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?

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: elliot.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

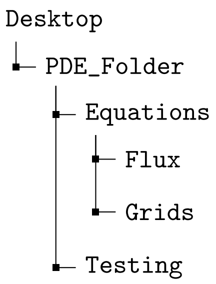
To begin, let us recall some simple bash commands for using the terminal.

- `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.

ampleFolderStructure⟩

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

```
MacBookPro:PDE_Folder elliotmarshall$ ls
Equations Testing
```

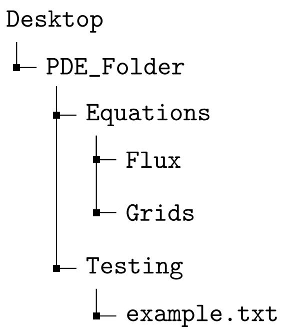
I could then navigate to the Testing folder and create a text file by typing

```
MacBookPro:PDE_Folder elliotmarshall$ cd Testing
MacBookPro:Testing elliotmarshall$ touch example.txt
```

My folder structure would now be given by Figure 2.

ifiedFolderStructure⟩

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 it[1]

```
MacBookPro:~ elliotmarshall$ cd Desktop
MacBookPro:Desktop elliotmarshall$ mkdir -p Example_Project
MacBookPro:Desktop elliotmarshall$ cd Example_Project
```

[1]The -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

```
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

```
program hello
    print *, 'hello'
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

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

This creates an executable file `Hello` which we can run using

```
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

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

We get the output

```
Pi =    3.1415926535897931
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,

```fortran
module compute_area
    implicit none
    real (kind=8), parameter :: pi = acos(-1.d0)
    public pi, area
    private

    contains

    real (kind=8) function area(r)
        real (kind=8), intent(in) :: r

        area = pi*r**2

    end function area

end module compute_area
```

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

```fortran
program maths
    use compute_area, only: area, pi
    implicit none
    real (kind = 8) :: r

    r = 2.0

    print *, "Pi = ", pi
    print *, "The area of the circle is ", area(r)

end program maths
```

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

```
$ 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.

```
$ gfortran -o area.o -c area.f90
$ gfortran -o maths.o -c maths.f90
$ 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`,

```
$ touch Makefile
```

Now, inside this file we can write

```
VAR=hello
default:
    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

```
echo hello
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,

```
VAR=hello
VAR2=goodbye
target1:
    @echo ${VAR}
target2:
    @echo ${VAR}
```

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

```
$ make target1
```

the output is

```
hello
```

Similarly, if we run

```
$ make target2
```

the output is

```
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

```
VAR=hello
VAR2=goodbye
target1:
    @echo ${VAR}
target2: target1
    @echo ${VAR}
```

If we now run

```
$ make target2
```

the output is

```
hello
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

```
main.exe: area.o maths.o
  gfortran -o maths maths.o area.o
area.o:
  gfortran -o area.o -c area.f90
maths.o:
  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

```
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

```
gfortran -O3  -c maths.f90
maths.f90:2:9:

    2 |     use compute_area, only: area, pi
      |         1
Fatal Error: Cannot open module file 'compute_area.mod' for reading at (1): No such file or directory
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,

```
FC = gfortran
FFLAGS = -O3 # This is an optimisation flag
SRC =  maths.o area.o

maths: $(SRC)
  $(FC) $(FFLAGS) -o maths $(SRC)
%.o : %.f90
  $(FC) $(FFLAGS) -o $@ -c $<
maths.o: area.o
clean:
  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.

```fortran
! This module sets up the floating point precision used.
module real_type_mod

! sp contains the kind value for single precision.
  integer, parameter :: sp = kind(1.0)

! dp contains the kind value for double precision.
  integer, parameter :: dp = kind(1.0d0)
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,

```fortran
module arrays
    implicit none

    contains

    subroutine array_operations()
        use real_type_mod, only: dp
        real(dp) :: x(3)
        real(dp) :: z(2,2), y(2,2)

        x = 1.0_dp ! Set all entries of x to 1.0
        print *,"x is", x

        x(1) = 0.62_dp ! Set the first entry to 0.62

        print *,"The updated x is", x

        z = 1.0_dp ! Set all entries of z to 1.0

        print *, "z is ", z

        ! Set the first row of y to be -0.5 and the second to 0.5
        y(1:2,1) = -0.5_dp
        y(1:2,2) = 0.5_dp

        print *, "y is ", y

        ! Add y and z elementwise
        y = y + z

        print *, "y+z is ", y

    end subroutine array_operations
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`

```fortran
program main
    use arrays, only: array_operations
    implicit none

    call array_operations()

end program main
```

Our `Makefile` becomes

```
FC = gfortran
FFLAGS = -O3 # This is an optimisation flag
SRC = real_type_mod.o arrays.o main.o

main: $(SRC)
  $(FC) $(FFLAGS) -o main $(SRC)
%.o : %.f90
  $(FC) $(FFLAGS) -o $@ -c $<
clean:
  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

```
program io
    use real_type_mod, only: dp
    implicit none
    character(len=20) :: filename
    real(dp) ::  x,y

    x = 1.0_dp
    y = 12.0_dp

    filename = 'result.txt'

    ! Write to a text file
    open(unit=1,file=filename,status='replace',form='formatted')
    write(1,*) x,y
    close(1)

    ! Reset the variables
    x = 0.0_dp
    y = 0.0_dp

    ! Read in from text file
    open(unit=2,file=filename,status='old')
    read(2,*) x,y
    close(2)

    ! Print the variables
    print *, 'x = ', x, 'y = ', y

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

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

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

```python
import numpy as np
### Heun's Method
def rk2(y0,t0,dt,rhs):
    k1 = rhs(t0,y0)
    k2 = rhs(t0 + dt, y0 + dt*k1)
    y0 = y0 + (dt/2)*(k1 + k2)
    return y0

### Exponential Growth/Decay ODE
def ODE(t,y):
    dydt = 2*y
    return dydt

### Define initial parameters
t0 = 0.0
tend = 5.0
dt = 0.01
y0 = 0.01

### Create list to store solution
y_output = []
y_output.append(y0)
t_output = []
t_output.append(t0)

### Main integration loop
while t0 < tend:
    y0 = rk2(y0,t0,dt,ODE)
    t0 += dt
    y_output.append(y0)
    t_output.append(t0)

### Save solution as a text file
y_soln = np.array(y_output) # Convert list to numpy arrays
t_soln = np.array(t_output)
np.savetxt('python_soln.txt', y_soln)
np.savetxt('python_soln_times.txt', t_soln)
```

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

```
$ python3 -O 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

```fortran
module runge_kutta_2
    use real_type_mod, only: dp
    implicit none

    ! First we construct an abstract interface
    ! This allows us to pass arbitrary ODEs to
    ! the Runge-Kutta method
    abstract interface
        subroutine rhs_interface(t, y, neqn, dydt)
            import :: dp
            integer :: neqn ! The number of eqns to solve
            real(dp), intent(in) :: t, y(neqn)
            real(dp), intent(out):: dydt(neqn)
        end subroutine rhs_interface
    end interface

    contains

    ! 2nd order accurate Runge-Kutta method (Heun's method)
    subroutine rk2(t0,y0,dt,neqn,rhs)
        integer, intent(in) :: neqn ! Num of eqns to solve
        real(dp), intent(in) :: t0,dt ! Current time, timestep
        real(dp), intent(inout) :: y0(neqn) ! Current values
        real(dp) :: k1(neqn), k2(neqn) ! Runge-Kutta Substeps
        procedure(rhs_interface) :: rhs

        call rhs(t0,y0,neqn,k1)
        call rhs(t0 + 0.5_dp*dt, y0 + dt*k1, neqn, k2)

        y0 = y0 + dt*(0.5_dp*k1 + 0.5_dp*k2)

    end subroutine rk2

end module runge_kutta_2
```

Clearly this is already is more involved than the Python version. In Python, we can pass the rhs function as an argument directly. In order to do in 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 function[2]. 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

```fortran
program ode_solve
    use runge_kutta_2, only: rk2
    use real_type_mod, only: dp
    implicit none
    integer :: num_dims
    real(dp), allocatable :: y0(:)
    real(dp) :: t0, tend, dt
    character(len=30) :: filename_solution, filename_times

```

[2]We 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.

```fortran
    ! Set up files for outputting data
    filename_solution = "fortran_soln.txt"
    filename_times = "fortran_soln_times.txt"

    ! Number of equations to solve
    num_dims = 1

    ! Set simulation parameters
    allocate(y0(num_dims)) ! Set the dimension of the initial data vector
    y0 = 0.01_dp
    t0 = 0.0_dp
    tend = 5.0_dp
    dt = 0.01_dp

    ! Save initial data values
    open(unit=10, file=filename_solution, status="replace", action="write")
    write(10, *) y0

    open(unit=11, file=filename_times, status="replace", action="write")
    write(11, *) t0

    ! Main rk2 loop
    do while (t0 < tend)
        call rk2(t0,y0,dt,num_dims,ODE)
        t0 = t0 + dt

        ! write solution data to txt files
        write(10, *) y0
        write(11, *) t0

    end do
    close(10)
    close(11)

contains

! Exponential growth/decay ODE
! Solution is exp(2*t)
subroutine ODE(t,y,neqn,dydt)
    implicit none
    integer, intent(in) :: neqn
    real(dp), intent(in) :: t, y(neqn)
    real(dp), intent(out) :: dydt(neqn)

    dydt = 2*y

end subroutine ODE

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`

```makefile
FC = gfortran
FFLAGS = -O3 # This is an optimisation flag
```

```
SRC = real_type_mod.o rk2.o ode_solve.o

ode_solve: $(SRC)
  $(FC) $(FFLAGS) -o $@ $(SRC)
%.o : %.f90
  $(FC) $(FFLAGS) -o $@ -c $<
clean:
  rm *.o *.mod
```

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

```python
import numpy as np
from matplotlib import pyplot as plt

python_soln = np.loadtxt('python_soln.txt')
python_times = np.loadtxt('python_soln_times.txt')

fortran_soln = np.loadtxt('fortran_soln.txt')
fortran_times = np.loadtxt('fortran_soln_times.txt')

### Plot outputs against the exact solution
plt.plot(python_times,python_soln, 'x',label='Python Solution')
plt.plot(fortran_times,fortran_soln, '.',label='Fortran Solution')
plt.plot(fortran_times, 0.01*np.exp(2*fortran_times), label='Exact Solution')
plt.xlabel("Time")
plt.ylabel("Solution")
plt.legend()
plt.show()
```

We get the following output:

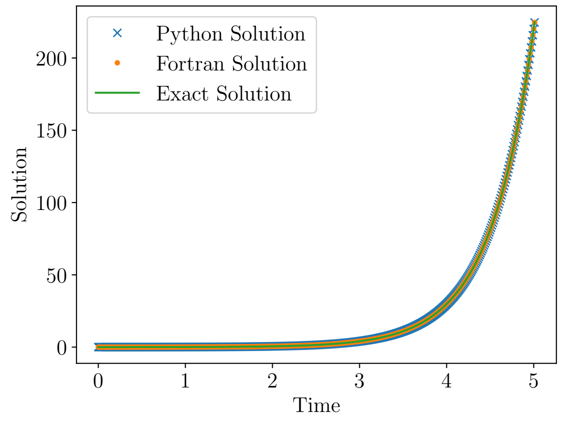


FIGURE 3. Solution of (6.1) using our Python and Fortran routines

_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 faster[3] 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,

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

[3]Probably? 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,

```fortran
    ! Module containing the finite difference operations
module finite_difference
    use real_type_mod, only: dp
    implicit none

    contains

    subroutine FD_C4(dx,Nx,Ngz,u,du)
        ! Inputs:
        ! dx - The spatial step size, real(dp)
        ! Nx - The number of grid points, integer
        ! Ngz - The number of ghost points, integer
        ! u - The function to be differentiated, real(dp) dimension(1,1-Ngz:Nx+Ngz)
        ! Output:
        ! du - The derivative, real(dp) dimension(1,1-Ngz:Nx+Ngz)
        implicit none
        real(dp), intent(in) :: dx
        integer, intent(in) :: Nx, Ngz
        real(dp), intent(in) :: u(1-Ngz:Nx+Ngz,1)
        real(dp), intent(out) :: du(1-Ngz:Nx+Ngz,1)
        ! integer :: i

        ! ### Loop Version
        ! do i = 1, Nx
        !     du(1,i) = (u(1,i-2) - 8.0_dp*u(1,i-1) + 8.0_dp*u(1,i+1) &
        !       - u(1, i+2))/(12.0_dp*dx)
        ! end do

        ! ### Slicing Version
        du(1:Nx,:) = (u(-1:Nx-2,:) - 8.0_dp*u(0:Nx-1,:) + 8.0_dp*u(2:Nx+1,:) &
                - u(3:Nx+2,:))/(12.0_dp*dx)


    end subroutine FD_C4

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,

```fortran
    ! Module containing the advection equation subroutine
module advection
    use real_type_mod, only: dp
    use finite_difference, only: FD_C4
    implicit none

    contains

    subroutine compute_rhs(Nx,Ngz,dx,t,u,dtu)
        implicit none
        integer, intent(in) :: Nx, Ngz
        real(dp), intent(in) :: t, dx
        real(dp), intent(in) :: u(1-Ngz:Nx+Ngz,1)
        real(dp) :: du(1-Ngz:Nx+Ngz,1)
        real(dp), intent(out) :: dtu(1-Ngz:Nx+Ngz,1)


       call  FD_C4(dx,Nx,Ngz,u,du) ! Compute the derivative

       dtu = -1.0_dp*du



    end subroutine compute_rhs

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.

```fortran
    ! Module containing the boundary conditions
module boundary_conditions
    use real_type_mod, only: dp
    implicit none

    contains

    subroutine periodic_bc(Nx,Ngz,u)
        ! Inputs:
        ! dx - The spatial step size, real(dp)
        ! Nx - The number of grid points, integer
        ! u - The function we apply the BCs to, real(dp) dimension(1,1-Ngz:Nx+Ngz)
        ! Output:
        ! u - The function with updated boundaries, real(dp) dimension(1,1-Ngz:Nx+
    Ngz)
        implicit none
        integer, intent(in) :: Nx, Ngz
        real(dp), intent(inout) :: u(1-Ngz:Nx+Ngz,1)
        ! integer :: i

        u(Nx,1) = u(1,1) ! First and last grid points are identified

        ! Right Boundary
        u(Nx+1:Nx+Ngz,1) = u(1+1:1+Ngz,1)

        ! Left Boundary
        u(1-Ngz:0,1) = u(Nx-Ngz:Nx-1,1)

        ! ### Loop version
        ! do i = 1,Ngz

        !     ! Right Boundary
        !     u(1,Nx+i) = u(1,1+i)

        !     ! Left Boundary
        !     u(1,1-i) = u(1,Nx-i)

        ! end do

    end subroutine periodic_bc

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.

```fortran
    ! Module containing the time stepping routines
module time_stepping
    use real_type_mod, only: dp
    use advection, only: compute_rhs
    use boundary_conditions, only: periodic_bc


    contains

    subroutine rk4(Nx,Ngz,dx,t0,y0,dt)
        integer, intent(in) :: Nx,Ngz ! Num of grid/ghost points
        real(dp), intent(in) :: t0,dt, dx ! Current time, time step, spatial step
        real(dp), intent(inout) :: y0(1-Ngz:Nx+Ngz,1) ! Current values
        real(dp) :: k1(1-Ngz:Nx+Ngz,1), k2(1-Ngz:Nx+Ngz,1), k3(1-Ngz:Nx+Ngz,1), &
            k4(1-Ngz:Nx+Ngz,1) ! Runge-Kutta Substeps
        real(dp) :: y1(1-Ngz:Nx+Ngz,1), y2(1-Ngz:Nx+Ngz,1), y3(1-Ngz:Nx+Ngz,1)

        ! Step 1
        call compute_rhs(Nx,Ngz,dx,t0,y0,k1)
        y1 = y0 + 0.5*dt*k1
        call periodic_bc(Nx,Ngz,y1) ! Update Boundary Conditions

        ! Step 2
        call compute_rhs(Nx,Ngz,dx,t0+0.5_dp*dt,y1,k2)
        y2 = y0 + 0.5*dt*k2
        call periodic_bc(Nx,Ngz,y2)

        ! Step 3
        call compute_rhs(Nx,Ngz,dx,t0+0.5_dp*dt,y2,k3)
        y3 = y0 + dt*k3
        call periodic_bc(Nx,Ngz,y3)

        ! Step 4
        call compute_rhs(Nx,Ngz,dx,t0+dt,y3,k4)

        ! Update y0
        y0 = y0 + dt*((1.0_dp/6.0_dp)*k1 + (1.0_dp/3.0_dp)*k2 &
            + (1.0_dp/3.0_dp)*k3  + (1.0_dp/6.0_dp)*k4)
        call periodic_bc(Nx,Ngz,y0)


    end subroutine rk4

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,

```fortran
program evolve_advection
    use real_type_mod, only: dp
    use time_stepping, only: rk4
    use boundary_conditions, only: periodic_bc
    use finite_difference, only: FD_C4
    implicit none

    integer :: Nx, Ngz, i, k, beginning, end, steps
    real(dp), parameter :: pi = acos(-1.0_dp)
    real(dp) :: x_min, x_max, dx, CFL, t0, tend, dt, rate
    real(dp), allocatable ::  grid(:)
    real(dp), allocatable :: y0(:,:)
    character(len=30) :: filename_solution, filename_times

    call system_clock(beginning, rate)

    !#########################################
    ! Set simulation parameters
    !########################################
    Nx = 501 ! Grid points
    Ngz = 2 ! Ghost points

    ! Boundaries of grid [x_min,x_max]
    x_min = 0.0_dp
    x_max = 2.0_dp*pi

    ! Spatial step size, CFL number, timestep size
    dx = (x_max - x_min)/(real(Nx,dp)-1.0_dp)
    CFL = 0.5_dp
    dt = CFL*dx ! Set timestep with CFL condition

    ! Compute grid point values
    allocate(grid(1-Ngz:Nx+Ngz))
    do i = 1-Ngz, Nx+Ngz
        grid(i) = x_min + (real(i,dp) -1.0_dp)*dx
    end do

    !#########################################
    ! Set initial data
    !########################################
    allocate(y0(1-Ngz:Nx+Ngz,1))

    y0(:,1) = sin(grid)

    call periodic_bc(Nx,Ngz,y0)

    !#########################################
    ! Time interval
    !########################################
    t0 = 0.0_dp
    tend = 20.0_dp

    !#########################################
    ! Set up files for saving data
    !########################################

    ! Name files
    filename_solution = "advection_soln.dat"
    filename_times = "advection_soln_times.dat"

    ! Save initial data
    open(unit=10, file=filename_solution, status="replace", action="write")
    do k = 1, Nx
        write(10, '(es21.12E3,a1)',advance='NO') y0(k,1)
    end do
    write(10,*)

    open(unit=11, file=filename_times, status="replace", action="write")
    write(11,'(es21.12E3,a1)') t0

    ! close(10)
    ! close(11)

    !#########################################
    ! Evolution Routine
    !########################################
    steps = 0
    do while (t0<tend)

        steps = steps + 1
        call rk4(Nx,Ngz,dx,t0,y0,dt)
        t0 = t0 + dt

        if (steps == 100) then
            do k = 1, Nx
                write(10, '(es21.12E3,a1)',advance='NO') y0(k,1)
            end do
            write(10, *)
            ! write(10) y0
            write(11,'(es21.12E3,a1)') t0
            steps = 0
        end if

    end do

    close(10)
    close(11)

    call system_clock(end)
    print *, "elapsed time: ", real(end - beginning) / real(rate)



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.
- 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`

```
FC = gfortran
FFLAGS = -O3 # This is an optimisation flag
SRC = real_type_mod.o finite_difference_mod.o \
  boundary_conditions_mod.o advection_mod.o \
  time_stepping_mod.o evolve_advection.o

.PHONY: clean

main: $(SRC)
  $(FC) $(FFLAGS) -o $@ $(SRC)
%.o : %.f90
  $(FC) $(FFLAGS) -o $@ -c $<
clean:
  @rm *.o *.mod main
```

and a simple Python script to plot the output

```
import numpy as np
from matplotlib import pyplot as plt

fortran_soln = np.loadtxt('advection_soln.dat')
fortran_times = np.loadtxt('advection_soln_times.dat')
# Subtract the number of ghost points when defining the grid:
grid = np.linspace(0,2*np.pi,np.shape(fortran_soln)[1]-4)




### Plotting parameters I use when saving a single figure
### for a paper

# plt.rcParams.update({"text.usetex": True,
#      "font.family": "serif",
#      "font.serif": "Computer Modern",
#      "savefig.bbox": "tight",
#      "savefig.format": "pdf"})
# plt.rc('font', size=16)


### Plot the solution
for i in range(np.shape(fortran_times)[0]):
    plt.plot(grid,fortran_soln[i,2:-2],'x',linestyle=None)
    plt.xlabel('x')
    plt.title("t = " +str(fortran_times[i]))
    plt.ylim([-1.5,1.5])
    plt.draw()
    plt.pause(0.01)
    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`.

```
&Grid_Params
x_min = 0.0
x_max = 2.0
Nx = 10
Ngz = 2
/

```

`&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.

```fortran
module read_parameters
    use real_type_mod, only: dp
    implicit none
    real(dp) :: x_min, x_max, dx
    integer :: Nx, Ngz, i
    real(dp), allocatable :: grid(:)

    contains

    subroutine read_params

        ! Load in parameters from namelist
        namelist /Grid_Params/ x_min, x_max, Nx, Ngz

        open(file='params.nml',status='old',unit=10,action='read')
        read(10,nml=Grid_Params)
        close(10)

        ! Created derived parameters
        dx = (x_max - x_min)/(real(Nx,dp))

        allocate(grid(1-Ngz:Nx+Ngz))

        do i = 1-Ngz, Nx+Ngz
            grid(i) = x_min + (real(i,dp) -0.5_dp)*dx
        end do


    end subroutine read_params



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:

```fortran
program namelist_read
    use read_parameters
    implicit none

    call read_params

    ! Print out the parameters:
    print *, "The grid domain is: ", x_min, x_max
    print *, "The number of cell centres is: ", Nx
    print *, "The number of ghost points at each boundary is: ", Ngz
    print *, "The step size is: ", dx
    print *, "The cell centres are: ", grid

end program namelist_read
```

The output is given by

```
The grid domain is:    0.0000000000000000        2.0000000000000000
 The number of cell centres is:           10
 The number of ghost points at each boundary is:            2
 The step size is:   0.20000000000000001
 The cell centres are:  -0.30000000000000004      -0.10000000000000001
    0.10000000000000001       0.30000000000000004       0.50000000000000000
    0.70000000000000007       0.90000000000000002        1.1000000000000001
    1.3000000000000000        1.5000000000000000        1.7000000000000002
    1.9000000000000001        2.1000000000000001        2.3000000000000003
```

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

```python
### Define Grid Parameters

nml_1 = "&Grid_Params"
x_min = 0.0
x_max = 2.0
Nx = 10
Ngz = 2


### Create namelist file
f = open("params.nml","w")

f.write(nml_1+"\n")
f.write("x_min = " + str(x_min) + "\n")
f.write("x_max = " + str(x_max) + "\n")
f.write("Nx = " + str(Nx) + "\n")
f.write("Ngz = " + str(Ngz) + "\n")
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

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

```python
import subprocess
import numpy as np

### Define Grid Parameters
nml_1 = "&Grid_Params"
x_min = 0.0
x_max = 2.0*np.pi
Nx = 10
Ngz = 2

### Create namelist file
f = open("params.nml","w")

f.write(nml_1+"\n")
f.write("x_min = " + str(x_min) + "\n")
f.write("x_max = " + str(x_max) + "\n")
f.write("Nx = " + str(Nx) + "\n")
f.write("Ngz = " + str(Ngz) + "\n")
f.write("/\n")
f.close() # Need to close namelist file before running Fortran!

### Run our Fortran script
subprocess.run("./main")
```

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

```
$ python3 -O 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: `elliot.marshall@monash.edu`