2.29 Finite Volume MATLAB Framework Documentation

Manual written by: Matt Ueckermann, Pierre Lermusiaux

November 20, 2011

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 for an incompressible fluid (constant density) or for a Boussinesq fluid 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 the perturbation density multiplied by g.

2 Download and Installation

2.1 Downloading

The code can be downloaded from `http://mseas.mit.edu/codes`. You will have to register to get access to the site. Once you are logged in, you may download the latest Framework from the "'Downloads::Dynamic Models"' link:
`http://mseas.mit.edu/codes/downloads/dynamic-models`. The code is also directly available on Stellar for 2.29 students.

2.2 Installing on Linux

Installation is simple on a Linux machine. Simply untar the file in the directory of your choice, for example:

```
% cd $install-root\
% tar xvfz FV_MATLAB_Framework_vX-XXX.tar.gz
```

where `$install-root` is a directory of your choice, and `X-XXX` indicates the version number.

Some files are coded in C++ for efficiency and these need to be compiled. If you are lucky, the files that come with the framework that we compiled on our machine will work on your machine. But if this is not the case, you will need to "mex" the routines yourself.

On Linux this is a simple task.

```
% cd $install-root\
% matlab -nodesktop -nosplash
>> setup
>> cd Src/Advect
>> mex Narrowband_Advect_LS_mx.cpp
>> mex Narrowband_LS_mx.cpp
>> mex SCAadvect_CDS_mx.cpp
>> mex SCAadvect_mx.cpp
>> mex SCAadvect_TVD_mx.cpp
>> mex UVadvect_DO_CDS_mx.cpp
>> mex UVadvect_DO_mx.cpp
>> mex UVadvect_TVD_mx.cpp
>> cd ../Util
>> mex dsegment.cpp
>> cd ../..
```

and you are ready to go!

If this does not work, it is possible that you do not have a C++ compiler installed on your system. In that case, you may have to install a compiler and set it up in MATLAB. For example, if you are on an Ubuntu system, the following is a start:

```
% sudo apt-get install build-essential
% matlab -nosplash -nodesktop
>> mex -setup
```

at which point you are on your own, good luck!

2.3 Installing on Mac

Installation is a little more complicated a Mac. Start by uncompressing the file in the directory of your choice.

Some files are coded in C++ for efficiency and they have to be compiled. If you are lucky, the files that come with the framework that we compiled on our machine will work on your machine. You can first try things out, but if our compiled routines don't work on your machine, you will need to "mex" the routines yourself.

On a mac, you first have to install Xcode, including the Developer tools. Once you have these installed, you will have a "'/Developer/usr"' directory on your Mac. Once properly installed, you should be able to mex the required routines. Start by opening MATLAB, and from the command window type:

```
>> setup
>> cd Src/Advect
>> mex Narrowband_Advect_LS_mx.cpp
>> mex Narrowband_LS_mx.cpp
>> mex SCAadvect_CDS_mx.cpp
>> mex SCAadvect_mx.cpp
>> mex SCAadvect_TVD_mx.cpp
>> mex UVadvect_DO_CDS_mx.cpp
>> mex UVadvect_DO_mx.cpp
>> mex UVadvect_TVD_mx.cpp
>> cd ../Util
>> mex dsegment.cpp
>> cd ../..
```

and you are ready to go!

If this does not work, it is possible that the /Developer directory is not in you path. To check, type

```
>> !echo $PATH
```

From MATLAB. If you do not see /Developer/usr/bin and /Developer/usr/lib/, then MATLAB will not be able to find the compilers. To fix this, open a new X11 terminal. Start by checking if the directories are in the path, if not add, them, then open MATLAB from the terminal:

```
bash-3.2$ !echo $PATH
bash-3.2$ export PATH=$PATH:/Developer/usr/bin
bash-3.2$ export PATH=$PATH:/Developer/usr/lib
bash-3.2$ cd /Applications/MATLAB_R2010b/bin
bash-3.2$ ./matlab
```

Then change folders until you get to the trunk/Src/Advect folder, and issue the commands above.

2.4 Installing on Windows

Installation on Windows is a little more complex than on a Linux machine. First the downloaded file needs to be untarred, and for this you will need a program such as WinRaR `http://www.rarlab.com/` or IZArc. Once you have an appropriate program, untar the file to a directory of your choice `$install-root` (for example, `$install-root=C:\Users \usrname \Documents \MIT2-29FV \`).

Some files are coded in C++ for efficiency and they have to be compiled. If you are lucky, the files that come with the framework that we compiled on our machine will work on your machine. But if this is not the case, you will need to "mex" the routines yourself. You may be able to use the C-compiler that comes with MATLAB, but it may not be the best. To use it, start with:

```
>> mex -setup
```

Select [y] → [1] Lcc-win32 C 2.4.1 in C:\PROGRA 1\MATLAB\R2009a\sys\lcc → [y]. Then you may compile the mexed routines as follows:

```
>> setup
>> cd Src/Advect
>> mex Narrowband_Advect_LS_mx.cpp
>> mex Narrowband_LS_mx.cpp
>> mex SCAadvect_CDS_mx.cpp
>> mex SCAadvect_mx.cpp
>> mex SCAadvect_TVD_mx.cpp
>> mex UVadvect_DO_CDS_mx.cpp
>> mex UVadvect_DO_mx.cpp
>> mex UVadvect_TVD_mx.cpp
>> cd ../Util
>> mex dsegment.cpp
>> cd ../..
```

If this does not work, the first task in to install another compiler. For this you could either install Mingw (for helpful instructions see `http://gnumex.sourceforge.net/`) or the Microsoft compilers, such as Visual C++ Studio Express
(`http://www.microsoft.com/express/windows/`).
MP personally likes the Microsoft compiler, because the Visual Studio Express debugger can be used to debug MATLAB mex files.

After installing a compiler, you need to set up the compiler in MATLAB. This is, unfortunately, not a completely straightforward task, and seems to change for every version of MATLAB and every version of Visual Studio Express. While not completely straightforward, once found, the solution is simple. From MATLAB type

```
>> mex -setup
```

You could choose option [y] at this stage, and perhaps MATLAB will correctly detect the correct compiler, but in case that does not work, choose "n".

Now MATLAB gives you a choice, the correct move here is to choose the compiler that most closely relates to the one you just installed. For example, choose `[8] Microsoft Visual C++ 2008 Express`.

Unfortunately, the default installation directory is not correct, so one has to manually enter the installation directory for his/her compiler (for example, `C:\Program Files (x86)\Microsoft Visual Studio 10.0`). After doing this, one can mex the necessary files as above. At this point, you should hopefully be ready to go!

3 Quick Start Guide

After starting MATLAB, browse to the ⟨`install-root`⟩ directory (directory containing `setup.m`). The first task is to properly setup all the paths so that MATLAB knows where to find the necessary source files. This is done by running the `setup.m` script from the MATLAB prompt.

NOTE: To use this framework, setup has to be run every time you start MATLAB.

To see if the Framework is correctly installed with all components working, you can run the 21 examples we have provided by issuing the following commands. From the ⟨`install-root`⟩ directory (directory containing `setup.m`) run:

```
>> setup
>> TestCase(?);
```

where `?` should be replaced by any integer from 1-18.

The `TestCase.m` is basically a high-level interface to all the solvers and examples that we provide. For detailed information about the inputs, outputs, different examples and solvers, type

```
>> help TestCase
```

at the MATLAB prompt. It is not necessary to run `TestCase` for all applications, but the solvers may be called directly. However, the examples in `TestCase` have been tested thoroughly, and should work without error when using all default parameters.

`TestCase` may be invoked with 1, 2, or 3 inputs. By running `TestCase.m` with only 1 input, default values are used for the solver, resolution, diffusivity, and other parameters. The second input is a structure with which you may adjust many of the other solution parameters. If you adjust these values and the solution no longer works, please see the Troubleshooting section of this document. With the third input, a different solver may be chosen instead of the default.

For example, you may want to run the Lock-Exchange example with the default solver, but a different ν. This can be done as follows:

```
>> app.nu = 0.1
>> TestCase(6,app);
```

that is, simply enter "[]" for the solver.

3.1 Brief description of provided examples

1. Heated plate demo: This test-case demonstrates the different boundary conditions and geometry that can be used on the different staggered grids (see §4.2). Its main purpose, initially, was to validate that the diffusion operator was working correctly.

2. Tracer advection demo with QUICK scheme: This example is used for a convergence study to verify the correctness of the advection scheme. This demo uses the Quadratic Upwind Interpolation for Convective Kinetics (QUICK) scheme (see §5.1.3). Its performance is contrasted with the upwind scheme in the next example.

3. Tracer advection demo with Upwind scheme: This example is included to highlight the poor performance of the low-order upwind scheme (see §5.1.2), which is still commonly used in many CFD codes, compared to the previous example using the QUICK scheme.

4. Analytical Stokes Example: This example is used to verify the correctness of the Stokes solver implementations. It is described in detail in §4.5.

5. Poiseuille flow test case: This example solves the classical Poiseuille flow in a pipe. It tests the implementation of the open boundary condition.

6. Lock exchange experiment: This example tests Boussinesq solvers, that is, density driven flows. The setup is as follows: initially a dense and light fluid are separated by a barrier or a "lock." At time 0, the barrier is instantaneously removed, and the flow evolves.

7. Sudden expansion: This example demonstrates the accuracy of a control-volume analysis compared to a Bernoulli analysis for predicting the pressure drop across a sudden expansion in a pipe. Students in MIT course 2.29 solve this problem analytically using the two approaches, and verify the answer numerically at a later point in the course. The Bernoulli approach is incorrect in this case, because the recirculation zones that develop violate the - no viscous effects assumption - upon which Bernoulli is based. This example ensures that the numerical solution remains symmetric up to a transition Reynolds number where numerical roundoff can become sufficiently large to cause flow perturbations. It is also used to test the open boundary condition, and it demonstrates that the domain masking works correctly. The setup script for this example also shows how to create more complex geometries. This case can readily

be modified for the classical flow over a cylinder test-case. The setup file is described in detail in §4.6.

8. Buoyant sudden expansion: This is nearly the same as the previous example, but a lighter density fluid is allowed to enter the domain, causing fluid to rise upwards after the expansion. This introduces an asymmetry into the problem, and vortex shedding is seen using the default parameter values.

9. Warm rising bubble: This is another density driven flow example. Here, the solution is initialized with a small bubble of warm water surrounded with cold water in a tube with closed ends. The bubble rises and diffuses/mixes with the warm water as the flow evolves. The plot script for this example shows an elegant method for saving outputs (such as images of .mat files). It involves adding additional fields to the `app` data-structure.

10. Lid-driven cavity flow: This example is a standard test-case for CFD codes. Fluid initially at rest is inside a closed box (or cavity). At the time 0, the lid (or top boundary) instantaneously moves with a fixed velocity, and continues to do so until the end of the simulation. For low Reynolds numbers, a steady state is reached, whereas for sufficiently high Reynolds numbers, a steady state is not reached.

11. Stochastic lid-driven cavity flow: This is the same setup as the previous example, but adds uncertain initial conditions. The uncertainties are then allowed to evolve using the Dynamically Orthogonality (DO) equations (see Sapsis and Lermusiaux (2009)). The setup script for this case demonstrates how to manually initialize DO simulations for specific applications.

12. Stochastic lid-driven cavity flow: This is the same setup as the previous two examples, except in this case the uncertain initial conditions are automatically initialized, and an adaptive number of modes are used. That is, during the simulation, a variable number of terms are kept in the expansion for the velocity (see Sapsis and Lermusiaux (2011)).

13. Stochastic Double-Gyre test case: This example makes use of the solvers that include the Coriolis forces. It uses automatic initialization for the DO modes, and an adaptive number of modes. The fluid is initially at rest, and at time 0, a wind instantaneously starts to force the 2D horizontal flow. The 2D wind on top of the fluid is the shape of a cosine function, with maximum amplitudes at the top, middle, and bottom of the 2D horizontal domain. The direction of the wind at the top and bottom are to the left, whereas in the middle the direction is to the right. This drives the flow to form a double-gyre, which becomes unstable after long integration times for high enough Reynolds numbers. The default parameter values do no reach the point of instability, since simulations that do take considerable time to complete.

14. Deterministic Double-Gyre test case: Same example as above, but using the deterministic solver.

15. Stochastic Double-Gyre test case with path planning: This has the same setup as the above two examples, except here a vehicle is added which is affected by the flow. This example demonstrates the use of Level Sets to advance a wavefront (representing possible vehicle positions) in a dynamic flow field. From the level sets, a path can be calculated to desired targets, which is demonstrated for two targets in this example.

16. Path planning with analytically-provided flow-field (1): This example demonstrates path planning, but in this case the flowfield is provided through an analytic expression as a function of position and time. This example has a rotating, time-varying flowfield.

17. Path planning with analytically-provided flow-field (1): This example is the same as the previous one, except in this case the flowfield is less complicated and not time-varying. It has a highway flow, and illustrates how two very different paths can reach similar end points at the same time.

18. Thermohaline circulation: This example solve the Boussinesq equations with the density determined through a non-linear state equation which is a function of pressure, salinity and temperature. The 2D cross-section represents a zonally averaged meridional slice through the earth, that is, the south-pole is on the left and the north-pole is on the right, the top is the surface of the ocean, and the bottom is the ocean seabed. The circulation here is driven by the temperature difference between the equator and the poles. This example is a step towards idealized stochastic climate studies. The setup file for this example also illustrates how to have a variable (non-constant) boundary condition on the top boundary.

19. Flow over a Square Cylinder in a Confined channel: Very similar to the sudden expansion example, this illustrates the classical flow-over-a-cylinder test-case, showing von Karman vortex streets for $\mathrm{Re} > 50$ and a steady recirculation zone for $\mathrm{Re} < 40$.

20. Deterministic Rayleigh-Bernard Instability: This is a buoyancy-driven flow, where to top is being cooled and the bottom heated – resulting in an unstable density stratification which drives flow. This classic test-case can have multiple solutions depending on the input parameters (for example, clockwise versus counter-clockwise circulation).

21. Stochastic Rayleigh-Bernard Instability: Same test as above using the Stochastic solver. Here multiple solutions are captures within a single run.

4 Setting up new examples

This section is for users who wish to set up new problems using this framework or want to have a better understanding of the inner working of this code. We remind the reader that this is not an exhaustive manual, but should be sufficient for students who have been introduced to numerical fluid mechanics. In general the MATLAB functions are well-commented, and for further study, users are encouraged to read the code.

4.1 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].

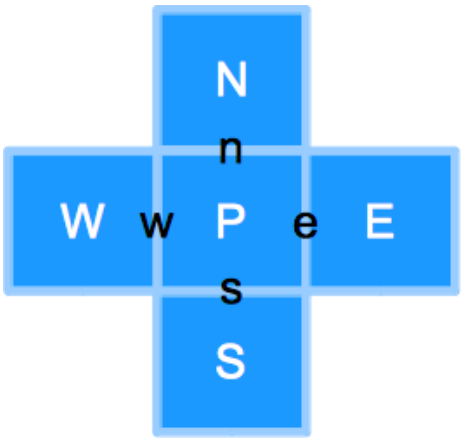

Figure 1: Naming conventions

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.

4.2 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.

4.3 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, and coded through the `updateBcs` function.

The application data-structure `app` is used as the general input structure. By using a structure, users are allowed to pass in whatever information they desire to be used in the setup scripts. For example, one could include an `app.bcsDu` field that controls the u-velocity boundary condition.

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

4.4 Domain, Masking and Boundary conditions: General Rules and guidelines

Here we describe the overall rules and guidelines required in the set-up of the domain, masking and boundary conditions for all simulations and test cases. Some users may want to check sub-sections on a forced Stokes flow (sub-section 4.5) and on a Sudden Expansion flows (sub-section 4.6) for more detailed examples. In what follows, the description is a bit more generic.

4.4.1 Domain

The domain is set-up as a rectangle. The call to mesh the domain is:

```
[XP, YP, XU, YU, XV, YV, dx, dy] = meshdomain(app, [0, a], [0, b])
```

We note that domains are always rectangles/squares, so up to the number of finite volumes, the set-up of the domain is pretty much the same for all examples. What varies from one

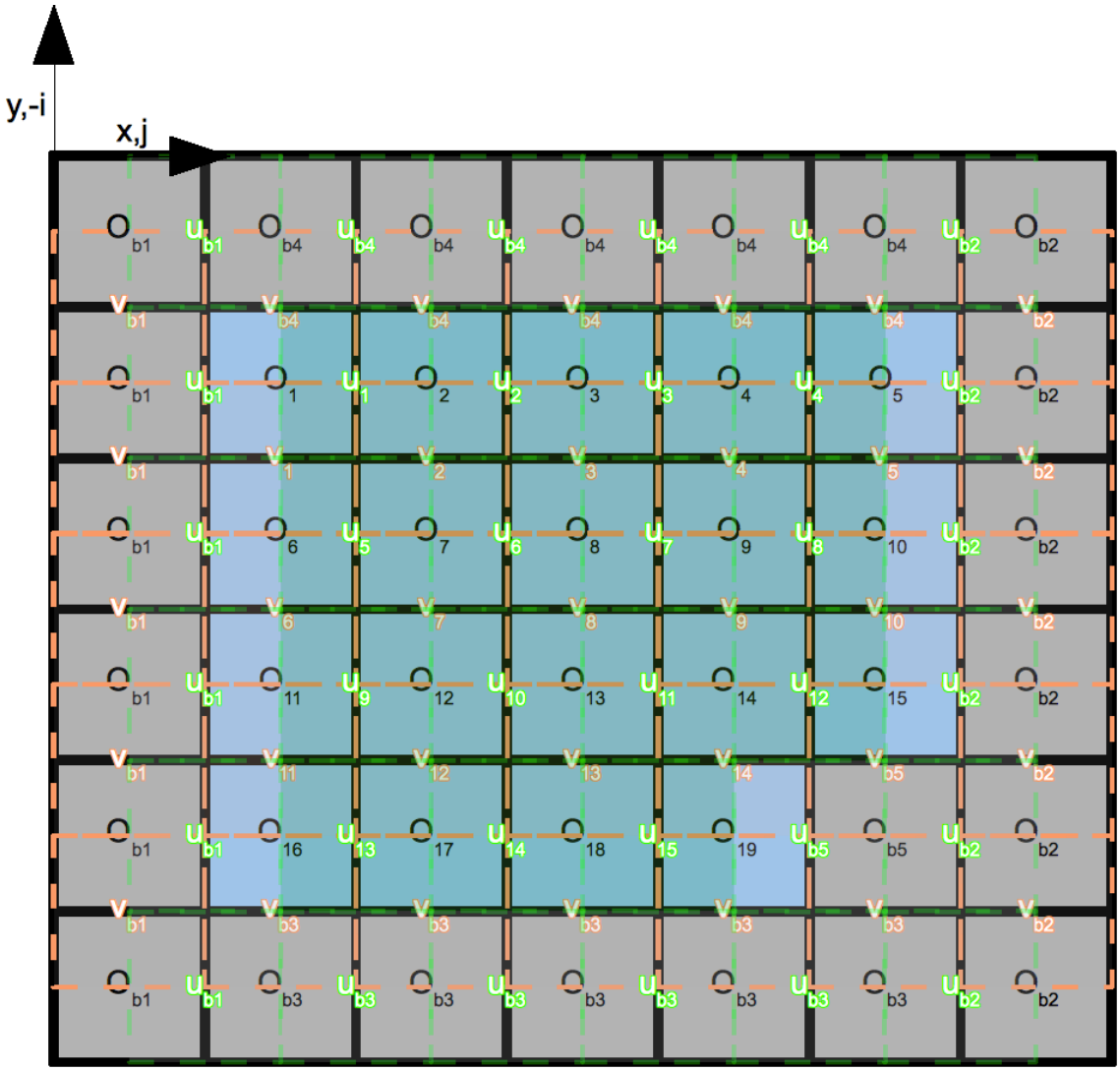

Figure 2: Grid setup

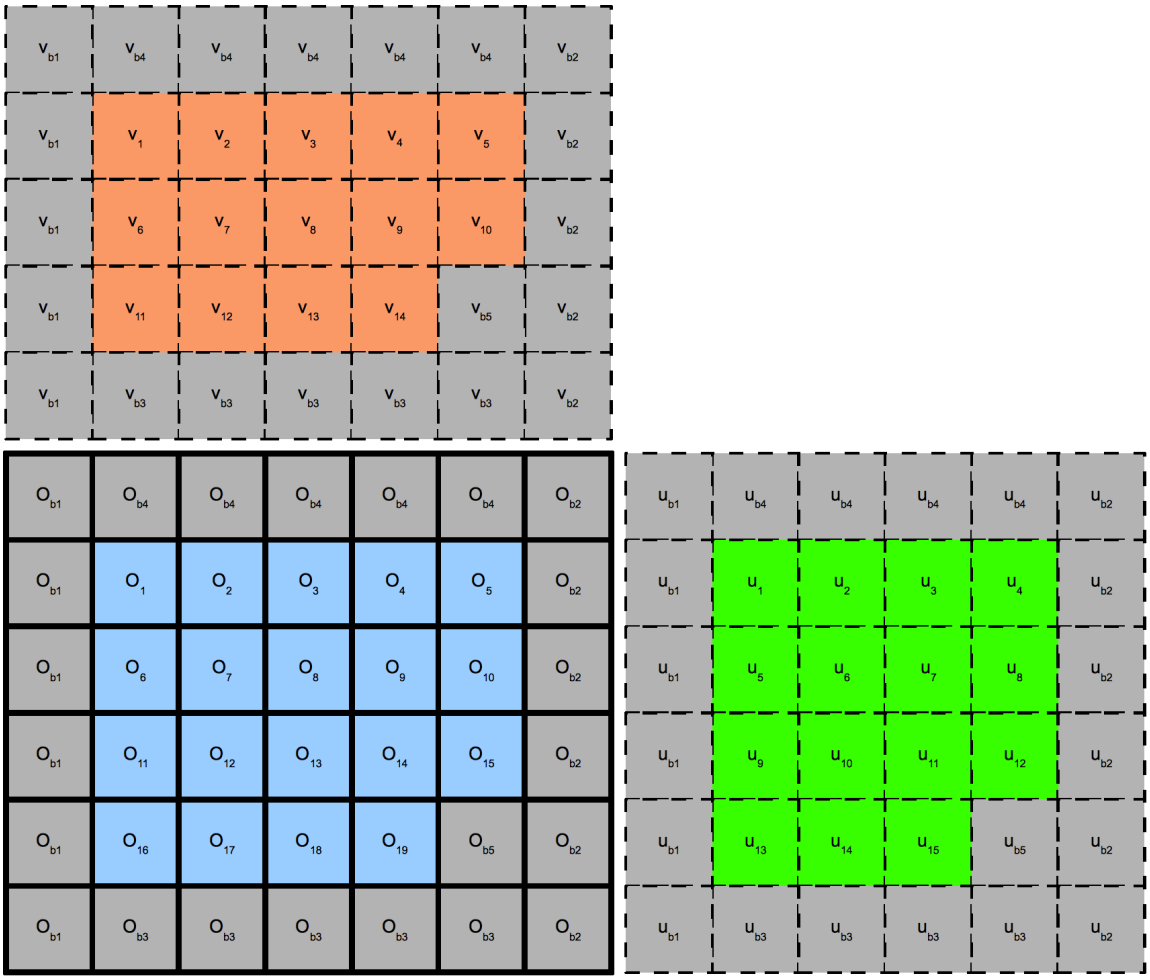

Figure 3: Grid setup

example to the next are the masking and boundary/initial conditions, and of course the equations used.

4.4.2 Masking

This mask is user-created. Zeros are unmasked, and non-zero integers indicate a masked region. The four domain sides are not considered as masked, but they have a separate number assigned to them in the Node numbering arrays.

To allow for different boundary conditions at the edges of each masked regions, the interior of each masked region must have a different integer number. If two masked region have the same number, they will be considered as a single masked region with its own definition of boundary condition at the edges of the mask.

First, the whole mask is created that fills the whole rectangular domain. Then the user can define additional masked regions using the position arrays XP and YP, and Boolean rules.

For example, let's assume we would like to define a domain for sudden expansion, but with a elliptical obstacle (Fig. 4). We then need to define three masked region, two rectangles and one ellipse. Each one of these are defined using their positions and Boolean rules.

For the two rectangles, we have:

```
top_rect = @(x, y) logical((y \ge 2 / 3) .* (x \le 4));
bot_rect = @(x, y) logical((y \le 1 / 3) .* (x \le 4));
```

For the ellipse, we have:

```
ellipse = @(x, y) (((x - 10) .^2 / 20 + (y - 0.5) .^ 2) < 0.1);
```

We then assign numbers to the three masked region, for example, associate 1 with the top domain, 2 for the bottom domain and 3 for the circle.

```
Mask(top_rect(XP, YP)) = 1;
Mask(bot_rect(XP, YP)) = 2;
Mask(ellipse(XP, YP)) = 3;
```

4.4.3 Boundary Conditions

The type of boundary conditions needs to be specified. The available types are:

1. 'D': Dirichlet with a single value
2. 'Dmv': Dirichlet with multiple values (i.e. spatially variable)
3. 'O': Open boundary

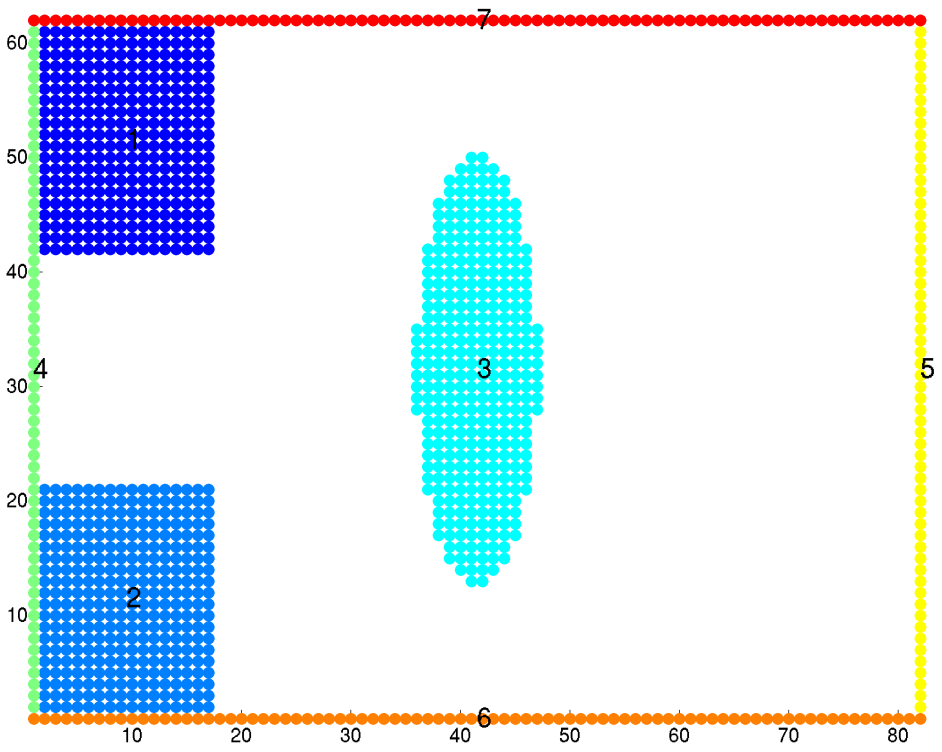

Figure 4: Sudden Expansion Boundary ID setup with an additional cirle region masked.

4. 'N': Neumann with a single value

5. 'Nmv': Neumann with multiple values (i.e. spatially variable)

The boundary type is specified in a cell array `bcid2type{i}`, where `i` corresponds to the boundary condition id number. For example, the ellipse has boundary id=3 (Fig. 4), so to set a Neumann boundary on u-velocity for the ellipse, you would ensure that `bcid2type_u{4}='N'`. Each of Pressure, u-velocity, v-velocity, and optionally density should have its own array. The boundary condition type is set using the `set_bcs` function.

Initially, all BC values are set to zero everywhere. Some of these boundary conditions may perhaps not be used in a specific example (for example, if density or temperature is not used), but it is still fine to initialize them.

```
[bcsDP, bcsOP, bcsNP] = init_bcs(nbcP);
[bcsDu, bcsOu, bcsNu] = init_bcs(nbcu);
[bcsDv, bcsOv, bcsNv] = init_bcs(nbcv);
[bcsDrho, bcsOrho, bcsNrho] = init_bcs(nbcrho);
```

Hence, the above set-to-zero lines should in general not be changed in the set-up part of the code.

Next, the user can modify these zero boundary conditions to the values that they wish to use. First we obtain the coordinates for the boundary points. Each of the XY arrays contain two columns, the first gives the x-coordinate, the second gives the y-coordinate associated with a particular boundary (D, O, or N).

The four sides of the domains are set-up directly, without the need of using boolean selections based on position arrays XP and YP, or XU and YU. However, for all masked regions, first, we re-select the region of the masks that needs to be assigned to a value, this is done as in the mask definition themselves. Then, a value is assigned to this. For example, to set the top half of the inlet boundary condition to 1, and the bottom half to 0.5, you would first ensure that `bcid2type_u{4} = 'Dmv'`, then:

```
%Get the boundary coordinates
[XYD, XYO, XYN] = get_bc_coord('u', XU, YU, Nodeu, bcsDu, bcsOu, bcsNu);
ids = (XYD(:, 1) == 0);
bcsDu(ids & XYD(:, 2) > 0.5) = 1;
bcsDu(ids & XYD(:, 2) <= 0.5) = 0.5;
```

For the masked regions in the interior, you can also add boundary values that vary in space. For example, to add a Dirichlet parabolic density profile to the ellipse, you would first ensure that `bcid2type_u{3} = 'Dmv'`, then:

```
[XYD, XYO, XYN] = get_bc_coord('P', XP, YP, NodeRho, bcsDrho, bcsOrho, bcsNrho);
ids = ellipse(XYD(:, 1), XYD(:, 2));
bcsDrho(ids) = (XYD(ids, 1) - mean(XYD(ids, 1))).^2;
```

We note that the script `set_bcs` ensures that all conventions needed for the code are followed. In most cases, the user, thus, does not need to worry about this: she/he can use his/her own mask numbers and `set_bcs` will reset them (it is only in some rare cases that it may need some updates). The conventions that the arrays `NodeP`, `Nodeu`, `Nodev`, and `NodeRho`, should follow once they come out of the script `set_bcs` are:

1. No Neumann boundary condition can have a number smaller than a Dirichlet or Open boundary condition. That is we require $\mathrm{BCid}_{\mathrm{Neumann}} > \mathrm{BCid}_{\mathrm{Dirichlet}}$.

2. No Open boundary condition can have a number smaller than a Dirichlet condition, or larger than a Neumann condition. That is we require $\mathrm{BCid}_{\mathrm{Neumann}} > \mathrm{BCid}_{\mathrm{Open}} > \mathrm{BCid}_{\mathrm{Dirichlet}}$.

3. No Dirichlet boundary condition can have a number higher than an Open or Neumann boundary condition. That is we require $\mathrm{BCid}_{\mathrm{Neumann}} > \mathrm{BCid}_{\mathrm{Open}} > \mathrm{BCid}_{\mathrm{Dirichlet}}$.

4. The number of boundary conditions for velocity, pressure, and density *have to be the same.*

5. The `bcsD` arrays need to include open boundary conditions, and the `bcsO` arrays need to indicate which of the Dirichlet boundaries are actually open boundaries.

6. Time-varying boundary conditions are implemented through the user-provided `updateBcs` function. If this function is not provided, boundary conditions will not be time-dependent. The function needs to have the following outputs, and take the following inputs:

```
[bcsDP, bcsNP, bcsDrho, bcsNrho, bcsDu, bcsNu,bcsDv, bcsNv]...
              = updateBcs(time, XP, YP, XU, YU, XV, YV, app, P, rho, u, v,...
                          NodeP, NodeRho, Nodeu, Nodev);
```

The rest of this section walks through how to create the SuddenExpansion and StokesForcing input files, including also the initial conditions.

4.5 StokesForcing test case:

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

$$u = \pi \sin(t) \sin(2\pi y) \sin^2(\pi x) \quad (4)$$

$$v = \pi \sin(t) \sin(2\pi x) \sin^2(\pi y) \quad (5)$$

$$P = \sin(t) \cos(\pi x) \sin(\pi y) \quad (6)$$

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

The solver function requires 3 inputs: `app,SetupScript,PlotScript`. The first input is a structure, where the fields `app.Nx, app.Ny, app.T` and `app.dt` define the discretization, giving the number of INTERIOR points in the x-direction, y-direction, the duration of the simulation, and the size of the time step, respectively. The value of the kinematic viscosity (which is analogous to the inverse Reynolds number for the full NS equations) is set through the `app.nu` field. The `app.PlotIntrvl` field is an integer number that indicates the number of time integrations steps between calling the script given by the string input `PlotScript`. The second 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 variables for 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 `app.PlotIntrvl` steps of the integration. While `PlotScript` is normally used for plotting the solution, it can also be used for saving the solution, and various other user-defined tasks. 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:

```
% Stokes Flow forcing function test case

%% Forcing Function
app.nu = 1;
Fu = @(time,p) ...
              pi.*cos(time+1).*sin(2.*pi.*p(:,2)).*sin(pi.*p(:,1)).^2-...
        2.*pi.^3.*sin(time+1).*sin(2.*pi.*p(:,2)).*cos(pi.*p(:,1)).^2+...
        6.*pi.^3.*sin(time+1).*sin(2.*pi.*p(:,2)).*sin(pi.*p(:,1)).^2-...
                  sin(time+1).*sin(pi.*p(:,1)).*pi.*sin(pi.*p(:,2));
Fv = @(time,p) ...
             -pi.*cos(time+1).*sin(2.*pi.*p(:,1)).*sin(pi.*p(:,2)).^2-...
        6.*pi.^3.*sin(time+1).*sin(2.*pi.*p(:,1)).*sin(pi.*p(:,2)).^2+...
        2.*pi.^3.*sin(time+1).*sin(2.*pi.*p(:,1)).*cos(pi.*p(:,2)).^2+...
                  sin(time+1).*cos(pi.*p(:,1)).*cos(pi.*p(:,2)).*pi;

%% Exact Solution (used in plotting function)
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)) ;

%% Set domain size
a    = 2;          b    = 2;
dx   = a./(app.Nx);    dy   = b./(app.Ny);

%% Set boundary conditions
bcsDP= [];         bcsNP= [0 0 0 0];
bcsDu= [0 0 0 0];  bcsNu= [];
bcsDv= [0 0 0 0];  bcsNv= [];
```

```
%% Set number of timesteps
Nt = app.T/app.dt;

%% create geometry mask
Nbcs = length(bcsDP) + length(bcsNP);
Mask = zeros(app.Ny, app.Nx);
%create Node matrices
[NodeP, Nodeu, Nodev, idsP, idsu, idsv] = NodePad(Mask,[1 1 1 1],1);


%% Initialize fields
%create vectors of coordinates for p, u, and v control volumes
xp = linspace(dx/2/a-0.5, 0.5-dx/2/a, app.Nx  )*a;
yp = fliplr(linspace(dy/2/b-0.5, 0.5-dy/2/b, app.Ny  ))*b;
xu = linspace( dx/a-0.5   ,0.5-dx/a , app.Nx-1)*a;
yu = fliplr(linspace(dy/2/b-0.5, 0.5-dy/2/b, app.Ny  ))*b;
xv = linspace(dx/2/a-0.5, 0.5-dx/2/a, app.Nx  )*a;
yv = fliplr(linspace(dy/b-0.5    , 0.5-dy/b, app.Ny-1))*b;

%create matrix of x and y coordinates
[XP YP] = meshgrid(xp,yp);
[XU YU] = meshgrid(xu,yu);
[XV YV] = meshgrid(xv,yv);

%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)])];
Pex = [bcsDP';bcsNP';pexact(0,[XP(idsP) YP(idsP)])];
uex = [bcsDu';bcsNu';uexact(0,[XU(idsu) YU(idsu)])];
vex = [bcsDv';bcsNv';vexact(0,[XV(idsv) YV(idsv)])];
```

4.5.1 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:

```
Fu = @(time,p) ...
              pi.*cos(time+1).*sin(2.*pi.*p(:,2)).*sin(pi.*p(:,1)).^2-...
        2.*pi.^3.*sin(time+1).*sin(2.*pi.*p(:,2)).*cos(pi.*p(:,1)).^2+...
        6.*pi.^3.*sin(time+1).*sin(2.*pi.*p(:,2)).*sin(pi.*p(:,1)).^2-...
                  sin(time+1).*sin(pi.*p(:,1)).*pi.*sin(pi.*p(:,2));
Fv = @(time,p) ...
             -pi.*cos(time+1).*sin(2.*pi.*p(:,1)).*sin(pi.*p(:,2)).^2-...
        6.*pi.^3.*sin(time+1).*sin(2.*pi.*p(:,1)).*sin(pi.*p(:,2)).^2+...
        2.*pi.^3.*sin(time+1).*sin(2.*pi.*p(:,1)).*cos(pi.*p(:,2)).^2+...
                  sin(time+1).*cos(pi.*p(:,1)).*cos(pi.*p(:,2)).*pi;
```

You will note we used "time+1" in the forcing functions, and this is only to start the simulation with a non-zero forcing.

4.5.2 Setting up the grid

The first step in creating the grid is setting the constant discretization sizes $\Delta x, \Delta y$, and number of time-integration steps Nt, which is accomplished simply as:

```
%% Set domain size
a    = 2;          b    = 2;
dx   = a./(app.Nx);    dy   = b./(app.Ny);

%% Set number of timesteps
Nt = app.T/app.dt;
```

since here the domain is 2×2 units long, and we are integrating for $T/\Delta t$ timesteps.

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 (that is no obstructions or holes in the interior of the domain), 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 control volumes (or nodes) in the x-direction, and Ny interior control volumes in the y-direction. The mask is only created for the *interior* control volume. 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:

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

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

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

4.5.3 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 (see §4.4). 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 from how 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= [];
```

4.5.4 Initial conditions

To create the initial conditions, first "coordinate" matrices, which contain the coordinates of the control volume centres, are created as follows:

```
%% Initialize fields
%create vectors of coordinates for p, u, and v control volumes
xp = linspace(dx/2/a-0.5, 0.5-dx/2/a, app.Nx  )*a;
yp = fliplr(linspace(dy/2/b-0.5, 0.5-dy/2/b, app.Ny  ))*b;
xu = linspace( dx/a-0.5   ,0.5-dx/a , app.Nx-1)*a;
yu = fliplr(linspace(dy/2/b-0.5, 0.5-dy/2/b, app.Ny  ))*b;
xv = linspace(dx/2/a-0.5, 0.5-dx/2/a, app.Nx  )*a;
yv = fliplr(linspace(dy/b-0.5    , 0.5-dy/b, app.Ny-1))*b;

%create matrix of x and y coordinates
[XP YP] = meshgrid(xp,yp);
[XU YU] = meshgrid(xu,yu);
[XV YV] = meshgrid(xv,yv);
```

note here we are using the MATLAB `meshgrid` function to create the matrices, and that these matrices 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 §4.2.

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)])];
```

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

4.6 Sudden Expansion

Note: This section is now a bit out-dated and could be updated for easier use with the new boundary condition interface. However, it still works as is, so we leave in the manual for now.

This example demonstrates the accuracy of a control-volume analysis compared to a Bernoulli analysis for predicting the pressure drop across a sudden expansion in a pipe. Students in MIT course 2.29 solve this problem analytically using the two approaches, and verify the answer numerically at a later point in the course. The Bernoulli approach is incorrect in this case, because the recirculation zones that develop violate the energy conservation principle upon which Bernoulli is based. This example ensures that 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 setup script for this example also shows how to create more complex geometries. This case can readily be modified for the classical flow over a cylinder test-case.

The Navier Stokes solvers require 3 inputs: `app,SetupScript` and `PlotScript`. The first input is a structure, where the fields `app.Nx, app.Ny, app.T` and `app.dt` define the discretization, giving the number of INTERIOR points in the x-direction, y-direction, the duration of the simulation, and the size of the time step, respectively. The value of the kinematic viscosity (which is analogous to the inverse Reynolds number) is set through the `app.nu` field, whereas the kinematic diffusivity for the density is set through `app.kappa`. The `app.PlotIntrvl` field is an integer number that indicates the number of time integrations steps between calling the script given by the string input `PlotScript`. The second input is the main focus of this section, 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 `app.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:

```
% Sudden Expansion test case

%# of interior CV's
%Check of app.Ny is a multiple of 3
if mod(app.Ny,3)
    fprintf('NOTE: app.Ny was not a multiple of 3, change app.Ny from %g to %g\n',...
        app.Ny,app.Ny-mod(app.Ny,3));
    app.Ny = app.Ny-mod(app.Ny,3);
end
if app.Ny/3<=2
   display('app.Ny needs to be at least 9 for this test case, app.Ny changed to 9.')
   app.Ny = 9;
end

%Define Forcing functions
Fu = @(time,p) 0;
Fv = @(time,p) 0;

%Set size of domain
a    = 20;          b    = 1;
dx   = a./(app.Nx);    dy   = b./(app.Ny);

%Set boundary conditions
bcsDrho = [0 0 0 0];    bcsNrho = [0];
bcsDP   = [0];          bcsNP = [0 0 0 0];
bcsOP   = [1];
bcsDu   = [0 1 0 0 0];  bcsNu= [];
bcsOu   = [5];
bcsDv   = [0 0 0 0];    bcsNv=  [0];

%Set number of timesteps
Nt = ceil(app.T/app.dt);

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

%create Node matrices
[NodeP, Nodeu, Nodev, idsP, idsu, idsv] = NodePad(Mask,bnd,1);
NodeRho=NodeP;
NodeP(find(NodeP==1))=-1;NodeP(find(NodeP==3))=1;NodeP(find(NodeP==-1))=3;
Nodeu(find(Nodeu==3))=-1;Nodeu(find(Nodeu==5))=3;Nodeu(find(Nodeu==-1))=5;
Nodev(find(Nodev==3))=-1;Nodev(find(Nodev==5))=3;Nodev(find(Nodev==-1))=5;
NodeRho(find(NodeRho==3))=-1;NodeRho(find(NodeRho==5))=3;NodeRho(find(NodeRho==-1))=5;
```

```
%% Initialize fields
xp = linspace(dx/2, a-dx/2, app.Nx  );yp = fliplr(linspace(dy/2, b-dy/2, app.Ny  ));
xu = linspace( dx   ,a-dx , app.Nx-1);yu = fliplr(linspace(dy/2, b-dy/2, app.Ny  ));
xv = linspace(dx/2, a-dx/2, app.Nx  );yv = fliplr(linspace(dy    , b-dy, app.Ny-1));
[XP YP] = meshgrid(xp,yp);[XU YU] = meshgrid(xu,yu);[XV YV] = meshgrid(xv,yv);
rhoinit=@(X,Y) 0*((X>0.5) - (X<0.5))*0.5;
pinit = @(X,Y) zeros(size(X));
uinit = @(X,Y) 1*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 to be divergence-free
u(Nodeu(2:end-1,round(app.Nx/5)+2:end-1))=1/3;
```

4.6.1 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;
```

4.6.2 Setting up the grid

The first step in creating the grid is setting the constant discretization sizes $\Delta x, \Delta y$, and number of time integration steps Nt, which is accomplished simply as:

```
%Set size of domain
a    = 20;          b    = 1;
dx   = a./(app.Nx);    dy   = b./(app.Ny);

%Set number of timesteps
Nt = ceil(app.T/app.dt);
```

since here the domain is 20×1 units long, and we are integrating for $T/\Delta t$ timesteps.

The Navier-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 (for example 1, in this case, see `NodePad.m` for more information) to the `NodePad` function as follows:

```
%% create geometry mask
Nbcs = length(bcsDP) + length(bcsNP);
bnd = [1 1 1 1];
Mask = zeros(app.Ny, app.Nx);
Mask(1:round(app.Ny/3),1:round(app.Nx/5))=1;
Mask(end-round(app.Ny/3-1):end,1:round(app.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 (see §4.4), 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 number
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`:

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

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

4.6.3 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 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. The reason for this boundary condition is that it is a very weak condition on the solution. It essentially requires that the second derivative vanishes as the exit, that is, no curvature.

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];
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];
```

4.6.4 Initial conditions

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

```
%% Initialize fields
xp = linspace(dx/2, a-dx/2, Nx  );yp = fliplr(linspace(dy/2, b-dy/2, Ny  ));
xu = linspace( dx   ,a-dx , Nx-1);yu = fliplr(linspace(dy/2, b-dy/2, Ny  ));
xv = linspace(dx/2, a-dx/2, Nx  );yv = fliplr(linspace(dy    , b-dy, Ny-1));
[XP YP] = meshgrid(xp,yp);[XU YU] = meshgrid(xu,yu);[XV YV] = meshgrid(xv,yv);
```

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 §4.2.

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.

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;
```

4.6.5 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)))`.

5 Discretizations: Schemes and Implementation

This section summarizes several of the discretizations utilized in the framework. Again, this is not an exhaustive manual, but should be sufficient for students who have been introduced to numerical fluid mechanics. In general the MATLAB functions are well-commented, and for further study, users are encouraged to read the code.

5.1 Advection Schemes

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

The Central, UPWIND and QUICK schemes for tracer advection and u, v-velocity advection are implemented.

The discrete finite volume problem is as follows:

$$\begin{aligned} \frac{\partial \phi_{x,y}}{\partial t}\Delta V &= \left\{u_{x-\Delta x/2,y}\hat{\phi}_{x-\Delta x/2,y} - u_{x+\Delta x/2,y}\hat{\phi}_{x+\Delta x/2,y}\right\} + \\ &\quad \left\{v_{x,y-\Delta y/2}\hat{\phi}_{x,y-\Delta y/2} - v_{x,y+\Delta y/2}\hat{\phi}_{x,y+\Delta y/2}\right\} \end{aligned} \tag{7}$$

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

$$\frac{\partial \phi_{x,y}}{\partial t}\Delta V = \left\{u_W\hat{\phi}_w - u_E\hat{\phi}_e\right\} + \left\{v_S\hat{\phi}_s - v_N\hat{\phi}_n\right\} \tag{8}$$

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.

5.1.1 Central scheme

The central flux scheme uses a linear interpolation to determine the value of the function at interfaces. The Central Difference Scheme (CDS) scheme is then given by:

$$\hat{\phi}_e = \frac{\phi_P - \phi_E}{2} \tag{9}$$

The CDS scheme is implemented in `SCAadvect_CDS_mex.m` and `UVadvect_CDS_mex.m`.

5.1.2 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:

$$\hat{\phi}_e = \begin{cases} \phi_P & u_E > 0 \\ \phi_E & u_E < 0 \end{cases} \tag{10}$$

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.

$$\hat{\phi}_e = \frac{1}{2}\left\{u_E(\phi_E + \phi_P) - |u_E|(\phi_E - \phi_P)\right\} \tag{11}$$

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

For the tracer 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.

5.1.3 QUICK scheme

The QUICK scheme interpolates the value of the function at the midpoint of the control surfaces using a quadratic interpolation. This is similar to the unstable central flux scheme which uses a linear interpolation, with the difference that an upwind bias is introduced. Since three points are required for the interpolation, two points are chosen from the upwind direction, and only one is chosen from the upwind direction. The QUICK scheme is then given by:

$$\hat{\phi}_e = \begin{cases} 6/8\phi_P - 1/8\phi_W + 3/8\phi_E & u_E > 0 \\ 6/8\phi_E - 1/8\phi_{EE} + 3/8\phi_P & u_E < 0 \end{cases} \tag{12}$$

Or using the naming convention upwind = U, up-upwind = UU, downwind = D:

$$\hat{\phi}_e = 6/8\phi_U - 1/8\phi_{UU} + 3/8\phi_D \tag{13}$$

The code is essentially the same as the upwind case. The major difference is that first the interpolated values are found. Then, instead of using the upwind and downwind values of the function given by the CV centers, the interpolated upwind and downwind values of the functions are used.

The exact same procedure is used for the u, v-velocity advection, except (again) care must be taken with the staggering of the control volumes. This scheme is implemented in `SCAadvect_mex.m` and `UVadvect_DO_mex.m`.

5.1.4 TVD scheme

The TVD scheme uses a mixture between the CDS and UW schemes. TVD stands for Total Variation Diminishing. Basically, what happens is the slope is "'limited"' to ensure that the maximum value of the function is never exceeded, and the minimum value of the function is never undershot.

We use a standard Total Variation Diminishing (TVD) scheme, with an monotonized central (MC) symmetric flux limiter (Van Leer, 1977). The scheme can be written for a variable η as:

$$\begin{aligned} F(\eta_{i-\frac{1}{2}}) = u_{i-\frac{1}{2}} & \frac{\eta_i + \eta_{i-1}}{2} \\ & - \left|u_{i-\frac{1}{2}}\right| \frac{\eta_i - \eta_{i-1}}{2} \left[1 - \left(1 - \left|u_{i-\frac{1}{2}} \frac{\Delta t}{\Delta x}\right|\right) \Psi(r_{i-\frac{1}{2}})\right], \end{aligned} \tag{14}$$

where the MC slope limiter $\Psi(r)$ is defined as

$$\Psi(r) = \max\left\{0, \min\left[\min\left(\frac{1+r}{2}, 2\right), 2r\right]\right\},$$

and the variable r as

$$r_{i-\frac{1}{2}} = \frac{\left[\frac{1}{2}\left(u_{i-\frac{1}{2}} + \left|u_{i-\frac{1}{2}}\right|\right)(\eta_{i-1} + \eta_{i-2}) + \frac{1}{2}\left(u_{i-\frac{1}{2}} - \left|u_{i-\frac{1}{2}}\right|\right)(\eta_{i+1} + \eta_i)\right]}{u_{i-\frac{1}{2}}(\eta_i - \eta_{i-1})},$$

where $u_{i-\frac{1}{2}}$ is used without interpolation for the density advection (due to the grid staggering) while a second-order central interpolation scheme is used for the non-linear u and v advection. For more information about TVD schemes, we refer to (Leveque, 2002).

The exact same procedure is used for the u, v-velocity advection, except (again) care must be taken with the staggering of the control volumes. This scheme is implemented in `SCAadvect_TVD_mex.m` and `UVadvect_TVD_mex.m`.

5.1.5 Advection schemes for Diagonally Orthogonal Implementation

For the implementation of the stochastic Diagonally Orthogonal schemes, we require more flexibility in the advection schemes. For this reason, the DO specific routines `UVadvect_DO.m`, `UVadvect_DO_UW.m` and `UVadvect_DO_CDS.m` are provided. Basically, functions allow for either of the following choices for the u advection.

$$u_i \frac{\partial u_j}{\partial x} + v_i \frac{\partial u_j}{\partial y} \tag{15}$$

$$u_j \frac{\partial u_i}{\partial x} + v_j \frac{\partial u_i}{\partial y} \tag{16}$$

5.1.6 Mexed Routines

To increase the efficiency of the solvers, it was found that the MATLAB implementations of the advection schemes was slow compared to a C++ implementation. Therefore, most of the advection schemes were coded in C, compiled, and linked to MATLAB (mexing).

Note, that you may have to compile the binaries yourself. This is a simple matter on a Linux or Mac system with a

```
>> mex filename.cpp
```

whereas on a Windows system, first an appropriate C++ library needs to be installed and configured to work with MATLAB. Refer to the Installation section for more details on this.

5.1.7 The CFL condition

The Courant-Friedrichs-Lewy is a stability condition on the solution on the solution of the advection operator when using an explicit time integration scheme. Mathematically speaking, this condition states that the numerical domain of dependence has to contain the physical domain of dependence. What this means, essentially, is that the maximum allowable numerical advection speed has to be greater than the actual advection speed, or

$$\frac{\Delta t}{\Delta x} C \leq 1 \tag{17}$$

where C is the physical advection speed.

If this condition is not satisfied, the numerical solution may become unstable. This means that if a simulation fails, it likely means that a smaller timestep, `dt`, is required.

5.2 Matrix Operators

To solve the Navier Stokes equations using the Projection Methods as used in this code, the Laplacian and Diffusion 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 Diffusion matrix is easily build from the Laplacian by adding a diagonal component, hence the major matrix that needs to be built and stored is the Laplacian matrix.

5.2.1 Gradient Operations

For solving the Navier Stokes and Stokes equations, only the gradient of the Pressure is required. This is implemented in `Grad2D.m`. A simple second order central difference (finite difference) scheme is used to find the gradient of the pressure:

$$\left.\frac{\partial \phi}{\partial x}\right|_{x+\frac{\Delta x}{2}} = \frac{\phi(x+\Delta x)-\phi(x)}{\Delta x} \tag{18}$$

$$\left.\frac{\partial \phi}{\partial y}\right|_{y+\frac{\Delta y}{2}} = \frac{\phi(y+\Delta y)-\phi(y)}{\Delta y} \tag{19}$$

The gradient is calculated for the entire domain, including any interior masked points, and then only the active interior point are selected and outputted.

5.2.2 Divergence Operations

For solving the Navier Stokes and Stokes equations, only the divergence of the velocity is required. This is implemented in `Div2D.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 Neumann 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.

5.2.3 Laplace Operator

The discrete version of the Laplacian operator ∇^2 is built using the `mk_DiffusionOperator` 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 `FixDiffuvDbcs`. Also, open boundary conditions are corrected for using `FixDiffPObcs` (open boundaries set the second derivative equal zero).

The second derivative is approximated using the following:

$$-\left\{\frac{\partial^2 \phi}{\partial x^2}+\frac{\partial^2 \phi}{\partial y^2}\right\}_{x,y} = \frac{-\phi(x+\delta_x,y)+2\phi(x,y)-\phi(x-\delta_x,y)}{\Delta x^2} + \frac{-\phi(x,y+\delta_y)+2\phi(x,y)-\phi(x,y-\delta_y)}{\Delta y^2} \tag{20}$$

note that the negative of this operator is implemented in the code.

5.3 Stokes Solvers

We use a projection method to solve the Stokes problem. That is, the time integration proceeds in three steps, and is as follows:

$$\left[\frac{\mathbf{I}}{\Delta t}-\nu\nabla^2\right]\tilde{\mathbf{u}}^{k+1} = \frac{\mathbf{u}^k}{\Delta t}+\mathbf{F}^{k+1} \tag{21}$$

$$\nabla^2 P^{k+1} = \frac{1}{\Delta t}\nabla\cdot\tilde{\mathbf{u}}^{k+1} \tag{22}$$

$$\mathbf{u}^{k+1} = \tilde{\mathbf{u}}^{k+1}-\Delta t\nabla P^{k+1} \tag{23}$$

where P in this case is only the "Pseudo-pressure," and does not actually solve the correct pressure. An improved algorithm is the Incremental Pressure correction scheme, and we use the rotational form:

$$\left[\frac{\mathbf{I}}{\Delta t} - \nu\nabla^2\right]\tilde{\mathbf{u}}^{k+1} = \frac{\mathbf{u}^k}{\Delta t} + \nabla P^k + \mathbf{F}^{k+1} \tag{24}$$

$$\nabla^2(q^{k+1}) = \frac{1}{\Delta t}\nabla\cdot\tilde{\mathbf{u}}^{k+1} \tag{25}$$

$$\mathbf{u}^{k+1} = \tilde{\mathbf{u}}^{k+1} - \Delta t\nabla q^{k+1} \tag{26}$$

$$P^{k+1} = q^{k+1} + P^k - \nu\nabla\cdot\tilde{\mathbf{u}}^{k+1} \tag{27}$$

For the Navier Stokes equations we treat the non-linear terms explicitly. That is, to solve the Navier Stokes equations, we set $\mathbf{F}^{k+1} \approx \mathbf{u}^k\cdot\nabla\mathbf{u}^k$. For an excellent review of Projection method, see Guermond and Minev (2006).

5.4 Dynamical Orthogonality condition stochastic solvers

In the DO representation, the velocity field is decomposed into a finite sum

$$\mathbf{u} = \bar{\mathbf{u}} + \sum_{i=1}^{S} Y_i\mathbf{u}_i \tag{28}$$

where $\bar{\mathbf{u}}$ is the mean field, $\mathbf{u}_i$ are modes such that $\int_\Omega \mathbf{u}_i\mathbf{u}_j d\Omega = \delta_{ij}$, and Y_i are zero-mean stochastic coefficients, that is $\mathbb{E}[Y_i] = 0$. The novelty with the DO approach is that both the stochastic coefficients, and the modes are evolving in time. With other methods, either the coefficients or the modes are normally fixed. This allows for a very efficient representation of stochastic information. For further details, please refer to Sapsis and Lermusiaux (2009).

6 Troubleshooting

1. I get an error that MATLAB cannot find certain files?
   Did you remember to run setup at the start?
2. I changed `app.Nx` and `app.Ny` when running one of the examples in `TestCase`, but now my solution is wrong/blows up, and I get NaNs. What's going on?
   You are probably violating the CFL condition (See Section §5.1.7). Try setting `app.dt`, and keep decreasing the value until your solution is no longer wrong/blows up.
3. My Boundary conditions aren't working? What's going on?
   Did you carefully read the rules concerning Boundary conditions? See §4.4.
4. None of the above worked? What do I do now?
   Make sure that you are not inputting any non-physical parameter (for example, a negative dt). Check over your code, and try a few more things, but if you are stuck, please try the forum or submit a bug report on `http://mseas.mit.edu/codes`.

References

Guermond, J. and Minev, P. (2006). An overview of projection methods for incompressible flow. *Comput. Methods Appl. Mech. Engrg.*, 195:6011–6045.

Leveque, R. J. (2002). *Finite Volume Methods for Hyperbolic Problems.* Cambridge University Press.

Sapsis, T. and Lermusiaux, P. (2009). Dynamically orthogonal field equations for continuous stochastic dynamical systems. *Physica D*, 238:2347–2360.

Sapsis, T. P. and Lermusiaux, P. F. J. (2011). Dynamical criteria for the evolution of the stochastic dimensionality in flows with uncertainty. *Physica D*, In. Press.

Van Leer, B. (1977). Towards the ultimate conservative difference scheme. IV. A new approach to numerical convection. *J. Comput. Phys.*, 23(3):276 – 299.