2.29 BELGIUM CHOCOLATE NAVIER STOKES SOLVER MATLAB PACKAGE

MATT UECKERMANN

1. Introduction

This set of MATLAB packages and scripts solve advection-diffusion-reaction (ADR) equations eq. [1], the Stokes equations eq. [2] and the Navier-Stokes equations eq. [3].

$$\frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi - \kappa \nabla^2 \phi = f(\phi, \mathbf{x}, t) \tag{1}$$

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} &= \nu \nabla^2 \mathbf{u} - \nabla P \\ \nabla \cdot \mathbf{u} &= 0 \end{aligned} \tag{2}$$

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} &= \nu \nabla^2 \mathbf{u} - \nabla P^* - \rho^* \hat{k} + F(\mathbf{x}, t) \\ \nabla \cdot \mathbf{u} &= 0 \\ \frac{\partial \rho^*}{\partial t} + \mathbf{u} \cdot \nabla \rho^* - \kappa \nabla^2 \rho^* &= 0 \end{aligned} \tag{3}$$

where P^* is the dynamic pressure (hydrostatic component removed) divided by the average density ρ_0, and ρ^* is perturbation density multiplied by g.

2. Quick Start Guide

This Guide walks through how to create the SuddenExpansion and StokesForcing test cases.

2.1. **StokesForcing test case:** This is an analytical test case used for testing the convergence of various Stokes solvers. The solution is:

$$u = \pi \sin(t) \sin(2\pi y) \sin^2(\pi x) \tag{4}$$

$$v = \pi \sin(t) \sin(2\pi x) \sin^2(\pi y) \tag{5}$$

$$P = \sin(t) \cos(\pi x) \sin(\pi y) \tag{6}$$

and is solved on a square square domain of size $[-1, 1] \times [-1, 1]$.

The solver function requires 7 inputs: `Nx,Ny,Nt,nu,PlotIntrvl,SetupScript,PlotScript`. The first three inputs define the discretization, giving the number of INTERIOR points in the x-direction, y-direction, and in time. The fourth input is the value of the kinematic viscosity (which is analogous to modifying the Reynolds number for the full NS

equations). The fifth input is an integer number that indicates the number of time integrations steps between calling the script given by the string input `PlotScript`. The sixth input is the main focus of the quick-start guide, and it is a string giving the name of a MATLAB script which sets up the necessary parts of the problem to be solved. That is, the solver calls `eval(SetupScript)` to setup the problem at hand. Similarly, the solver calls `eval(PlotScript)` every `PlotIntrvl` steps of the integration. While `PlotScript` is normally used for plotting the solution, it can also be used for saving the solution. Also, any uncleared variables in `SetupScript` will be available for use in `PlotScript`. Therefore, a function handle created in `SetupScript` can be used in `PlotScript` without causing an error.

The setup file is in its entirety as follows:

```
TODO
```

2.1.1. *Setting up forcing functions.* The forcing functions can take time and space as the input variables. The spatial variable is expected to be a matrix with two columns. The first column should contain the x locations of control volume (CV) centres, and the second column should contain the y locations of CV centres. To set up the forcing functions corresponding to the solutions eq. [6] we proceed as follows:

```
TODO
```

2.1.2. *Setting up the grid.* The Stokes solvers require 3 node numbering matrices, `NodeP`, `Nodeu`, and `Nodev` for the Pressure, u-velocity, and v-velocity. A simple way to create these matrices is by providing a "masking" matrix to the `NodePad` function. Since there is no interior geometry for this test-case, we can provide a mask with all-zero values the size of the domain to the `NodePad` function as follows:

```
Mask = zeros(Ny, Nx);
%create Node matrices
[NodeP, Nodeu, Nodev, idsP, idsu, idsv] = NodePad(Mask,[1 1 1 1],1);
```

In this case there are Nx interior nodes in the x-direction, and Ny interior nodes in the y-direction. The mask is only created for the *interior* nodes. Therefore `NodePad` "pads" the masking matrix to add the four boundaries conditions on each edge. The second input to `NodePad` is a logical array, and tells the function which of the boundaries to pad ([left, right, bottom, top]), and the last input is a logical scalar which checks for periodicity, automatically adding periodic boundaries if required.

Note that the ID of the first interior degree of freedom is $Nbcs+1$, where $Nbcs$ is a scalar variable containing the number of boundary conditions, in this case $Nbcs = 4$. Therefore the first interior node ID is 5, and this will be the first unknown that will be solved (that is, the first row in the A matrix).

An example of `NodeP, Nodeu`, and `Nodev` for a $Nx = 5,\ Ny = 4$ domain is as follows:

$$NodeP = \begin{bmatrix} 1 & 4 & 4 & 4 & 4 & 4 & 2 \\ 1 & 5 & 9 & 13 & 17 & 21 & 2 \\ 1 & 6 & 10 & 14 & 18 & 22 & 2 \\ 1 & 7 & 11 & 15 & 19 & 23 & 2 \\ 1 & 8 & 12 & 16 & 20 & 24 & 2 \\ 1 & 3 & 3 & 3 & 3 & 3 & 2 \end{bmatrix}$$

$$Nodeu = \begin{bmatrix} 1 & 4 & 4 & 4 & 4 & 2 \\ 1 & 5 & 9 & 13 & 17 & 2 \\ 1 & 6 & 10 & 14 & 18 & 2 \\ 1 & 7 & 11 & 15 & 19 & 2 \\ 1 & 8 & 12 & 16 & 20 & 2 \\ 1 & 3 & 3 & 3 & 3 & 2 \end{bmatrix}$$

$$Nodev = \begin{bmatrix} 1 & 4 & 4 & 4 & 4 & 4 & 2 \\ 1 & 5 & 8 & 11 & 14 & 17 & 2 \\ 1 & 6 & 9 & 12 & 15 & 18 & 2 \\ 1 & 7 & 10 & 13 & 16 & 19 & 2 \\ 1 & 3 & 3 & 3 & 3 & 3 & 2 \end{bmatrix}$$

2.1.3. *Boundary Conditions.* *Significant care must be taken with properly setting up the boundary conditions*

There are a number of conventions related to the boundary conditions. First, let us define the variables used for the boundary conditions, in each case these variables are one-dimensional double arrays:

```
bcsDP: Pressure Dirichlet boundary conditions
bcsNP: Pressure Neumann boundary conditions
bcsOP: ID of Pressure Dirichlet boundary that should be an open boundary
bcsDu: U-velocity Dirichlet boundary conditions
bcsNu: U-velocity Neumann boundary conditions
bcsOu: ID of U-velocity Dirichlet boundary that should be an open boundary
bcsDv: V-velocity Dirichlet boundary conditions
bcsNv: V-velocity Neumann boundary conditions
bcsOv: ID of V-velocity Dirichlet boundary that should be an open boundary
```

The numbering convention comes about from the way that the node numbering matrices are used. The first *Nbcs* numbers are reserved for the boundary conditions. So, in this case, the first 4 numbers are reserved for boundaries. `bcsDP`, then is an array that contains the VALUE of the Dirichlet boundary conditions. Since this problem contains only uniform Dirichlet boundaries for velocity and uniform Neumann conditions for Pressure, `bcsDu, bcsDv`, and `bcsNP` are all 1×4 arrays with zero entries. The remaining arrays are empty. This is implemented as follows:

```
bcsDP= [];          bcsNP= [0 0 0 0];
```

```
bcsDu= [0 0 0 0];  bcsNu= [];
bcsDv= [0 0 0 0];  bcsNv= [];
```

2.1.4. *Initial conditions.* To create the initial conditions, first "coordinate" matrices that contain the coordinates of the control volume centres are created as follows:

`TODO`

note here we are using the MATLAB `meshgrid` function to create the matrices, and that these matrixes are the size of the INTERIOR nodes only, that is, they do not go all the way up to the boundaries. Also note, the vertical coordinate starts from the maximum value and decreases, and this is due to the node numbering convection described in Section 3.2.

When initializing the variables, the boundary condition values are also stored in the vector of unknowns. Therefore, when initializing the vector of unknowns, the values of the boundary conditions have to be included as well.

For the general case there may be interior masked nodes, and then the created coordinate matrices will have too many entries, and will not necessarily correspond to the IDs of the unknowns. Here the `idsP, idsu,` and `idsv` outputs from the `NodePad` function is useful for selecting the correct elements of the coordinate matrices.

Initializing the vector of unknowns using the analytical solution to the problem at $t = 0$ is accomplished as follows:

```
%% Solution
uexact = @(time,x)  pi*sin(time+1).*(sin(2*pi*x(:,2)).*(sin(pi*x(:,1))).^2);
vexact = @(time,x) -pi*sin(time+1).*(sin(2*pi*x(:,1)).*(sin(pi*x(:,2))).^2);
pexact = @(time,x)     sin(time+1).* cos(  pi*x(:,1)).* sin(pi*x(:,2)) ;
%Initialize vectors
P = [bcsDP';bcsNP';pexact(0,[XP(idsP) YP(idsP)])];
u = [bcsDu';bcsNu';uexact(0,[XU(idsu) YU(idsu)])];
v = [bcsDv';bcsNv';vexact(0,[XV(idsv) YV(idsv)])];
```

2.1.5. *Misc. and Plotting.* To plot the solution we can simply use `surf(P(NodeP))` in the plotting script. To plot the solution at the correct (x, y) coordinates for the interior we can use `surf(XP,YP,P(NodeP(2:end-1,2:end-1)))`.

This problem requires $\nu = 1$, and to ensure this happens, we explicitly set $\nu = 1$ in the `SetupScript` to guarantee this happens.

We also create vectors `Pex, uex, vex` in the `SetupScript`, and these are again used in `PlotScript` for plotting the exact solution.

2.2. **Sudden Expansion.** This test case is useful for ensuring the numerical solution remains symmetric up to a transition Reynolds number where numerical roundoff is sufficiently large enough to cause flow perturbations. It is also used to test the open boundary condition, and it demonstrates that the domain masking works correctly.

The Navier Stokes solver functions require 8 inputs: `Nx,Ny,Nt,nu,kappa,PlotIntrvl,SetupScript,Pl`
The first three inputs define the discretization, giving the number of INTERIOR points in the x-direction, y-direction, and in time. The fourth input is the value of the kinematic

viscosity (which is analogous to modifying the Reynolds number for the full NS equations), and the fifth is the diffusivity of the tracer density. The sixth input is an integer number that indicates the number of time integrations steps between calling the script given by the string input `PlotScript`. The seventh input is the main focus of the quick-start guide, and it is a string giving the name of a MATLAB script which sets up the necessary parts of the problem to be solved. That is, the solver calls `eval(SetupScript)` to setup the problem at hand. Similarly, the solver calls `eval(PlotScript)` every `PlotIntrvl` steps of the integration. While `PlotScript` is normally used for plotting the solution, it can also be used for saving the solution. Also, any uncleared variables in `SetupScript` will be available for use in `PlotScript`. Therefore, a function handle created in `SetupScript` can be used in `PlotScript` without causing an error.

The setup file is in its entirety as follows:

```
Todo
```

2.2.1. *Setting up forcing functions.* The forcing functions can take time and space as the input variables. For this test case there are no forcing function contributions, so we simply set:

```
Fu = @(time,p) 0;
Fv = @(time,p) 0;
```

2.2.2. *Setting up the grid.* The Stokes solvers require 4 node numbering matrices, `Noderho, NodeP, Nodeu`, and `Nodev` for the density, Pressure, u-velocity, and v-velocity. A simple way to create these matrices is by providing a "masking" matrix to the `NodePad` function. Since there we now have interior geometry for this test-case, we need to set parts of the masking matrix equal to a non-zero number to the `NodePad` function as follows:

```
%% create geometry mask
Nbcs = length(bcsDP) + length(bcsNP);
bnd = [1 1 1 1];
Mask = zeros(Ny, Nx);
Mask(1:round(Ny/3),1:round(Nx/5))=1;
Mask(end-round(Ny/3-1):end,1:round(Nx/5))=1;

%create Node matrices
[NodeP, Nodeu, Nodev, idsP, idsu, idsv] = NodePad(Mask,bnd,1);
Noderho=NodeP;
```

In this case there are Nx interior nodes in the x-direction, and Ny interior nodes in the y-direction, minus the nodes that are masked. Note that only the first fifth ($N_x/5$) of the domain in the x-direction is masked, and the top and bottom third ($N_y/3$) of the domain in the y-direction is masked. The mask is only created for the *interior* nodes. Therefore `NodePad` "pads" the masking matrix to add the four boundaries conditions on each edge. The second input to `NodePad` is a logical array, and tells the function which of the boundaries to pad ([left, right, bottom, top]), and the last input is a logical scalar which checks for periodicity, automatically adding periodic boundaries if required.

Note that the ID of the first interior degree of freedom is $Nbcs+1$, where $Nbcs$ is a scalar variable containing the number of boundary conditions, in this case $Nbcs = 5$. Therefore the first interior node ID is 6, and this will be the first unknown that will be solved (that is, the first row in the A matrix).

An example of `NodeP` and `Nodeu` for a $Nx = 10,\ Ny = 9$ domain is as follows:

$$NodeP = \begin{bmatrix} 2 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 3 \\ 2 & 1 & 1 & 13 & 21 & 30 & 39 & 48 & 57 & 66 & 75 & 3 \\ 2 & 1 & 1 & 13 & 22 & 31 & 40 & 49 & 58 & 67 & 76 & 3 \\ 2 & 1 & 1 & 14 & 23 & 32 & 41 & 50 & 59 & 68 & 77 & 3 \\ 2 & 6 & 9 & 15 & 24 & 33 & 42 & 51 & 60 & 69 & 78 & 3 \\ 2 & 7 & 10 & 16 & 25 & 34 & 43 & 52 & 61 & 70 & 79 & 3 \\ 2 & 8 & 12 & 17 & 26 & 35 & 44 & 53 & 62 & 71 & 80 & 3 \\ 2 & 1 & 1 & 18 & 27 & 36 & 45 & 54 & 63 & 72 & 81 & 3 \\ 2 & 1 & 1 & 19 & 28 & 37 & 46 & 55 & 64 & 73 & 82 & 3 \\ 2 & 1 & 1 & 20 & 29 & 38 & 47 & 56 & 65 & 74 & 83 & 3 \\ 2 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 3 \end{bmatrix}$$

$$Nodeu = \begin{bmatrix} 2 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 3 \\ 2 & 1 & 1 & 13 & 21 & 30 & 39 & 48 & 57 & 66 & 3 \\ 2 & 1 & 1 & 13 & 22 & 31 & 40 & 49 & 58 & 67 & 3 \\ 2 & 1 & 1 & 14 & 23 & 32 & 41 & 50 & 59 & 68 & 3 \\ 2 & 6 & 9 & 15 & 24 & 33 & 42 & 51 & 60 & 69 & 3 \\ 2 & 7 & 10 & 16 & 25 & 34 & 43 & 52 & 61 & 70 & 3 \\ 2 & 8 & 12 & 17 & 26 & 35 & 44 & 53 & 62 & 71 & 3 \\ 2 & 1 & 1 & 18 & 27 & 36 & 45 & 54 & 63 & 72 & 3 \\ 2 & 1 & 1 & 19 & 28 & 37 & 46 & 55 & 64 & 73 & 3 \\ 2 & 1 & 1 & 20 & 29 & 38 & 47 & 56 & 65 & 74 & 3 \\ 2 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 3 \end{bmatrix}$$

With the boundary conditions chosen for this problem, however, the arrangement given above is not possible (explanation is following section), hence the boundary condition numbers are re-arranged using the following code:

```
%Re-arrange pressure boundary node numbers suchs that open boundary has lowest number
NodeP(find(NodeP==1))=-1;NodeP(find(NodeP==3))=1;NodeP(find(NodeP==-1))=3;
%Re-arrange u-velocity boundary node numbers such that open boundary has highest number
Nodeu(find(Nodeu==3))=-1;Nodeu(find(Nodeu==5))=3;Nodeu(find(Nodeu==-1))=5;
%Re-arrange v-velocity boundary node numbers such that Neumann boundary has highest numb
Nodev(find(Nodev==3))=-1;Nodev(find(Nodev==5))=3;Nodev(find(Nodev==-1))=5;
%Re-arrange density boundary node numbers such that Neumann boundary has highest number
NodeRho(find(NodeRho==3))=-1;NodeRho(find(NodeRho==5))=3;NodeRho(find(NodeRho==-1))=5;
```

and gives the following results for `NodeP` and `Nodeu`:

$$NodeP = \begin{bmatrix} 2 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 1 \\ 2 & 3 & 3 & 13 & 21 & 30 & 39 & 48 & 57 & 66 & 75 & 1 \\ 2 & 3 & 3 & 13 & 22 & 31 & 40 & 49 & 58 & 67 & 76 & 1 \\ 2 & 3 & 3 & 14 & 23 & 32 & 41 & 50 & 59 & 68 & 77 & 1 \\ 2 & 6 & 9 & 15 & 24 & 33 & 42 & 51 & 60 & 69 & 78 & 1 \\ 2 & 7 & 10 & 16 & 25 & 34 & 43 & 52 & 61 & 70 & 79 & 1 \\ 2 & 8 & 12 & 17 & 26 & 35 & 44 & 53 & 62 & 71 & 80 & 1 \\ 2 & 3 & 3 & 18 & 27 & 36 & 45 & 54 & 63 & 72 & 81 & 1 \\ 2 & 3 & 3 & 19 & 28 & 37 & 46 & 55 & 64 & 73 & 82 & 1 \\ 2 & 3 & 3 & 20 & 29 & 38 & 47 & 56 & 65 & 74 & 83 & 1 \\ 2 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 1 \end{bmatrix}$$

$$Nodeu = \begin{bmatrix} 2 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 5 \\ 2 & 1 & 1 & 13 & 21 & 30 & 39 & 48 & 57 & 66 & 5 \\ 2 & 1 & 1 & 13 & 22 & 31 & 40 & 49 & 58 & 67 & 5 \\ 2 & 1 & 1 & 14 & 23 & 32 & 41 & 50 & 59 & 68 & 5 \\ 2 & 6 & 9 & 15 & 24 & 33 & 42 & 51 & 60 & 69 & 5 \\ 2 & 7 & 10 & 16 & 25 & 34 & 43 & 52 & 61 & 70 & 5 \\ 2 & 8 & 12 & 17 & 26 & 35 & 44 & 53 & 62 & 71 & 5 \\ 2 & 1 & 1 & 18 & 27 & 36 & 45 & 54 & 63 & 72 & 5 \\ 2 & 1 & 1 & 19 & 28 & 37 & 46 & 55 & 64 & 73 & 5 \\ 2 & 1 & 1 & 20 & 29 & 38 & 47 & 56 & 65 & 74 & 5 \\ 2 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 5 \end{bmatrix}$$

2.2.3. *Boundary Conditions.* *Significant care must be taken with properly setting up the boundary conditions*

There are a number of conventions related to the boundary conditions. First, let us define the variables used for the boundary conditions, in each case these variables are one-dimensional double arrays:

```
bcsDrho: Density Dirichlet boundary conditions
bcsNrho: Density Neumann boundary conditions
bcsDP: Pressure Dirichlet boundary conditions
bcsNP: Pressure Neumann boundary conditions
bcsOP: ID of Pressure Dirichlet boundary that should be an open boundary
bcsDu: U-velocity Dirichlet boundary conditions
bcsNu: U-velocity Neumann boundary conditions
bcsOu: ID of U-velocity Dirichlet boundary that should be an open boundary
bcsDv: V-velocity Dirichlet boundary conditions
bcsNv: V-velocity Neumann boundary conditions
bcsOv: ID of V-velocity Dirichlet boundary that should be an open boundary
```

The numbering convention comes about from the way that the node numbering matrices are used. The first *Nbcs* numbers are reserved for the boundary conditions. So, in this case,

the first 5 numbers are reserved for boundaries. `bcsDP`, then is an array that contains the VALUE of the Pressure Dirichlet boundary conditions with one exception (for open boundaries), and `bcsNP` contains the VALUE of the Pressure Neumann boundary conditions. `bcsOP` contains the IDs of Dirichlet boundary conditions that are actually open boundary conditions, and this is where the exception comes about for Dirichlet boundaries. This may seem strange, but due to the way open boundary conditions are implemented, they are first treated as Dirichlet boundaries and the matrices are modified afterwards to open boundary conditions. The numbering rules are as follow: The lowest numbered boundary conditions must be of Dirichlet or Open type, and the highest numbered boundary conditions must be of Neumann type. That is, there can never be a Dirichlet boundary that has a larger number than a Neumann boundary. This explains why the Node numbering matrices had to be renumbered in the grid-creation section. Open boundary conditions must be the highest numbered Dirichlet boundary condition, that is, no true Dirichlet boundary condition may have a higher number than an open boundary condition. Therefore, the numbering order (from lowest-numbered to highest numbered) is then as follows:

(1) Dirichlet
(2) Open
(3) Neumann

.

The open boundary condition used sets $\frac{\partial^2 \phi}{\partial n^2} = 0$, where ϕ is any quantity and n is in the normal direction.

Therefore, to implement the following boundary conditions

$$\begin{aligned}
\rho &= 0 \ \forall x \neq 20 \ on \ \partial\Omega \\
\frac{\partial \rho}{\partial n} &= 0 \ \forall x = 20 \ on \ \partial\Omega \\
\frac{\partial P}{\partial n} &= 0 \ \forall x \neq 20 \ on \ \partial\Omega \\
\frac{\partial^2 P}{\partial n^2} &= 0 \ \forall x = 20 \ on \ \partial\Omega \\
u &= 0 \ \forall y = 0 | y = 1 \ on \ \partial\Omega \\
u &= 1 \ \forall x = 0 \ on \ \partial\Omega \\
\frac{\partial^2 u}{\partial n^2} &= 0 \ \forall x = 20 \ on \ \partial\Omega \\
v &= 0 \ \forall x \neq 20 \ on \ \partial\Omega \\
\frac{\partial v}{\partial n} &= 0 \ \forall x = 20 \ on \ \partial\Omega
\end{aligned}$$

we use the code below:

```
%Density
bcsDrho = [0 -0.0 0 0];
bcsNrho = [0];
%Pressure
```

```
bcsDP= [0];
bcsNP = [0 0 0 0];
bcsOP= [1];
%Velocities
bcsDu= [0 1 0 0 0];
bcsNu= [];
bcsOu= [5];
bcsDv= [0 0 0 0];
bcsNv=  [0];
```

2.2.4. *Initial conditions.* When initializing the variables, the boundary condition values are also stored in the vector of unknowns. Therefore, when initializing the vector of unknowns, the values of the boundary conditions have to be included as well.

In this case there are interior masked nodes, which means that the created coordinate matrices will have too many entries, and will not necessarily correspond to the IDs of the unknowns. Here the `idsP, idsu,` and `idsv` outputs from the `NodePad` function needs to be used to select the correct elements of the coordinate matrices in order to properly initialize.

Initializing the vector of unknowns using a uniform u-velocity is accomplished as follows:

```
rhoinit=@(X,Y)zeros(size(X));
pinit = @(X,Y) zeros(size(X));
uinit = @(X,Y)  ones(size(X));
vinit = @(X,Y) zeros(size(X));

rho=[bcsDrho' ;bcsNrho' ;rhoinit(XP(idsP), YP(idsP))];
P = [bcsDP';bcsNP';  pinit(XP(idsP), YP(idsP))];
u = [bcsDu';bcsNu';  uinit(XU(idsu), YU(idsu))];
v = [bcsDv';bcsNv';  vinit(XV(idsv), YV(idsv))];
%Correct u-velocity IC to be divergence-free
u(Nodeu(2:end-1,round(Nx/5)+2:end))=1/3;
```

2.2.5. *Misc. and Plotting.* To plot the solution we can simply use `surf(P(NodeP))` in the plotting script. To plot the solution at the correct (x, y) coordinates for the interior we can use `surf(XP,YP,P(NodeP(2:end-1,2:end-1)))`.

3. Detailed documentation

3.1. **Naming conventions.** In the code two naming conventions are used. When dealing with a single control volume centered at (x, y) we name the locations $(x-\Delta x/2, y)$,$(x+\Delta x/2, y)$,$(x, y-\Delta y/2)$, and $(x, y+\Delta y/2)$ as [west/left], [east,right], [south,bottom], [north,top], respectively, using *LOWERCASE letters*. Similarly, we name the locations $(x-\Delta x, y)$,$(x+\Delta x, y)$,$(x, y-\Delta y)$, and $(x, y+\Delta y)$ as [West/Left], [East,Right], [South,Bottom], [North,Top], respectively, using *UPPERCASE letters*. Finally, it is customary to label the value of the function centered at the present CV as "Present" or just "P". We note that this may be

confusing with the Pressure, however, Pressure is never used as a subscript, so hopefully the meaning will be clear from the context. This naming convention is illustrated in Figure [1].

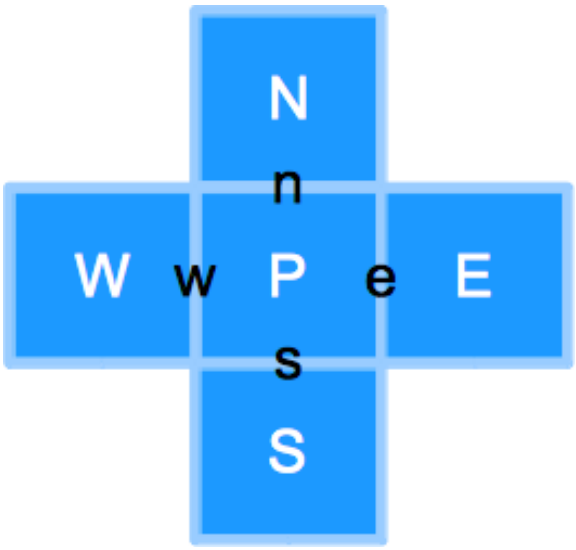

FIGURE 1. Naming conventions

Usually a single letter of the alphabet is used for integer numbers used in for loops, and generally the first for loop is started using the letter "i". Also, generally "i" is used for the rows of matrices, and "j" is used for the columns of matrices.

3.2. **Node number convention.** The tracer, u-velocity and v-velocity grids are shown superimposed in Figure [2] and separately in Figure [3]. The numbering convention used in the code is illustrated in Figure [2]. A node matrix will be used and it will be in the form `Node(i,j)`. Therefore, the "i" index will be used for the y-direction, and the "j" index will be used for the x-direction. This convention was chosen such that the way a matrix is printed on screen in MATLAB mirrors the numbering convention of the physical domain. It is nearly ideal, since increasing the column number of the matrix j increases the x spatial direction. However, to increase the y spatial direction, the row number of the matrix i has to decrease. Note, nowhere are the node-numbering matrices required, but these are used for coding convenience, and (hopefully) clarity.

3.3. **Data-Structures.** Since structured grids are used, the data-structures are reasonably simple. Unknowns are stored in 1-dimensional arrays. The first $Nbcs$ entries are reserved for boundary conditions, and are used to specify the value of boundaries. Time varying boundary conditions are allowed in this way, but these are not explicitly coded (although, it *could* be handled using `PlotScript`.

The remaining data-structures are restricted to the matrix operators. All matrix operators are stored as sparse two-dimensional arrays.

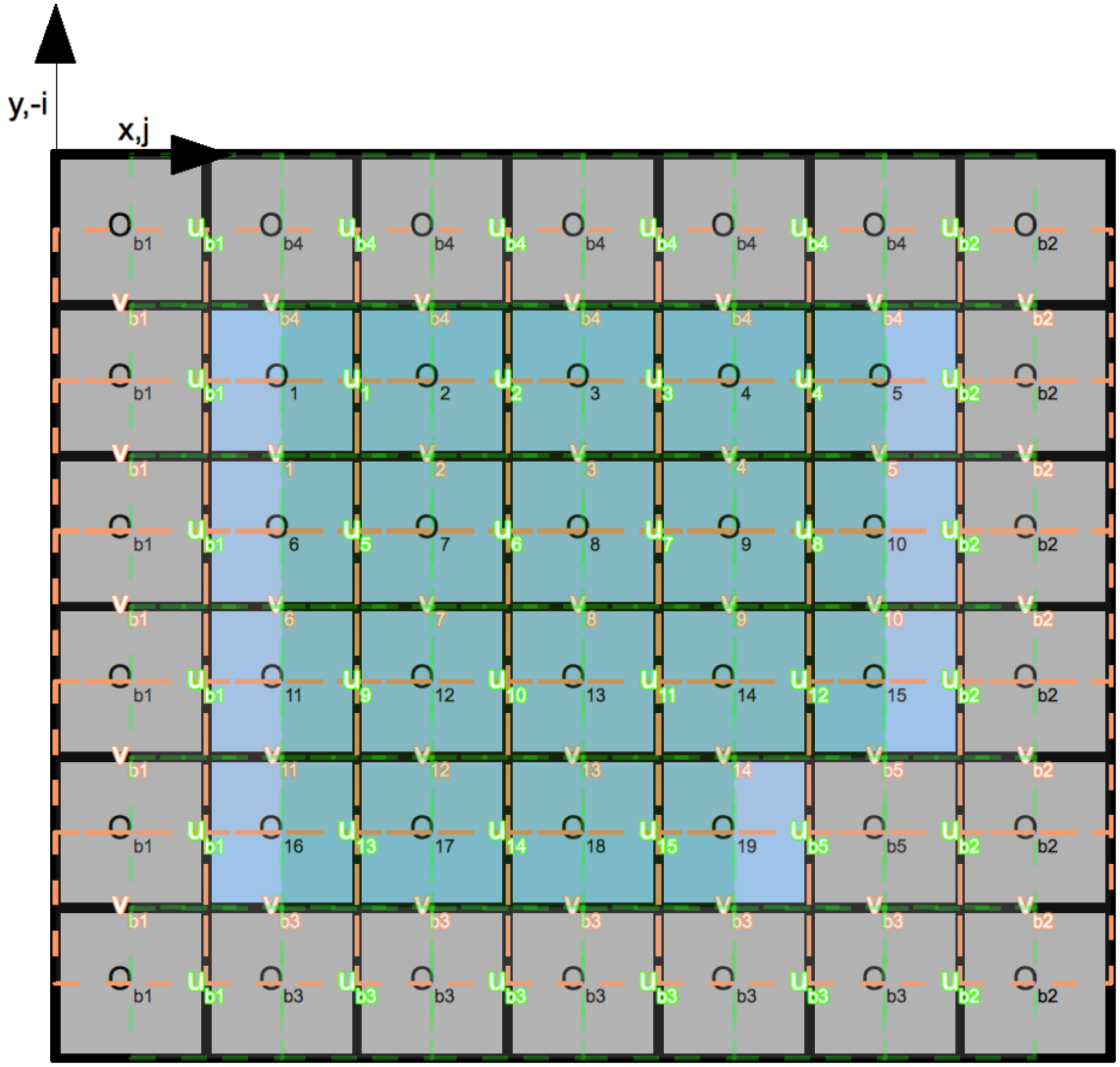

FIGURE 2. Grid setup

3.4. **Coordinate Transformation.** The main difference between the Vanilla code and this one is the use of a generalized coordinate transformation. This transformation requires functions $x = f(\xi, \eta), y = g(\xi, \eta)$, where ξ, η are defined on a uniform cartesian grid with $\xi = [0, 1], \eta = [0, 1]$, which is staggered as shown in Figure [2].

The conservation equation, which in cartesian coordinates reads:

$$\frac{\partial(\rho\phi)}{\partial t} + \frac{\partial}{\partial x_j}\left(\rho u_j \phi - \nu \frac{\partial \phi}{\partial x_j}\right) = S_\phi \tag{7}$$

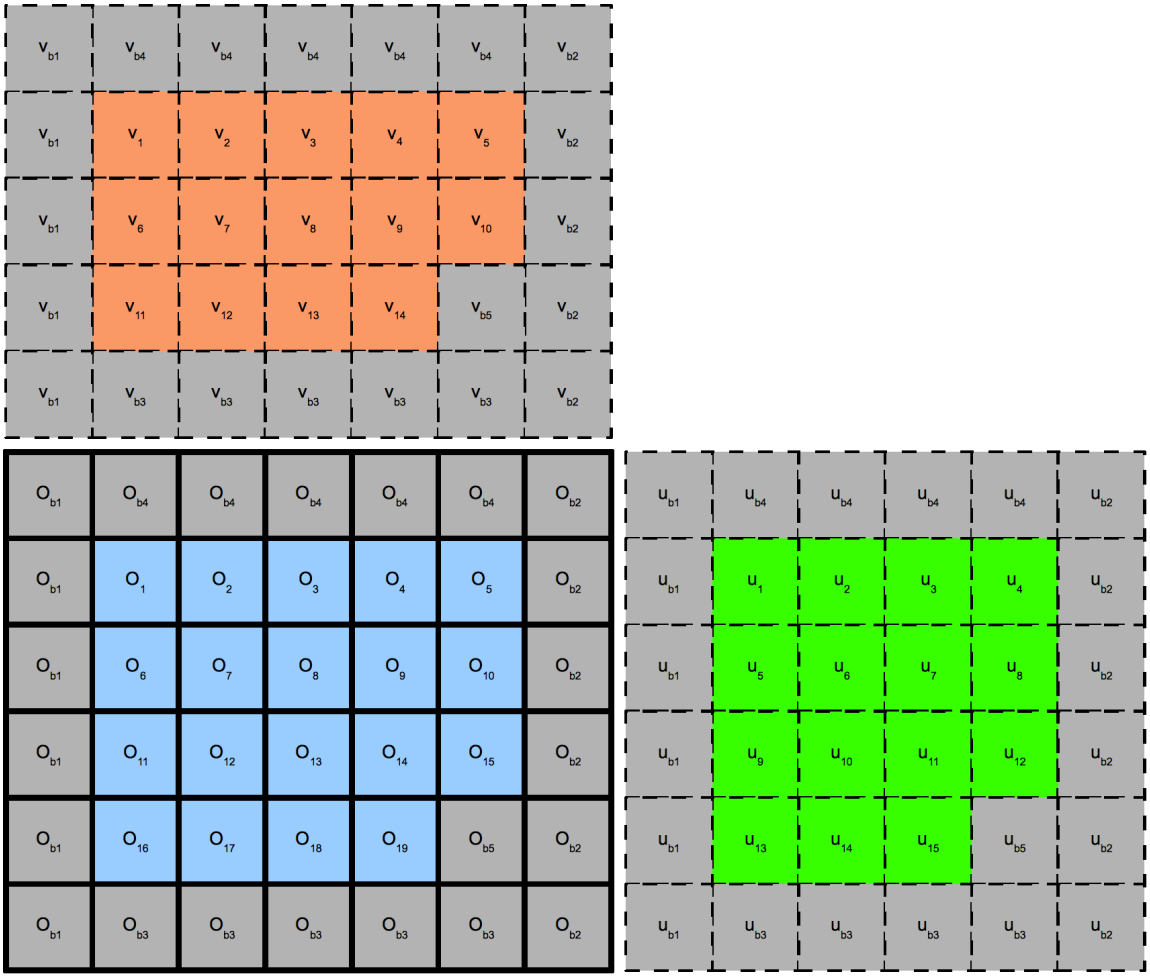

FIGURE 3. Grid setup

becomes

$$\frac{\partial(\rho\phi)}{\partial t} + \frac{1}{J}\frac{\partial}{\partial \xi_j}\left[\rho U_j \phi - \frac{\nu}{J}\left(\frac{\partial \phi}{\partial \xi_m} B^{mj}\right)\right] = S_\phi \tag{8}$$

where

$$J = det\left(\frac{\partial x_i}{\partial \xi_j}\right) = \begin{vmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{vmatrix} = \frac{\partial x}{\partial \xi}\frac{\partial y}{\partial \eta} - \frac{\partial x}{\partial \eta}\frac{\partial y}{\partial \xi} \tag{9}$$

$$U_j = u_k \beta^{kj} = u\beta^{1j} + v\beta^{2j} \tag{10}$$

$$B^{mj} = \beta^{kj}\beta^{km} = \beta^{1j}\beta^{1m} + \beta^{2j}\beta^{2m} \tag{11}$$

$$\beta^{ij} = \begin{bmatrix} \frac{\partial y}{\partial \eta} & -\frac{\partial y}{\partial \xi} \\ -\frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \xi} \end{bmatrix} \tag{12}$$

3.5. **Advection Schemes.** The different advection schemes implemented are described in this section. The functions discussed can be found under `<app root>/Advect>`, where `<app root>` is the root folder where these MATLAB scripts were uncompressed.

Both the UPWIND and QUICK (TODO) schemes for scalar advection and u, v-velocity advection are implemented.

The discrete finite volume problem is as follows:

$$\frac{\partial \phi_{x,y}}{\partial t}\Delta V = \left\{u_{x-\Delta x/2,y}\hat{\phi}_{x-\Delta x/2,y} - u_{x+\Delta x/2,y}\hat{\phi}_{x+\Delta x/2,y}\right\} + \left\{v_{x,y-\Delta y/2}\hat{\phi}_{x,y-\Delta y/2} - v_{x,y+\Delta y/2}\hat{\phi}_{x,y+\Delta y/2}\right\} \tag{13}$$

Or using the East, West, North, South naming convention:

$$\frac{\partial \phi_{x,y}}{\partial t}\Delta V = \left\{u_W\hat{\phi}_w - u_E\hat{\phi}_e\right\} + \left\{v_S\hat{\phi}_s - v_N\hat{\phi}_n\right\} \tag{14}$$

where the problem now is selecting appropriate values of the fluxes $\hat{\phi}_{x\pm\Delta x/2,y\pm\Delta y}$, which are approximately equal to $\hat{\phi}_i \approx \frac{\int_{A_i}\phi dA_i}{A_i}$.

A logical choice would be $\hat{\phi}_{x+\Delta x/2,y} = \frac{\phi_{x+\Delta x,y}+\phi_{x,y}}{2}$ and this is known as the "Central" flux. However, for general cases, the central flux is shown to be unstable.

For the following sections, we will use the east flux as the example flux.

3.5.1. *UPWIND schemes.* The upwind scheme works by choosing the upwind value of the function as the true value of the function at the interface of a control volume. The discrete flux function then takes the value:

$$\hat{\phi}_e = \begin{cases} \phi_P & u_E > 0 \\ \phi_E & u_E < 0 \end{cases} \tag{15}$$

To actually implement this in the code, a trick involving absolute values are used to avoid use of "if" statements, and this is as follows.

$$\hat{\phi}_e = \frac{1}{2}\left\{u_E(\phi_E + \phi_P) - |u_E|(\phi_E + \phi_P)\right\} \tag{16}$$

The upwind scalar and u, v-velocity advection functions are implemented in `SCAadvect_UW.m` and `UVadvect_UW.m` respectively.

For the scalar advection functions, the velocities at the center of the control surfaces are conveniently available, hence only the upwind value of the function needs to be selected in two dimensions for each of the four surfaces. To to this, the indices were carefully selected using integers "w, e, s, n" for the west, east, south, and north faces. Setting one of these integers equal to 1 and the rest to 0 then selects the correct indices for the corresponding face. Therefore, the only change in the code from one flux to the next is setting these integers, as well as selecting either the u or v component of velocity.

The final line selects only the active interior nodes, and returns the flux for those nodes. The advection is performed for all nodes in the domain, including inactive boundary nodes in the interior if present.

Essentially the same procedure is followed in the function performing the u, v-velocity advection, however in this case we need to consider the staggering of the grid more carefully. That is, sometimes it is necessary to average the vertical or horizontal velocity to obtain a value at the desired location. The code for the u and v velocity advection is very similar, in fact, once the u-velocity code was working it was copied, pasted, and with minor modifications was used for the v-velocity.

3.6. **Matrix Operators.** To solve the Navier Stokes equations using the Projection Methods as used in this code, the Laplacian and Helmholz matrices need to be inverted, and gradients and divergences need to be taken. The divergence and gradient operations do not require inversion, and are therefore implemented as functions in a matrix-free format. The Helmholz matrix is easily build from the Laplacian by adding a diagonal component, hence the major matrix that needs to be built and stores is the Laplacian matrix.

3.6.1. *Gradient Operations.* For solving the Navier Stokes and Stokes equations, only the gradient of the Pressure is required. This is implemented in `mk_Grad_Div_t.m`. A simple second order central difference (finite difference) scheme is used to find the gradient of the pressure. The gradient is calculated for the entire domain, including any interior masked points, and then only the active interior point are selected and outputted. For the transformed coordinate frame, the equations are as follows:

$$\begin{aligned}
\frac{\partial \phi}{\partial x} &= \frac{\partial \phi}{\partial \xi}\frac{\beta^{11}}{J} + \frac{\partial \phi}{\partial \eta}\frac{\beta^{12}}{J} \\
&\approx \frac{\phi_{i,j+\frac{1}{2}} - \phi_{i,j-\frac{1}{2}}}{\Delta\xi}\frac{\beta^{11}}{J} + \frac{\beta^{12}}{4J\Delta\eta}\left(\phi_{i-1,j+\frac{1}{2}} - \phi_{i+1,j+\frac{1}{2}} + \phi_{i-1,j-\frac{1}{2}} - \phi_{i+1,j-\frac{1}{2}}\right) \qquad (17) \\
\frac{\partial \phi}{\partial y} &= \frac{\partial \phi}{\partial \xi}\frac{\beta^{21}}{J} + \frac{\partial \phi}{\partial \eta}\frac{\beta^{22}}{J} \\
&\approx \frac{\beta^{21}}{4J\Delta\xi}\left(\phi_{i+\frac{1}{2},j+1} - \phi_{i+\frac{1}{2},j-1} + \phi_{i-\frac{1}{2},j+1} - \phi_{i-\frac{1}{2},j-1}\right) + \frac{\phi_{i-\frac{1}{2},j} - \phi_{i+\frac{1}{2},j}}{\Delta\eta}\frac{\beta^{22}}{J} \qquad (18)
\end{aligned}$$

There are a few things to note. First, this finds the derivative at the point i, j which would be a velocity point, NOT a pressure point. Therefore, the equation above is written using

the velocity point indices. Also, keep in mind that in our notation i is for $n-s$ (up down) and j is for $e-w$ (right left), and i decrease going north (up) while j increases going east (right). This gives the following computational stencils (now written in Pressure indices)

	j	$j+1$
$i-1$	$\frac{\beta^{12}}{4J\Delta\eta}$	$\frac{\beta^{12}}{4J\Delta\eta}$
i	$-\frac{\beta^{11}}{J\Delta\xi}$	$\frac{\beta^{11}}{J\Delta\xi}$
$i+1$	$-\frac{\beta^{12}}{4J\Delta\eta}$	$-\frac{\beta^{12}}{4J\Delta\eta}$

TABLE 1. Computational cell for $\frac{\partial P}{\partial x}$ in Pressure indices

	$j-1$	j	$j+1$
i	$-\frac{\beta^{21}}{4J\Delta\xi}$	$\frac{\beta^{22}}{J\Delta\eta}$	$\frac{\beta^{21}}{4J\Delta\xi}$
$i+1$	$-\frac{\beta^{21}}{4J\Delta\xi}$	$-\frac{\beta^{22}}{J\Delta\eta}$	$\frac{\beta^{21}}{4J\Delta\xi}$

TABLE 2. Computational cell for $\frac{\partial P}{\partial y}$ in Pressure indices

3.6.2. *Divergence Operations.* For solving the Navier Stokes and Stokes equations, only the divergence of the velocity is required. This is implemented in `mk_Grad_Div_t.m`.The basic divergence operator implementation is essentially the same as the Gradient operator, since it also use a second-order accurate finite difference approximation of the derivatives. However, in the case where Neumann boundary conditions are used, additional steps are required to set the correct value of $\frac{\partial phi}{\partial n}$ at the boundary. Therefore, in the code, first the derivatives are found while ignoring the boundary conditions. Then the IDs of the Neumann boundaries are found. Finally, on Neuman boundaries, the calculated derivative is replaced by the specified values of the Neumann boundaries.

The implementation looks, perhaps, more complicated than it should be. However, this implementation is generally applicable to any interior masked domain. The computational cells remain the same, with some minor changes in indices due to the grid staggering.

	$j-1$	j
$i-1$	$\frac{\beta^{12}}{4J\Delta\eta}$	$\frac{\beta^{12}}{4J\Delta\eta}$
i	$-\frac{\beta^{11}}{J\Delta\xi}$	$\frac{\beta^{11}}{J\Delta\xi}$
$i+1$	$-\frac{\beta^{12}}{4J\Delta\eta}$	$-\frac{\beta^{12}}{4J\Delta\eta}$

TABLE 3. Computational cell for $\frac{\partial u}{\partial x}$ in u-velocity indices

	$j-1$	j	$j+1$
$i-1$	$-\frac{\beta^{21}}{4J\Delta\xi}$	$\frac{\beta^{22}}{J\Delta\eta}$	$\frac{\beta^{21}}{4J\Delta\xi}$
i	$-\frac{\beta^{21}}{4J\Delta\xi}$	$-\frac{\beta^{22}}{J\Delta\eta}$	$\frac{\beta^{21}}{4J\Delta\xi}$

TABLE 4. Computational cell for $\frac{\partial v}{\partial y}$ in v-velocity indices

3.6.3. *Laplace Operator.* The discrete version of the Laplacian operator ∇^2 is built using the `mk_Laplace` function. The base discrete matrix is built for either Neumann or Dirichlet boundary conditions. However, due to staggering, the Dirichlet boundary conditions will be incorrect for some faces of the u, v CVs, and all faces of the Pressure CVs. These are fixed using `fixdbcs_t.m`. Also, open boundary conditions are not yet implemented, but should be corrected for using `FixDiffPObcs`.

	$j-1$	j	$j+1$
$i-1$	$\frac{\varepsilon^{12}_{k,m-\frac{1}{2}}+\varepsilon^{12}_{k+\frac{1}{2},m}}{4J\Delta\xi\Delta\eta}$	$-\frac{\varepsilon^{22}_{k+\frac{1}{2},m}}{J\Delta\eta^2}+\frac{\varepsilon^{12}_{k,m-\frac{1}{2}}-\varepsilon^{12}_{k,m+\frac{1}{2}}}{4J\Delta\xi\Delta\eta}$	$\frac{-\varepsilon^{12}_{k,m+\frac{1}{2}}-\varepsilon^{12}_{k+\frac{1}{2},m}}{4J\Delta\xi\Delta\eta}$
i	$-\frac{\varepsilon^{11}_{k,m-\frac{1}{2}}}{J\Delta\xi^2}+\frac{\varepsilon^{12}_{k+\frac{1}{2},m}-\varepsilon^{12}_{k-\frac{1}{2},m}}{4J\Delta\xi\Delta\eta}$	$\frac{\varepsilon^{11}_{k,m+\frac{1}{2}}+\varepsilon^{11}_{k,m-\frac{1}{2}}}{J\Delta\xi^2}+\frac{\varepsilon^{22}_{k+\frac{1}{2},m}+\varepsilon^{22}_{k-\frac{1}{2},m}}{J\Delta\eta^2}$	$-\frac{\varepsilon^{11}_{k,m+\frac{1}{2}}}{J\Delta\xi^2}+\frac{\varepsilon^{12}_{k-\frac{1}{2},m}-\varepsilon^{12}_{k+\frac{1}{2},m}}{4J\Delta\xi\Delta\eta}$
$i+1$	$\frac{-\varepsilon^{12}_{k,m-\frac{1}{2}}-\varepsilon^{12}_{k-\frac{1}{2},m}}{4J\Delta\xi\Delta\eta}$	$-\frac{\varepsilon^{22}_{k-\frac{1}{2},m}}{J\Delta\eta^2}+\frac{\varepsilon^{12}_{k,m+\frac{1}{2}}-\varepsilon^{12}_{k,m-\frac{1}{2}}}{4J\Delta\xi\Delta\eta}$	$\frac{\varepsilon^{12}_{k,m+\frac{1}{2}}+\varepsilon^{12}_{k-\frac{1}{2},m}}{4J\Delta\xi\Delta\eta}$

TABLE 5. Computational cell for ∇^2 in transformed coordinates. $\varepsilon^{ij}_{k,m} = \frac{\nu}{J}B^{ij}|_{k\Delta\eta,m\Delta\xi,}$.

This cell highlights one of the unfortunate effects of our choice of coordinates systems. While i decrease, we require k to increase, which can lead to some confusion. Particularly, the function used in the code for $\varepsilon^{ij}_{k,m}$ is defined as `E`^{ij}`(`$m\xi$`,`$k\eta$`)`, that is, the subscript and the function call order are reversed. Finally, notice that $\varepsilon^{ij} = \varepsilon^{ji}$ is symmetric.