

SOLVING PDES WITH FORTRAN

ELLIOT MARSHALL

ABSTRACT. The purpose of these notes is to give a simple introduction to numerically solving PDEs using Fortran. Since I am very new to writing code in Fortran, it is likely these notes will contain errors or do things in a suboptimal way! Right now, it is mainly just to remind myself of how to setup and run simple codes.

CONTENTS

1. Why Fortran?	1
2. Simple Bash Commands	1
3. My First Fortran Programs	3
4. Makefiles	4
4.1. Makefiles for Compiling	6
5. Fortran Features	7
5.1. Floating Point Precision	7
5.2. Arrays	7
5.3. Saving to Files	8
6. A Simple ODE Solver	9
7. Advection Equation Solver	12
8. Namelists	19
References	22

1. WHY FORTRAN?

If you are like me (and many other mathematicians), you have probably learnt to code in a dynamically typed language such as Matlab or Python. While these languages are simple to learn and have a lot of nice features (e.g. standardised packages such as numpy) they are not particularly fast. This particularly becomes an issue when attempting to solve large systems of equations, such as the EFEs, in more than one spatial dimension. In this case, it is often necessary to use statically typed languages, such as C++ and Fortran. I have chosen to use Fortran as many of the resources I regularly use, such as Randall LeVeque's textbook on finite volume methods [2], describe algorithms in Fortran. Additionally, there are many existing PDE packages written in Fortran, such as Clawpack [1], which I find useful to look at when learning how to implement a new PDE method or algorithm.

This document is being written as I learn Fortran and likely contains many errors and foolish ways of implementing things. If you find errors, typos etc. let me know by emailing: elliott.marshall@monash.edu.

2. SIMPLE BASH COMMANDS

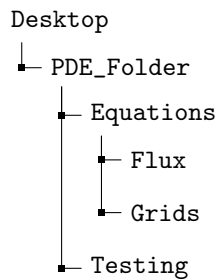
All the instructions in this section are intended to run on a Mac using Bash. They might work for Unix systems more generally, but I don't know enough about this to be sure

To begin, let us recall some simple bash commands for using the terminal.

- `cd` - Changes the current directory for the terminal.
- `cd ..` - Move one directory up
- `ls` - Lists all files in the current directory.
- `pwd` - Prints the current directory (i.e. the file path).
- `echo` - Prints text to the terminal
- `touch` - Create a file
- `mv` - Move a file
- `rm` - Remove a file
- `mkdir` - Create a new folder

For example, suppose we have the folder layout shown in Figure 1.

FIGURE 1. An example folder structure.



From the home directory, I could navigate to `PDE_Folder` by typing `cd Desktop/PDE_Folder` into the terminal. If I then type `ls`, we get

```

1 MacBookPro:PDE_Folder elliotmarshall$ ls
2 Equations Testing

```

I could then navigate to the `Testing` folder and create a text file by typing

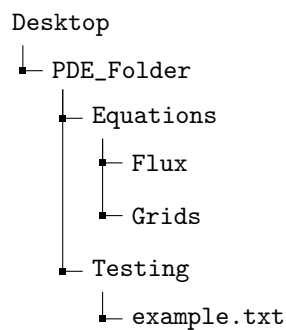
```

1 MacBookPro:PDE_Folder elliotmarshall$ cd Testing
2 MacBookPro:Testing elliotmarshall$ touch example.txt

```

My folder structure would now be given by Figure 2.

FIGURE 2. The modified folder structure.



It is straightforward to play around with the other commands and see how they work. Let's now create a project folder for our first Fortran project. To begin, we will create the folder `Example_Project` and navigate to it¹

```

1 MacBookPro:~ elliotmarshall$ cd Desktop
2 MacBookPro:Desktop elliotmarshall$ mkdir -p Example_Project
3 MacBookPro:Desktop elliotmarshall$ cd Example_Project

```

¹The `-p` flag after `mkdir` creates any necessary parent directories if they do not already exist.

We can now create an empty Fortran file by typing

```
1 MacBookPro:Example_Project elliotmarshall$ touch hello.f90
```

3. MY FIRST FORTRAN PROGRAMS

Let's now create our first Fortran program. Inside our hello.f90 file, we write

```
1 program hello
2   print *, 'hello'
3 end program hello
```

Unlike Python or Matlab, a Fortran file must be compiled before we can run it. We will use the gfortran compiler. In the terminal, we write

```
1 MacBookPro:Example_Project elliotmarshall$ gfortran -o Hello hello.f90
```

This creates an executable file Hello which we can run using

```
1 MacBookPro:Example_Project elliotmarshall$ ./Hello
```

Let's consider an example which calculates the area of a circle with radius 2. We will create a file maths.f90 with the following program

```
1 program maths
2   implicit none
3   real (kind = 8) :: pi, r, area
4
5   ! compute pi as arc-cosine of -1:
6   pi = acos(-1.d0) ! need -1.d0 for full precision!
7   r = 2.0
8   area = pi*r**2
9
10  print *, "Pi = ", pi
11  print *, "The area of the circle is ", area
12 end program maths
```

Compile and run the program using

```
1 $ gfortran -o maths maths.f90
2 $ ./maths
```

We get the output

```
1 Pi =      3.1415926535897931
2 The area of the circle is      12.566370614359172
```

Some remarks about the structure of this program:

- We used `implicit none` at the start of our program. This means any variable we use in the program has to be explicitly declared.
- In line 3, we declared the variables `pi`, `r`, and `area` to be real floating point numbers with 8 bits of precision. This corresponds (**Roughly true? Compiler dependent...**) to double precision. We will discuss a better way of specifying floating point precision later on.
- Comments in Fortran are denoted by `!`
- While Fortran contains many built-in functions, it does not contain π . In line 6, we define it using the built-in arc-cos function to double precision.
- Fortran is **not case-sensitive!** For example `pi` and `Pi` would refer to the same variable!

Often, we will want to make our code modular with individual files containing only a few functions or subroutines. Let's now create a file `area.f90` with a module which contains a function to calculate the area,

```

1 module compute_area
2   implicit none
3   real (kind=8), parameter :: pi = acos(-1.d0)
4   public pi, area
5   private
6
7   contains
8
9   real (kind=8) function area(r)
10    real (kind=8), intent(in) :: r
11
12    area = pi*r**2
13
14  end function area
15
16 end module compute_area

```

Now, we re-write `maths.f90` to use this module

```

1 program maths
2   use compute_area, only: area, pi
3   implicit none
4   real (kind = 8) :: r
5
6   r = 2.0
7
8   print *, "Pi = ", pi
9   print *, "The area of the circle is ", area(r)
10
11 end program maths

```

In order to create the executable, we have to tell the compiler to link the `area.f90` and `maths.f90` files:

```

1 $ gfortran -o maths area.f90 maths.f90

```

Some remarks:

- In the `compute_area` module we declared `pi` to be a parameter. This means that `pi` is simply an alias for the (fixed) numerical value which we can now use throughout the program.
- We explicitly stated which parts of the module are `public` (available to use in other parts of the program) and `private` (only available to use inside the module).
- The function `area` is declared with a type. Its output is determined by setting the name of the function (in this case `area`) somewhere in the body of the function.
- The `intent(in)` statement tells the compiler that the variable `r` will not be modified by the function.
- We import the parameter `pi` and the function `area` into `maths.f90` using the `use` and `only` statements. This is similar to import statements in Python.

4. MAKEFILES

In the previous example, we needed to tell the compiler that both `area.f90` and `maths.f90` were required to create the program. Clearly this method of compiling is untenable for a large project using many files. Moreover, we don't want to recompile the whole project if we have only changed one file. We can avoid the latter by first compiling our Fortran files into *object* files. We then *link* the object files to create our executable, e.g.

```

1 $ gfortran -o area.o -c area.f90
2 $ gfortran -o maths.o -c maths.f90
3 $ gfortran -o maths maths.o area.o

```

This is better, but not much. In order to make this process more efficient, we can use **make**.

make is a separate programming language which allows us to automate the compiling process. First, we create a file in the working directory with the name **Makefile**,

```
1 $ touch Makefile
```

Now, inside this file we can write

```
1 VAR=hello
2 default:
3     echo ${VAR}
```

It is important to note that the line `echo $VAR` is indented using a tab. Now, if we type **make** into the terminal this prints out

```
1 echo hello
2 hello
```

That is, it prints out the bash command and then the result of that command. Makefiles have two key features, *targets* and *dependencies*. In the previous example **default** was a target and we had no dependencies. Let's consider another example,

```
1 VAR=hello
2 VAR2=goodbye
3 target1:
4     @echo ${VAR}
5 target2:
6     @echo ${VAR}
```

Here we have two targets, **target1** and **target2**. The `@` symbol stops the bash command itself being printed. If we run

```
1 $ make target1
```

the output is

```
1 hello
```

Similarly, if we run

```
1 $ make target2
```

the output is

```
1 goodbye
```

If we just used the command **make** without specifying a target, then the first target is run by default. Dependencies come after the colon of a target. A dependency can be another target or a file, for example

```
1 VAR=hello
2 VAR2=goodbye
3 target1:
4     @echo ${VAR}
5 target2: target1
6     @echo ${VAR}
```

If we now run

```
1 $ make target2
```

the output is

```
1 hello
2 goodbye
```

So **target2** runs its dependencies first before running its own commands.

4.1. Makefiles for Compiling. Ultimately, the **Makefile** sends a set of instructions to the terminal to be executed. A simple **Makefile** to compile and run the **maths** program might be

```
1 main.exe: area.o maths.o
2   gfortran -o maths maths.o area.o
3 area.o:
4   gfortran -o area.o -c area.f90
5 maths.o:
6   gfortran -o maths.o -c maths.f90
```

which we could run by simply typing **make** into the terminal. We can simplify this process by defining variables for example

```
1 FC = gfortran
2 FFLAGS = -O3 # This is an optimisation flag
3 SRC = area.o maths.o
4
5 maths: $(SRC)
6   $(FC) -o maths $(SRC)
7 %.o : %.f90
8   $(FC) $(FFLAGS) -o $@ -c $<
9 clean:
10  rm *.o *.mod
```

Line 7 is called a wildcard rule. It tells **make** how to construct an object file from the corresponding **.f90** file. The target **clean** removes any object and mod files after compilation. We can run it by typing **make clean** into the terminal. If we run the **Makefile** again without changing any of the source files, we get the message

```
1 make: 'maths' is up to date.
```

What would happen if we swapped the order of **area.o** and **maths.o** in the **SRC** declaration? We get the following error

```
1 gfortran -O3 -c maths.f90
2 maths.f90:2:9:
3
4   2 |      use compute_area, only: area, pi
5     |      1
6 Fatal Error: Cannot open module file 'compute_area.mod' for reading at (1): No such
   file or directory
7 compilation terminated.
```

This is because **maths.f90** depends on **area.f90**, so if we try to compile **maths.f90** before **area.f90** we get an error! One option is to explicitly specify the dependencies in the make file,

```
1 FC = gfortran
2 FFLAGS = -O3 # This is an optimisation flag
3 SRC = maths.o area.o
4
5 maths: $(SRC)
6   $(FC) $(FFLAGS) -o maths $(SRC)
7 %.o : %.f90
8   $(FC) $(FFLAGS) -o $@ -c $<
9 maths.o: area.o
10 clean:
11  rm *.o *.mod
```

This ensures that **area.f90** is compiled first. There are more advanced ways to automatically declare dependencies but we will not go into this further here.

5. FORTRAN FEATURES

5.1. Floating Point Precision. In Fortran, for floating point numbers, we have to specify the level of precision otherwise single precision is used by default. In our previous examples we did this by using `(kind=8)`. However this is compiler dependent and does not necessarily correspond to double precision. Instead we will create a module to define the floating point precision.

```

1 ! This module sets up the floating point precision used.
2 module real_type_mod
3
4 ! sp contains the kind value for single precision.
5   integer, parameter :: sp = kind(1.0)
6
7 ! dp contains the kind value for double precision.
8   integer, parameter :: dp = kind(1.0d0)
9 end module real_type_mod

```

We can then import this module to our other files to define the required floating point precision.

5.2. Arrays. When solving PDEs numerically, we almost exclusively work with arrays. Let's define a module to do some simple array operations,

```

1 module arrays
2   implicit none
3
4   contains
5
6   subroutine array_operations()
7     use real_type_mod, only: dp
8     real(dp) :: x(3)
9     real(dp) :: z(2,2), y(2,2)
10
11     x = 1.0_dp ! Set all entries of x to 1.0
12     print *, "x is", x
13
14     x(1) = 0.62_dp ! Set the first entry to 0.62
15
16     print *, "The updated x is", x
17
18     z = 1.0_dp ! Set all entries of z to 1.0
19
20     print *, "z is ", z
21
22     ! Set the first row of y to be -0.5 and the second to 0.5
23     y(1:2,1) = -0.5_dp
24     y(1:2,2) = 0.5_dp
25
26     print *, "y is ", y
27
28     ! Add y and z elementwise
29     y = y + z
30
31     print *, "y+z is ", y
32
33   end subroutine array_operations
34 end module arrays

```

Observe that we use our module `real_type_mod` to define the floating precision. We can then call the `array_operations` subroutine from a new file called `main`

```

1 program main
2   use arrays, only: array_operations

```

```

3  implicit none
4
5  call array_operations()
6
7  end program main

```

Our Makefile becomes

```

1  FC = gfortran
2  FFLAGS = -O3 # This is an optimisation flag
3  SRC = real_type_mod.o arrays.o main.o
4
5  main: $(SRC)
6  $(FC) $(FFLAGS) -o main $(SRC)
7  %.o : %.f90
8  $(FC) $(FFLAGS) -o $@ -c $<
9  clean:
10 rm *.o *.mod

```

Some comments:

- Indexing is 1 based (Python is zero based)
- Slicing is inclusive, e.g. `a(1:5)` runs over the indices 1 to 5 *including* index 5.
- As in numpy, array operations act *element-wise*.
- Unlike functions, we must write `call` in front of subroutine calls.

5.3. Saving to Files. We can write to and read from files using the `open`, `close`, `write`, and `read` commands. This example saves some data to a text file and then reads the data from that file

```

1  program io
2      use real_type_mod, only: dp
3      implicit none
4      character(len=20) :: filename
5      real(dp) :: x,y
6
7      x = 1.0_dp
8      y = 12.0_dp
9
10     filename = 'result.txt'
11
12     ! Write to a text file
13     open(unit=1,file=filename,status='replace',form='formatted')
14     write(1,*) x,y
15     close(1)
16
17     ! Reset the variables
18     x = 0.0_dp
19     y = 0.0_dp
20
21     ! Read in from text file
22     open(unit=2,file=filename,status='old')
23     read(2,*) x,y
24     close(2)
25
26     ! Print the variables
27     print *, 'x = ', x, 'y = ', y
28
29 end program io

```

To do: Figure out how to save to HDF5 files...

6. A SIMPLE ODE SOLVER

Before we move on to PDEs, a natural first step is to build an ODE solver. In particular, if we discretise PDEs using the method of lines we should be able to re-use some of the time stepping routines. As a simple example, consider the autonomous ODE

$$\begin{aligned} y'(t) &= 2y, \\ y(t_0) &= y_0 \end{aligned} \tag{6.1} \text{eqn:ode_example}$$

which has the solution

$$y(t) = y_0 e^{2t}.$$

A standard way to numerically solve ODEs is with Runge-Kutta multistep methods. In this case we will use a 2nd order algorithm called *Heun's method*,

$$\begin{aligned} y^{n+1} &= y^n + \frac{\Delta t}{2}(k_1 + k_2), \\ k_1 &:= f(t^n, y^n), \\ k_2 &:= f(t^n + \Delta t, y^n + \Delta t k_1), \end{aligned}$$

where Δt is our time step size. A simple python script to implement this would be

```
1 import numpy as np
2 ### Heun's Method
3 def rk2(y0,t0,dt,rhs):
4     k1 = rhs(t0,y0)
5     k2 = rhs(t0 + dt, y0 + dt*k1)
6     y0 = y0 + (dt/2)*(k1 + k2)
7     return y0
8
9 ### Exponential Growth/Decay ODE
10 def ODE(t,y):
11     dydt = 2*y
12     return dydt
13
14 ### Define initial parameters
15 t0 = 0.0
16 tend = 5.0
17 dt = 0.01
18 y0 = 0.01
19
20 ### Create list to store solution
21 y_output = []
22 y_output.append(y0)
23 t_output = []
24 t_output.append(t0)
25
26 ### Main integration loop
27 while t0 < tend:
28     y0 = rk2(y0,t0,dt,ODE)
29     t0 += dt
30     y_output.append(y0)
31     t_output.append(t0)
32
33 ### Save solution as a text file
34 y_soln = np.array(y_output) # Convert list to numpy arrays
35 t_soln = np.array(t_output)
36 np.savetxt('python_soln.txt', y_soln)
37 np.savetxt('python_soln_times.txt', t_soln)
```

We can save this file as `python_version.py` and run it using

```
1 $ python3 -0 python_version.py
```

It is straightforward to generalise this script to work for systems of ODEs. Additionally, we could improve the efficiency by exclusively working with numpy arrays instead of native python. Let's now look at a Fortran implementation. First, we will write a module with the RK2 integrator

```
1 module runge_kutta_2
2   use real_type_mod, only: dp
3   implicit none
4
5   ! First we construct an abstract interface
6   ! This allows us to pass arbitrary ODEs to
7   ! the Runge-Kutta method
8   abstract interface
9     subroutine rhs_interface(t, y, neqn, dydt)
10      import :: dp
11      integer :: neqn ! The number of eqns to solve
12      real(dp), intent(in) :: t, y(neqn)
13      real(dp), intent(out):: dydt(neqn)
14    end subroutine rhs_interface
15  end interface
16
17  contains
18
19  ! 2nd order accurate Runge-Kutta method (Heun's method)
20  subroutine rk2(t0,y0,dt,neqn,rhs)
21    integer, intent(in) :: neqn ! Num of eqns to solve
22    real(dp), intent(in) :: t0,dt ! Current time, timestep
23    real(dp), intent(inout) :: y0(neqn) ! Current values
24    real(dp) :: k1(neqn), k2(neqn) ! Runge-Kutta Substeps
25    procedure(rhs_interface) :: rhs
26
27    call rhs(t0,y0,neqn,k1)
28    call rhs(t0 + 0.5_dp*dt, y0 + dt*k1, neqn, k2)
29
30    y0 = y0 + dt*(0.5_dp*k1 + 0.5_dp*k2)
31
32  end subroutine rk2
33
34 end module runge_kutta_2
```

Clearly this is already more involved than the python version. In python, we can pass the rhs function as an argument directly. In order to do this in Fortran, we must first use an interface block. This block tells the compiler the types of the inputs and outputs of the rhs function². The RK2 subroutine itself is actually very similar. First, we declare the types of the input and output variables. Notice that y0 has intent(inout). This means it is both an input and output of the subroutine. We also declare the rhs input using the previously defined interface block. Now, let's look at our main program file

```
1 program ode_solve
2   use runge_kutta_2, only: rk2
3   use real_type_mod, only: dp
4   implicit none
5   integer :: num_dims
6   real(dp), allocatable :: y0(:)
7   real(dp) :: t0, tend, dt
8   character(len=30) :: filename_solution, filename_times
9
```

²We have also passed an additional input neqn for the number of equations. This means our Fortran code can be easily modified to work for systems of equations.

```

10      ! Set up files for outputting data
11      filename_solution = "fortran_soln.txt"
12      filename_times = "fortran_soln_times.txt"
13
14      ! Number of equations to solve
15      num_dims = 1
16
17      ! Set simulation parameters
18      allocate(y0(num_dims)) ! Set the dimension of the initial data vector
19      y0 = 0.01_dp
20      t0 = 0.0_dp
21      tend = 5.0_dp
22      dt = 0.01_dp
23
24      ! Save initial data values
25      open(unit=10, file=filename_solution, status="replace", action="write")
26      write(10, *) y0
27
28      open(unit=11, file=filename_times, status="replace", action="write")
29      write(11, *) t0
30
31      ! Main rk2 loop
32      do while (t0 < tend)
33          call rk2(t0,y0,dt,num_dims,ODE)
34          t0 = t0 + dt
35
36          ! write solution data to txt files
37          write(10, *) y0
38          write(11, *) t0
39
40      end do
41      close(10)
42      close(11)
43
44      contains
45
46      ! Exponential growth/decay ODE
47      ! Solution is exp(2*t)
48      subroutine ODE(t,y,neqn,dydt)
49          implicit none
50          integer, intent(in) :: neqn
51          real(dp), intent(in) :: t, y(neqn)
52          real(dp), intent(out) :: dydt(neqn)
53
54          dydt = 2*y
55
56      end subroutine ODE
57
58      end program ode_solve

```

This essentially follows the structure of the python code. One difference to note is that we declare `y0` to be an allocatable vector. This means the dimension of `y0` is declared later in the program using the statement `allocate(y0(num_dims))`. This lets us change the size of `y0` by simply changing the value of `num_dims`. For example, if we had two equations to solve, we could change `num_dims` to 2 and then `y0` would automatically be assigned the correct shape. For completeness, here is the Makefile

```

1 FC = gfortran
2 FFLAGS = -O3 # This is an optimisation flag

```

```

3 SRC = real_type_mod.o rk2.o ode_solve.o
4
5 ode_solve: $(SRC)
6     $(FC) $(FFLAGS) -o $@ $(SRC)
7     %.o : %.f90
8     $(FC) $(FFLAGS) -o $@ -c $<
9 clean:
10     rm *.o *.mod

```

Now, let's compare the output of the two programs. We can plot the solutions using a simple python script

```

1 import numpy as np
2 from matplotlib import pyplot as plt
3
4 python_soln = np.loadtxt('python_soln.txt')
5 python_times = np.loadtxt('python_soln_times.txt')
6
7 fortran_soln = np.loadtxt('fortran_soln.txt')
8 fortran_times = np.loadtxt('fortran_soln_times.txt')
9
10 ### Plot outputs against the exact solution
11 plt.plot(python_times,python_soln, 'x',label='Python Solution')
12 plt.plot(fortran_times,fortran_soln, '.',label='Fortran Solution')
13 plt.plot(fortran_times, 0.01*np.exp(2*fortran_times), label='Exact Solution')
14 plt.xlabel("Time")
15 plt.ylabel("Solution")
16 plt.legend()
17 plt.show()

```

We get the following output:

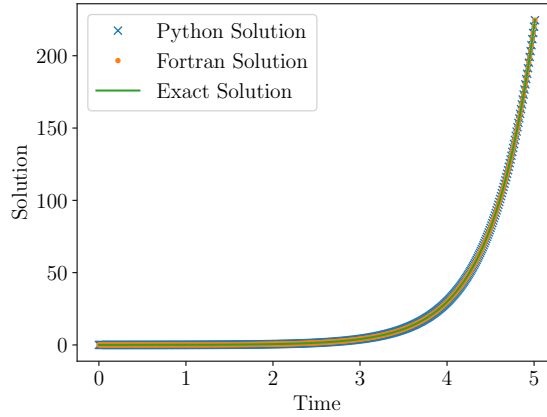


FIGURE 3. Solution of (6.1) using our Python and Fortran routines

_Python_ODE_Compare)?

It is important to note that the Fortran code is not remotely as optimised as it could be. For example, it would be faster³ to pre-allocate arrays for the outputs and only save to a text file once at the end of the program.

7. ADVECTION EQUATION SOLVER

Let's now look at solving at the advection equation,

$$\partial_t u + \partial_x u = 0,$$

³Probably? I only tested speed comparisons without saving to files. In this case Fortran is significantly faster.

$$u(0, x) = \sin(x)$$

on the domain $x \in [0, 2\pi]$ with periodic boundary conditions. We will use centred fourth-order finite differences to discretise our spatial derivatives and a fourth-order Runge-Kutta method to evolve in time. First, we will build a module for the finite difference operator,

```

1  ! Module containing the finite difference operations
2  module finite_difference
3      use real_type_mod, only: dp
4      implicit none
5
6      contains
7
8      subroutine FD_C4(dx, Nx, Ngz, u, du)
9          ! Inputs:
10         ! dx - The spatial step size, real(dp)
11         ! Nx - The number of grid points, integer
12         ! Ngz - The number of ghost points, integer
13         ! u - The function to be differentiated, real(dp) dimension(1,1-Ngz:Nx+Ngz)
14         ! Output:
15         ! du - The derivative, real(dp) dimension(1,1-Ngz:Nx+Ngz)
16         implicit none
17         real(dp), intent(in) :: dx
18         integer, intent(in) :: Nx, Ngz
19         real(dp), intent(in) :: u(1-Ngz:Nx+Ngz,1)
20         real(dp), intent(out) :: du(1-Ngz:Nx+Ngz,1)
21         ! integer :: i
22
23         ! ### Loop Version
24         ! do i = 1, Nx
25         !     du(1,i) = (u(1,i-2) - 8.0_dp*u(1,i-1) + 8.0_dp*u(1,i+1) &
26         !         - u(1, i+2))/(12.0_dp*dx)
27         ! end do
28
29         ! ### Slicing Version
30         du(1:Nx,:) = (u(-1:Nx-2,:) - 8.0_dp*u(0:Nx-1,:) + 8.0_dp*u(2:Nx+1,:) &
31             - u(3:Nx+2,:))/(12.0_dp*dx)
32
33
34     end subroutine FD_C4
35
36 end module finite_difference

```

There are a few things to note about this program. First we declare our arrays to have size $(1-Ngz:Nx+Ngz,1)$. This means our arrays start at index $1-Ngz$. For example, in the case $Ngz = 2$, our arrays would start at index -1 . This is different from Python, where arrays always start at index 0 .

Additionally, we choose our first dimension to be the number of grid points. This is because arrays in Fortran are stored in *column-major order* which means it is fastest to loop over columns. This is somewhat irrelevant in this example, since we only have 1 equation, but it would become important if we had multiple equations or the spatial dimension was greater than 1. Finally, we note that we have two implementations: one using loops and one using vectorised slicing operations. In principle, the vectorised operations should be faster but I have not noticed a difference in this example.

Remark 7.1. The $\&$ symbol is used to break long lines of code in Fortran.

Now, let's write a module to compute the right-hand side of the evolution equation,

```

1  ! Module containing the advection equation subroutine
2  module advection
3      use real_type_mod, only: dp
4      use finite_difference, only: FD_C4
5      implicit none
6
7      contains
8
9      subroutine compute_rhs(Nx,Ngz,dx,t,u,dtu)
10         implicit none
11         integer, intent(in) :: Nx, Ngz
12         real(dp), intent(in) :: t, dx
13         real(dp), intent(in) :: u(1-Ngz:Nx+Ngz,1)
14         real(dp) :: du(1-Ngz:Nx+Ngz,1)
15         real(dp), intent(out) :: dtu(1-Ngz:Nx+Ngz,1)
16
17
18         call FD_C4(dx,Nx,Ngz,u,du) ! Compute the derivative
19
20         dtu = -1.0_dp*du
21
22
23
24     end subroutine compute_rhs
25
26 end module advection

```

Since we have used a subroutine to calculate the derivative, we must call `FD_C4` before updating the evolution equation. However, since this routine has only one output it might be simpler to replace it with a function instead.

At each time step, we will need to update the boundary conditions. Here, it does make sense to modify the solution in place using a subroutine and the `intent(inout)` declaration.

```

1  ! Module containing the boundary conditions
2  module boundary_conditions
3      use real_type_mod, only: dp
4      implicit none
5
6      contains
7
8      subroutine periodic_bc(Nx,Ngz,u)
9          ! Inputs:
10         ! dx - The spatial step size, real(dp)
11         ! Nx - The number of grid points, integer
12         ! u - The function we apply the BCs to, real(dp) dimension(1,1-Ngz:Nx+Ngz)
13         ! Output:
14         ! u - The function with updated boundaries, real(dp) dimension(1,1-Ngz:Nx+
15         Ngz)
16         implicit none
17         integer, intent(in) :: Nx, Ngz
18         real(dp), intent(inout) :: u(1-Ngz:Nx+Ngz,1)
19         ! integer :: i
20
21         u(Nx,1) = u(1,1) ! First and last grid points are identified
22
23         ! Right Boundary
24         u(Nx+1:Nx+Ngz,1) = u(1+1:1+Ngz,1)

```

```

25      ! Left Boundary
26      u(1-Ngz:0,1) = u(Nx-Ngz:Nx-1,1)
27
28      ! ### Loop version
29      ! do i = 1,Ngz
30
31      !      ! Right Boundary
32      !      u(1,Nx+i) = u(1,1+i)
33
34      !      ! Left Boundary
35      !      u(1,1-i) = u(1,Nx-i)
36
37      ! end do
38
39      end subroutine periodic_bc
40
41 end module boundary_conditions

```

Once again, we can implement this with either slicing or a loop. Now, let's look at our Runge-Kutta routine.

```

1      ! Module containing the time stepping routines
2 module time_stepping
3     use real_type_mod, only: dp
4     use advection, only: compute_rhs
5     use boundary_conditions, only: periodic_bc
6
7
8     contains
9
10    subroutine rk4(Nx,Ngz,dx,t0,y0,dt)
11        integer, intent(in) :: Nx,Ngz ! Num of grid/ghost points
12        real(dp), intent(in) :: t0,dt, dx ! Current time, time step, spatial step
13        real(dp), intent(inout) :: y0(1-Ngz:Nx+Ngz,1) ! Current values
14        real(dp) :: k1(1-Ngz:Nx+Ngz,1), k2(1-Ngz:Nx+Ngz,1), k3(1-Ngz:Nx+Ngz,1), &
15            k4(1-Ngz:Nx+Ngz,1) ! Runge-Kutta Substeps
16        real(dp) :: y1(1-Ngz:Nx+Ngz,1), y2(1-Ngz:Nx+Ngz,1), y3(1-Ngz:Nx+Ngz,1)
17
18        ! Step 1
19        call compute_rhs(Nx,Ngz,dx,t0,y0,k1)
20        y1 = y0 + 0.5*dt*k1
21        call periodic_bc(Nx,Ngz,y1) ! Update Boundary Conditions
22
23        ! Step 2
24        call compute_rhs(Nx,Ngz,dx,t0+0.5_dp*dt,y1,k2)
25        y2 = y0 + 0.5*dt*k2
26        call periodic_bc(Nx,Ngz,y2)
27
28        ! Step 3
29        call compute_rhs(Nx,Ngz,dx,t0+0.5_dp*dt,y2,k3)
30        y3 = y0 + dt*k3
31        call periodic_bc(Nx,Ngz,y3)
32
33        ! Step 4
34        call compute_rhs(Nx,Ngz,dx,t0+dt,y3,k4)
35
36        ! Update y0
37        y0 = y0 + dt*((1.0_dp/6.0_dp)*k1 + (1.0_dp/3.0_dp)*k2 &
38            + (1.0_dp/3.0_dp)*k3 + (1.0_dp/6.0_dp)*k4)
39        call periodic_bc(Nx,Ngz,y0)

```

```

40
41
42     end subroutine rk4
43
44 end module time_stepping

```

Notice we have directly imported the subroutines `compute_rhs` and `periodic_bc` rather than defining an interface block. While this is simpler to implement, it means we would have to update (and recompile) the subroutine `rk4` every time we want to change the evolution equation or boundary conditions. Now that all the appropriate modules have been defined, we can create a main program file which sets up the initial data and runs the evolution loop,

```

1  program evolve_advection
2      use real_type_mod, only: dp
3      use time_stepping, only: rk4
4      use boundary_conditions, only: periodic_bc
5      use finite_difference, only: FD_C4
6      implicit none
7
8      integer :: Nx, Ngz, i, k, beginning, end, steps
9      real(dp), parameter :: pi = acos(-1.0_dp)
10     real(dp) :: x_min, x_max, dx, CFL, t0, tend, dt, rate
11     real(dp), allocatable :: grid(:)
12     real(dp), allocatable :: y0(:, :)
13     character(len=30) :: filename_solution, filename_times
14
15     call system_clock(beginning, rate)
16
17     !#####
18     ! Set simulation parameters
19     !#####
20     Nx = 501 ! Grid points
21     Ngz = 2 ! Ghost points
22
23     ! Boundaries of grid [x_min,x_max]
24     x_min = 0.0_dp
25     x_max = 2.0_dp*pi
26
27     ! Spatial step size, CFL number, timestep size
28     dx = (x_max - x_min)/(real(Nx,dp)-1.0_dp)
29     CFL = 0.5_dp
30     dt = CFL*dx ! Set timestep with CFL condition
31
32     ! Compute grid point values
33     allocate(grid(1-Ngz:Nx+Ngz))
34     do i = 1-Ngz, Nx+Ngz
35         grid(i) = x_min + (real(i,dp) - 1.0_dp)*dx
36     end do
37
38     !#####
39     ! Set initial data
40     !#####
41     allocate(y0(1-Ngz:Nx+Ngz,1))
42
43     y0(:,1) = sin(grid)
44
45     call periodic_bc(Nx,Ngz,y0)
46
47     !#####

```



```

48      ! Time interval
49      !#####
50      t0 = 0.0_dp
51      tend = 20.0_dp
52
53      !#####
54      ! Set up files for saving data
55      !#####
56
57      ! Name files
58      filename_solution = "advection_soln.dat"
59      filename_times = "advection_soln_times.dat"
60
61      ! Save initial data
62      open(unit=10, file=filename_solution, status="replace", action="write")
63      do k = 1, Nx
64          write(10, '(es21.12E3,a1)', advance='NO') y0(k,1)
65      end do
66      write(10,*)
67
68      open(unit=11, file=filename_times, status="replace", action="write")
69      write(11,'(es21.12E3,a1)') t0
70
71      ! close(10)
72      ! close(11)
73
74      !#####
75      ! Evolution Routine
76      !#####
77      steps = 0
78      do while (t0<tend)
79
80          steps = steps + 1
81          call rk4(Nx,Ngz,dx,t0,y0,dt)
82          t0 = t0 + dt
83
84          if (steps == 100) then
85              do k = 1, Nx
86                  write(10, '(es21.12E3,a1)', advance='NO') y0(k,1)
87              end do
88              write(10, *)
89              ! write(10) y0
90              write(11,'(es21.12E3,a1)') t0
91              steps = 0
92          end if
93
94      end do
95
96      close(10)
97      close(11)
98
99      call system_clock(end)
100     print *, "elapsed time: ", real(end - beginning) / real(rate)
101
102
103
104 end program evolve_advection

```

Some comments:

- We set our initial data array `y0` to be allocatable, so we can adjust the number of grid points without having to change the type declaration.
- Fortran does not have a built-in `linspace` function like Matlab or numpy, so we construct our grid array using a loop.
- We are saving to binary files instead of text. This is both faster and results in smaller files.
- The inbuilt `system_clock` subroutine can be used to time our code.

Without saving to files, this implementation was about 5 times faster than a comparable python implementation using numpy. If we save to a file, the code is approximately twice as fast. This could probably be improved by pre-assigning a array to store data and only saving to file after the array has been filled. For completeness, here is the **Makefile**

```

1 FC = gfortran
2 FFLAGS = -O3 # This is an optimisation flag
3 SRC = real_type_mod.o finite_difference_mod.o \
4     boundary_conditions_mod.o advection_mod.o \
5     time_stepping_mod.o evolve_advection.o
6
7 .PHONY: clean
8
9 main: $(SRC)
10     $(FC) $(FFLAGS) -o $@ $(SRC)
11 %.o : %.f90
12     $(FC) $(FFLAGS) -o $@ -c $<
13 clean:
14     @rm *.o *.mod main

```

and a simple python script to plot the output

```

1 import numpy as np
2 from matplotlib import pyplot as plt
3
4 fortran_soln = np.loadtxt('advection_soln.dat')
5 fortran_times = np.loadtxt('advection_soln_times.dat')
6 # Subtract the number of ghost points when defining the grid:
7 grid = np.linspace(0,2*np.pi,np.shape(fortran_soln)[1]-4)
8
9
10
11
12 ### Plotting parameters I use when saving a single figure
13 ### for a paper
14
15 # plt.rcParams.update({"text.usetex": True,
16 #     "font.family": "serif",
17 #     "font.serif": "Computer Modern",
18 #     "savefig.bbox": "tight",
19 #     "savefig.format": "pdf"})
20 # plt.rc('font', size=16)
21
22
23 ### Plot the solution
24 for i in range(np.shape(fortran_times)[0]):
25     plt.plot(grid,fortran_soln[i,2:-2], 'x', linestyle=None)
26     plt.xlabel('x')
27     plt.title("t = " +str(fortran_times[i]))
28     plt.ylim([-1.5,1.5])
29     plt.draw()
30     plt.pause(0.01)
31 plt.cla()

```

One downside of this implementation is that our simulation parameters are defined in the main program file. This means we must recompile the program if we want to update them. A better way is to set parameters from an external file and read them in.

8. NAMELISTS

Namelists are a built-in method for reading blocks of data in Fortran from an external namelist file. We will use this feature to read in simulation parameters. In particular, this will allow us to change the parameters *without* recompiling the main Fortran code.

First, let's look at an example namelist file, which we will save as `params.nml`.

```

1 &Grid_Params
2 x_min = 0.0,
3 x_max = 2.0,
4 Nx = 10
5 Ngz = 2
6 /
7

```

`&Grid_Params` is the name of our namelist. We define the four parameters that we want to read in and finish with `/` **and** a blank line. The blank line was necessary when using `gfortran` to compile my Fortran code. Now, let's create a Fortran module to read in the parameters and create derived quantities such as the spatial step size and array of cell centres.

```

1 module read_parameters
2   use real_type_mod, only: dp
3   implicit none
4   real(dp) :: x_min, x_max, dx
5   integer :: Nx, Ngz, i
6   real(dp), allocatable :: grid(:)
7
8   contains
9
10  subroutine read_params
11
12      ! Load in parameters from namelist
13      namelist /Grid_Params/ x_min, x_max, Nx, Ngz
14
15      open(file='params.nml', status='old', unit=10, action='read')
16      read(10, nml=Grid_Params)
17      close(10)
18
19      ! Created derived parameters
20      dx = (x_max - x_min)/(real(Nx, dp))
21
22      allocate(grid(1-Ngz:Nx+Ngz))
23
24      do i = 1-Ngz, Nx+Ngz
25          grid(i) = x_min + (real(i, dp) - 0.5_dp)*dx
26      end do
27
28
29  end subroutine read_params
30
31
32
33 end module read_parameters

```

Our `namelist` and its corresponding variables are declared in line 13. We then open the `params.nml` file using `open`. To fill in the namelist variables, we use the `read` command with the keyword `nml`. Now, let's create a simple program which calls the `read_params` subroutine and prints our grid variables:

```

1 program namelist_read
2   use read_parameters
3   implicit none
4
5   call read_params
6
7   ! Print out the parameters:
8   print *, "The grid domain is: ", x_min, x_max
9   print *, "The number of cell centres is: ", Nx
10  print *, "The number of ghost points at each boundary is: ", Ngz
11  print *, "The step size is: ", dx
12  print *, "The cell centres are: ", grid
13
14 end program namelist_read

```

The output is given by

```

1 The grid domain is:      0.0000000000000000      2.0000000000000000
2 The number of cell centres is:      10
3 The number of ghost points at each boundary is:      2
4 The step size is:      0.20000000000000001
5 The cell centres are:  -0.30000000000000004      -0.10000000000000001
6      0.10000000000000001      0.30000000000000004      0.50000000000000000
7      0.70000000000000007      0.90000000000000002      1.10000000000000001
8      1.30000000000000000      1.50000000000000000      1.70000000000000002
9      1.90000000000000001      2.10000000000000001      2.30000000000000003

```

We can also create a python script, which we save as `write_namelist.py`, to write the namelist for us,

```

1 ### Define Grid Parameters
2
3 nml_1 = "&Grid_Params"
4 x_min = 0.0
5 x_max = 2.0
6 Nx = 10
7 Ngz = 2
8
9
10 ### Create namelist file
11 f = open("params.nml","w")
12
13 f.write(nml_1+"\n")
14 f.write("x_min = " + str(x_min) + "\n")
15 f.write("x_max = " + str(x_max) + "\n")
16 f.write("Nx = " + str(Nx) + "\n")
17 f.write("Ngz = " + str(Ngz) + "\n")
18 f.write("/\n")

```

Now, let's we change `x_max` to 2π in our python script. We run our python script to update the namelist file followed by our Fortran program *without* recompiling. We obtain the following output

```

1 The grid domain is:      0.0000000000000000      6.2831853071795862
2 The number of cell centres is:      10
3 The number of ghost points at each boundary is:      2
4 The step size is:      0.62831853071795862

```

5	The cell centres are: -0.94247779607693793 -0.31415926535897931		
	0.31415926535897931	0.94247779607693793	1.5707963267948966
	2.1991148575128552	2.8274333882308138	3.4557519189487724
	4.0840704496667311	4.7123889803846897	5.3407075111026483
	5.9690260418206069	6.5973445725385655	7.2256631032565242

Finally, we can eliminate the need to separately run the Fortran file by using the `subprocess` package in Python. Our new script is given by

```

1 import subprocess
2 import numpy as np
3
4 ### Define Grid Parameters
5 nml_1 = "&Grid_Params"
6 x_min = 0.0
7 x_max = 2.0*np.pi
8 Nx = 10
9 Ngz = 2
10
11 ### Create namelist file
12 f = open("params.nml","w")
13
14 f.write(nml_1+"\n")
15 f.write("x_min = " + str(x_min) + "\n")
16 f.write("x_max = " + str(x_max) + "\n")
17 f.write("Nx = " + str(Nx) + "\n")
18 f.write("Ngz = " + str(Ngz) + "\n")
19 f.write("/\n")
20 f.close() # Need to close namelist file before running Fortran!
21
22 ### Run our Fortran script
23 subprocess.run("./main")

```

Now we can create our namelist and immediately run our Fortran program by simply typing

```
1 $ python3 -0 write_namelist.py
```

into the command line.

Remark 8.1. Instead of creating our own code to write a namelist file, there is also a Python package `f90nml` which includes several functions to convert between Python dictionaries and Fortran namelists. I have not tried this, but it seems to be fairly well maintained.

REFERENCES

- [1] Clawpack Development Team, *Clawpack software*, 2024. Version 5.11.0.
- [2] R.J. LeVeque, *Finite Volume Methods for Hyperbolic Problems*, Cambridge Texts in Applied Mathematics, Cambridge University Press, 2002.

SCHOOL OF MATHEMATICS, 9 RAINFOREST WALK, MONASH UNIVERSITY, VIC 3800, AUSTRALIA
Email address: `elliott.marshall@monash.edu`