, default=1, the number of edge values set by interpolation at the edge regions of each patch. The default is one (suitable for microscale lattices with only nearest neighbour interactions).
- **EdgyInt**, true/false, *optional*, default=false. If true, then interpolate to left/right edge-values from right/left next-to-edge values. If false or omitted, then interpolate from centre-patch values.
- **nEnsem**, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.
- **hetCoeffs**, *optional*, default empty. Supply a 1D or 2D array of microscale heterogeneous coefficients to be used by the given microscale **fun** in each patch. Say the given array **cs** is of size $m_x \times n_c$, where n_c is the number of different sets of coefficients. The coefficients are to be the same for each and every patch; however, macroscale variations are catered for by the n_c coefficients being n_c parameters in some macroscale formula.
 - If **nEnsem** = 1, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the first point in each patch. Best accuracy usually obtained when the periodicity of the coefficients is a factor of **nSubP-2*nEdge** for **EdgyInt**, or a factor of $(\text{nSubP}-\text{nEdge})/2$ for not **EdgyInt**.
 - If **nEnsem** > 1 (value immaterial), then reset **nEnsem** := m_x and construct an ensemble of all m_x phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When **EdgyInt** is true, and when the coefficients are diffusivities/elasticities, then this coupling cunningly preserves symmetry.
- **nCore**, *optional-experimental*, default one, but if more, and only for non-**EdgyInt**, then interpolates from an average over the core of a patch, a

core of size $??$. Then edge values are set according to interpolation of the averages $??$ or so that average at edges is the interpolant $??$

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

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

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

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

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

⁷When 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.

5.1 If no arguments, then execute an example

```

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

```

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

1. `configPatches1`
2. `ode15s` integrator \mapsto `patchSys1` \mapsto user's PDE
3. process results

Establish global patch data struct to point to and interface with a function coding Burgers' PDE: to be solved on 2π -periodic domain, with eight patches, spectral interpolation couples the patches, with micro-grid spacing 0.06, and with seven microscale points forming each patch.

```

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

```

Set some initial condition, with some microscale randomness.

```

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

```

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

```

298 if ~exist('OCTAVE_VERSION','builtin')
299 [ts,us] = ode15s( @patchSys1,[0 0.5],u0(:));
300 else % octave version
301 [ts,us] = ode0cts(@patchSys1,[0 0.5],u0(:));
302 end

```

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

```

314 figure(1),clf
315 if 1, patches.x([1 end],:,:,:) = nan; us=us.';
316 else us=reshape(patchEdgyInt1(us.'),[],length(ts));
317 end
318 mesh(ts,patches.x(:),us)
319 view(60,40), colormap(0.7*hsv)
320 title('Burgers PDE: patches in space, continuous time')
321 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')

```

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

```

335 ifOurCf2eps(mfilename)
336 return
337 end%if nargin==0

```

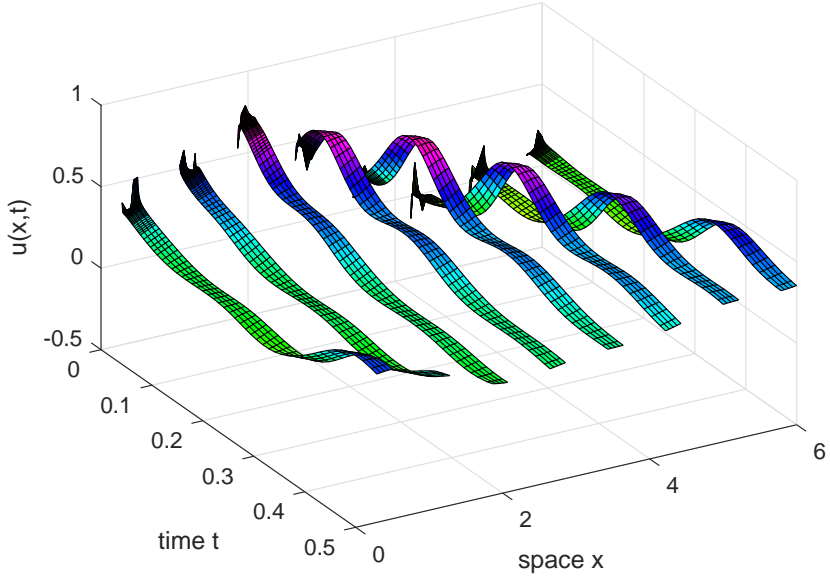
5.2 Parse input arguments and defaults

```

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

```

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



```

354 addRequired(p,'fun',fnValidation);
355 addRequired(p,'Xlim',@isnumeric);
356 %addRequired(p,'Dom'); % nothing yet decided
357 addRequired(p,'nPatch',@isnumeric);
358 addRequired(p,'ordCC',@isnumeric);
359 addRequired(p,'dx',@isnumeric);
360 addRequired(p,'nSubP',@isnumeric);
361 addParameter(p,'nEdge',1,@isnumeric);
362 addParameter(p,'EdgyInt',false,@islogical);
363 addParameter(p,'nEnsem',1,@isnumeric);
364 addParameter(p,'hetCoeffs',[],@isnumeric);
365 addParameter(p,'parallel',false,@islogical);
366 addParameter(p,'nCore',1,@isnumeric);
367 parse(p,fun,Xlim,nPatch,ordCC,dx,nSubP,varargin{:});

```

Set the optional parameters.

```

373 patches.nEdge = p.Results.nEdge;
374 patches.EdgyInt = p.Results.EdgyInt;
375 patches.nEnsem = p.Results.nEnsem;

```

```

376 cs = p.Results.hetCoeffs;
377 patches.parallel = p.Results.parallel;
378 patches.nCore = p.Results.nCore;

    Check parameters.

385 assert(Xlim(1)<Xlim(2) ...
386         , 'two entries of Xlim must be ordered increasing')
387 assert((mod(ordCC,2)==0)|(patches.nEdge==1) ...
388         , 'Cannot yet have nEdge>1 and staggered patch grids')
389 assert(3*patches.nEdge<=nSubP ...
390         , 'too many edge values requested')
391 assert(rem(nSubP,patches.nEdge)==0 ...
392         , 'nSubP must be integer multiple of nEdge')
393 if ~patches.EdgeyInt, assert(rem(nSubP/patches.nEdge,2)==1 ...
394         , 'for non-edgyInt, nSubP/nEdge must be odd integer')
395     end
396 if (patches.nEnsem>1)&(patches.nEdge>1)
397     warning('not yet tested when both nEnsem and nEdge non-one')
398     end
399 if patches.nCore>1
400     warning('nCore>1 not yet tested in this version')
401     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.

```

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

```

Default macroscale conditions are periodic with evenly spaced patches.

```

421 if isempty(Dom), Dom=struct('type','periodic'); end
422 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.

```

430 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.

```

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

```

5.3 The code to make patches and interpolation

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

```

471 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 .

```

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

```

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

```

488 patches.stag=mod(ordCC,2);
489 ordCC=ordCC+patches.stag;
490 patches.ordCC=ordCC;

```

Check for staggered grid and periodic case.

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

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

```
507 switch Dom.type
```

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

```
515 case 'periodic'  
516     X=linspace(Xlim(1),Xlim(2),nPatch+1);  
517     DX=X(2)-X(1);  
518     X=X(1:nPatch)+diff(X)/2;  
519     pEI=patches.EdgeInt;% abbreviation  
520     pnE=patches.nEdge; % abbreviation  
521     if pre2023, dx = ratio*DX/(nSubP-pnE*(1+pEI))*(2-pEI);  
522     else      ratio = dx/DX*(nSubP-pnE*(1+pEI))/(2-pEI); end  
523     patches.ratio=ratio;
```

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

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

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

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

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

The Chebyshev case is spaced according to the Chebyshev distribution in order to reduce macro-interpolation errors, $X_i \propto -\cos(i\pi/N)$, but with the extreme edges aligned with the spatial domain boundaries, modified by the offset, and modified by possible ‘boundary layers’.⁹

```

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

```

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

```

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

```

Assign the centre-patch coordinates.

```

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

```

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

```

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

```

⁹ However, maybe overlapping patches near a boundary should be viewed as some sort of spatial analogue of the ‘christmas tree’ of projective integration and its projection to a slow manifold. Here maybe the overlapping patches allow for a ‘christmas tree’ approach to the boundary layers. Needs to be explored??

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

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

5.4 Set ensemble inter-patch communication

For `EdgyInt` or centre interpolation respectively,

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

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

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

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

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

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

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

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

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

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


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

```

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

```

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

```

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

```

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

```

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

```

End the two if-statements.

```

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

```

If parallel code then first assume this is not within an `spmd`-environment, and so we invoke `spmd...end` (which starts a parallel pool if not already started). At this point, the global `patches` is copied for each worker processor and so it becomes *composite* when we distribute any one of the fields. Hereafter, *all fields in the global variable patches must only be referenced within an spmd-environment*.¹⁰

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

Fin

```

782 end% function

```

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

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

```

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

```

Input

- **u** is a vector/array of length $\mathbf{nSubP} \cdot \mathbf{nVars} \cdot \mathbf{nEnsem} \cdot \mathbf{nPatch}$ where there are $\mathbf{nVars} \cdot \mathbf{nEnsem}$ field values at each of the points in the $\mathbf{nSubP} \times \mathbf{nPatch}$ grid.
- **t** is the current time to be passed to the user's time derivative function.
- **patches** a struct set by `configPatches1()` with the following information used here.
 - **.fun** is the name of the user's function `fun(t,u,patches,...)` that computes the time derivatives on the patchy lattice. The array **u** has size $\mathbf{nSubP} \times \mathbf{nVars} \times \mathbf{nEnsem} \times \mathbf{nPatch}$. Time derivatives should be computed into the same sized array, then herein the patch edge values are overwritten by zeros.
 - **.x** is $\mathbf{nSubP} \times 1 \times 1 \times \mathbf{nPatch}$ array of the spatial locations x_i of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale.
- **varargin**, optional, is arbitrary number of parameters to be passed onto the users time-derivative function as specified in `configPatches1`.

Output

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

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

```

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

```

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

```

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

```

Fin.

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

Couples 2D patches across 2D space by computing their edge values via macroscale interpolation. Research (Roberts, MacKenzie, and Bunder 2014; Bunder et al. 2021) indicates the patch centre-values are sensible macroscale variables, and macroscale interpolation of these determine patch-edge values. However, for computational homogenisation in multi-D, interpolating patch next-to-edge values appears better (Bunder, Kevrekidis, and Roberts 2021). This function is primarily used by `patchSys2()` but is also useful for user graphics.¹¹

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

```

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

```

Input

- `u` is a vector/array of length $\text{prod}(\text{nSubP}) \cdot \text{nVars} \cdot \text{nEnsem} \cdot \text{prod}(\text{nPatch})$ 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 .

¹¹Script `patchEdgeInt2test.m` verifies this code.

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

Output

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

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

```

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

```

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

```

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

```

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

```

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

```

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

```

160 %x = patches.x;
161 %if patches.nEdge(1)>1
162 % m = patches.nEdge(1);
163 % x = x(1:m:nx, :, :, :, :);
164 % nx = nx/m;
165 % u = reshape(u,m,nx,ny,nVars,nEnsem,Nx,Ny);
166 % nVars = nVars*m;
167 % u = reshape( permute(u,[2:3 1 4:7]) ...
168 %             ,nx,ny,nVars,nEnsem,Nx,Ny);
169 %end%if patches.nEdge(1)
170 %y = patches.y;
171 %if patches.nEdge(2)>1
172 % m = patches.nEdge(2);
173 % y = y(:,1:m:ny, :, :, :);
174 % ny = ny/m;
175 % u = reshape(u,nx,m,ny,nVars,nEnsem,Nx,Ny);
176 % nVars = nVars*m;
177 % u = reshape( permute(u,[1 3 2 4:7]) ...
178 %             ,nx,ny,nVars,nEnsem,Nx,Ny);
179 %end%if patches.nEdge(2)
180 x = patches.x;

```

```

181 y = patches.y;
182 if mean(patches.nEdge)>1
183     mx = patches.nEdge(1);
184     my = patches.nEdge(2);
185     x = x(1:mx:nx,:,:,:,:);
186     y = y(:,1:my:ny,:,:,:,:);
187     nx = nx/mx;
188     ny = ny/my;
189     u = reshape(u,mx,nx,my,ny,nVars,nEnsem,Nx,Ny);
190     nVars = nVars*mx*my;
191     u = reshape( permute(u,[2 4 1 3 5:8]) ...
192                 ,nx,ny,nVars,nEnsem,Nx,Ny);
193 end%if patches.nEdge

```

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

```

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

```

7.1 Periodic macroscale interpolation schemes

```

211 if patches.periodic

```

Get the size ratios of the patches.

```

217 rx = patches.ratio(1);
218 ry = patches.ratio(2);

```

7.1.1 Lagrange interpolation gives patch-edge values

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

```

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

```

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

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

```

247     if patches.EdgeInt % interpolate next-to-face values
248         U = u([2 nx-1],2:(ny-1),:,:,I,J);
249     else % interpolate centre-cross values
250         U = u(i0,2:(ny-1),:,:,I,J);
251     end;%if patches.EdgeInt

```

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

```

259     szU0=size(U); szU0=[szU0 ones(1,6-length(szU0)) 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.

```

269     if ~patches.parallel, dmU = zeros(szU0); % 7D
270     else dmU = zeros(szU0,patches.codist); % 7D
271     end%if patches.parallel

```

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

```

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

```

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


```

297   for k = 3:ordCC
298       dmu(:,:,,:,I,:,k) =      dmu(:,:,,:,Ip,:,k-2) ...
299       -2*dmu(:,:,,:,I,:,k-2) +dmu(:,:,,:,Im,:,k-2);
300   end

```

Interpolate macro-values to be Dirichlet edge values for each patch (Roberts and Kevrekidis 2007; Bunder, Roberts, and Kevrekidis 2017), using weights computed in `configPatches2()`. Here interpolate to specified order.

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

```

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

```

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

```

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

```

Adjoin extra dimension for the array of differences.

```

341   szU0=size(U); szU0=[szU0 ones(1,6-length(szU0)) ordCC];

```

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

```

348   if ~patches.parallel, dmu = zeros(szU0); % 7D
349   else dmu = zeros(szU0,patches.codist); % 7D
350   end%if patches.parallel

```

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

```

356 if patches.stag % use only odd numbered neighbours
357     error('polynomial interpolation not yet for staggered patch coupling')
358 else %disp('starting standard interpolation')
359     dmu(:,:,,:,J,1) = (U(:,:,,:,Jp) ...
360         -U(:,:,,:,Jm))/2; %\mu\delta
361     dmu(:,:,,:,J,2) = (U(:,:,,:,Jp) ...
362         -2*U(:,:,,:,J) +U(:,:,,:,Jm)); %\delta^2
363 end% if stag

```

Recursively take δ^2 .

```

369 for k = 3:ordCC
370     dmu(:,:,,:,J,k) = dmu(:,:,,:,Jp,k-2) ...
371         -2*dmu(:,:,,:,J,k-2) +dmu(:,:,,:,Jm,k-2);
372 end

```

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

```

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

```

7.1.2 Case of spectral interpolation

Assumes the domain is macro-periodic.

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

Compute the Fourier transform of the centre-cross values. Unless doing patch-edge 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`.

```

469 if ~patches.EdgyInt
470 Cm = fft( u(i0,iy,:,:,,:) ,[],5);
471 Cp = Cm;

```

```

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

```

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

```

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

```

y-normal edge values Set wavenumbers into a vector.

```

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

```

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

```

503 if ~patches.EdgyInt
504     Cm = fft( u(:,j0,:,:,,:) ,[],6);
505     Cp = Cm;
506 else
507     Cm = fft( u(:,2 ,:,patches.bo,:,:) ,[],6);
508     Cp = fft( u(:,ny-1 ,:,patches.to,:,:) ,[],6);
509 end%if ~patches.EdgyInt

```

Invert the Fourier transforms to complete interpolation.

```

515 u(:,ny,:,:,,:) = uclean( ifft( ...
516     Cm.*exp(1i*(stagShift+kr)) ,[],6) );
517 u(:, 1,:,:,,:) = uclean( ifft( ...
518     Cp.*exp(1i*(stagShift-kr)) ,[],6) );
524 end% if ordCC>0 else, so spectral

```

7.2 Non-periodic macroscale interpolation

```

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

```

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

```

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

```

7.2.1 x -direction values

Set function values in first ‘column’ of the tables for every variable and across ensemble. For `EdgyInt`, the ‘reversal’ of the next-to-edge values are because their values are to interpolate to the opposite edge of each patch. ¹²

```

569 F = nan(patches.EdgyInt+1,ny-2,nVars,nEnsem,Nx,Ny,px+1);
570 if patches.EdgyInt % interpolate next-to-edge values
571     F(:, :, :, :, :, 1) = u([nx-1 2],iy, :, :, :, :);
572     X = x([nx-1 2], :, :, :, :, :);
573 else % interpolate mid-patch cross-patch values
574     F(:, :, :, :, :, 1) = u(i0,iy, :, :, :, :);
575     X = x(i0, :, :, :, :, :);
576 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.

```

587 for q = 1:px
588     i = 1:Nx-q;
589     F(:, :, :, :, i, :, q+1) ...
590     = (F(:, :, :, :, i+1, :, q)-F(:, :, :, :, i, :, q)) ...
591     ./ (X(:, :, :, :, i+q, :) - X(:, :, :, :, i, :));
592 end

```

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

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

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

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

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

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

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

7.2.2 *y*-direction values

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

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

Form tables of divided differences.

```
647 for q = 1:py
648     j = 1:Ny-q;
649     F(:,:,:,j,q+1) ...
650     = (F(:,:,:,j+1 ,q)-F(:,:,:,j,q)) ...
651     ./(Y(:,:,:,j+q) -Y(:,:,:,j));
652 end
```

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

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

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

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

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

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

7.2.3 Optional NaNs for safety

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

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

End of the non-periodic interpolation code.

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

Unfold multiple edges No need to restore x, y .

```

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

```

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

```

727 end% function patchEdgeInt2

```

8 configPatches2(): configures spatial patches in 2D

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys2()`. [Section 8.1](#) lists an example of its use.

```

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

```

Input If invoked with no input arguments, then executes an example of simulating a nonlinear diffusion PDE relevant to the lubrication flow of a thin layer of fluid—see [Section 8.1](#) for an example code.

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)` or `fun(t,u,patches,...)`, that computes time-derivatives (or time-steps) of quantities on the 2D micro-grid within all the 2D patches.
- `Xlim` array/vector giving the rectangular macro-space domain of the computation, namely $[Xlim(1),Xlim(2)] \times [Xlim(3),Xlim(4)]$. 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/top/bottom macroscale boundaries are aligned to the left/right/top/bottom edges of the corresponding extreme patches, but offset by `.bcOffset` of the sub-patch micro-grid spacing. For example, use `bcOffset=0` when the micro-code applies Dirichlet boundary values on the extreme edge micro-grid points, whereas use `bcOffset=0.5` when the microcode applies Neumann boundary conditions halfway between the extreme edge micro-grid points. Similarly for the top and bottom edges.

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

- `.X`, optional vector/array with `nPatch(1)` elements, in the case `'usergiven'` it specifies the x -locations of the centres of the patches—the user is responsible the locations makes sense.
 - `.Y`, optional vector/array with `nPatch(2)` elements, in the case `'usergiven'` it specifies the y -locations of the centres of the patches—the user is responsible the locations makes sense.
- `nPatch` sets the number of equi-spaced spatial patches: if scalar, then use the same number of patches in both directions, otherwise `nPatch(1:2)` gives the number of patches (≥ 1) in each direction.
 - `ordCC` is the ‘order’ of interpolation for inter-patch coupling across empty space of the macroscale patch values to the edge-values of the patches: currently must be 0, 2, 4, \dots ; where 0 gives spectral interpolation.

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

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

- **nSubP** is the number of equi-spaced microscale lattice points in each patch: if scalar, then use the same number in both directions, otherwise **nSubP(1:2)** gives the number in each direction. If not using **EdgyInt**, then **nSubP./nEdge** must be odd integer(s) so that there is/are centre-patch lattice lines. So for the defaults of **nEdge** = 1 and not **EdgyInt**, then **nSubP** must be odd.
- '**nEdge**', *optional* (integer—scalar or two element), default=1, the width of edge values set by interpolation at the edge regions of each patch. If two elements, then respectively the width in x, y -directions. The default is one (suitable for microscale lattices with only nearest neighbour interactions).
- **EdgyInt**, true/false, *optional*, default=false. If true, then interpolate to left/right/top/bottom edge-values from right/left/bottom/top next-to-edge values. If false or omitted, then interpolate from centre cross-patch lines.
- **nEnsem**, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.
- **hetCoeffs**, *optional*, default empty. Supply a 2D or 3D array of microscale heterogeneous coefficients to be used by the given microscale **fun** in each patch. Say the given array **cs** is of size $m_x \times m_y \times n_c$, where n_c is the number of different sets of coefficients. For example, in heterogeneous diffusion, $n_c = 2$ for the diffusivities in the *two* different spatial directions (or $n_c = 3$ for the diffusivity tensor). The coefficients are to be the same for each and every patch; however, macroscale variations are catered for by the n_c coefficients being n_c parameters in some macroscale formula.

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

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

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

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

```
123 if nargout==0, global patches, end
124 patches.version = version;
```

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

¹³When 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**, only for macro-periodic conditions, are the $\text{ordCC} \times 2$ -array of weights for the inter-patch interpolation onto the right/top and left/bottom edges (respectively) with patch:macroscale ratio as specified or as derived from **dx**.
- **.x** (6D) is $\text{nSubP}(1) \times 1 \times 1 \times 1 \times \text{nPatch}(1) \times 1$ array of the regular spatial locations x_{iI} of the microscale grid points in every patch.
- **.y** (6D) is $1 \times \text{nSubP}(2) \times 1 \times 1 \times 1 \times \text{nPatch}(2)$ array of the regular spatial locations y_{jJ} of the microscale grid points in every patch.
- **.ratio** 1×2 , only for macro-periodic conditions, are the size ratios of every patch.
- **.nEdge** 1×2 , is the width of edge values set by interpolation at the edge regions of each patch, in the x, y -directions respectively.
- **.le**, **.ri**, **.bo**, **.to** determine inter-patch coupling of members in an ensemble. Each a column vector of length **nEnsem**.
- **.cs** either
 - [] 0D, or
 - if **nEnsem** = 1, $(\text{nSubP}(1) - 1) \times (\text{nSubP}(2) - 1) \times n_c$ 3D array of microscale heterogeneous coefficients, or
 - if **nEnsem** > 1, $(\text{nSubP}(1) - 1) \times (\text{nSubP}(2) - 1) \times n_c \times m_x m_y$ 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.

8.1 If no arguments, then execute an example

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

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

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

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

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

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

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

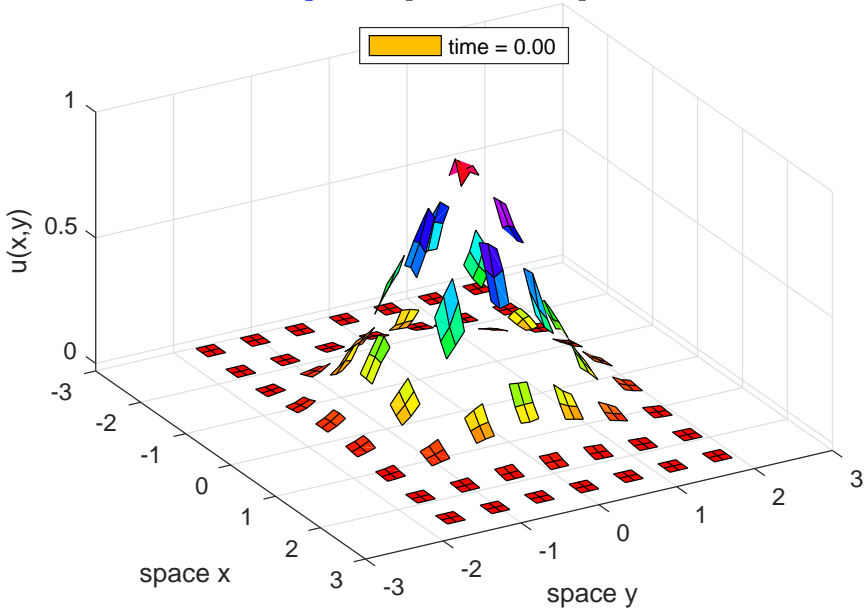
Initiate a plot of the simulation using only the microscale values interior to the patches: optionally set x and y -edges to `nan` to leave the gaps between patches.

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

Start by showing the initial conditions of Figure 10 while the simulation computes.

```
349 u = reshape(permute(squeeze(u0) ...
350     , [1 3 2 4]), [numel(x) numel(y)]);
```

Figure 10: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to a nonlinear ‘diffusion’ PDE: [Figure 11](#) plots the computed field at time $t = 3$.



```

351 hsurf = mesh(x(:),y(:),u');
352 axis([-3 3 -3 3 -0.03 1]), view(60,40)
353 legend('time = 0.00','Location','north')
354 xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
355 colormap(hsv)
356 ifOurCf2eps([mfilename 'ic'])

```

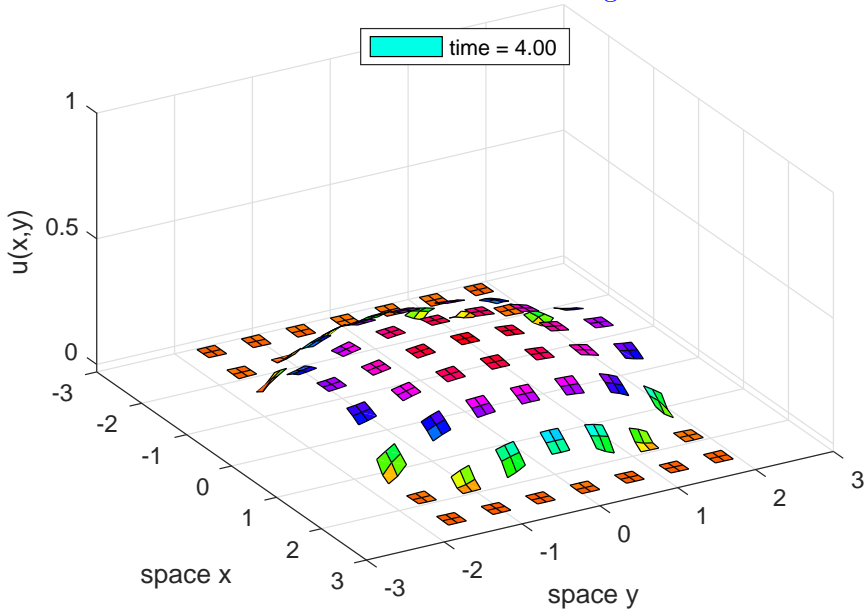
Integrate in time to $t = 4$ using standard functions. In MATLAB `ode15s` would be natural as the patch scheme is naturally stiff, but `ode23` is quicker (Maclean, Bunder, and Roberts [2021](#), Fig. 4). Ask for output at non-uniform times because the diffusion slows.

```

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

```

Figure 11: field $u(x,y,t)$ at time $t = 3$ of the patch scheme applied to a nonlinear ‘diffusion’ PDE with initial condition in [Figure 10](#).



```

380     [ts,us] = ode0cts(@patchSys2,[0 1],u0(:));
381 end

```

Animate the computed simulation to end with [Figure 11](#). Use `patchEdgeInt2` to interpolate patch-edge values.

```

389 for i = 1:length(ts)
390     u = patchEdgeInt2(us(i,:));
391     u = reshape(permute(squeeze(u) ...
392         ,[1 3 2 4]), [numel(x) numel(y)]);
393     set(hsurf,'ZData', u');
394     legend(['time = ' num2str(ts(i),'%.2f')])
395     pause(0.1)
396 end
397 if0urCf2eps([mfilename 't3'])

```

Upon finishing execution of the example, exit this function.

```

412 return
413 end%if no arguments

```

8.2 Parse input arguments and defaults

```
427 p = inputParser;
428 fnValidation = @(f) isa(f, 'function_handle');%test for fn name
429 addRequired(p,'fun',fnValidation);
430 addRequired(p,'Xlim',@isnumeric);
431 %addRequired(p,'Dom'); % nothing yet decided
432 addRequired(p,'nPatch',@isnumeric);
433 addRequired(p,'ordCC',@isnumeric);
434 addRequired(p,'dx',@isnumeric);
435 addRequired(p,'nSubP',@isnumeric);
436 addParameter(p,'nEdge',1,@isnumeric);
437 addParameter(p,'EdgyInt',false,@islogical);
438 addParameter(p,'nEnsem',1,@isnumeric);
439 addParameter(p,'hetCoeffs',[],@isnumeric);
440 addParameter(p,'parallel',false,@islogical);
441 %addParameter(p,'nCore',1,@isnumeric); % not yet implemented
442 parse(p,fun,Xlim,nPatch,ordCC,dx,nSubP,varargin{:});
```

Set the optional parameters.

```
448 patches.nEdge = p.Results.nEdge;
449 if numel(patches.nEdge)==1
450     patches.nEdge = repmat(patches.nEdge,1,2);
451 end
452 patches.EdgyInt = p.Results.EdgyInt;
453 patches.nEnsem = p.Results.nEnsem;
454 cs = p.Results.hetCoeffs;
455 patches.parallel = p.Results.parallel;
456 %patches.nCore = p.Results.nCore;
```

Initially duplicate parameters for both space dimensions as needed.

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

Check parameters.

```
474 assert(Xlim(1)<Xlim(2) ...
475     , 'first pair of Xlim must be ordered increasing')
```



```

476 assert(Xlim(3)<Xlim(4) ...
477         , 'second pair of Xlim must be ordered increasing')
478 assert((mod(ordCC,2)==0)|all(patches.nEdge==1) ...
479         , 'Cannot yet have nEdge>1 and staggered patch grids')
480 assert(all(3*patches.nEdge<=nSubP) ...
481         , 'too many edge values requested')
482 assert(all(rem(nSubP,patches.nEdge)==0) ...
483         , 'nSubP must be integer multiple of nEdge')
484 if ~patches.EdgeyInt, assert(all(rem(nSubP./patches.nEdge,2)==1) ...
485         , 'for non-edgyInt, nSubP./nEdge must be odd integer')
486     end
487 if (patches.nEnsem>1)&all(patches.nEdge>1)
488     warning('not yet tested when both nEnsem and nEdge non-one')
489 end
490 %if patches.nCore>1
491 %    warning('nCore>1 not yet tested in this version')
492 %    end

```

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

```

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

```

Default macroscale conditions are periodic with evenly spaced patches.

```

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

```

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

```

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

```

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

```

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

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

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

```

8.3 The code to make patches

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

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

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

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

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

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

Check for staggered grid and periodic case.

```
613 if patches.stag, assert(all(mod(nPatch,2)==0), ...
614         'Require an even number of patches for staggered grid')
615 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.

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

Distribution depends upon Dom.type:

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

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

```
643 case 'periodic'
644     Q=linspace(Xlim(qq),Xlim(qq+1),nPatch(q)+1);
645     DQ=Q(2)-Q(1);
```

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

```

646 Q=Q(1:nPatch(q))+diff(Q)/2;
647 pEI=patches.EdgyInt; % abbreviation
648 pnE=patches.nEdge(q);% abbreviation
649 if pre2023, dx(q) = ratio(q)*DQ/(nSubP(q)-pnE*(1+pEI))*(2-pEI);
650 else      ratio(q) = dx(q)/DQ*(nSubP(q)-pnE*(1+pEI))/(2-pEI);
651 end
652 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.

```

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

```

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

```

686 case 'chebyshev'
687     halfWidth=dx(q)*(nSubP(q)-1)/2;
688     Q1 = Xlim(1)+halfWidth-Dom.bcOffset(qq)*dx(q);
689     Q2 = Xlim(2)-halfWidth+Dom.bcOffset(qq+1)*dx(q);
690 % 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`.

```

699 pEI=patches.EdgyInt; % abbreviation
700 pnE=patches.nEdge(q);% abbreviation

```

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

```

701 width=(1+pEI)/2*(nSubP(q)-pnE*(1+pEI))*dx(q);
702 for b=0:2:nPatch(q)-2
703     DQmin=(Q2-Q1-b*width)/2*( 1-cos(pi/(nPatch(q)-b-1)) );
704     if DQmin>width, break, end
705 end%for
706 if DQmin<width*0.999999
707     warning('too many Chebyshev patches (mid-domain overlap)')
708 end

```

Assign the centre-patch coordinates.

```

714 Q =[ Q1+(0:b/2-1)*width ...
715       (Q1+Q2)/2-(Q2-Q1-b*width)/2*cos(linspace(0,pi,nPatch(q)-b)) ...
716       Q2+(1-b/2:0)*width ];

```

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

```

725 case 'usergiven'
726     if q==1, Q = reshape(Dom.X,1,[]);
727     else    Q = reshape(Dom.Y,1,[]);
728     end%if
729 end%switch Dom.type

```

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

```

736 if q==1, X=Q; end
737 if q==2, Y=Q; end
738 end%for q

```

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

```

754 xs = dx(1)*( (1:nSubP(1))-mean(1:nSubP(1)) );
755 patches.x = reshape( xs'+X ...
756                      ,nSubP(1),1,1,1,nPatch(1),1);
757 ys = dx(2)*( (1:nSubP(2))-mean(1:nSubP(2)) );
758 patches.y = reshape( ys'+Y ...
759                      ,1,nSubP(2),1,1,1,nPatch(2));

```

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

```

770 if patches.periodic
771     ratio = reshape(ratio,1,2); % force to be row vector
772     patches.ratio=ratio;
773     if ordCC>0
774         [Cwtsr,Cwtsl] = patchCwts(ratio,ordCC,patches.stag);
775         patches.Cwtsr = Cwtsr; patches.Cwtsl = Cwtsl;
776     end%if
777 end%if patches.periodic

```

8.4 Set ensemble inter-patch communication

For EdgyInt or centre interpolation respectively,

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

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

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

```

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

```

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

```

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

```

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

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

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

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

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

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

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

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

```
17
872 if prod(ratio)*patches.nEnsem>0.9, warning( ...
873 'Probably poor scale separation in ensemble of coupled phase-shifts')
874 scaleSeparationParameter = ratio*patches.nEnsem
875 end
```

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

End the two if-statements.

```
881     end%if-else nEnsem>1
882 end%if not-empty(cs)
```

If parallel code then first assume this is not within an `spmd`-environment, and so we invoke `spmd...end` (which starts a parallel pool if not already started). At this point, the global `patches` is copied for each worker processor and so it becomes *composite* when we distribute any one of the fields. Hereafter, *all fields in the global variable patches must only be referenced within an spmd-environment.*¹⁸

```
901 if patches.parallel
902 % theparpool=gcp()
903     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.

```
912     [~,pari]=max(nPatch+0.01*(1:2));
913     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.

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

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

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

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

Fin

end% function

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

To simulate in time with 2D spatial patches we often need to interface a users time derivative function with time integration routines such as `ode23` or `PIRK2`. This function provides an interface. Communicate patch-design variables ([Section 8](#)) either via the global struct `patches` or via an optional third argument. `patches` is required for the parallel computing of `spmd`, or if parameters are to be passed though to the user microscale function.

```
function dudt = patchSys2(t,u,patches,varargin)
if nargin<3, global patches, end
```

Input

- `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 $\text{nSubP}(1) \times \text{nSubP}(2) \times \text{nPatch}(1) \times \text{nPatch}(2)$ grid.
- `t` is the current time to be passed to the user's time derivative function.
- `patches` a struct set by `configPatches2()` with the following information used here.
 - `.fun` is the name of the user's function `fun(t,u,patches,...)` that computes the time derivatives on the patchy lattice. The array `u` has size $\text{nSubP}(1) \times \text{nSubP}(2) \times \text{nVars} \times \text{nEsem} \times \text{nPatch}(1) \times \text{nPatch}(2)$. Time derivatives must be computed into the same sized array, although herein the patch edge-values are overwritten by zeros.
 - `.x` is $\text{nSubP}(1) \times 1 \times 1 \times \text{lnPatch}(1) \times 1$ array of the spatial locations x_i of the microscale (i,j) -grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and micro-scales.
 - `.y` is similarly $1 \times \text{nSubP}(2) \times 1 \times 1 \times 1 \times \text{nPatch}(2)$ array of the spatial locations y_j of the microscale (i,j) -grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and micro-scales.

- **varargin**, optional, is arbitrary list of parameters to be passed onto the users time-derivative function as specified in `configPatches2`.

Output

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

Reshape the fields **u** as a 6D-array, and sets the edge values from macroscale interpolation of centre-patch values. [Section 7](#) describes `patchEdgeInt2()`.

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

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

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

Fin.

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

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

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

¹⁹Script `patchEdgeInt3test.m` verifies this code.

```

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

```

Input

- **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 $\text{nSubP1} \cdot \text{nSubP2} \cdot \text{nSubP3} \cdot \text{nPatch1} \cdot \text{nPatch2} \cdot \text{nPatch3}$ multiscale spatial grid on the $\text{nPatch1} \cdot \text{nPatch2} \cdot \text{nPatch3}$ array of patches.
- **patches** a struct set by `configPatches3()` which includes the following information.
 - **.x** is $\text{nSubP1} \times 1 \times 1 \times 1 \times 1 \times \text{nPatch1} \times 1 \times 1$ array of the spatial locations x_{iI} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index i , but may be variable spaced in macroscale index I .
 - **.y** is similarly $1 \times \text{nSubP2} \times 1 \times 1 \times 1 \times 1 \times \text{nPatch2} \times 1$ array of the spatial locations y_{jJ} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index j , but may be variable spaced in macroscale index J .
 - **.z** is similarly $1 \times 1 \times \text{nSubP3} \times 1 \times 1 \times 1 \times 1 \times \text{nPatch3}$ array of the spatial locations z_{kK} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index k , but may be variable spaced in macroscale index K .
 - **.ordCC** is order of interpolation, currently only $\{0, 2, 4, \dots\}$
 - **.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).

- `.nEdge`, three elements, the width of edge values set by interpolation at the x, y, z -face regions, respectively, of each patch (default is one all x, y, z -faces).
- `.nEnsem` the number of realisations in the ensemble.
- `.parallel` whether serial or parallel.

Output

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

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

```

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

```

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

```

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

```

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

```

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

```

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

```

170 %x = patches.x;
171 %if patches.nEdge(1)>1
172 % m = patches.nEdge(1);
173 % x = x(1:m:nx, :, :, :, :, :, :);
174 % nx = nx/m;
175 % u = reshape(u,m,nx,ny,nz,nVars,nEnsem,Nx,Ny,Nz);
176 % nVars = nVars*m;
177 % u = reshape( permute(u,[2:4 1 5:9]) ...
178 %             ,nx,ny,nz,nVars,nEnsem,Nx,Ny,Nz);
179 %end%if patches.nEdge(1)
180 %y = patches.y;
181 %if patches.nEdge(2)>1
182 % m = patches.nEdge(2);
183 % y = y(:,1:m:ny, :, :, :, :, :);
184 % ny = ny/m;
185 % u = reshape(u,nx,m,ny,nz,nVars,nEnsem,Nx,Ny,Nz);
186 % nVars = nVars*m;
187 % u = reshape( permute(u,[1 3:4 2 5:9]) ...
188 %             ,nx,ny,nz,nVars,nEnsem,Nx,Ny,Nz);
189 %end%if patches.nEdge(2)
190 %z = patches.z;
191 %if patches.nEdge(3)>1
192 % m = patches.nEdge(3);
193 % z = z(:, :, 1:m:nz, :, :, :, :);
194 % nz = nz/m;
195 % u = reshape(u,nx,ny,m,nz,nVars,nEnsem,Nx,Ny,Nz);
196 % nVars = nVars*m;
197 % u = reshape( permute(u,[1:2 4 3 5:9]) ...
198 %             ,nx,ny,nz,nVars,nEnsem,Nx,Ny,Nz);
199 %end%if patches.nEdge(3)
200 x = patches.x;
201 y = patches.y;
202 z = patches.z;
203 if mean(patches.nEdge)>1
204     mx = patches.nEdge(1);

```

```

205     my = patches.nEdge(2);
206     mz = patches.nEdge(3);
207     x = x(1:mx:nx, :, :, :, :, :, :);
208     y = y(:, 1:my:ny, :, :, :, :, :);
209     z = z(:, :, 1:mz:nz, :, :, :, :);
210     nx = nx/mx;
211     ny = ny/my;
212     nz = nz/mz;
213     u = reshape(u, mx, nx, my, ny, mz, nz, nVars, nEnsem, Nx, Ny, Nz);
214     nVars = nVars*mx*my*mz;
215     u = reshape( permute(u, [2:2:6 1:2:5 7:11]) ...
216                 , nx, ny, nz, nVars, nEnsem, Nx, Ny, Nz);
217 end%if patches.nEdge

```

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

```

226     i0 = round((nx+1)/2);
227     j0 = round((ny+1)/2);
228     k0 = round((nz+1)/2);

```

10.1 Periodic macroscale interpolation schemes

```

237 if patches.periodic

```

Get the size ratios of the patches in each direction.

```

243 rx = patches.ratio(1);
244 ry = patches.ratio(2);
245 rz = patches.ratio(3);

```

10.1.1 Lagrange interpolation gives patch-face values

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

```

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

```

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

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

```

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

```

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

```

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

```

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

```

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

```

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

```

303     if patches.stag % use only odd numbered neighbours
304         error('polynomial interpolation not yet for staggered patch coupl
305 %     dmux(:,:,:,I,::,1) = (Ux(:,:,:,Ip,::) +Ux(:,:,:,Im,::))/2;
306 %     dmux(:,:,:,I,::,2) = (Ux(:,:,:,Ip,::) -Ux(:,:,:,Im,::))/2;
307 %     Ip = Ip(Ip); Im = Im(Im); % increase shifts to \pm2
308 %     dmuy(:,:,:,J,::,1) = (Ux(:,:,:,Jp,::)+Ux(:,:,:,Jm,::))/2;
309 %     dmuy(:,:,:,J,::,2) = (Ux(:,:,:,Jp,::)-Ux(:,:,:,Jm,::))/2;
310 %     Jp = Jp(Jp); Jm = Jm(Jm); % increase shifts to \pm2
311 %     dmuz(:,:,:,K,1) = (Ux(:,:,:,Kp)+Ux(:,:,:,Km))/2;
312 %     dmuz(:,:,:,K,2) = (Ux(:,:,:,Kp)-Ux(:,:,:,Km))/2;
313 %     Kp = Kp(Kp); Km = Km(Km); % increase shifts to \pm2
314     else %disp('starting standard interpolation')
315         dmu(:,:,:,I,::,1) = (U(:,:,:,Ip,::) ...
316                             -U(:,:,:,Im,::))/2; %\mu\delta
317         dmu(:,:,:,I,::,2) = (U(:,:,:,Ip,::) ...
318                             -2*U(:,:,:,I,::) +U(:,:,:,Im,::)); %\delta^2
319     end% if stag

```

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

```

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

```

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

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

```

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

```

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

```

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

```

Adjoin extra dimension for the array of differences.

```

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

```

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

```

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

```


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

```

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

```

Recursively take δ^2 .

```

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

```

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

```

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

```

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

```

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

```

Adjoin extra dimension for the array of differences.

```

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

```

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

```

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

```

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

```

454     if patches.stag % use only odd numbered neighbours
455         error('polynomial interpolation not yet for staggered patch coupling')
456     else %disp('starting standard interpolation')
457         dmu(:,:,,:,K,1) = (U(:,:,,:,Kp) ...
458                             -U(:,:,,:,Km))/2; %\mu\delta
459         dmu(:,:,,:,K,2) = (U(:,:,,:,Kp) ...
460                             -2*U(:,:,,:,K) +U(:,:,,:,Km)); %\delta^2
461     end% if stag

```

Recursively take δ^2 .

```

467     for k = 3:ordCC
468         dmu(:,:,,:,K,k) = dmu(:,:,,:,Kp,k-2) ...
469                             -2*dmu(:,:,,:,K,k-2) +dmu(:,:,,:,Km,k-2);
470     end

```

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

```

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

```

10.1.2 Case of spectral interpolation

Assumes the domain is macro-periodic.

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

y-normal face values Set wavenumbers into a vector.

```

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

```

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

```

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

```

Invert the Fourier transforms to complete interpolation.

```

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

```

z-normal face values Set wavenumbers into a vector.

```

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

```

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

```

634 if ~patches.EdgeyInt
635     Cm = fft( u(:,:,k0,:,:,:,:,:) ,[],8);
636     Cp = Cm;
637 else
638     Cm = fft( u(:,:,2 ,: ,patches.ba,:,:,:) ,[],8);
639     Cp = fft( u(:,:,nz-1 ,: ,patches.fr,:,:,:) ,[],8);
640 end%if ~patches.EdgeyInt

```

Invert the Fourier transforms to complete interpolation.

```

646 u(:,:,nz,:,:,:,:,:) = uclean( ifft( ...
647     Cm.*exp(1i*(stagShift+kr)) ,[],8) );
648 u(:,:, 1,:,:,:,:,:) = uclean( ifft( ...
649     Cp.*exp(1i*(stagShift-kr)) ,[],8) );
655 end% if ordCC>0

```

10.2 Non-periodic macroscale interpolation

```

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

```

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

```

682 if patches.ordCC<1
683     px = Nx-1; py = Ny-1; pz = Nz-1;
684 else px = min(patches.ordCC,Nx-1);
685     py = min(patches.ordCC,Ny-1);
686     pz = min(patches.ordCC,Nz-1);
687 end
688 % interior indices of faces (ix n/a)
689 ix=2:nx-1; iy=2:ny-1; iz=2:nz-1;

```

10.2.1 *x*-direction values

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

```

702 F = nan(patches.EdgyInt+1,ny-2,nz-2,nVars,nEnsem,Nx,Ny,Nz,px+1);
703 if patches.EdgyInt % interpolate next-to-face values
704     F(:, :, :, :, :, :, :, 1) = u([nx-1 2],iy,iz, :, :, :, :);
705     X = x([nx-1 2], :, :, :, :, :, :);
706 else % interpolate mid-patch cross-patch values
707     F(:, :, :, :, :, :, :, 1) = u(i0,iy,iz, :, :, :, :);
708     X = x(i0, :, :, :, :, :, :);
709 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.

```

722 for q = 1:px
723     i = 1:Nx-q;
724     F(:, :, :, :, :, i, :, :, q+1) ...
725     = ( F(:, :, :, :, :, i+1, :, :, q)-F(:, :, :, :, :, i, :, :, q)) ...
726     ./ (X(:, :, :, :, :, i+q, :, :)-X(:, :, :, :, :, i, :, :));
727 end

```

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

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

```

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

```

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

```

747 i = max(1,min(1:Nx,Nx-ceil(px/2))-floor(px/2));
748 Uface = F(:,:,:,:,i,:,:px+1);
749 for q = px:-1:1
750     Uface = F(:,:,:,:,i,:,:q) ...
751     +(Xface-X(:,:,:,:,i+q-1,:,:)).*Uface;
752 end

```

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

```

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

```

10.2.2 y -direction values

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

```

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

```

Form tables of divided differences.

```

784 for q = 1:py
785     j = 1:Ny-q;
786     F(:,:,:,:,j,:q+1) ...
787     = ( F(:,:,:,:,j+1,:q)-F(:,:,:,:,j,:q)) ...
788     ./ (Y(:,:,:,:,j+q,:) -Y(:,:,:,:,j,:));
789 end

```

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

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

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

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

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

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

10.2.3 z -direction values

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

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

Form tables of divided differences.

```
842 for q = 1:pz
843     k = 1:Nz-q;
844     F(:, :, :, :, :, k, q+1) ...
845     = ( F(:, :, :, :, :, k+1, q)-F(:, :, :, :, :, k, q)) ...
846     ./ (Z(:, :, :, :, :, k+q) -Z(:, :, :, :, :, k));
847 end
```


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

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

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

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

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

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

10.2.4 Optional NaNs for safety

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

```
889 if isfield(patches, 'intTest') && patches.intTest
890     else % usual case
891         u( 1, :, :, :, :, 1, :, :) = nan;
892         u(nx, :, :, :, :, Nx, :, :) = nan;
893         u(:, 1, :, :, :, :, 1, :) = nan;
894         u(:, ny, :, :, :, :, Ny, :) = nan;
895         u(:, :, 1, :, :, :, :, 1) = nan;
896         u(:, :, nz, :, :, :, :, Nz) = nan;
897     end%if
```

End of the non-periodic interpolation code.

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

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

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

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

927 end% function patchEdgeInt3
```

11 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 11.1](#) and ?? list examples of its use.

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

Input If invoked with no input arguments, then executes an example of simulating a heterogeneous wave PDE—see [Section 11.1](#) for an example code.

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)` or `fun(t,u,patches,...)`, that computes time-derivatives (or time-steps) of quantities on the 3D micro-grid within all the 3D patches.
- `Xlim` array/vector giving the rectangular-cuboid macro-space domain of the computation: namely $[Xlim(1),Xlim(2)] \times [Xlim(3),Xlim(4)] \times [Xlim(5),Xlim(6)]$. If `Xlim` has two elements, then the domain is the cubic domain of the same interval in all three directions.
- `Dom` sets the type of macroscale conditions for the patches, and reflects the type of microscale boundary conditions of the problem. If `Dom` is

NaN or [], then the field `u` is triply macro-periodic in the 3D spatial domain, and resolved on equi-spaced patches. If `Dom` is a character string, then that specifies the `.type` of the following structure, with `.bcOffset` set to the default zero. Otherwise `Dom` is a structure with the following components.

- `.type`, string, of either `'periodic'` (the default), `'equispace'`, `'chebyshev'`, `'usergiven'`. For all cases except `'periodic'`, users *must* code into `fun` the micro-grid boundary conditions that apply at the left/right/bottom/top/back/front faces of the leftmost/rightmost/bottommost/topmost/backmost/frontmost patches, respectively.
- `.bcOffset`, optional one, three or six element vector/array, in the cases of `'equispace'` or `'chebyshev'` the patches are placed so the left/right macroscale boundaries are aligned to the left/right faces of the corresponding extreme patches, but offset by `bcOffset` of the sub-patch micro-grid spacing. For example, use `bcOffset=0` when the micro-code applies Dirichlet boundary values on the extreme face micro-grid points, whereas use `bcOffset=0.5` when the microcode applies Neumann boundary conditions halfway between the extreme face micro-grid points. Similarly for the top, bottom, back, and front faces.

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

- `.X`, optional vector/array with `nPatch(1)` elements, in the case `'usergiven'` it specifies the x -locations of the centres of the patches—the user is responsible the locations makes sense.
- `.Y`, optional vector/array with `nPatch(2)` elements, in the case `'usergiven'` it specifies the y -locations of the centres of the patches—the user is responsible the locations makes sense.
- `.Z`, optional vector/array with `nPatch(3)` elements, in the case `'usergiven'` it specifies the z -locations of the centres of the patches—the user is responsible the locations makes sense.

- **nPatch** sets the number of equi-spaced spatial patches: if scalar, then use the same number of patches in all three directions, otherwise **nPatch(1:3)** gives the number (≥ 1) of patches in each direction.
- **ordCC** is the ‘order’ of interpolation for inter-patch coupling across empty space of the macroscale patch values to the face-values of the patches: currently must be 0, 2, 4, ...; where 0 gives spectral interpolation.
- **dx** (real—scalar or three elements) is usually the sub-patch micro-grid spacing in x , y and z . If scalar, then use the same **dx** in all three directions, otherwise **dx(1:3)** gives the spacing in each of the three directions.

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

- **nSubP** is the number of equi-spaced microscale lattice points in each patch: if scalar, then use the same number in all three directions, otherwise **nSubP(1:3)** gives the number in each direction. If not using **EdgyInt**, then **nSubP./nEdge** must be odd integer(s) so that there is/are centre-patch lattice planes. So for the defaults of **nEdge** = 1 and not **EdgyInt**, then **nSubP** must be odd.
- **'nEdge'**, *optional* (integer—scalar or three element), default=1, the width of face values set by interpolation at the face regions of each patch. If two elements, then respectively the width in x, y -directions. The default is one (suitable for microscale lattices with only nearest neighbour interactions).
- **'EdgyInt'**, true/false, *optional*, default=false. If true, then interpolate to left/right/top/bottom/front/back face-values from right/left/bottom/top/back/front next-to-face values. If false or omitted, then interpolate from centre-patch planes.
- **'nEnsem'**, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.

- **'hetCoeffs'**, *optional*, default empty. Supply a 3D or 4D array of microscale heterogeneous coefficients to be used by the given microscale **fun** in each patch. Say the given array **cs** is of size $m_x \times m_y \times m_z \times n_c$, where n_c is the number of different arrays of coefficients. For example, in heterogeneous diffusion, $n_c = 3$ for the diffusivities in the *three* different spatial directions (or $n_c = 6$ for the diffusivity tensor). The coefficients are to be the same for each and every patch. However, macroscale variations are catered for by the n_c coefficients being n_c parameters in some macroscale formula.
 - If **nEnsem** = 1, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the (1, 1, 1)-point in each patch. Best accuracy usually obtained when the periodicity of the coefficients is a factor of **nSubP-2*nEdge** for **EdgyInt**, or a factor of **(nSubP-nEdge)/2** for not **EdgyInt**.
 - If **nEnsem** > 1 (value immaterial), then reset **nEnsem** := $m_x \cdot m_y \cdot m_z$ and construct an ensemble of all $m_x \cdot m_y \cdot m_z$ phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When **EdgyInt** is true, and when the coefficients are diffusivities/elasticities in x, y, z -directions, respectively, then this coupling cunningly preserves symmetry.
- **'parallel'**, true/false, *optional*, default=false. If false, then all patch computations are on the user's main CPU—although a user may well separately invoke, say, a GPU to accelerate sub-patch computations.

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

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

as a global variable.²¹

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

- `.fun` is the name of the user's function `fun(t,u,patches)` or `fun(t,u)` or `fun(t,u,patches,...)` that computes the time derivatives (or steps) on the patchy lattice.
- `.ordCC` is the specified order of inter-patch coupling.
- `.periodic`: either true, for interpolation on the macro-periodic domain; or false, for general interpolation by divided differences over non-periodic domain or unevenly distributed patches.
- `.stag` is true for interpolation using only odd neighbouring patches as for staggered grids, and false for the usual case of all neighbour coupling—not yet implemented.
- `.Cwtsr` and `.Cwtsl` are the `ordCC × 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 $\text{nSubP}(1) \times 1 \times 1 \times 1 \times 1 \times \text{nPatch}(1) \times 1 \times 1$ array of the regular spatial locations x_{iI} of the microscale grid points in every patch.
- `.y` (8D) is $1 \times \text{nSubP}(2) \times 1 \times 1 \times 1 \times 1 \times \text{nPatch}(2) \times 1$ array of the regular spatial locations y_{jJ} of the microscale grid points in every patch.
- `.z` (8D) is $1 \times 1 \times \text{nSubP}(3) \times 1 \times 1 \times 1 \times 1 \times \text{nPatch}(3)$ array of the regular spatial locations z_{kK} of the microscale grid points in every patch.
- `.ratio` 1×3 , only for macro-periodic conditions, are the size ratios of every patch.
- `.nEdge` 1×3 , is the width of face values set by interpolation at the face regions of each patch, in the x, y, z -directions respectively.
- `.le`, `.ri`, `.bo`, `.to`, `.ba`, `.fr` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.

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

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

11.1 If no arguments, then execute an example

```

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

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

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

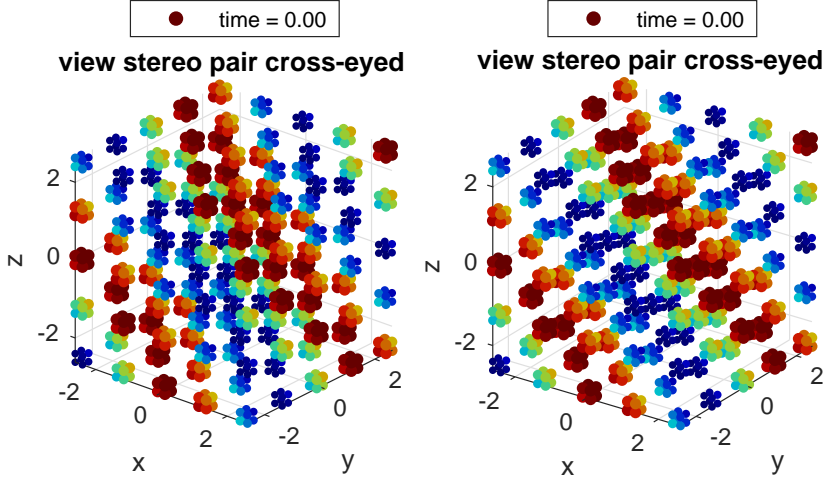
Set random heterogeneous coefficients of period two in each of the three directions. Crudely normalise by the harmonic mean so the macro-wave time scale is roughly one.

```

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

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

Figure 12: initial field $u(x, y, z, t)$ at time $t = 0$ of the patch scheme applied to a heterogeneous wave PDE: Figure 13 plots the computed field at time $t = 6$.



```

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

```

Set a wave initial state using auto-replication of the spatial grid, and as Figure 12 shows. This wave propagates diagonally across space. Concatenate the two u, v -fields to be the two components of the fourth dimension.

```

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

```

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

```

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

```



```

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

```

Animate the computed simulation to end with [Figure 13](#). Use `patchEdgeInt3` to obtain patch-face values in order to most easily reconstruct the array data structure.

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

```

401 figure(1), clf, colormap(0.8*jet)
402 xs = patches.x+0*patches.y+0*patches.z;
403 ys = patches.y+0*patches.x+0*patches.z;
404 zs = patches.z+0*patches.y+0*patches.x;
405 if 1, xs([1 end],:,:)=nan;
406     xs(:,[1 end],:,:)=nan;
407     xs(:,:,[1 end],:)=nan;
408 end;%option
409 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.

```

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

```

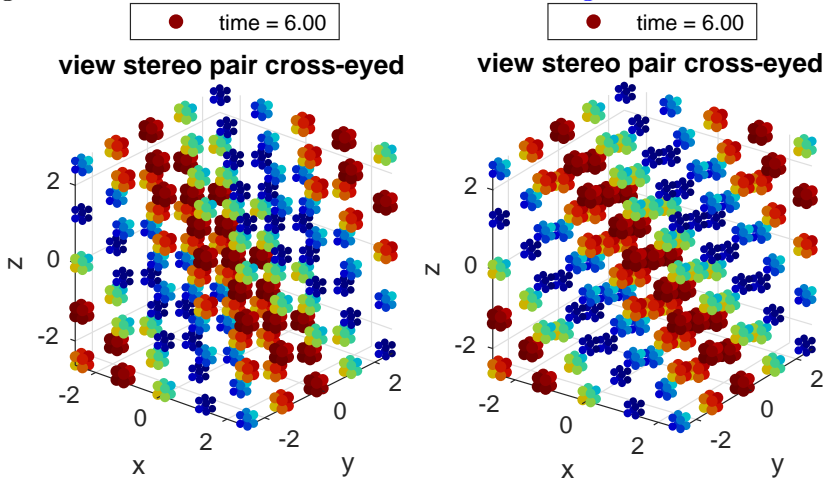
Loop to plot at each and every time step.

```

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

```

Figure 13: 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 12](#).



```

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

```

Optionally save the initial condition to graphic file for [Figure 10](#), and optionally save the last plot.

```

446 if i==1,
447     ifOurCf2eps([mfilename 'ic'])
448     disp('Type space character to animate simulation')
449     pause
450 else pause(0.05)
451 end
452 end% i-loop over all times
453 ifOurCf2eps([mfilename 'fin'])

```

Upon finishing execution of the example, exit this function.

```

468 return
469 end%if no arguments

```

11.2 Parse input arguments and defaults

```

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

```

```

488 addRequired(p,'fun',fnValidation);
489 addRequired(p,'Xlim',@isnumeric);
490 %addRequired(p,'Dom'); % too flexible
491 addRequired(p,'nPatch',@isnumeric);
492 addRequired(p,'ordCC',@isnumeric);
493 addRequired(p,'dx',@isnumeric);
494 addRequired(p,'nSubP',@isnumeric);
495 addParameter(p,'nEdge',1,@isnumeric);
496 addParameter(p,'EdgyInt',false,@islogical);
497 addParameter(p,'nEnsem',1,@isnumeric);
498 addParameter(p,'hetCoeffs',[],@isnumeric);
499 addParameter(p,'parallel',false,@islogical);
500 %addParameter(p,'nCore',1,@isnumeric); % not yet implemented
501 parse(p,fun,Xlim,nPatch,ordCC,dx,nSubP,varargin{:});

```

Set the optional parameters.

```

507 patches.nEdge = p.Results.nEdge;
508 if numel(patches.nEdge)==1
509     patches.nEdge = repmat(patches.nEdge,1,3);
510 end
511 patches.EdgyInt = p.Results.EdgyInt;
512 patches.nEnsem = p.Results.nEnsem;
513 cs = p.Results.hetCoeffs;
514 patches.parallel = p.Results.parallel;
515 %patches.nCore = p.Results.nCore;

```

Initially duplicate parameters for three space dimensions as needed.

```

523 if numel(Xlim)==2,    Xlim = repmat(Xlim,1,3); end
524 if numel(nPatch)==1, nPatch = repmat(nPatch,1,3); end
525 if numel(dx)==1,     dx = repmat(dx,1,3); end
526 if numel(nSubP)==1,  nSubP = repmat(nSubP,1,3); end

```

Check parameters.

```

533 assert(Xlim(1)<Xlim(2) ...
534         , 'first pair of Xlim must be ordered increasing')
535 assert(Xlim(3)<Xlim(4) ...
536         , 'second pair of Xlim must be ordered increasing')
537 assert(Xlim(5)<Xlim(6) ...

```

```

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

```

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

```

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

```

Default macroscale conditions are periodic with evenly spaced patches.

```

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

```

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

```

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

```

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

```

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

```

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

```

605 patches.periodic=false;
606 for p=1:3
607     switch Dom.type(p,:)
608     case 'periodic'
609         patches.periodic=true;
610         if isfield(Dom,'bcOffset')
611             warning('bcOffset not available for Dom.type = periodic'), end
612             msg=' not available for Dom.type = periodic';
613             if isfield(Dom,'X'), warning(['X' msg]), end
614             if isfield(Dom,'Y'), warning(['Y' msg]), end
615             if isfield(Dom,'Z'), warning(['Z' msg]), end
616         case {'equispace','chebyshev'}
617             if ~isfield(Dom,'bcOffset'), Dom.bcOffset=zeros(2,3); end
618             % for mixed with usergiven, following should still work
619             if numel(Dom.bcOffset)==1
620                 Dom.bcOffset= repmat(Dom.bcOffset,2,3); end
621             if numel(Dom.bcOffset)==3
622                 Dom.bcOffset= repmat(Dom.bcOffset(:)',2,1); end
623             msg=' not available for Dom.type = equispace or chebyshev';
624             if (p==1)& isfield(Dom,'X'), warning(['X' msg]), end
625             if (p==2)& isfield(Dom,'Y'), warning(['Y' msg]), end
626             if (p==3)& isfield(Dom,'Z'), warning(['Z' msg]), end
627         case 'usergiven'
628             % if isfield(Dom,'bcOffset')
629             % warning('bcOffset not available for usergiven Dom.type'), end
630             msg=' required for Dom.type = usergiven';
631             if p==1, assert(isfield(Dom,'X'),['X' msg]), end
632             if p==2, assert(isfield(Dom,'Y'),['Y' msg]), end
633             if p==3, assert(isfield(Dom,'Z'),['Z' msg]), end
634         otherwise
635             error(['Dom.type ' is unknown Dom.type'])
636     end%switch Dom.type
637 end%for p

```

11.3 The code to make patches

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

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

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

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

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

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

Check for staggered grid and periodic case.

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

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

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

Distribution depends upon `Dom.type`:

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

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

```

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

```

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

```

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

```

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

```

752 case 'chebyshev'
753     halfWidth=dx(q)*(nSubP(q)-1)/2;
754     Q1 = Xlim(1)+halfWidth-Dom.bcOffset(qq)*dx(q);
755     Q2 = Xlim(2)-halfWidth+Dom.bcOffset(qq+1)*dx(q);
756 %   Q = (Q1+Q2)/2-(Q2-Q1)/2*cos(linspace(0,pi,nPatch));

```

Search for total width of ‘boundary layers’ so that in the interior the patches are non-overlapping Chebyshev. But the width for assessing overlap of patches is the following variable `width`.

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

```

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

```

Assign the centre-patch coordinates.

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```



```

824 ys = dx(2)*( (1:nSubP(2))-mean(1:nSubP(2)) );
825 patches.y = reshape( ys'+Y ...
826                     ,1,nSubP(2),1,1,1,1,nPatch(2),1);
827 zs = dx(3)*( (1:nSubP(3))-mean(1:nSubP(3)) );
828 patches.z = reshape( zs'+Z ...
829                     ,1,1,nSubP(3),1,1,1,1,nPatch(3));

```

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

```

841 if patches.periodic
842     ratio = reshape(ratio,1,3); % force to be row vector
843     patches.ratio = ratio;
844     if ordCC>0
845         [Cwtsr,Cwtsl] = patchCwts(ratio,ordCC,patches.stag);
846         patches.Cwtsr = Cwtsr; patches.Cwtsl = Cwtsl;
847     end%if
848 end%if patches.periodic

```

11.4 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()`.

```

877 nE = patches.nEnsem;
878 patches.le = 1:nE; patches.ri = 1:nE;
879 patches.bo = 1:nE; patches.to = 1:nE;
880 patches.ba = 1:nE; patches.fr = 1:nE;

```

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

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

```

938 ri = mod(mmx-mod(nx-2,mx),mx)+1;
939 patches.ri = reshape( ri+mx*(mmy+my*mmz) , [],1);
940 bo = mod(mmy+mod(ny-2,my),my)+1;
941 patches.bo = reshape( 1+mmx+mx*(bo-1+my*mmz) , [],1);
942 to = mod(mmy+mod(ny-2,my),my)+1;
943 patches.to = reshape( 1+mmx+mx*(to-1+my*mmz) , [],1);
944 ba = mod(mmz+mod(nz-2,mz),mz)+1;
945 patches.ba = reshape( 1+mmx+mx*(mmy+my*(ba-1)) , [],1);
946 fr = mod(mmz+mod(nz-2,mz),mz)+1;
947 patches.fr = reshape( 1+mmx+mx*(mmy+my*(fr-1)) , [],1);

```

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

```

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

```

End the two if-statements.

```

964 end%if-else nEnsem>1
965 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](#)}

```

984 if patches.parallel
985     spmd

```

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

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

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

```

994     [~,pari]=max(nPatch+0.01*(1:3));
995     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.

```

1005     switch pari
1006     case 1, patches.x=codistributed(patches.x,patches.codist);
1007     case 2, patches.y=codistributed(patches.y,patches.codist);
1008     case 3, patches.z=codistributed(patches.z,patches.codist);
1009     otherwise
1010         error('should never have bad index for parallel distribution')
1011     end%switch
1012     end%spmd

```

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

```

1020 else% not parallel
1021     if isfield(patches,'codist'), rmfield(patches,'codist'); end
1022 end%if-parallel

```

Fin

```

1031 end% function

```

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

To simulate in time with 3D spatial patches we often need to interface a users time derivative function with time integration routines such as `ode23` or `PIRK2`. This function provides an interface. Communicate patch-design variables ([Section 11](#)) either via the global struct `patches` or via an optional third argument. `patches` is required for the parallel computing of `spmd`, or if parameters are to be passed though to the user microscale function.

```

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

```

Input

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

Output

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

Sets the edge-face values from macroscale interpolation of centre-patch values, and if necessary, reshapes the fields `u` as a 8D-array. [Section 10](#) describes `patchEdgeInt3()`.

```

104 sizeu = size(u);
105 u = patchEdgeInt3(u,patches);

```

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

```

116 dudt = patches.fun(t,u,patches,varargin{:});
117 m = patches.nEdge(1);
118 dudt([1:m end-m+1:end],:,:,:) = 0;
119 m = patches.nEdge(2);
120 dudt(:,[1:m end-m+1:end],:,:) = 0;
121 m = patches.nEdge(3);
122 dudt(:,:,[1:m end-m+1:end],:) = 0;
123 dudt = reshape(dudt,sizeu);

```

Fin.

13 New interpolation tests

13.1 patchEdgeInt1test: test the 1D patch coupling

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

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

```

22 clear all, close all
23 global patches
24 nRealise = 20

```

13.1.1 Check divided difference interpolation

```

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

```

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

```

47 for iReal=1:nRealise
48     nEdge = randi(3)% =1,2, or 3
49     edgyInt = rand<0.5
50     nSubP = nEdge*( (2-edgyInt)*randi(2)+1+edgyInt )
51     ordCC = 2*randi(4)
52     nPatch = ordCC+randi([2 4])
53     Domain=5*[-rand rand]
54     dx=rand*diff(Domain)/nPatch/nSubP
55     configPatches1(@sin,Domain,'equispace',nPatch,ordCC,dx,nSubP ...
56         , 'EdgyInt',edgyInt,'nEdge',nEdge);
57     patches.intTest = true;

```

Displace patches to a random non-uniform spacing.

```

63 H = diff(patches.x(1,:,: ,1:2));
64 patches.x = patches.x+0.8*H*(rand(1,1,1,nPatch)-0.5);
65 %H = squeeze( diff(patches.x(1,:,: ,:)) )% for information only

```

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

```

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

```

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

```

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

```

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

```

92     ui=patchEdgeInt1(u(:));
93     ui=squeeze(ui);

```

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

```

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

```

End the for-loop over random parameters.

```

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

```

13.1.2 Test standard spectral interpolation

```

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

```

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

```

142 for iReal=1:nRealise
143     nEdge=randi(3)% =1,2, or 3
144     edgyInt = rand<0.5
145     nSubP = nEdge*( (2-edgyInt)*randi(2)+1+edgyInt )
146     nPatch=randi([5 10])
147     Len=10*rand
148     dx=0.5*rand*Len/nPatch/nSubP
149     configPatches1(@sin,[0 Len], 'periodic', nPatch, 0, dx, nSubP ...
150     , 'EdgyInt', edgyInt, 'nEdge', nEdge); % random Edgy or not
151     if mod(nPatch,2)==0, fprintf('\nAvoiding highest wavenumber\n'),
152     kMax=floor((nPatch-1)/2);

```

Test single field Set a profile, and evaluate the interpolation.

```

160 for k=-kMax:kMax
161     u0=exp(1i*k*patches.x*2*pi/Len);
162     u=u0; u([1:nEdge end-nEdge+1:end],:)=nan;
163     ui=patchEdgeInt1(u(:));
164     normError=rms(ui(:)-u0(:));
165     if abs(normError)>5e-14

```



```

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

```

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

```

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

```

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

```

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

```

End the for-loop over various geometries.

```

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

```

13.1.3 Now test spectral interpolation on staggered grid

```

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

```

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

```

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

```

Identify which microscale grid points are h or u values.

```

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

```

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

```

257 fprintf('Staggered: single field-pair test.\n')
258 for k=-kMax:kMax
259     U0=nan(nSubP,nPatch);
260     U0(hPts)=rand*exp(+1i*k*patches.x(hPts)*2*pi/Len);
261     U0(uPts)=rand*exp(-1i*k*patches.x(uPts)*2*pi/Len);
262     U=U0;
263     U([1:nEdge end-nEdge+1:end],:)=nan;
264     Ui=patchEdgeInt1(U0(:));
265     normError=norm(Ui(:)-U0(:));
266     if abs(normError)>5e-14
267         normError=normError
268         patches=patches
269         error(['staggered: failed single sys interpolation k=' num2str(k)
270             end
271         end

```

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

```

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

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

293 fprintf('Staggered: Two field-pairs test.\n')
294 x0=patches.x((nSubP+1)/2,1);
295 patches.x=patches.x-x0;
296 oddP=1:2:nPatch; evnP=2:2:nPatch;
297 for k=1:kMax
298     U0=nan(nSubP,1,1,nPatch); V0=U0;
299     U0(hPts)=rand*sin(k*patches.x(hPts)*2*pi/Len);
300     U0(uPts)=rand*sin(k*patches.x(uPts)*2*pi/Len);
301     U=U0; U([1:nEdge end-nEdge+1:end],:)=nan;
302     V0(hPts)=rand*cos(k*patches.x(hPts)*2*pi/Len);
303     V0(uPts)=rand*cos(k*patches.x(uPts)*2*pi/Len);
304     V=V0; V([1:nEdge end-nEdge+1:end],:)=nan;
305     UVi=patchEdgeInt1([U0 V0]);
306     normuError=[normDiff(UVi(:,1,:,oddP),U0(:,:,:oddP),U0(:,:,:evnP))
307                 normDiff(UVi(:,1,:,evnP),U0(:,:,:evnP),U0(:,:,:oddP))]' ;
308     normvError=[normDiff(UVi(:,2,:,oddP),V0(:,:,:oddP),V0(:,:,:evnP))
309                 normDiff(UVi(:,2,:,evnP),V0(:,:,:evnP),V0(:,:,:oddP))]' ;
310     if norm(normuError)+norm(normvError)>2e-13
311         normuError=normuError, normvError=normvError
312         patches=patches
313         error(['staggered: failed double field interpolation k=' num2str(
314             k)
315         end
316     end

End for-loop over patches

```

13.1.4 Check standard finite width interpolation

```

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

```

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

```

345 for iReal=1:nRealise
346     nEdge=randi(3)% =1,2, or 3
347     edgyInt = rand<0.5
348     nSubP = nEdge*( (2-edgyInt)*randi(2)+1+edgyInt )
349     ordCC = 2*randi(4)
350     nPatch = ordCC+randi([2 4])
351     Domain=5*[-rand rand]
352     dx=0.5*rand*diff(Domain)/nPatch/nSubP
353     configPatches1(@sin,Domain,'periodic',nPatch,ordCC,dx,nSubP ...
354         , 'EdgyInt',edgyInt,'nEdge',nEdge);

```

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

```

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

```

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

```

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

```

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

```

379     ui=patchEdgeInt1(u(:));
380     ui=squeeze(ui);

```

The interior patches should have zero error.

```

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

```

End the for-loops over various parameters.

```

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

```

13.1.5 Now test finite width interpolation on staggered grid

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

Must have even number of patches for a staggered grid.

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

Identify which microscale grid points are h or u values.

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

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

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

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

```

469 i0=(nSubP+1)/2; % centre-patch index
470 fprintf('Two field-pairs test.\n')
471 x0=patches.x((nSubP+1)/2,1);
472 patches.x=patches.x-x0;
473 for k=1:nPatch/4
474     U0=nan(nSubP,1,1,nPatch); V0=U0;
475     U0(hPts)=rand*sin(k*patches.x(hPts)*2*pi/Len);
476     U0(uPts)=rand*sin(k*patches.x(uPts)*2*pi/Len);
477     V0(hPts)=rand*cos(k*patches.x(hPts)*2*pi/Len);
478     V0(uPts)=rand*cos(k*patches.x(uPts)*2*pi/Len);
479     UVi=patchEdgeInt1([U0 V0]);
480     Ui=squeeze(UVi(:,1,1,:));
481     Vi=squeeze(UVi(:,2,1,:));
482     normuError=norm(Ui(:,1:2:nPatch)-U0(:,1:2:nPatch))*norm(U0(i0,2:2:nPatch))
483         +norm(Ui(:,2:2:nPatch)-U0(:,2:2:nPatch))*norm(U0(i0,1:2:nPatch));
484     normvError=norm(Vi(:,1:2:nPatch)-V0(:,1:2:nPatch))*norm(V0(i0,2:2:nPatch))
485         +norm(Vi(:,2:2:nPatch)-V0(:,2:2:nPatch))*norm(V0(i0,1:2:nPatch));
486     if abs(normuError)+abs(normvError)>2e-13
487         normuError=normuError, normvError=normvError
488         error(['failed double field interpolation k=' num2str(k)])
489     end
490 end

    End for-loop over the realisations
497 end

```

13.1.6 Finish

If no error messages, then all OK.

```

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

```

13.2 patchEdgeInt2test: tests 2D patch coupling

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

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

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

```
21 clear all, close all
22 global patches
23 nRealise = 20
```

13.2.1 Check divided difference interpolation

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

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

```
46 maxErrors=[];
47 for realisation = 1:nRealise
48     nEdge = randi(3,1,2)% =1,2, or 3
49     edgyInt = (rand>0.5)
50     Lx = 1+3*rand; Ly = 1+3*rand;
51     xyLim = [0 Lx 0 Ly]-[rand*[1 1] rand*[1 1]]
52     nSubP = nEdge.*( (2-edgyInt)*randi(3,1,2)+1+edgyInt )
53     ordCC = 2*randi(4)
54     nPatch = ordCC+randi(4,1,2)
55     dx = [Lx Ly]./nPatch./nSubP.*rand(1,2)/2
56     configPatches2(@sin,xyLim,'equispace',nPatch,ordCC ...
57         ,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);
```

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

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

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

```

81     ox=min(ordCC,nPatch(1)-1);
82     oy=min(ordCC,nPatch(2)-1);
83     [ps,qs]=ndgrid(0:ox,0:oy);
84     ps=reshape(ps,1,1,[]);
85     qs=reshape(qs,1,1,[]);
86     cs=2*rand(size(ps))-1;
87     u0=cs.*patches.x.^ps.*patches.y.^qs;

```

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

```

94     u=u0;
95     u([1:nEdge(1) end-nEdge(1)+1:end],:,:)=inf;
96     u(:,[1:nEdge(2) end-nEdge(2)+1:end],:)=inf;
97     ui=patchEdgeInt2(u(:));

```

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

```

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

```

End the for-loops over various parameters.

```

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

```


13.2.2 Test standard spectral interpolation

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

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

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

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

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

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

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

Copy and nan the edges, then interpolate

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

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

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

End the for-loop over realisations

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

13.2.3 Check polynomial finite width interpolation

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

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

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

```

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

```

Then evaluate the interpolation.

```

261 ui=patchEdgeInt2(u0(:));

```

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

```

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

```

End the for-loops over various parameters.

```

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

```

13.2.4 Finished

If no error messages, then all OK.

```

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

```

13.3 patchEdgeInt3test: tests 3D patch coupling

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

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

```
21 clear all, close all
22 global patches
23 nRealise = 10
```

13.3.1 Check divided difference interpolation

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

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

```
46 maxErrors=[];
47 for realisation = 1:nRealise
48     nEdge = randi(3,1,3)% =1,2, or 3
49     edgyInt = (rand>0.5)
50     Lx = 1+3*rand; Ly = 1+3*rand; Lz = 1+3*rand;
51     xyzLim = [0 Lx 0 Ly 0 Lz]-[rand*[1 1] rand*[1 1] rand*[1 1]]
52     nSubP = nEdge.*( (2-edgyInt)*randi(2,1,3)+1+edgyInt )
53     ordCC = 2*randi(3)
54     nPatch = ordCC+randi(4,1,3)
55     dx = [Lx Ly Lz]./nPatch./nSubP.*rand(1,3)/2
56     configPatches3(@sin,xyzLim,'equispace',nPatch,ordCC ...
57         ,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);
```

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

```
63 Hx = diff(patches.x(1,1,1,:,: ,1:2,1,1));
64 patches.x = patches.x+0.8*Hx*(rand(1,1,1,1,1,nPatch(1),1,1)-0.5);
65 Hx = squeeze( diff(patches.x(1,1,1,:,: ,1,1)) )';% for information o
66 Hy = diff(patches.y(1,1,1,:,: ,1,1:2,1));
67 patches.y = patches.y+0.8*Hy*(rand(1,1,1,1,1,1,nPatch(2),1)-0.5);
68 Hy = squeeze( diff(patches.y(1,1,1,:,: ,1,1)) )';% for information o
69 Hz = diff(patches.z(1,1,1,:,: ,1,1,1:2));
70 patches.z = patches.z+0.8*Hz*(rand(1,1,1,1,1,1,1,nPatch(3))-0.5);
71 Hz = squeeze( diff(patches.z(1,1,1,:,: ,1,1,1)) )';% for information o
```

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

```

85     ox=min(ordCC,nPatch(1)-1);
86     oy=min(ordCC,nPatch(2)-1);
87     oz=min(ordCC,nPatch(3)-1);
88     [ps,qs,rs]=ndgrid(0:ox,0:oy,0:oz);
89     ps=reshape(ps,1,1,1,[]);
90     qs=reshape(qs,1,1,1,[]);
91     rs=reshape(rs,1,1,1,[]);
92     cs=2*rand(size(ps))-1;
93     u0=cs.*patches.x.^ps.*patches.y.^qs.*patches.z.^rs;

```

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

```

100    u=u0;
101    u([1:nEdge(1) end-nEdge(1)+1:end],:,:)=inf;
102    u(:,[1:nEdge(2) end-nEdge(2)+1:end],:,:)=inf;
103    u(:,:[1:nEdge(3) end-nEdge(3)+1:end],:)=inf;
104    ui=patchEdgeInt3(u(:));

```

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

```

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

```

End the for-loops over various parameters.

```

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

```

13.3.2 Test standard spectral interpolation

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

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

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

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

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

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

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

Copy and nan the faces, then interpolate

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

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

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

End the for-loop over realisations

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

13.3.3 Check polynomial finite width interpolation

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

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

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

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

Then evaluate the interpolation.

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

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

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

End the for-loops over various parameters.

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

13.3.4 Finished

If no error messages, then all OK.

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


References

- Bunder, J. E., I. G. Kevrekidis, and A. J. Roberts (July 2021). “Equation-free patch scheme for efficient computational homogenisation via self-adjoint coupling”. In: *Numerische Mathematik* 149.2, pp. 229–272. DOI: [10.1007/s00211-021-01232-5](https://doi.org/10.1007/s00211-021-01232-5) (cit. on pp. [20](#), [44](#), [74](#)).
- 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. [3](#), [7](#), [11](#), [20](#), [24](#), [49](#), [80](#)).
- Bunder, J.E. et al. (2021). “Large-scale simulation of shallow water waves with computation only on small staggered patches”. In: *International Journal for Numerical Methods in Fluids* 93.4, pp. 953–977. DOI: [10.1002/flid.4915](https://doi.org/10.1002/flid.4915) (cit. on pp. [44](#), [74](#)).
- Hu, Zhenfang et al. (Apr. 2016). “Sparse Principal Component Analysis via Rotation and Truncation”. In: *IEEE Transactions on Neural Networks and Learning Systems* 27.4, pp. 875–890. ISSN: 2162-237X. DOI: [10.1109/TNNLS.2015.2427451](https://doi.org/10.1109/TNNLS.2015.2427451) (cit. on p. [15](#)).
- 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. [62](#), [96](#)).
- 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. [20](#)).
- 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. [20](#), [24](#), [49](#), [80](#)).
- 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. [44](#), [61](#), [74](#)).
- Wikipedia (2022). *Divided differences*. https://en.wikipedia.org/wiki/Divided_differences (visited on 12/28/2022) (cit. on pp. [27](#), [53](#), [86](#)).