Fortran: FORmula TRANslation

Introductory Notes & Programs

Andrew R. Winters
Division of Computational Mathematics
Department of Mathematics
Linköping University
`andrew.ross.winters (at) liu.se`

Contents

Chapter 1: Introduction

Hello, and welcome to "Everything I wanted to know about Fortran, but was afraid to ask." These notes operate as an introductory, crash course in Fortran for applied mathematicians. Though you may encounter that Fortran is seen as antiquated by some, know that Fortran is a fast and efficient language largely used in the applied math and scientific computing communities. The advantages of Fortran are especially prevalent in vector/matrix operations.

The name Fortran comes from FORmula TRANslation. That is, the language was originally developed for easy implementation of mathematical formulae, vector and matrix operations in particular. The flexibility of Fortran to natively handle arrays makes your life much easier when coding basic routines, like matrix-vector products, to more advanced routines like linear solvers or conjugate gradient.

This introduction serves as an outline of Fortran's different features. In particular, the notes will mainly focus on programming in the `Fortran95` standard (although some of the functionality discussed in the "Advanced Topics" Chapter uses more modern Fortran). This is done because the `Fortran95` distribution is very stable and supported by nearly every compiler available. Thus, coding practices for Fortran learned from these notes will easily transfer between machines or compilers. Know that these notes ***do not*** include all aspects of Fortran, but it will be enough to get up and running, solving problems, and coding with organization. The notes are structured to learn Fortran through example, with commentary along the way. We color code the discussion of Fortran code examples in the following way:

- Program structure components are red.
- Variable declarations are green.
- Intrinsic Fortran functions (like sine) are light blue.
- Comments are gray.
- Numbers, written output, and formatting are pink.

Furthermore, we use a set of style conventions to make example codes in the notes easier to read. ***This does not affect whether the code compiles, but will make the code more readable and easier to debug***. Loops, if statements, and other nested control sequences are indented with three spaces. The indentations help identify when each event opens and closes. Also, we capitalize Fortran control structures and variable declarations. We write variable names in (mostly) lower case and, since variable names in `Fortran95` can be as much as 32 characters in length, we make variable names as descriptive as possible to make the code easy to read/debug. It is important to always remember: ***Fortran is not case sensitive, so the variables with the name a and A are the same, as far as the compiler is concerned***.

In Chap. 2, we begin with a couple introductory examples of Fortran programs. This chapter also introduces how code examples in thiese notes are set-up. For the most part, examples are first introduced as psuedocode which is then translated into Fortran code that can be compiled. The ability to inspect pseudocode and translate it into working Fortran code is one of the main goals of this note set.

Chap 3. outlines, in depth, the components of a Fortran program. This includes the data types available to a program, the control sequences used (like loops), and how to read and write information from the terminal or a file. We then provide an example of a quadrature method to practice some of the new program components at our disposal.

Next, in Chap. 4, we examine the organization of a Fortran program. To keep programs from becoming bloated and hard to debug we split the code into functions and subroutines. This includes introducing some of the intrinsic functions available in Fortran as well as external functions, i.e., functions that are written by a user. Subroutines are always written by users. We then show how to pass an external function to another function or program. Next, we introduce modules, an important concept that allows one to collect similar functions and subroutines in a single source file. Finally, we outline a few options of how to compile these multiple source files, as the organization techniques will produce multiple Fortran source files.

In Chap. 5 we demonstrate the ease with which Fortran handles arrays. This includes how to manipulate data stored in arrays, intrinsic functions, printing, and how to pass arrays to functions or subroutines.

Chap. 6 introduces object oriented programming in Fortran using modules. To do so we introduce derived types and demonstrate how to improve the usability of Fortran code contained in modules using interfaces.

In Chap. 7 we cover some of the more advanced features of Fortran. This includes overloading operators (using interfaces), dynamic arrays, optional input arguments in functions and subroutines, formatted file I/O, recursive functions, and the associate construct.

Chap. 8 covers some basic philosophy and examples when it come to debugging Fortran code. This includes an explanation of some of the compiler flags available as well as specific examples of common bugs that can be present in a scientific computing project implemented in Fortran.

In Chap. 9 we bring everything together in a larger coding project example. By the end of this Chapter we will implement a one dimensional finite difference style solver for the linear advection equation. The purpose of this project is to practice the process of begin given a numerical recipe to solve a problem, think about how to divide the problem apart into smaller pieces, organize a Fortran code around these pieces, and then implement and compile the project.

Next, Chap. 10 dives deeper into the object oriented capabilities of Fortran. To do so, it introduces some specific, selected examples useful to implement a discontinuous Galerkin spectral element method (DGSEM) in one spatial dimension. The extension to multiple dimensions is straightforward. We also provide several implementations to output data to a file for plotting in different programs.

Finally, in Chap. 11, we introduce the topic of source code management on local and remote repositories using `git`. Although not specifically a Fortran topic, source code management is an important component of computer programming and scientific computing.

1.1 Getting Started

To begin coding in Fortran we'll need two things:

1. A Fortran compiler.
2. An environment to edit source files.

There are many Fortran compilers available like `absoft`, `ifort`, or `gfortran`. However, the `gfortran` compiler is a good first choice because it is free, open source, and can be found on pretty much all versions of Linux, Unix, and in cygwin for Windows. The binaries are available at `http://gcc.gnu.org/wiki/GFortranBinaries` along with installation instructions. However, building the binaries on your own can be a bit of a headache. Therefore, we provide a short walkthrough and some useful links for the three major OS distributions. For Linux or Mac it is assumed that compilation will be done using a terminal. For Windows, because there is not a true terminal environment, an Integrated Development Environment (IDE) is also installed that is used to edit source code as well as compile using the `make` utility. The IDE generates any makefiles automatically, so you don't have to manage that part of the project. If you want, similar IDEs are available for Linux or Mac as well.

1.1.1 Linux Installation

It is straightforward to install `gfortran` on a Linux machine (particularly running a distribution like Ubuntu or Mint) because of the synaptic package manager. Simply open a terminal and type

```
sudo apt-get install gfortran
```

enter your password and it will install the binaries of the latest stable release of the compiler for you.

1.1.2 Mac Installation

To install `gfortran` on an OSX machine you first need to install XCode and the Mac developer tools. A complete walkthrough of the installation for OSX is provided by David Whipp at `https://wiki.helsinki.fi/display/HUGG/GNU+compiler+install+on+Mac+OS+X`.

1.1.3 Windows Installation

For Windows we will install the Eclipse IDE software with Photran plugins. The IDE will make compilation and running on a Windows machine much easier. The installation process is lengthy, but a full description and step-by-step instructions are available at `https://wiki.eclipse.org/PTP/photran/documentation/photran8installation`. This also includes instructions on the installation of Cygwin.

1.1.4 Editing Source Files

Now that we have a compiler installed we need a way to edit source code. In the end, for the small examples and projects discussed in this note set, the editing program you choose is not too important. Pick your favorite among the different options: Any text editor works to edit source files (like gEdit on Linux, TextEdit on Mac, or notepad on Windows). There are also developer tools available (like XCode on Mac) or the more barebones terminal based code editors like `vim` or `emacs`.

If you decide to use an IDE like Eclipse there are a few more steps before you are up and running. First, you need to start a new Fortran project. This creates a repository for all the source files as well as other components of the program. Eclipse has a built in editor for the source files (with certain features like auto-completion). When you are ready to compile the program you press the "build all" button. Once the program builds (assuming no errors are thrown) you press the "run" button to execute the program. There is a small output environment at the bottom of the Eclipse window that will display information as the program runs.

1.2 Moving from MATLAB to Fortran

These notes are constructed from a point-of-view that the reader is familiar with the basics of computer programming and scientific computing. Further, the presentation of the notes assumes that the reader is moving from coding and running programs in MATLAB and wants to learn Fortran. Here we collect some of the major differences (obviously not all) when we make this move. Throughout the notes we will also note some of the smaller subtle differences between MATLAB and Fortran.

1.2.1 Compiler Flags

If you are used to coding in MATLAB you may not realize how much flexibility you have when compiling your source code. Most importantly you can tell the Fortran compiler to optimize (which you can read as "speedup") your code. This optimization process is automated with certain compiler flags, but know (at least if you ever move to a high performance computing (HPC) environment) that it involves loop unrolling, peeling, and splitting, among other things.

Compiler flags are also available for debugging purposes. If you are used to the graphical debugger in MATLAB you are in for a surprise. Debugging Fortran is a little more nuanced. Personally, I don't use a true debugger. I just use a combination of printing information to the screen and compiler flags to debug my code. We will revisit some debugging practices and techniques later in Chap. 8.

Under the assumption that we use the `gfortran` compiler we provide a short description of the some of the most useful compiler flags. Know that there is a complete list of available compiler flags at `http://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html#Invoking-GCC`. Some important compiler flags are:

- `-o`: allows the user to name the executable produced by compilation. If this flag is absent the program is given the default name `a.out`
- `-O0`: No optimization. Useful for debugging purposes.
- `-O`: Optimize. The compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.
- `-O2`: Optimize even more. Performs all optimization of `-O` as well as all supported optimizations that do not involve a space-speed tradeoff.
- `-O3`: Optimize yet more. Turns on all optimizations specified by `-O2` with more aggressive management of memory.

- `-Ofast`: Disregards strict standards compliance. Enables all `-O3` optimizations. Also enables optimizations that are not valid for all standard-compliant programs. It turns on `-ffast-math` which further optimizes intrinsic functions.
- `-fcheck-bounds`: Used for debugging. Tells the compiler to examine array bounds closely and inform the user if the program overruns outside of allocated memory. Especially useful in finding segmentation faults.

Note that optimization ***usually*** makes a program faster, but this is not always true.

1.2.2 Plotting Program Output

Another jarring aspect of switching from MATLAB to Fortran is that there is not a native plot command. You can't simply display your results while the program is running and inspect them. In fact, you need to print results to an external file (typically with a specific data format) and then use an independent piece of plotting software to visualize the results. We'll outline how to use Fortran to print results to a file in these notes. Three good, free plotting programs for program data are:

- `gnuplot`: A simple, terminal based plotter. The syntax can be a little obtuse and Google will be your friend for the more advanced options.
- `matplotlib`: A package for Python, this plotting option requires you to learn a little bit of the Python programming language. One advantage is that this package creates vector graphics, so you can rescale the images without fear of rasterization (i.e. images won't look pixelated as you scale them).
- `VisIt`: Basically a free version of TecPlot made by Laurence Livermore National Laboratory (LLNL). It is available for download at `http://wci.llnl.gov/simulation/computer-codes/visit`. I like this program, but it is difficult to Google information about because of the name. If you are not specific you will end with search results like "visit Sweden." Always include LLNL in any inquiry about `VisIt`. An advantage of `VisIt` is the ease with which you can make movies. Also, it plots all major file types, most importantly TecPlot files, which makes the transition easier.

We focus more on plotting routines in Chap. 10. There, we present a `VistIt` implementation in one, two, and three spatial dimensions as well as examples in `matplotliob`.

1.3 Further resources

These notes are self-contained and cover many features of Fortran, but there is always more to learn. Two excellent books that offer in-depth discussions of Fortran:

Fortran 90 Programming (International Computer Science Series), by T. M. R. Ellis, I. R. Phillips, and T. M. Lahey. Offers comprehensive tutorials and information about programming in Fortran.

Numerical Computing With Modern Fortran (Applied Mathematics), by R. J. Hanson and T. Hopkins. This is a more advanced book that highlights Fortran in the context of HPC and how to develop scalable implementations in modern Fortran on supercomputers.

Now, with the `gfortran` ***compiler, a preferred method for source code creation and editing, and what to expect when moving away from MATLAB you are ready to learn Fortran!***

Chapter 2: The Program

Let's jump right into Fortran programming. We start with a simple introductory example to learn program structure, some basic Fortran syntax, and how to compile/run a Fortran program. These notes are designed to teach the `Fortran95` standard. However, any source code files will end with a `.f90` extension (as is the standard in the Fortran community).

2.1 Example: Adding Two Integers

Below we see the pseudocode for a program that adds two fixed integers.

Algorithm 1: *A First Example*: Add two integers and print

```
Procedure A First Example
$i \leftarrow 3$
$j \leftarrow 7$
$k \leftarrow i + j$
Print $k$
End Procedure A First Example
```

Now we translate this pseudocode into functional Fortran code, which we save as the program "firstExample.f90'.

`firstExample.f90`

```
PROGRAM firstExample
   IMPLICIT NONE
   INTEGER :: i,j,k
!
   i = 3
   j = 7
   k = i + j
   WRITE(*,*)k
!
END PROGRAM firstExample
```

A few quick notes on this simple example:

1. An executable source file will start with PROGRAM PROGRAM_NAME and end with END PROGRAM PROGRAM_NAME
2. Fortran is a sequential language, thus we always declare variables, like INTEGER at the top of the program
3. The variable declaration will always indicate the type followed by two colons and then the variable name, e.g., INTEGER :: `i,j,k`
4. We'll look at more on I/O in Chap. 3, but for now know that in the WRITE statement the first * means "print to terminal" and the second * means that the output is "unformatted". We discuss formatted writing of data later in Chap. 7.4.2. In Fortran, an alternative to the WRITE statement above is the PRINT* command to produce an unformatted output of information to the terminal window.
5. Again, ***be aware that Fortran is not case sensitive***. So the output for firstExample.f90 would be the same, for example, if we had written `J = 7`.

It is important to note that just after the program is initialized we will ***always include the command*** IMPLICIT NONE.

```
PROGRAM program_Name
   IMPLICIT NONE
   ...
```

The command IMPLICIT NONE makes the use of undeclared variables illegal, leading to compiler errors. In Fortran, any undeclared variables that begin with letters between $i - n$ or $I - N$ are cast to integers, all other undeclared variables are cast to single precision. This can lead to bugs and lots of headaches, so we always ***turn off*** this feature by including IMPLICIT NONE.

Note, turning off implicit variable typing should be done within ***any and all*** procedures including the main PROGRAM as well as any MODULE, SUBROUTINE, or FUNCTION. The compiler flag `-fimplicit-none` in `gfortran` deactivates implicit typing globally. However, such a flag is not available in all Fortran compilers (for example no such flag exists in `ifort`). Thus, it is best practice to turn off implicit typing manually.

Once we save our simple program as firstExample.f90, we are ready to run. We open a terminal and move to the correct folder, where the source file is located, and then type in the command line:

`firstExample.f90 - Commands and Output`

```
gfortran firstExample.f90 -o firstExample
./firstExample
        10
```

Let us breakdown what was just done and gather general information on how to run Fortran programs:

1. When we run the compiler (the `gfortran` command above) it creates an executable file
2. The compiler flag `-o` lets us name the executable, in this case "firstExample". If you ***don't*** use this flag the executable has the default name "a.out"
3. To run the executable from the command line we use the ./command

2.2 Example: Add Two Real Numbers (in Double Precision)

Let's do another quick example to reinforce the Fortran syntax we just learned. Also, this example will introduce a general approach in Fortran used to manipulate real numbers. We have the pesudocode for a routine that adds two double precision real numbers:

Algorithm 2: *Real Numbers*: Add two real numbers and print

```
Procedure Real Numbers
$x \leftarrow \pi$
$y \leftarrow 2.5$
$z \leftarrow x + y$
Print $z$
End Procedure Real Numbers
```

Now we translate this second piece of pseudocode into workable a program. We save this program as "realAdd.f90".

`realAdd.f90`

```
PROGRAM realAdd
   IMPLICIT NONE
   INTEGER,PARAMETER       :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP)           :: x,y,z
   REAL(KIND=RP),PARAMETER :: pi = 4.0_RP*ATAN(1.0_RP)
!  Let's add some real numbers
   x = pi
   y = 2.5_RP
   z = x + y
   PRINT*,z
!
END PROGRAM realAdd
```

There are a few important aspects of this implementation for adding two real numbers that require discussion:

1. We define the number of digits of accuracy for real numbers in a rather unusual way. This is important to maintain calculations in double precision, which has sixteen digits of accuracy (so we input 15 in the intrinsic function because we start counting at 0). In this way, we define a parameter to indicate what is meant within a Fortran program by "real precision" called `RP`. Then we specify the accuracy of the REAL data type by including KIND=`RP`.

   In Fortran, if we don't specify a KIND and simply declare a real number with REAL`::x` the compiler defaults the data type of `x` to be a single precision value. We note that the flag `-fdefault-real-8` will tell the compiler to "upgrade" standard REAL declarations to double precision as well as any non-double constants like `1.0`. Just like with implicit typing, such flags are ***different*** across compilers. Thus, it is best to specify the KIND manually so it is always explicitly known what type a variable has along with its precision.

   Note, there is a DOUBLE PRECISION variable type in Fortran, but ***don't use it. It is deprecated.*** The reason is because double precision means different things on different machines. However, if you specify 16 digits with the intrinsic function SELECTED_REAL_KIND, you know that no matter what machine you use that you'll always have that many digits of accuracy. Thus, it makes your code portable and easy to change (if, say, you wanted 32 digits of accuracy you would simply change 15 to 31).

2. What does it mean for a variable to be a PARAMETER? It means, we define a number, in this case π, that will remain constant throughout a run of the program. One may be tempted to hard-code π to a certain number of digits, e.g. π `= 3.14159265359`. However, on different machines with different precisions you may lose accuracy when a hard-coded constant is truncated. To maintain portability of code ***try not hard-code any constants***. An accurate value of the constant π is easily recovered from standard trigonometric properties. But sometimes it may be unavoidable to truncate and hard-code a constant's value due to a complicated definition such as with the Euler-Mascheroni constant γ (it is an improper integral).

3. Inside the program instructions, how do we tell the compiler to treat a fixed number (often called a ***literal***) as a REAL(KIND=`RP`)? If we want the compiler to treat this number as a double precision value we must write `1.0_RP`. If we only type `1` in the program, then the compiler will treat `1` as an integer. If we type `1.0`, then the compiler treats `1.0` as a ***single precision*** value. Both of these treatments of literals can introduces spurious round-off errors in a program. As a final note, the syntax `1.d0` will treat the literal correctly as a double precision value. However, in these notes we prefer the explicit attachment of `_RP` to specify the precision of literals for clarity as well as flexibility of the code.

4. What's with all the exclamation points? The exclamation point, !, is the comment character in Fortran. Thus, anything that appears after an exclamation point is ignored by the compiler. Comments are there to make the code more readable for the author as well as for other people. Always remember, codes can never have too many comments so be as liberal with them as you like. It will only make your life easier when you go back to read code and remember what exactly it does weeks, months, or years down the road.

Finally, we compile and run the second example of adding two real numbers.

realAdd.f90 - Commands and Output

```
gfortran realAdd.f90 -o realAdd
./realAdd
        5.6415926535897931
```

Chapter 3: The Elements of a Program

We start this chapter with a couple of laundry lists (often with mini-examples) that outline the data types available in fortran, the syntax for common control sequences, and simple, unformatted file I/O. After we walk through the different program elements we will finish the chapter with an example problem from numerical integration.

3.1 Data Types

The most common data types in fortran are:

- INTEGER – Positive or negative whole number
- CHARACTER – Standard text ASCII character with one byte per character
- CHARACTER(LEN=4) – String of a specified length
- LOGICAL – Set to either .TRUE. or .FALSE.
- REAL(4) – Floating point number with 6 significant digits
- REAL(8) – Floating point number with 12 significant digits
- REAL(KIND=RP) – Floating point number with significant digits specified by RP = SELECTED_REAL_KIND(15), where 15 yields double precision (16 significant digits)
- COMPLEX(KIND=RP) – Two floating point numbers with significant digits specified by RP

Keep in mind, we can make any of the preceding data types a PARAMETER value in the PROGRAM as well.

Now that we know the data types available in fortran, we need to be careful when variables with different data types interact. This is because precision can be ***lost*** in such manipulations (for example when a REAL and an INTEGER interact). Note, this is another difference moving away from MATLAB! Let's examine a few situations of how fortran handles casting one variable type to another:

typesExample.f90

```fortran
PROGRAM typesExample
   IMPLICIT NONE
   INTEGER,PARAMETER :: RP = SELECTED_REAL_KIND(15)
   INTEGER           :: i,j,k
   REAL(KIND=RP)     :: x,y,z
!
   i = 7
   j = 4
   x = 3.7_RP
   y = 4.0_RP
   k = j**i
   PRINT* ! puts a carriage return for spacing
   PRINT*,'4**7= ',k ! to take a value to a power use the ** operator
   k = i/j
   PRINT*,'7/4= ',k ! division with integers always truncates down
   k = x
   PRINT*,'automatic conversion 3.7 to INT= ',k ! always truncated towards zero
   z = i
   PRINT*,'automatic conversion 7 to REAL= ',z ! converts integer to real
   z = i/j
   PRINT*,'automatic conversion 7/4 to REAL= ',z ! truncates then converts to real
   z = i/4.0_RP
   PRINT*,'automatic conversion 7/4.0_RP to REAL= ',z ! retains accuracy of a real
   z = 7.0_RP/j
   PRINT*,'automatic conversion 7.0_RP/4 to REAL= ',z ! retains accuracy of a real
   z = REAL(i,RP)/REAL(j,RP) ! compiler to treat the integers as reals
   PRINT*,'automatic conversion 7_RP/4_RP= ',z
!
END PROGRAM typesExample
```

We compile and run this example to illustrate how the different variable types are cast between one another. The output of the program also serves to clarify the meaning of the comments above.

typesExample.f90 - Commands and Output

```
gfortran typesExample.f90 -o typesExample
./typesExample

 4**7=        16384
 7/4=            1
 automatic conversion 3.7 to INT=            3
 automatic conversion 7 to REAL=    7.0000000000000000
 automatic conversion 7/4 to REAL=    1.0000000000000000
 automatic conversion 7/4.0_RP to REAL=    1.7500000000000000
 automatic conversion 7.0_RP/4 to REAL=    1.7500000000000000
 automatic conversion 7_RP/4_RP to REAL=    1.7500000000000000
```

3.2 Control Sequences

Next, we give a list of the most common control sequences available in fortran:

- IF/THEN/ELSE – Execute certain pieces of code based on a logical condition(s). The main logical logical operators are:
  - ⊛ Less than: $<$ or .LT.
  - ⊛ Less than or equal to: $<=$ or .LE.
  - ⊛ Greater than: $>$ or .GT.
  - ⊛ Greater than or equal to: $>=$ or .GE.
  - ⊛ Equal to: $==$ or .EQ.
  - ⊛ Not Equal: $/=$ or .NE.
  - ⊛ Logical and: .AND.
  - ⊛ Logical or: .OR.
  - ⊛ Logical not: .NOT.

  IF/THEN/ELSE Statement

  ```
  IF (i.EQ.7) THEN
     do something
  ELSE IF ((i.LT.5).AND.(i.GT.2)) THEN
     do something else
  ELSE
     and now for something completely different
  END IF
  ```

  If the condition only needs to execute a single expression the IF statement can be shortened and does not need a THEN qualifier. Also, this short example demonstrates how a LOGICAL type variable can be used.

Shortened IF Statement
```fortran
j = 6
isPositive = .TRUE.
IF (isPositive) PRINT*,j
```

- DO loops – Perform a piece of code in the loop structure a specified number of times. The bounds of the loop are INTEGERS.

DO Loop
```fortran
DO i = 1,13
   PRINT*,i
END DO! i
!  Loops can also run backwards
DO j = 9,0,-1 ! This means step backwards from 9 to 0, decrement by 1 each iteration
   PRINT*,j
END DO! j
!  In general, we have the bounds on the loop
DO j = start_value,end_value,increment ! the default increment is 1
   CODE
END DO! j
```

After the END DO it is helpful to note in a comment which iteration variable stops. This is purely a convention, but it helps organize larger loop nests for debugging purposes.

Loop Nest
```fortran
DO i = 1,4
   DO j = 2,5
      DO k = 6,0,-2
         PRINT*,i*j*k
      END DO! k
   END DO! j
END DO! i
```

We note that it is possible in fortran to put multiple items on a single line of code to reduce the size of the source code when we separate the commands with a semicolon. This can be done when assigning variable values, initiating loop nests, etc. In practice as well as throughout these notes we do not utilize this feature because it often makes the source code more difficult to understand. However, for completeness, we mention this possibility as one might encounter it when working on larger fortran projects. The following condensed loop nest will produce the same output as the previous loop nest:

Condensed Loop Nest
```fortran
DO i = 1,4; DO j = 2,5; DO k = 6,0,-2
   PRINT*,i*j*k
END DO; END DO; END DO
```

- DO WHILE loops – Perform a piece of code in the loop structure until a logical condition is met.

DO WHILE Loop
```fortran
b = 2
DO WHILE (b.NE.128)
   b = b*b
   PRINT*,b
END DO
```

- EXIT – Exit out of the current DO loop based on a logical argument.

EXIT Example
```
DO i = 1,35
   PRINT*,i
   IF (i**2.GT.55) THEN
      EXIT
   END IF
! alternatively could use shortened logical IF (i*i.GT.55) EXIT
END DO! i
```

- CYCLE – Increment to the next iteration in a DO loop based on a logical argument.

CYCLE Example
```
DO i = 1,6
   IF (i.EQ.4) THEN
      CYCLE
   END IF
   PRINT*,i
END DO! i
```

- STOP – This statement will halt the program. One example of its use is to prevent an infinite DO WHILE loop inside an iterative method. Once a program reaches a set `maxIts` the program ceases.

STOP Example
```
numIts = 0
maxIts = 500
DO WHILE (err.GT.tol) ! check the error against a preset tolerance threshold
...
perform an iterative method like Gauss-Seidel
...
numIts = numIts + 1
IF (numIts.GT.maxIts) THEN
   PRINT*,'Iteration failed to converge'
   STOP
END IF
END DO
```

3.3 Input/Output Constructs

Next, we examine the different options fortran offers for the input of data into a program to be operated on and output of data to the screen or to a file for visualization or other analysis. For now this section only covers unformatted read/write commands, but know that it is possible to format the data to suit one's needs. We cover formatted reads/writes in Chap. 7 Advanced Topics, as it is somewhat complicated. For the tasks in this note set as well as most fortran projects unformatted I/O will be sufficient.

3.3.1 I/O to the Screen

- Input – We can ask the user to provide information from the terminal. To do so, we use a READ(*,*) command to inform the PROGRAM that it should expect to read-in and store data from the screen inputted by the user. Again, the first * means "the screen" and the second * means "unformatted".

readScreenExample.f90
```
PROGRAM readExample
   IMPLICIT NONE
   INTEGER,PARAMETER :: RP = SELECTED_REAL_KIND(15)
   INTEGER           :: j
   REAL(KIND=RP)     :: x
   CHARACTER(LEN=20) :: name
!
   READ(*,*)'Enter an integer',j ! if you don't input an integer, an error is thrown
   READ(*,*)'Enter a real number',x
   READ(*,*)'Enter your name',name
!
   WRITE(*,*)j,x,name
!
END PROGRAM readExample
```

- Output – We've used this a couple times already, but it is restated here for reference purposes. It is important to print to the screen to inform the user what is happening in the PROGRAM, what part of the execution the PROGRAM has reached, etc. Also, we often print to the screen for debugging purposes. For example, we print variable values set during execution to check if they make sense in the context of the problem begin solved. Because output to the screen is used primarily for these kind of sanity checks, we only worry about unformatted output to the screen.

Unformatted Printing to Screen

```
a = 24601
! Both commands will produce the same output
PRINT*,'The number is',a
! or
WRITE(*,*)'The number is',a
```

3.3.2 I/O to a File

Having the user input runtime values might seem like a good idea at first. However, if you are actively working on a PROGRAM debugging and/or adding features it is extremely annoying to constantly input data for *every, single, run.* This is one reason putting runtime values into a file is useful as the user can quickly change one (or a few) values and immediately run the PROGRAM again to see the effect. Also, as mentioned previously, it is important to write the data produced by a PROGRAM to a file. This allows the user to analyze the data, create figures, etc. more easily than trying to interpret a large amount of data that is simply spat out onto the screen.

In order to input or output data to a file we first have to tell the PROGRAM to open a file. To do so, we use the OPEN command, which assigns an integer (sometimes called the `fileUnit`) to reference a file's name. If the file to be opened does not exist the PROGRAM will ***create*** an empty instance of the file with the given extension and assigned name. The file will be created in whatever directory contains the executable. It is also possible to open a file contained in a sub-folder from the directory containing the executable. However, the folder ***must*** already exist. If the file does not exist inside the sub-folder, then the PROGRAM will create the file just as before. If you try and open a file from a folder that does not exist, e.g. OPEN(13, `FILE='folder/file.dat'`), the source code will compile but at runtime the execution will throw an error.

Error Opening a File from a Non-Existent Folder

```
Fortran runtime error: Cannot open file 'folder/file.dat': No such file or directory
```

When we go to open a file for I/O it is important to not set the `fileUnit` to be 0, 5 or 6 as most compilers reserve those values in the following way:

- ***Standard Error*** is 0 – Used by programs to output error messages or diagnostics.
- ***Standard In*** is 5 – Used by programs to input data from the terminal, similar to READ(*,*).
- ***Standard Out*** is 6 – Used by programs to output data to the terminal, similar to WRITE(*,*).

Always remember, anytime you OPEN a file you should CLOSE it once you are done reading/writing. In particular, this is important because closing a file will free up the `fileUnit` for reuse elsewhere in the PROGRAM. Now, we can examine examples of unformatted I/O in Fortran.

- Input – To read in from a file we replace the first * in the READ statement with the `fileUnit` that has been assigned to a particular file name. By default Fortran reads in data line-by-line, advancing to a new line after each variable that is read. In Chap. 7.4 we examine how to alter this behavior. For the purpose of this example we assume that we have a file with the data

  testInput.dat
  ```
  7
  34.4635
  ```

  Unfortunately, we cannot put comments into `testInput.dat` to indicate what the values represent or where they should be assigned. We will revisit file reading later in Chap. 7.4 where we will develop nice file reading routines useful across Fortran projects. Note that Fortran will check if the data read-in from a file ***matches*** the data type of the variable the PROGRAM tries to save it in. If there is a mismatch Fortran will throw an error and the execution will stop. For example, if we tried to read the value 34.4635 from `testInput.dat` into the integer `j` we get the error of the form

  Error Read-In Data Mismatch
  ```
  Fortran runtime error: Bad integer for item 1 in list input
  ```

  For now we have the simple functionality of reading from a file.

  readFileExample.f90
  ```
  PROGRAM readFileExample
     IMPLICIT NONE
     INTEGER,PARAMETER :: RP = SELECTED_REAL_KIND(15)
     INTEGER           :: j
     REAL(KIND=RP)     :: x
  !
     OPEN(UNIT=33,FILE='testInput.dat')
     READ(33,*)j,x
     CLOSE(33)
  !
     PRINT*,j,x
  !
  END PROGRAM readFileExample
  ```

- Output – Similar to the input example we replace the first * in the WRITE statement with the `fileUnit`. Note, to output data to a file we must use a WRITE statement. The PRINT statement is only able to output information to the screen.

  writeFileExample.f90
  ```
  PROGRAM writeFileExample
     IMPLICIT NONE
     INTEGER,PARAMETER :: RP = SELECTED_REAL_KIND(15)
     REAL(KIND=RP)     :: x,y,z
  !
     x = 6.25_RP
     y = 1.15_RP
     z = x + y
  !
     OPEN(UNIT=75,FILE='testOutput.dat')
     WRITE(75,*)z
     CLOSE(75)
  !
  END PROGRAM writeFileExample
  ```

3.4 Example: Quadrature

We wrap-up this discussion with an example from numerical integration. For simplicity, this example will utilize I/O from the screen. Although, it is a good exercise to think about: How would we convert this example to use an input file?

Let us use a Riemann sum with left endpoints to approximate the value of the definite integral

$$I = \int_a^b \frac{x^2}{3}\,dx \approx \sum_{i=0}^{N-1} \frac{x_i^2}{3}\Delta x.$$

In order to approximate the value of this integral, we must decide how to divide apart the interval $[a, b]$ and distribute the sample points x_i, $i = 0, \ldots, N-1$. The simplest distribution of sample points is a uniform spacing such that $x_i = a + i\Delta x$, with

$$\Delta x = \frac{b-a}{N}.$$

We store the approximate integral value in the real variable "sum" and display the result to the screen. Next, we have pseudocode of the integral approximation, for a given value of a, b, and N:

Algorithm 3: *Left Riemann Sum*: Approximate an integral with the left Riemann sum.

```
Procedure Left Riemann Sum
Input: $a, b, N$

$\Delta x \leftarrow \frac{b-a}{N}$
$sum \leftarrow 0$
for $i = 0$ to $N-1$ do
    $x_i \leftarrow a + i\Delta x$
    $sum \leftarrow sum + (x_i^2/3)\cdot\Delta x$

Output: $sum$

End Procedure Left Riemann Sum
```

We next provide a Fortran implementation that reads in the values for a, b, and N from the screen and outputs the resulting integral approximation to the screen.

leftRiemann.f90

```fortran
PROGRAM leftRiemann
   IMPLICIT NONE
   INTEGER,PARAMETER :: RP =SELECTED_REAL_KIND(15)
   INTEGER           :: i,N
   REAL(KIND=RP)     :: a,b,x_i,dx,sum
!
   WRITE(*,*)' Enter a value for a, b, and number of sub-rectangles N'
   READ(*,*)a,b,N ! You don't include the RP when inputting real numbers
                  ! For example, just type -2.0,4.0,25
   dx  = (b-a)/N
   sum = 0.0_RP
   DO i = 0,N-1
      x_i = a + i*dx
      sum = sum + dx*(x_i*x_i/3.0_RP) ! could alternatively use x_i**2/3.0_RP
   END DO! i
!
   WRITE(*,*)' The integral is approximately',sum
!
END PROGRAM leftRiemann
```

Note, that we hard-coded the function that we wished to integrate. In the next Chapter, we will learn how to pass an arbitrary function $f(x)$ into the program. We save our program as leftRiemann.f90 and run the program to find:

```
leftRiemann.f90 - Commands and Output
gfortran leftRiemann.f90 -o leftRiemann
./leftRiemann
  Enter a value for a, b, and number of sub-rectangles N
-2.0,4.0,25
  The integral is approximately   7.5391999999999992
```

We can compare the Riemann approximation of the integral to the known solution which is

$$I = \int_{-2}^{4} \frac{x^2}{3}\, dx = \left.\frac{x^3}{9}\right|_{x=-2}^{4} = 8.$$

Chapter 4: Program Organization

Now that we know the basics of how to code in Fortran we will learn how to break a program into manageable parts. This makes code easier to read, easier to test, easier to debug, and easier to reuse. We hit the high notes of how to use the FUNCTION, SUBROUTINE, and MODULE constructs to break-up the workflow of a program. Here is a quick way to remember the difference between each type of construct:

- FUNCTION takes in multiple arguments and returns a single argument.
- SUBROUTINE takes in and returns multiple arguments.
- MODULE is a file that contains variable declarations, functions, and subroutines that can be made available to a program or another module.

When we discuss the FUNCTION and the SUBROUTINE we include special attributes that tell the compiler how to handle arguments passed to them by the user. There are three options:

- INTENT(IN) – Use with functions and subroutines. It informs the compiler that an argument may not be changed by the function/subroutine.
- INTENT(OUT) – Use with subroutines. It informs the compiler that an argument will return information from the subroutine to the calling procedure. The argument's value is ***undefined*** on entry to the procedure and must be given a value by some means.
- INTENT(INOUT) – Use with subroutines. It informs the compiler that an argument may transmit information into a subroutine, be operated upon (possibly even overwritten), and then returned to the calling procedure.

4.1 Functions

A FUNCTION in Fortran is a procedure that accepts multiple arguments and returns a single result. They come in two flavors intrinsic and external.

4.1.1 Intrinsic Functions

Intrinsic functions are built-in functions that do not need declared. There are ***lots*** of intrinsic functions available (a quick Google search reveals as much), some of them are

- EXP – exponential function
- SIN – sine, argument in radians (other trig functions available as well)
- ASIN – arcsine, argument in radians (other inverse trig functions available as well)
- LOG – natural logarithm
- LOG10 – common logarithm (base 10)
- ABS – absolute value
- SIGN – sign transfer function. This is a function of two variables with the definition

$$\text{SIGN}(x,y) = \begin{cases} \text{ABS}(x), & \text{if } \ y \geq 0, \\ -\text{ABS}(x), & \text{if } \ y < 0. \end{cases}$$

  The practical effect of this function is that SIGN(x,y) has the absolute value of x, but has the sign of y. Thus, the sign of y is *transferred* to x.

4.1.2 External Functions

These are procedures written by a user that can be called by another PROGRAM (or FUNCTION or SUBROUTINE). We'll write the external function and the PROGRAM that cals it in separate files. For this example we write a function to evaluate

$$f(x) = x^3 - x + \cos(x).$$

First, we write the FUNCTION

function.f90
```
FUNCTION f(x)
   IMPLICIT NONE
   INTEGER,PARAMETER        :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP),INTENT(IN) :: x
   REAL(KIND=RP)            :: f
!
   f = x**3 - x + COS(x)
!
END FUNCTION f
```

In `function.f90` we have a variable `f` with ***the same name*** as the FUNCTION. This is a special variable, known as the RESULT variable. It is the means by which a FUNCTION returns information to a calling procedure. Always remember that every FUNCTION ***must*** contain a variable that has the same name as the FUNCTION, and this variable ***must*** be assigned. In truth, the RESULT variable can have a different name from the function provided special syntax is invoked (explored in Chap. 7 Advanced Topics). But, for simplicity, we will use matching FUNCTION and RESULT variable names in this Chapter.

Next, we write the main program that invokes the function $f(x)$.

functionMain.f90
```
PROGRAM functionExample
   IMPLICIT NONE
   INTEGER,PARAMETER :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP)     :: x,y
   REAL(KIND=RP)     :: f
!
   x = 3.0_RP
   y = f(x)
   PRINT*,y ! the result is 23.010007503399553
!
END PROGRAM functionExample
```

Now that we have these two Fortran source files, how do we compile the program? We see that the main PROGRAM has a dependency of another source file that contains the FUNCTION $f(x)$. How do we ensure that the main PROGRAM has access to the other user written source code procedures? It turns out we have several options for compilation. We can:

1. Compile the files all at once on the command line. Note the main file `functionMain.f90` ***depends*** on a FUNCTION contained in `function.f90` file. Therefore, we put `function.f90` first in the compilation list and the main PROGRAM file last.

   Compile all files at once
   ```
   gfortran function.f90 functionMain.f90 -o functionExample
   ```

2. Compile the files one at a time using the command line. The compiler flag `-c` tells `gfortran` to compile a piece of code, but do not create an executable. This creates an additional file with a `.o` extension. We do the same to compile the main PROGRAM generating another `.o` file. When we go to compile the source to generate an executable we note that there is ***still*** a dependency of the main source file on the other. To

generate an executable of this PROGRAM we include the pre-compiled pieces of source code (in their proper order).

Compile files on at a time
```
! the -c flag means ‘compile’ but don’t create an executable
gfortran -c function.f90
gfortran -c functionMain.f90
gfortran function.o functionMain.o -o functionExample
```

3. Compile using a make file. For this example, one of the first two options are straightforward to compile the code and create an executable. However, once a Fortran project has a lot of source files with intertwined dependencies it becomes unwieldy due to the nature of ***inheritance***. Inheritance, essentially, boils down to a check: *if a program calls a function, then the function needs compiled* ***before*** *the program.* A make file makes the compilation of many source files simple because it automatically detects inheritance and which files need compiled. Later, in Chap. 4.4.1, we cover make file construction.

In general, a function in Fortran has the form:

General Function Format
```
FUNCTION function_name( passing arguments )
   IMPLICIT NONE
   INTEGER,PARAMETER    :: RP = SELECTED_REAL_KIND(15)
   data_type,INTENT(IN) :: arguments that shouldn’t be changed
   data_type            :: any arguments that can be changed
   data_type            :: function_name
! Local variables
   data_type            :: available locally, calling procedure doesn’t know of
                           these variables
   ...
   code
!
END FUNCTION function_name
```

4.2 Subroutines

Subroutines are more general than functions as they allow multiple input arguments ***and*** output results. However, we must be diligent and assign the correct INTENT of each argument in order to avoid bugs. Sometimes the the compiler will detect that an INTENT is incorrectly assigned and other times the compiler will not notice. This partially depends on the level of optimization as well as the types of compiler warnings activated by certain flags. Because we cannot always trust the compiler to detect a mistake in INTENT, it is best to manually keep track.

Let’s start with a simple SUBROUTINE example that mimics the FUNCTION we just wrote.

subroutine.f90
```
SUBROUTINE f(x,y)
   IMPLICIT NONE
   INTEGER,PARAMETER         :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP),INTENT(IN)  :: x
   REAL(KIND=RP),INTENT(OUT) :: y
!
   y = x**3 - x + COS(x) ! alternatively could use y = x*x*x - x + COS(x)
!
   RETURN
END SUBROUTINE f
```

In `subroutine.f90`, we have an argument that transmits the value of x into the subroutine and an argument y that returns information to the calling program. Note that the INTENT of each input argument matches accordingly.

Also, a subroutine includes the RETURN command, which passes control of the execution back to the calling procedure once the SUBROUTINE is completed.

Next, we implement another version of the main PROGRAM that calls the SUBROUTINE.

subroutineMain.f90

```fortran
PROGRAM subroutineExample
   IMPLICIT NONE
   INTEGER,PARAMETER :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP)     :: x,y
   REAL(KIND=RP)     :: f
!
   x = 3.0_RP
   CALL f(x,y)
   PRINT*,y ! the result is 23.010007503399553
!
END PROGRAM subroutineExample
```

The two source files for the SUBROUTINE example are compiled identically to the FUNCTION source code example (modulo file names). This is because the dependency of the source files between the two examples didn't change. Only the Fortran syntax did.

In general, a subroutine in Fortran has the form:

General Subroutine

```fortran
SUBROUTINE subroutine_name( passing arguments )
   IMPLICIT NONE
   INTEGER,PARAMETER        :: RP = SELECTED_REAL_KIND(15)
   data_type,INTENT(IN)     :: arguments that shouldn't be changed
   data_type,INTENT(INOUT)  :: arguments that may be changed
   data_type,INTENT(OUT)    :: arguments for output only
! Local variables
   data_type                :: available locally, calling procedure doesn't know of
                               these variables
   ...
   code
!
   RETURN
END SUBROUTINE subroutine_name
```

4.3 Passing Functions

Let's return to the example of using the left endpoint quadrature rule to approximate a definite integral. Recall we had to hard code the function we wanted to integrate. Now, however, with our knowledge of functions we can write a general quadrature routine and pass in different functions to integrate.

We divide the program into smaller pieces to help organize our thoughts as well as understand the dependencies between source files. We'll write two functions to integrate, the left hand rule for approximating a Riemann integral, and then a driver to collect the quadrature results. Let's consider two functions

$$f(x) = e^x - x^2 \quad \text{and} \quad g(x) = \sin(x) - \cos(x),$$

with corresponding source code

functions.f90

```fortran
FUNCTION f(x)
   IMPLICIT NONE
   INTEGER,PARAMETER         :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP),INTENT(IN)  :: x
   REAL(KIND=RP)             :: f
```

```
!
   f = EXP(x) - x**2 ! Alternatively could use x*x, faster than the "power" command
!
END FUNCTION f
!
FUNCTION g(x)
   IMPLICIT NONE
   INTEGER,PARAMETER        :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP),INTENT(IN) :: x
   REAL(KIND=RP)            :: g
!
   g = SIN(x) - COS(x)
!
END FUNCTION g
```

Next, we write a function that implements the left point rule quadrature method.

quadratureRules.f90

```
FUNCTION leftHandRule(a,b,N,func)
   IMPLICIT NONE
   INTEGER      ,PARAMETER  :: RP = SELECTED_REAL_KIND(15)
   INTEGER      ,INTENT(IN) :: N
   REAL(KIND=RP),INTENT(IN) :: a,b
   REAL(KIND=RP),EXTERNAL   :: func
   REAL(KIND=RP)            :: leftHandRule
! Local Variables
   INTEGER       :: i
   REAL(KIND=RP) :: dx,x
!
   dx  = (b-a)/N
   leftHandRule = 0.0_RP
   DO i = 0,N-1
      x   = a + i*dx
      leftHandRule = leftHandRule + func(x)*dx
   END DO! i
!
END FUNCTION leftHandRule
```

We applied the EXTERNAL attribute to the `func` input argument, which informs the compiler that the argument is a REAL FUNCTION and not a REAL variable. This, in effect, allows us to pass a function to another function.

Finally, we write a driver for the quadrature program.

QuadExample.f90

```
PROGRAM Quadrature
   IMPLICIT NONE
   INTEGER      ,PARAMETER :: RP = SELECTED_REAL_KIND(15)
   INTEGER      ,PARAMETER :: N = 150
   REAL(KIND=RP),PARAMETER :: a = -1.0_RP, b = 2.0_RP
   REAL(KIND=RP)           :: Integral
   REAL(KIND=RP),EXTERNAL  :: leftHandRule,f,g
!
   WRITE(*,*)
   Integral = lefthandRule(a,b,N,f)
   WRITE(*,*)'Approximation for f(x) integral: ',Integral
   WRITE(*,*)
   Integral = leftHandRule(a,b,N,g)
   WRITE(*,*)'Approximation for g(x) integral: ',Integral
!
END PROGRAM Quadrature
```

We finish the example by providing the output of the quadrature program for the two functions $f(x)$, $g(x)$.

QuadExample - Commands and Output

```
gfortran functions.f90 quadratureRules.f90 QuadExample.f90 -o quad
./quad

 Approximation for f(x) integral:    3.9809989288432961

 Approximation for g(x) integral:  -0.82136496727329411
```

4.4 Modules

Modules are similar to classes in C++. With modules we can easily organize code, allow for code reuse and obtain additional benefits we will discuss later (in Chap. 6). Modules are extremely useful and powerful and we will only scratch the surface of what modules can do. While source files can contain multiple modules or you can split one module between multiple files ***don't. Follow the philosophy of one file = one module***.

Modules have the general structure

General Module Structure

```
MODULE module_name
!  variable definitions
CONTAINS
!  functions and subroutines
END MODULE module_name
```

We briefly note that there is the option in a module to make data or routine PUBLIC or PRIVATE. If you set something to be PRIVATE than ***only*** the current module has access to it. Even if another module uses the current module anything set to PRIVATE will be "invisible" in some sense. The opposite is PUBLIC, which means that the data or routine is available for use outside the current module. By default everything in a module is set to be PUBLIC. This issue is more familiar to C/C++ programmers and is related to the issue of ***scope***, which is a word to describes which routines have access to data or other routines at a given point in the code.

Let's create a simple module that will make commonly needed constants available across a Fortran project. Thus far, in every FUNCTION, SUBROUTINE, and PROGRAM we've had to declare

```
INTEGER,PARAMETER :: RP = SELECTED_REAL_KIND(15)
```

It would be nice if we could always have the precision definition available to all procedures. We can use a module to achieve this goal. For good measure we'll also include the constant π, just in case we need it.

Constants.f90

```
MODULE Constants
   IMPLICIT NONE
   INTEGER      ,PARAMETER :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP),PARAMETER :: pi = 4.0_RP*ATAN(1.0_RP)
END MODULE Constants
```

Note that we still use IMPLICIT NONE in the module. Also, since the module Constants.f90 contains no functions or subroutines we omit the CONTAINS command.

Now in a main program (or a different module) we USE the module Constants.f90 to grant other pieces of source code access to its variables.

main.f90
```
PROGRAM main
   USE Constants
   IMPLICIT NONE
   ...
   code
END PROGRAM main
```

Notice we use the module ***before*** we invoke IMPLICIT NONE in `main.f90`. Also, we still use IMPLICIT NONE everywhere to ensure that implicit typing is always turned off.

To compile with modules is similar to compiling multiple `.f90` files, we simply have to worry about dependency and inheritance. Again, there is the option of compiling all files at once or individually. However, since we usually have more than a couple module (or source) files, there is an easier way to compile. We use makefiles.

4.4.1 Compiling Source Code with Makefiles

Makefiles are special format files that, together with the `make` utility, help automatically compile source code, get the correct dependencies and linking, and manage programming projects. A makefile has the basic components of defining the compiler to be used, any compiler flags, the name of the compiled executable, and then any objects to be built and their dependencies. Writing makefiles by hand is very tedious and we do it here only as an example. This discussion also reveals an advantage of using an IDE like Eclipse, because the makefile is ***automatically generated***.

Writing the syntax for a general makefile would be very unwieldy, confusing, and probably look like something found on the ship at Roswell. So instead, let's learn through an example. We'll create a makefile for the quadrature program written earlier in this chapter. The file name is `Makefile`. We state the compiler, any flags, and executable name at the top of the file, name the objects to built, and then state each source file to compile as well as any dependencies. The pound symbol `#` is the comment character for makefiles. We'll colorize the makefile to facilitate its readability. Also, we update the organization of the source files a bit and include the `Constants` module.

Makefile
```
F90 = gfortran
FFLAGS = #-O3, this is an optimizer which makes loops more efficient
EXECUTABLE = quad
# Object Files for build
OBJS = \
Constants.o \
functions.o \
quadratureRules.o \

$(EXECUTABLE) : $(OBJS)
        $(F90) $(FFLAGS) -o $(EXECUTABLE) ./QuadExample.f90 $(OBJS)

# Object dependencies and compilation
Constants.o : ./Constants.f90
        $(F90) $(FFLAGS) -c ./Constants.f90

functions.o : ./functions.f90 \
Constants.o
        $(F90) $(FFLAGS) -c ./functions.f90

quadratureRules.o : ./quadratureRules.f90 \
Constants.o
        $(F90) $(FFLAGS) -c ./quadratureRules.f90

# Utility targets
.PHONY: clean
```

```
clean:
	rm *.o *.mod
	rm quad
```

Now we show how to use a makefile to compile the program. If the make file has the name `Makefile`, then we simply type the command "`make`". It is also possible to give the make file a name, e.g. `quad.mak` where we use the `-f` flag to specify the file name "`make -f quad.mak`".

Makefile - Commands and Output

```
make
   gfortran  -c ./Constants.f90
   gfortran  -c ./functions.f90 Constants.o
   gfortran  -c ./quadratureRules.f90 Constants.o
   gfortran  -o quad ./QuadExample.f90 functions.o quadratureRules.o
./quad

   Approximation for f(x) integral:    3.9809989288432961

   Approximation for g(x) integral:  -0.82136496727329411
```

When we compile with `make` it detects which files have been altered and only compiles them. Thus, on an initial `make` all files are compiled. Then, say we implement another function

$$h(x) = \ln(x)\cos(2\pi x)$$

into `functions.f90`. Then a subsequent `build` would give

Makefile - New Function $h(x)$

```
make
   gfortran  -c ./functions.f90 Constants.o
   gfortran  -o quad ./QuadExample.f90 functions.o quadratureRules.o
```

Is it possible to do a "fresh" compilation of the PROGRAM? This is possible with the instructions under "Utility targets" comment in the `Makefile`. In this case this defines how to "clean" the project folder by removing any previously compiled `.o` and `.mod` files as well as the executable `quad`. The `clean` instruction is denoted .PHONY to specify to the `make` utility that the target is not a file. Once a project is cleaned the next compilation will touch all the source files generating new `.o`, `.mod` files, and an executable `quad`.

Clean and Re-Make

```
make clean
   rm *.o *.mod
   rm quad
make
   gfortran  -c ./Constants.f90
   gfortran  -c ./functions.f90 Constants.o
   gfortran  -c ./quadratureRules.f90 Constants.o
   gfortran  -o quad ./QuadExample.f90 functions.o quadratureRules.o
```

For this simple program a makefile is somewhat overkill. However, once we examine a lager project like in Chap. 9 or deal more with object oriented programming (Chaps. 6 and 10) makefiles are an excellent tool to manage the organization and compilation of Fortran projects.

Chapter 5: Arrays

The main advantage that Fortran has over other compiled programming languages is the ease with which it handles arrays. Because arrays are so easy to handle, Fortran is an ideal choice when writing code implementing mathematical formulae involving vector and matrix operations. Additionally, Fortran offers a great deal of freedom when indexing arrays, but keep in mind that the ***default initial index is*** 1.

It is worth noting that ***Fortran stores array information by column-by-column***, often referred to as ***column-major*** format. Knowing this helps when printing multi-dimensional arrays. Also, understanding how a programming language stores and manages memory helps write faster, more efficient code. For example, the order of loops inside a loop-nest has an impact on the speed of memory access and storage within cache. We won't go into any more detail on this topic, however it is something to keep in mind when writing programs later in particular when working with high performance computing (HPC) machines.

Note, since we know modules, that, ***if necessary, all the code examples from this point on will use the*** `Constants` ***module***.

5.1 Array Basics

In this chapter we will present examples of how to declare, manipulate, and print arrays. First, however, we list some useful intrinsic functions that we can use with or describe arrays:

- SIZE – Returns the total number of elements in an array.
- SHAPE – Returns the number of elements in each direction in an integer vector.
- LBOUND – Returns the lower index of each dimension of an array.
- UBOUND – Returns the upper index of each dimension of an array.
- MAXVAL – Returns the largest value in the array.
- MINVAL – Returns the smallest value in the array.
- MAXLOC – Returns the location of the largest value in an array.
- MINLOC – Returns the location of the smallest value in an array.
- SUM – Returns the sum of the elements of an array.
- TRANSPOSE – Returns the transpose of an array.
- DOT_PRODUCT – Returns the dot product of two one-dimensional array of the same size.
- MATMUL – Returns the product of two matrices. The dimensions must be consistent, i.e., (M, K) and (K, N). One nice thing is that the intrinsic function MATMUL can be used for matrix-matrix (BLAS3) as well as matrix-vector (BLAS2) operations. To compute a matrix-vector product you would provide the consistent dimensions, e.g., (N, N) and $(N, 1)$.

  Be aware that if you want to find the product of two matrices you ***must*** use the Fortran intrinsic function MATMUL. If you compute `A*B` in Fortran it will return the Hadamard (i.e. component wise) matrix product, so don't get confused.

  There is the question of optimality of the MATMUL function. Keep in mind that this intrinsic function assumes the matrices to be multiplied are dense. So, if you have a sparse matrix you ***want to exploit this sparsity*** for speed and memory reduction purposes and use a different routine, probably from `LAPACK`, a library that Fortran must properly link to use. The MATMUL function will be optimized by the compiler with many different flags. In `gfortran` the compiler flags: `-O`, `-O2`, `-O3`, `-Ofast` (listed from least to most aggressive) will all heavily optimize this function call. For modest matrix sizes (read as matrices smaller than 500×500) you won't need more sophisticated optimization for any matrix products provided the matrices dealt with are dense.

arrayExampleA.f90

```fortran
PROGRAM arrayExampleA
   USE Constants
   IMPLICIT NONE
! Can use the DIMENSION command or put an array's size after the variable name
   REAL(KIND=RP),DIMENSION(0:3)  :: array1
   REAL(KIND=RP),DIMENSION(4)    :: array2 ! same as 1:4
   REAL(KIND=RP),DIMENSION(-2:2) :: array3
   REAL(KIND=RP)                 :: array4(-1:1,0:2) ! same as DIMENSION(-1:1,0:2)
   REAL(KIND=RP)                 :: x,y
   INTEGER                       :: j,k

   x = 3.44_RP
   y = 1.25_RP

   WRITE(*,*)'size of array1',SIZE(array1)
   WRITE(*,*)'size of array2',SIZE(array2)
   WRITE(*,*)'size of array3',SIZE(array3)
   WRITE(*,*)'size of array4',SIZE(array4)
   WRITE(*,*)
   WRITE(*,*)'shape of array1',SHAPE(array1)
   WRITE(*,*)'shape of array2',SHAPE(array2)
   WRITE(*,*)'shape of array3',SHAPE(array3)
   WRITE(*,*)'shape of array4',SHAPE(array4)
   WRITE(*,*)
   WRITE(*,*)'lower index of array1',LBOUND(array1)
   WRITE(*,*)'lower index of array2',LBOUND(array2)
   WRITE(*,*)'lower index of array3',LBOUND(array3)
   WRITE(*,*)'lower indices of array4',LBOUND(array4)
   WRITE(*,*)
   WRITE(*,*)'upper index of array1',UBOUND(array1)
   WRITE(*,*)'upper index of array2',UBOUND(array2)
   WRITE(*,*)'upper index of array3',UBOUND(array3)
   WRITE(*,*)'upper indices of array4',UBOUND(array4)
   WRITE(*,*)
! Syntax to assign specific values to an array
   array1 = (/ -2.0_RP, 6.0_RP, pi, 1.1_RP /)
   array2 = (/ x-y, x+y, SIN(x)-EXP(y), y**x /)
! Assign values in a loop
   DO j = -2,2
      array3(j) = x**j
   END DO
! Can do array slicing using a colon, we assign each column of an array
   array4(-1,:) = (/  1.0_RP,-0.5_RP ,12.0_RP  /)
   array4(0,:)  = (/ -3.0_RP, 0.5_RP , 1.11_RP /)
   array4(1,:)  = (/  2.0_RP,-0.35_RP, 8.8_RP  /)
! Can print the array without loops
   WRITE(*,*)'array3 w/o loop',array3
   WRITE(*,*)'array4 w/o loop',array4
   WRITE(*,*)
! But it is always more readable if you print with loops and slice
   WRITE(*,*)'array3 w/loop'
   DO j = -2,2
      WRITE(*,*)array3(j)
   END DO! j
   WRITE(*,*)'array4 w/loop'
   DO j = -1,1
      WRITE(*,*),array4(j,:)
   END DO! j

   WRITE(*,*)'max of array1',MAXVAL(array1)
   WRITE(*,*)'location of max in array2',MAXLOC(array2)
   WRITE(*,*)'min of array3',MINVAL(array3)
   WRITE(*,*)'location of min in array4',MINLOC(array4)
END PROGRAM arrayExampleA
```

We compile and run this first array example and show the results. Note that the way you print arrays can turn gibberish into a nice two-dimensional output.

arrayExampleA.f90 - Output

```
 size of array1           4
 size of array2           4
 size of array3           5
 size of array4           9

 shape of array1           4
 shape of array2           4
 shape of array3           5
 shape of array4           3           3

 lower index of array1           0
 lower index of array2           1
 lower index of array3          -2
 lower indices of array4          -1           0

 upper index of array1           3
 upper index of array2           4
 upper index of array3           2
 upper indices of array4           1           2

 array3 w/o loop  8.45051379123850782E-002  0.29069767441860467        1.0000000000000000
   3.4399999999999999        11.833599999999999
 array4 w/o loop   1.0000000000000000       -3.0000000000000000        2.0000000000000000
 -0.50000000000000000       0.50000000000000000      -0.34999999999999998        12.0000000000
 00000        1.1100000000000001        8.8000000000000007

 array3 w/loop
  8.45051379123850782E-002
  0.29069767441860467
   1.0000000000000000
   3.4399999999999999
   11.833599999999999
 array4 w/loop
   1.0000000000000000      -0.50000000000000000        12.000000000000000
  -3.0000000000000000       0.50000000000000000        1.1100000000000001
   2.0000000000000000      -0.34999999999999998        8.8000000000000007
 max of array1   6.0000000000000000
 location of max in array2           2
 min of array3  8.45051379123850782E-002
```

```
location of min in array4           2           1
```

Next, we provide an example when we manipulate arrays which represent matrices. In the next example we also write a quick subroutine to help print the arrays in the style we want. Let us write a SUBROUTINE to print an array to the screen. Also, the next example explores how to manipulate values, specifically INTEGERs stored in a two-dimensional array.

arrayExampleB.f90

```
PROGRAM arrayExampleB
   IMPLICIT NONE
   INTEGER,DIMENSION(2,2) :: A,B
   INTEGER                :: i,j
! Initialize the two arrays. Note throughout this example the arrays are operated on in-place
! So these values get overwritten during execution
   A(1,:) = (/ 1 , 2 /)
   A(2,:) = (/ 3 , 4 /)
   B      = 1 ! sets every element in the array B to 1

   WRITE(*,*)'A='
   CALL PrintIntegerMatrix(A,2,2)
   WRITE(*,*)'B='
   CALL PrintIntegerMatrix(B,2,2)

   B(2,2) = B(2,2) + 3
   WRITE(*,*)'Add 3 to B(2,2)'
   CALL PrintIntegerMatrix(B,2,2)

   A = 2*A
   WRITE(*,*)'A=2*A'
   CALL PrintIntegerMatrix(A,2,2)

   A = A**2 - 1 ! These operations are done component-wise. This also applies for Fortran
                ! intrinsics like SIN, EXP, etc... as well
   WRITE(*,*)'A=A**2-1'
   CALL PrintIntegerMatrix(A,2,2)

   A = A+B
   WRITE(*,*)'A+B'
   CALL PrintIntegerMatrix(A,2,2)

   A = MATMUL(A,B)
   WRITE(*,*)'A=AB'
   CALL PrintIntegerMatrix(A,2,2)

   A = TRANSPOSE(A)
   WRITE(*,*)'A = A^T'
   CALL PrintIntegerMatrix(A,2,2)

   WRITE(*,*)'Dot product of col 1 of A with col 2 of A',DOT_PRODUCT(A(:,1),A(:,2))
   WRITE(*,*)'Dot product of row 1 of A with row 2 of A',DOT_PRODUCT(A(1,:),A(2,:))
END PROGRAM arrayExampleB
!
!//////////////////////////////////////////////////////////////////////
!
SUBROUTINE PrintIntegerMatrix(mat,N,M)
```

```fortran
   IMPLICIT NONE
   INTEGER,INTENT(IN) :: N,M
   INTEGER,INTENT(IN) :: mat(N,M)
! Local Variable
   INTEGER            :: i

   DO i = 1,N
      WRITE(*,*)mat(i,:)
   END DO! i
   WRITE(*,*)
!
   RETURN
END SUBROUTINE PrintIntegerMatrix
```

Again, we provide the output of the second array example for instructive purposes.

arrayExampleB.f90 - Commands and Output

```
gfortran arrayExampleB.f90 -o arrayB
./arrayB
 A=
           1           2
           3           4

 B=
           1           1
           1           1

 Add 3 to B(2,2)
           1           1
           1           4

 A=A*2
           2           4
           6           8

 A=A**2-1
           3          15
          35          63

 A+B
           4          16
          36          67

 A=AB
          20          68
         103         304

 A = A^T
          20         103
          68         304

 Dot product of col 1 of A with col 2 of A       22732
 Dot product of row 1 of A with row 2 of A       32672
```

5.2 Example: Interpolation

Next, we apply Fortran and its array capabilities for an example common to numerical analysis. That is, the creation of a polynomial interpolant of a function at a given set of interpolation nodes. Thus, this is a natural setting for using arrays, e.g., to store set of interpolation nodes in a one-dimensional array. For this example we also build upon the notion of PROGRAM structure and divide the process of creating a polynomial interpolant into smaller pieces. However, we first give some brief background on ***what kind*** of polynomial interpolation technique we will implement because there are several flavors available. Note this example implements a "classical" version of polynomial interpolation. Therefore, the interpolation nodes are taken to be unique and we assume no knowledge of the derivative of the function $f(x)$ (i.e. no Hermite type interpolation).

Generally, we wish to interpolate a function $f(x)$ given a set of interpolation points $\{x_i\}_{i=0}^N$ and the function evaluated at those points $\{y_i\}_{i=0}^N$, where $y_i = f(x_i)$. Omitting some detail, we know that Newton divided differences provide an efficient way to create an interpolating polynomial. Consider the two sets of points (x_0, y_0) and (x_1, y_1). The polynomial interpolant of order 1 is given by the straight line:

$$p_1(x) = y_0 + \underbrace{\frac{y_1 - y_0}{x_1 - x_0}}_{y[x_0,x_1]}(x - x_0),$$

where we introduce the standard notation for the first Newton divided difference coefficient $y[x_0, x_1]$. Notice that we have consistency at the endpoints, $p_1(x_0) = y_0$ and $p_1(x_1) = y_1$. A series of polynomials can then be constructed in a sequence based on their order, i.e., depending on the number of interpolation points available:

$$\begin{aligned} p_0(x) &= y_0, \\ p_1(x) &= y_0 + y[x_0, x_1](x - x_0), \\ p_2(x) &= y_0 + y[x_0, x_1](x - x_0) + y[x_0, x_1, x_2](x - x_0)(x - x_1), \\ &\vdots \end{aligned}$$

Then the interpolating polynomial, P_N, created with Newton divided differences is given by the sum of the preceding sequence, i.e.,

$$P_N(x) = \sum_{i=0}^{N} c_i \prod_{j=0}^{i-1}(x - x_j),$$

where the coefficients c_i, $i = 0, \ldots, N$

Next, we detail the procedures necessary to have a proper interpolation routine based on Newton divided differences. We slightly optimize how the coefficients and interpolating polynomial is constructed in the given subroutines. We then collect all the interpolation procedures into a module that we use with the main program.

Algorithm 4: *Newton Coefficients*: Compute interpolation coefficients from Newton divided differences.

```
Procedure Newton Coefficients
Input: $\{x_i\}_{i=0}^N, \{y_i\}_{i=0}^N, N$

$c_0 \leftarrow y_0$
for $k = 1$ to $N$ do
    $d \leftarrow x_k - x_{k-1}$
    $u \leftarrow c_{k-1}$
    for $j = k - 2$ to $0$ do
        $u \leftarrow u(x_k - x_j) + c_j$
        $d \leftarrow d(x_k - x_j)$

    $c_k \leftarrow (y_k - u)/d$

Output: $\{c_i\}_{i=0}^N$

End Procedure Newton Coefficients
```

Algorithm 5: *Newton Interpolating Polynomial*: Evaluate interpolant at a point

```
Procedure Newton Interpolating Polynomial
Input: $x, \{x_i\}_{i=0}^N, \{c_i\}_{i=0}^N, N$

$P(x) \leftarrow 0$
for $i = 0$ to $N$ do
    $p \leftarrow 1$
    for $j = 0$ to $i - 1$ do
        $p \leftarrow p(x - x_j)$

    $P(x) \leftarrow P(x) + c_i p$

Output: $P(x)$

End Procedure Newton Interpolating Polynomial
```

Algorithm 6: *Interpolated Values*: Interpolate to a set of nodes stored in $\mathbf{x}^{new}$.

```
Procedure Interpolated Values
Uses: Algorithm 5
Input: $\{c_i\}_{i=0}^N, \{x_i\}_{i=0}^N, \{x_i^{new}\}_{i=0}^N, N$

for $k = 0$ to $N$ do
    $y_k^{new} \leftarrow$ Newton Interpolating Polynomial$(x_k^{new}, \{x_i\}_{i=0}^N, \{c_i\}_{i=0}^N, N)$

Output: $\{y_i^{new}\}_{i=0}^N$

End Procedure Interpolated Values
```

We amalgamate the interpolation and convenience procedures for printing results to a file, for plotting purposes, into a MODULE. We then use this MODULE in a driving program to test the fidelity of the Newton divided difference interpolating polynomial. Note we only check the polynomial against the function $f(x)$ in the "eyeball" norm just as a sanity check. To be complete we should also perform an in-depth analysis of the approximation errors and convergence on different sets of interpolation nodes.

interpolationRoutines.f90

```fortran
MODULE InterpolationRoutines
   USE Constants
   IMPLICIT NONE
! No variable declarations
CONTAINS
   SUBROUTINE Interpolate(C,X,Xnew,Ynew,N)
      IMPLICIT NONE
      INTEGER                      ,INTENT(IN)  :: N
      REAL(KIND=RP),DIMENSION(0:N)  ,INTENT(IN)  :: C,X
      REAL(KIND=RP),DIMENSION(0:2*N),INTENT(OUT) :: Xnew,Ynew
      REAL(KIND=RP),EXTERNAL                     :: Poly_Interpolant
!  Local variables
      INTEGER                                    :: i

      DO i = 0,2*N
         Xnew(i) = i*(2.0_RP*pi)/N
         Poly_Interpolant(Xnew(i),X,C,N,Ynew(i))
      END DO! i
!
      RETURN
   END SUBROUTINE Interpolate
!
!//////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE NewtonCoeff(X,Y,C,N)
      IMPLICIT NONE
      INTEGER                      ,INTENT(IN)  :: N
      REAL(KIND=RP),DIMENSION(0:N),INTENT(IN)  :: X,Y
      REAL(KIND=RP),DIMENSION(0:N),INTENT(OUT) :: C
!  Local Variables
      REAL(KIND=RP)                            :: d,u
      INTEGER                                  :: i,j

      C(0) = Y(0)
      DO j = 1,N
         d = X(j) - X(j-1)
         u = C(j-1)
         DO i = j-2,0,-1
            u = u*(X(j)-X(i))+C(i)
            d = d*(X(j)-X(i))
         END DO! i
         C(j) = (Y(j)-u)/d
      END DO! j
!
      RETURN
   END SUBROUTINE NewtonCoeff
!
!//////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE Poly_Interpolant(x,nodes,coeff,N,y)
      IMPLICIT NONE
      INTEGER                      ,INTENT(IN)  :: N
      REAL(KIND=RP),DIMENSION(0:N),INTENT(IN)  :: nodes,coeff
      REAL(KIND=RP)                ,INTENT(IN)  :: x
      REAL(KIND=RP)                ,INTENT(OUT) :: y
!  Local variables
      INTEGER       :: i,j
      REAL(KIND=RP) :: temp

      y = 0.0_RP
      DO i = 0,N
         temp = 1.0_RP
         DO j = 0,i-1
            temp = temp*(x-nodes(j))
         END DO! j
         y = y + coeff(i)*temp
      END DO! i
!
      RUTURN
   END SUBROUTINE Poly_Interpolant
!
```

```
!////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE ReadInArrays(X,Y,N)
      IMPLICIT NONE
      INTEGER                       ,INTENT(IN)  :: N
      REAL(KIND=RP),DIMENSION(0:N),INTENT(OUT) :: X,Y
!  Local variables
      INTEGER :: i
!
      OPEN(12,FILE="poly.dat")
      DO i = 0,N
         READ(12,*)X(i),Y(i)
      END DO! i
!
      RETURN
   END SUBROUTINE ReadInArrays
!
!////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE WriteData(X,Y,N)
      IMPLICIT NONE
      INTEGER                       ,INTENT(IN) :: N
      REAL(KIND=RP),DIMENSION(0:N),INTENT(IN) :: X,Y
!  Local Variables
      INTEGER :: i
!
      OPEN(12,FILE='output.dat')
      DO i = 0,N
         WRITE(12,*)X(i),Y(i)
      END DO! i
      CLOSE(12)
!
      RETURN
   END SUBROUTINE WriteData
END MODULE InterpolationRoutines
```

With the interpolation module in place, we have the tools necessary to write and compile a driver program.

interpolationDriver.f90

```
PROGRAM interpolationDriver
   USE InterpolationRoutines
   IMPLICIT NONE
   INTEGER                        :: N = 12
   REAL(KIND=RP),DIMENSION(0:N)   :: X,Y,c
   REAL(KIND=RP),DIMENSION(0:2*N) :: newX,newY
   INTEGER                        :: i
! This ensures that the function data is available in the file poly.dat for the later routines
   OPEN(13,FILE='poly.dat')
   DO i = 0,N
      X(i) = i*(2.0_RP*pi)/N
      Y(i) = COS(X(i))
      WRITE(13,*)X(i),Y(i)
   END DO! i
   CLOSE(13)
!
```

```
   CALL ReadInArrays(X,Y,N)
   CALL NewtonCoeff(X,Y,c,N)
   CALL Interpolate(c,X,newX,newY,N)
   CALL WriteData(newX,newY,N)
!
END PROGRAM interpolationDriver
```

We compile and run the interpolation program and plot the result of the Newton divided difference polynomial of order 12. The result passes the eyeball norm, because the interpolation looks correct and approximates the function $f(x) = \cos(x)$ on the interval $[0, 2\pi]$. As previously mentioned, to fully verify the code and its interpolation functions properly, we ***need*** to compare the numerical errors, using multiple functions, to theoretical predictions and ensure consistency.

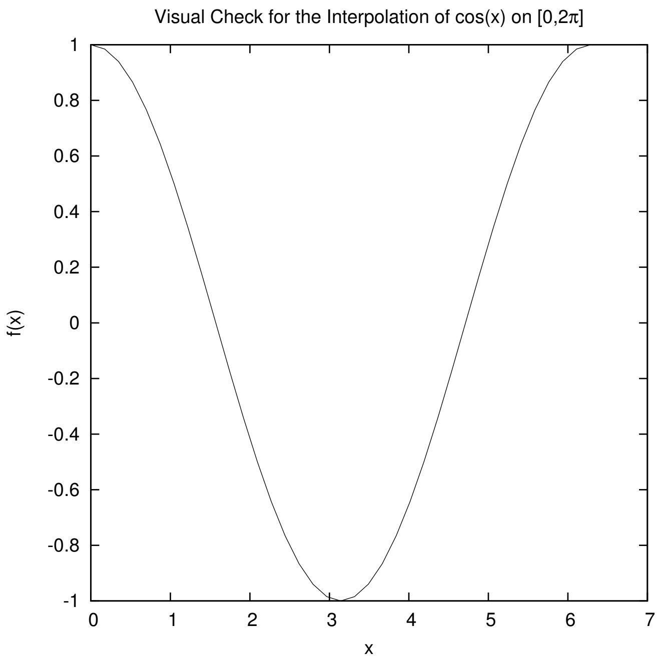

Figure 1: Newton interpolating polynomial run with $N = 12$.

Chapter 6: Object Oriented Programming

Object Oriented Programming (OOP) is a programming method that represents concepts as "objects" that contain data and procedures which operate on the data. Objects offer an effective and structured way to organize computer programs. We note that `Fortran95` is not *truly* object oriented, but can be "tricked" into behaving like it is. Object Oriented Fortran is available in later Fortran distributions (like `Fortran2003`); however, we do not discuss it in these notes.

In Fortran, modules are one important component of OOP. In this chapter we introduce the other component of OOP as well as expanding the power with which we use modules. Also, unlike with `C++`, Fortran programmers have to handle their own constructors and destructors for objects. Generally, this means one needs to be careful that any memory allocation (in a constructor) has a matching memory deallocation (in a destructor). This is especially important when using dynamic arrays, a topic discussed in Chap. 7.2.

6.1 Derived Types

We have seen the available intrinsic data types, like INTEGER, but is this all that is available to us? It turns out no! We can create our own data types. Because these user defined types contain instances of the intrinsic data types they are called ***derived types.*** We'll give a few examples of derived types that could come up in different computing projects. We also show how to access elements of a derived type using a `%` command.

DerivedEx1.f90

```fortran
PROGRAM DerivedEx1
   USE Constants
   TYPE Coordinate3D
      REAL(KIND=RP),DIMENSION(3) :: x ! stores (x,y,z)
   END TYPE Coordinate3D
   TYPE(Coordinate3D) :: point
!
   point%x(1) =  6.0_RP
   point%x(2) =  0.0_RP
   point%x(3) = -1.0_RP
!
   WRITE(*,*)point
!
END PROGRAM DerivedEx1
```

DerivedEx2.f90

```fortran
PROGRAM DerivedEx2
   USE Constants
   IMPLICIT NONE
   TYPE Coordinate3D
      REAL(KIND=RP),DIMENSION(3) :: x ! stores (x,y,z)
   END TYPE Coordinate3D
! One derived type can contain another derived type
   TYPE Pixel
      TYPE(Coordinate3D) :: position
      INTEGER            :: color(3) ! stores three integer RGB values
   END TYPE Pixel
!
   TYPE(Pixel) :: pixel1,pixel2
! set the RGB values for pixel 1
   pixel1%color(1) = 255
   pixel1%color(2) = 0
   pixel1%color(3) = 255
! set the position of the pixel. Note the nesting of the % command when we have a derived
! type that contains another derived type
   pixel1%position%x(1) = 2.0_RP
   pixel1%position%x(2) = 200.0_RP
   pixel1%position%x(3) = 50.0_RP
!
   WRITE(*,*)pixel1
! A more compact way to set values in the pixel derived type uses array type notation
   pixel2 = Pixel(Coordinate3D( (/ 15.0_RP/6.0_RP, 125.0_RP, 220.0_RP /) ),(/ 123,68,23 /))
!
   WRITE(*,*)pixel2
!
END PROGRAM DerivedEx2
```

6.2 Example: Matrices

Now let's combine the concepts of derived types and modules to create a matrix object as well as functions and subroutines that operate on said matrix object. In the module, which defines the matrix TYPE, we use a standard notational convention form OOP and refer to the object that is operated on as `this`.

MatrixModule.f90

```
MODULE MatrixModule
   USE Constants
   IMPLICIT NONE
   TYPE Matrix
      REAL(KIND=RP) :: element
   END TYPE Matrix
!
CONTAINS
!
   REAL(KIND=RP) FUNCTION OneNorm(this)
      TYPE(Matrix),INTENT(IN) :: this(:,:) ! colons as the dimension mean arbitrary size
!  Local variables
      REAL(KIND=RP) :: colSum
      INTEGER       :: j,m,n
!
      m = SIZE(this(:,1))
      n = SIZE(this(1,:))
      OneNorm = 0.0_RP
      DO j = 1,n
         colSum  = SUM(ABS(this%(:,j)%element))
         OneNorm = MAX(OneNorm,colSum)
      END DO! j
!
   END FUNCTION OneNorm
!
!//////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE matrix_real_scalar(this,real_scalar)
      REAL(KIND=RP),INTENT(IN)  :: real_scalar
      TYPE(Matrix) ,INTENT(OUT) :: this
!
      this%element = real_scalar
!
```

```fortran
      RETURN
   END SUBROUTINE matrix_real_scalar
!
!//////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE matrix_real_matrix(this,real_matrix)
      REAL(KIND=RP),INTENT(IN)  :: real_matrix
      TYPE(Matrix) ,INTENT(OUT) :: this
!
      this(:,:)%element = real_matrix(:,:)
!
      RETURN
   END SUBROUTINE matrix_real_scalar
!
END MODULE MatrixModule
```

MatrixMain.f90

```fortran
PROGRAM MatrixMain
   USE MatrixModule
   IMPLICIT NONE
   INTEGER,PARAMETER :: N = 2
   TYPE(Matrix)      :: mat1(N,N)
   INTEGER           :: i,j
   REAL(KIND=RP)     :: array(N,N)
!
   DO i = 1,N
      DO j = 1,N
         array(i,j) = 2.0_RP**i + 3.0_RP**j
      END DO! j
   END DO! i
   CALL matrix_real_matrix(mat1,array)
   WRITE(*,*)OneNorm(mat1)
   CALL matrix_real_scalar(mat1(1,2),2.3_RP)
!
END PROGRAM MatrixMain
```

These subroutines and function all work, and will yield correct values. However, it can become unwieldy to keep track of all the procedures in a module. We can simplify our lives if we use an INTERFACE.

6.3 Interfaces

We know the two major components of Object Oriented Programming in Fortran: Derived types and Modules that we use to organize data, functions, and subroutines. Now, we learn about the INTERFACE construct that will make modules easier to use.

MatrixModule2.f90

```fortran
MODULE MatrixModule
   USE Constants
   IMPLICIT NONE
   TYPE Matrix
      REAL(KIND=RP) :: element
   END TYPE Matrix
!
   INTERFACE ASSIGNMENT(=)
      MODULE PROCEDURE matrix_real_scalar,matrix_real_matrix
   END INTERFACE
!
CONTAINS
!
   REAL(KIND=RP) FUNCTION OneNorm(this)
      TYPE(Matrix),INTENT(IN) :: this(:,:) ! colons as the dimension mean arbitrary size
!   Local variables
      REAL(KIND=RP) :: colSum
      INTEGER       :: j,m,n
!
      m = SIZE(this(:,1))
      n = SIZE(this(1,:))
      OneNorm = 0.0_RP
      DO j = 1,n
         colSum  = SUM(ABS(this%(:,j)%element))
         OneNorm = MAX(OneNorm,colSum)
      END DO! j
   END FUNCTION OneNorm
!
!//////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE matrix_real_scalar(this,real_scalar)
      REAL(KIND=RP),INTENT(IN)  :: real_scalar
      TYPE(Matrix) ,INTENT(OUT) :: this
!
      this%element = real_scalar
!
      RETURN
   END SUBROUTINE matrix_real_scalar
!
!//////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE matrix_real_matrix(this,real_matrix)
      REAL(KIND=RP),INTENT(IN)  :: real_matrix
      TYPE(Matrix) ,INTENT(OUT) :: this
!
      this(:,:)%element = real_matrix(:,:)
!
      RETURN
   END SUBROUTINE matrix_real_scalar
END MODULE MatrixModule
```

MatrixMain2.f90

```
PROGRAM MatrixMain2
   USE MatrixModule
   IMPLICIT NONE
   INTEGER,PARAMETER :: N = 2
   TYPE(Matrix)      :: mat1(N,N)
   INTEGER           :: i,j
   REAL(KIND=RP)     :: array(N,N)
!
   DO i = 1,N
      DO j = 1,N
         array(i,j) = 2.0_RP**i + 3.0_RP**j
      END DO! j
   END DO! i
!   CALL matrix_real_matrix(mat1,array)
   mat1 = array
   WRITE(*,*)OneNorm(mat1)
!   CALL matrix_real_scalar(mat1(1,2),2.3_RP)
   mat1(1,1) = 2.3_RP
!
END PROGRAM MatrixMain2
```

It doesn't look like we have done a lot, but the INTERFACE construct associated two subroutines in the module MatrixModule2 with the assignment = operator. In effect we ***overloaded*** the = operator to use two different subroutines. So when we invoke the = operator the compiler will check to see if either of those subroutines work with the data types involved.

Further, it is possible to use an INTERFACE to simplify calls to Fortran's intrinsic functions. For example, let's change a call to MATMUL into the * operator. We'll add this capability to the previous module.

MatrixModule3.f90

```
MODULE MatrixModule
   USE Constants
   IMPLICIT NONE
!
   TYPE Matrix
      REAL(KIND=RP) :: element
   END TYPE Matrix
!
   INTERFACE ASSIGNMENT(=)
      MODULE PROCEDURE matrix_real_scalar,matrix_real_matrix
   END INTERFACE
!
   INTERFACE OPERATOR(*)
      MODULE PROCEDURE matrix_matrix_multiply
   END INTERFACE
!
CONTAINS
!
   REAL(KIND=RP) FUNCTION OneNorm(this)
      TYPE(Matrix),INTENT(IN) :: this(:,:) ! means arbitrary size
!  Local variables
      REAL(KIND=RP) :: colSum
      INTEGER       :: j,m,n
!
      m = SIZE(this(:,1))
      n = SIZE(this(1,:))
      OneNorm = 0.0_RP
      DO j = 1,n
         colSum  = SUM(ABS(this%(:,j)%element))
         OneNorm = MAX(OneNorm,colSum)
      END DO! j
!
   END FUNCTION OneNorm
!
!//////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE matrix_real_scalar(this,real_scalar)
```

```
      REAL(KIND=RP),INTENT(IN)  :: real_scalar
      TYPE(Matrix) ,INTENT(OUT) :: this
!
      this%element = real_scalar
!
      RETURN
   END SUBROUTINE matrix_real_scalar
!
!//////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE matrix_real_matrix(this,real_matrix)
      REAL(KIND=RP),INTENT(IN)  :: real_matrix
      TYPE(Matrix) ,INTENT(OUT) :: this
!
      this(:,:)%element = real_matrix(:,:)
!
      RETURN
   END SUBROUTINE matrix_real_scalar
!
!//////////////////////////////////////////////////////////////////////////
!
   REAL(KIND=RP) FUNCTION matrix_matrix_multiply(this1,this2) RESULT(this_prod)
      TYPE(Matrix),INTENT(IN) :: this1(:,:),this2(:,:)
      TYPE(Matrix)            :: mat_prod(SIZE(this1,1),SIZE(this2,2))!pull correct dimensions
!
      this_prod(:,:)%element = MATMUL(this1(:,:)%element,this2(:,:)%element)
!
   END FUNCTION matrix_matrix_multiply
!
END MODULE MatrixModule
```

In the FUNCTION written in the above module we see that there is not a variable with the same name as the function. Instead, we have a variable with the RESULT denomination. This lets the compiler know that the RESULT variable has the type of the FUNCTION (in this case REAL(KIND=RP) with the name this_prod. This feature is useful if the name of the FUNCTION is long or not descriptive of the information it returns.

MatrixMain3.f90
```
PROGRAM MatrixMain3
   USE MatrixModule
   IMPLICIT NONE
   INTEGER,PARAMETER :: N = 3
   TYPE(Matrix)      :: mat1(N,N)
   INTEGER           :: i,j
   REAL(KIND=RP)     :: array(N,N)
!
   DO i = 1,N
      DO j = 1,N
         array(i,j) = 2.0_RP**i + 3.0_RP**j
      END DO! j
   END DO! i
!   CALL matrix_real_matrix(mat1,array)
   mat1 = array
   PRINT*
   WRITE(*,*)'The 1-norm of the matrix A is 'OneNorm(mat1)
   PRINT*
```

```
!   CALL matrix_real_scalar(mat1(1,2),2.3_RP)
   mat1(1,1) = 2.3_RP
   mat1 = mat1*mat1
   PRINT*,'A*A = '
   DO i = 1,N
      PRINT*,mat1(i,:)
   END DO! i
   PRINT*
!
END PROGRAM MatrixMain3
```

We provide the output of the final matrix object for instructive purposes.

arrayExampleB.f90 - Commands and Output

```
gfortran MatrixModule3.f90 MatrixMain3.f90 -o MatrixMultiply
./MatrixMultiply

The 1-norm of the matrix A is   95.000000000000000

A*A =
421.00000000000000        691.00000000000000        1501.0000000000000
467.00000000000000        773.00000000000000        1691.0000000000000
559.00000000000000        937.00000000000000        2071.0000000000000
```

If desired, you could also add subroutines to this module, and possibly INTERFACE functionality. For example, one could add an inversion call A/B, a LU decomposition, the Thomas Algorithm, printing arrays, or other matrix operations to the Fortran code. This would give your Fortran implementation a more MATLAB feel.

Chapter 7: Advanced Topics

In this chapter we cover selected advanced topics in Fortran programming. All the topics come in handy to add extra functionality to programs, but the feature that you will use most often is dynamic arrays, i.e., an array the size of which can be redefined at execution time.

7.1 Overloading Operators

We've seen a few examples of overloading ***intrinsic*** assignment operators, like the = operator. However, we can also define our own operators using interfaces. For example, we can create dot and cross product operators for vector calculations:

Dot and Cross Product Operators

```
MODULE DotCrossProduct
   USE Constants
   IMPLICIT NONE
!  Operator syntax: u.DOT.v
   INTERFACE OPERATOR(.DOT.)
      MODULE PROCEDURE dotproduct
   END INTERFACE
!  Operator syntax: u.CROSS.v
   INTERFACE OPERATOR(.CROSS.)
      MODULE PROCEDURE crossproduct
   END INTERFACE
!!!
CONTAINS
!!!
   REAL(KIND=RP) FUNCTION dotproduct(vec1,vec2) RESULT(dot)
      REAL(KIND=RP),INTENT(IN) :: vec1(:),vec2(:)
      IF (SIZE(vec1).EQ.SIZE(vec2)) THEN
         dot = DOT_PRODUCT(vec1,vec2)
      ELSE
         dot = 0.0_RP
         PRINT*,'ERROR: Vector size mismatch'
      END IF
!
   END FUNCTION dotproduct
!
   REAL(KIND=RP) FUNCTION crossproduct(vec1,vec2)
      REAL(KIND=RP),INTENT(IN) :: vec1(3),vec2(3)
      REAL(KIND=RP)            :: cross(3)
      IF (SIZE(vec1).EQ.SIZE(vec2)) THEN
         cross(1) = vec1(2)*vec2(3) - vec1(3)*vec2(2)
         cross(2) = vec1(3)*vec2(1) - vec1(1)*vec2(3)
         cross(3) = vec1(1)*vec2(2) - vec1(2)*vec2(1)
      ELSE
         cross = 0.0_RP
         PRINT*,'ERROR: Vector size mismatch'
      END IF
!
   END FUCNTION crossproduct
END MODULE DotCrossProduct
```

7.2 Dynamic Arrays

When we dealt with arrays in Chap. 5 we made the implicit assumption that we already knew how large we wanted to make the array. We can remove such an assumption if we add the ALLOCATABLE attribute to a variable declaration. This attribute will tell the compiler to reserve a chunk of memory for possible use during execution. It also gives us the freedom to resize arrays during a PROGRAM without recompiling.

For example, if we want a three dimensional ALLOCATABLE array we use a colon, `:`, as a placeholder for the dimension size.

```
REAL(KIND=RP),ALLOCATABLE,DIMENSION(:,:,:) :: array1
```

To make the reserved memory available we use the ALLOCATE command. Always remember that if a PROGRAM, FUNCTION, or SUBROUTINE contains an ALLOCATE command it should have a corresponding DEALLOCATE command to release the memory. This will help prevent ***memory leaks***, which occur when a program mismanages memory. Memory leaks can slow down performance and can even exhaust available system memory (which leads to ***segmentation faults***). Fortran does offer some built-in error checking when allocating memory:

```
ALLOCATE(var_name(lowerBound:upperBound),STAT=ierr)
```

where the STAT option returns an INTEGER type. If `ierr` $\neq 0$, then there was an error allocating the memory for `var_name`.

It is important to note that there are compiler options that can assist you when debugging dynamic arrays (especially when tracking down ***segmentation faults***). Examples of compiler flags to help locate memory mismanagement for the `gfortran` compiler are `-fbounds-check`, `-g`, and `-fbacktrace`. With this flag activated the program will alert the user to any problems with arrays. In particular, the program will throw an error if it is ever detected that an operation reaches outside an allocated array's bound. As useful as this utility is, be sure to ***turn off*** these flags once you are confident the code is debugged because extra options like checking dynamic array bounds can hurt your code's performance. So, to ensure the program runs as quickly as possible make sure all the `debug` compiler options are turned off, and any optimizers you want activated turned on.

We also show a quick example of reallocating memory during a program's execution

AllocateExample.f90

```
PROGRAM AllocateExample
   IMPLICIT NONE
   INTEGER,ALLOCATABLE,DIMENSION(:)   :: array1
   INTEGER,ALLOCATABLE,DIMENSION(:,:) :: array2
!
   ALLOCATE(array1(-2:8),array2(-1:2,0:10))
   PRINT*,LBOUND(array1),UBOUND(array1)
   PRINT*,SHAPE(array2)
   DEALLOCATE(array1,array2)
!
   ALLOCATE(array1(0:100),array2(1:5,-5:5))
   PRINT*,LBOUND(array1),UBOUND(array1)
   PRINT*,SHAPE(array2)
   DEALLOCATE(array1,array2)
!
END PROGRAM AllocateExample
```

7.3 Optional Arguments

When we write a FUNCTION or SUBROUTINE sometimes an argument may not need to be present at all times. For example, in a function that prints a matrix we can write to a file, but default to the terminal if no file name is provided. In this way, we can make the file name an OPTIONAL argument. The procedure can then check if the argument is PRESENT (returns a LOGICAL) and operate accordingly.

OptionalPrint.f90
```fortran
SUBROUTINE PrintMatrix(mat,fileName,N)
   USE Constants,ONLY: RP
   IMPLICIT NONE
   INTEGER                     ,INTENT(IN) :: N
   REAL(KIND=RP),DIMENSION(N,N),INTENT(IN) :: mat
   CHARACTER(LEN=*),OPTIONAL   ,INTENT(IN) :: fileName
!  Local Variables
   INTEGER                                 :: i,fileUnit
!  Open a given file, if no file present print to screen
   IF (PRESENT(fileName)) THEN
      OPEN(UNIT=fileUnit,FILE=fileName)
   ELSE
      fileUnit = 6 ! recall 6 is the terminal screen
   END IF
!
   DO i = 1,N
      WRITE(fileUint,*)mat(i,:)
   END DO! i
!  Close the file if it was opened
   IF (PRESENT(fileName)) THEN
      CLOSE(fileUnit)
   END IF
!
   RETURN
END SUBROUTINE PrintMatrix
```

7.4 Advanced Input/Output Options

We know how to read in or write unformatted data to the terminal or to a file, but there are some useful I/O options we glossed over. Next we'll explore some of the advanced options available to us when inputting and outputting information.

7.4.1 Non-Advancing I/O

First is the ADVANCE option in READ or WRITE statements. Effectively, this will tell the READ/WRITE statement whether or not to advance to the next line. By default, ADVANCE is set to 'YES'. For example,

Advance NO
```fortran
READ(fileUnit,ADVANCE='NO')var1,var2,var3,var4
```

will read in four variables from the file pointed to by `fileUnit` where all variables are on the same line. However, if we read in using

Advance YES
```fortran
READ(fileUnit,*)var1,var2,var3,var4
```

reads in four variables from the file pointed to by `fileUnit` where each variables is on its own line, i.e., separated by a carriage return.

7.4.2 Formatted File I/O

Sometimes unformatted file reading and writing is insufficient. For example, Fortran can create and read binary files, which require special arguments. To format data for reading/writing Fortran uses a notation like 'A4', which means a character of length 4. Or in general, a variable type and the length of the data to be written. The most common variable outputs are

- A – characters/strings
- I – integer
- F – real number, decimal form
- E – real number, exponential form
- ES – real number, scientific notation
- EN – real number, engineering notation

For real number outputs we specify the total width of the number as well as the number of digits to appear after a decimal points, e.g., F10.6 is a real number with 10 digits, 6 of which appear after the decimal. Further, we can include multiple instances of real number formatting if we wish to output multiple numbers, i.e. 5F7.5 expects to output 5 real numbers, each with 7 digits, 5 of which appear after the decimal

We can combine several different formatting parameters by replacing the * option with '(parameters)' as we do in the next example

FormatPrint.f90

```
SUBROUTINE PrintMatrix_Formatted(mat,fileName,N)
   USE Constants,ONLY: RP
   IMPLICIT NONE
   INTEGER                     ,INTENT(IN) :: N
   REAL(KIND=RP),DIMENSION(N,N),INTENT(IN) :: mat
   CHARACTER(LEN=*),OPTIONAL   ,INTENT(IN) :: fileName
!  Local Variables
   INTEGER                                 :: i,j,fUnit
!  Open a given file, if no file present print to screen
   IF (PRESENT(fileName)) THEN
      OPEN(UNIT=fUnit,FILE=fileName)
   ELSE
      fUnit = 6 ! 6 is the terminal screen
   END IF
!
   DO i = 1,N
      DO j = 1,N
         WRITE(fUint,'(A2,I3,A2,I3,A2,F10.6,A1)',ADVANCE='NO')'A[',i,'][',j,']='mat(i,j),' '
      END DO! j
      WRITE(fUint,'(A2,I3,A2,I3,A2,F10.6)')'A[',i,'][',j,']='mat(i,j)
   END DO! i
!  Close the file if it was opened
   IF (PRESENT(fileName)) THEN
      CLOSE(fUnit)
   END IF
!
   RETURN
END SUBROUTINE PrintMatrix_Formatted
```

7.5 Recursive Procedures in Fortran

Many mathematical formulae lend themselves to a recursive formulation, like the Fast Fourier Transform (FFT). But, as we mentioned in Chap. 4, normally a FUNCTION or SUBROUTINE cannot reference itself, directly or indirectly. However, if we invoke that the procedure is RECURSIVE self-reference is possible.

7.5.1 Recursive Functions

We start with a canonical example of a recursive function to compute the factorial, $n!$, for some integer n. This example also introduces a SELECT CASE, which is a common alternative to IF statements.

Recursive Function
```
RECURSIVE FUNCTION factorial(n) RESULT(factorial_n)
   IMPLICIT NONE
   INTEGER,INTENT(IN) :: n
! Determine if recursion is required
   SELECT CASE(n)
   CASE (0)
! Recursion reached the end
      factorial_n = 1.0_RP
   CASE (1:) ! any integer above 0
! Recursion call(s) required
      factorial_n = n*factorial(n-1)
   CASE DEFAULT
! If n is negative, return error
      PRINT*,'ERROR: n is negative'
      factorial_n = 0.0_RP
   END SELECT
END FUNCTION factorial
```

7.5.2 Recursive Subroutines

We can also create recursive subroutines. Another example arises in the bisection method, where one recursively halves an interval based on the function $f(x)$ to locate the root of a function. We also enable the termination of the subroutine once we reach a certain number of iterations (interval halvings).

halveInterval.f90
```
RECURSIVE SUBROUTINE halveInterval(f,xL,xR,tol,iter_count,zero,delta,err)
   IMPLICIT NONE
   REAL(KIND=RP),INTENT(IN)    :: tol
   REAL(KIND=RP),INTENT(INOUT) :: xL,xR
   INTEGER      ,INTENT(INOUT) :: iter_count
   REAL(KIND=RP),INTENT(OUT)   :: zero,delta
   INTEGER      ,INTENT(OUT)   :: err
   REAL(KIND=RP),EXTERNAL      :: f
! Local Variables
   REAL(KIND=RP)               :: xM
!
   delta = 0.5_RP*(xR-xL)
! Check to see if you've reached the tolerance for the root
   IF (delta.LT.tol) THEN
! Yes! - Return result
      err  = 0
      zero = xL + delta
   ELSE
! No root yet - check iterations and halve again
      iter_count = iter_count - 1
      IF (iter_count.LT.0) THEN
! Max iterations w/o solution - return error
         err  = -2
         zero = xL + delta
      ELSE
```

```
! Keep iterating
         xM = xL + delta
         IF (f(xL)*f(xm).LT.0.0_RP) THEN
            CALL halveInterval(f,xL,xM,tol,iter_count,zero,delta,err)
         ELSE
            CALL halveInterval(f,xM,xR,tol,iter_count,zero,delta,err)
         END IF
      END IF
   END IF
!
   RETURN
END SUBROUTINE halveInterval
```

7.6 Using Specific Parts of a Module

We have already introduced how a PROGRAM or MODULE can use another MODULE to make variables, data structures, and routines available for use. However, it is often the case that a PROGRAM or SUBROUTINE does not use everything included in a module but only some of the data or a few routines. Next we demonstrate how to specify particular pieces of a given MODULE to be used elsewhere. There are several advantages to specifying what part of a MODULE is used where:

1. There is a built in list for each FUNCTION, SUBROUTINE, and PROGRAM of what precisely each routine uses and in which MODULEs the variables and/or routines lie. This makes it very clear to the programmer how the program is to be linked.
2. It makes debugging easier since you know precisely which routines could be causing the code to not run properly.
3. The code will compile faster.
4. The code will run faster (particularly with optimization enabled) because the compiler knows which routines are passed and can more effectively manage memory.

As a concrete example lets say we have a MODULE that contains several different routines to approximate an integral called `quadratureRules.f90` (this could include midpoint rule, Simpson's rule, Gauss quadrature, etc.) Then we can specify in the driver for the integral approximation which method we use.

useOnlyExample.f90

```
PROGRAM Quadrature
   USE Constants      ,ONLY: RP
   USE QuadratureRules,ONLY: SimpsonsRule
   USE Functions      ,ONLY: f ! = x^3 + cos(x)
   IMPLICIT NONE
   INTEGER      ,PARAMETER :: N = 100
   REAL(KIND=RP),PARAMETER :: a = -2.0_RP, b = 3.0_RP
   REAL(KIND=RP)           :: Integral
!
   Integral = SimpsonsRule(a,b,N,f)
   WRITE(*,*)'Approximation for f(x) integral: ',Integral
   WRITE(*,*)
!
END PROGRAM Quadrature
```

7.7 Pure Procedures

We have seen that a FUNCTION or a SUBROUTINE can modify their input arguments. We have some control over preventing this or dictating if it is allowed by prescribing the INTENT of an input variable. However, Fortran can place even stricter guidelines on routines to prevent changing data values outside of a specific routine. This is done by telling a Fortran a procedure is PURE, which means that there is ***no chance*** that the procedure will have any side effect on data outside the procedure. Either a FUNCTION or a SUBROUTINE can be defined as PURE, but the keyword ***must*** be used in the procedure declaration.

Due to their stringent nature on controlling data, there are some rules attached to making a procedure PURE. First, there can be no external I/O operations within a pure procedure. Also, a pure procedure cannot contain the STOP statement. If a PURE procedure is a FUNCTION, then each input argument ***must*** be declared as INTENT(IN). Further, if the PURE procedure is a SUBROUTINE each input argument ***must*** declare some form of the INTENT attribute (either IN, OUT, or INOUT).

Pure Function Example

```
PURE FUNCTION cube(x)
   USE Constants,ONLY: RP
   IMPLICIT NONE
   REAL(KIND=RP),INTENT(IN) :: x
   REAL(KIND=RP)            :: cube
   cube = x*x*x ! slower syntax would be x**3
END FUNCTION

PROGRAM pureFunctionExample
   USE Constant,ONLY: RP
   IMPLICIT NONE
   REAL(KIND=RP) :: a,b,cube
   a = 2.0_RP
   b = cube(a)
! After invoking the pure function we are certain nothing has changed besides assigning the
! output value to b. The value of a is guaranteed unchanged
END PROGRAM pureFunctionExample
```

Here, because the FUNCTION and the PROGRAM are contained within the same file, we do not have to assign the EXTERNAL statement to the `cube` function.

We also provide an example of a PURE SUBROUTINE. In this case it a procedure that converts a vector of conserved variables for the compressible Euler equations

$$\texttt{Cons} = (\rho\,,\,\rho u\,,\,\rho v\,,\,\rho w\,,\,E)\,,$$

where E is the total energy, into a vector of primitive variables (assuming an ideal gas)

$$\texttt{Prim} = (\rho\,,\,u\,,\,v\,,\,w\,,\,p)\,.$$

In practice, this is a useful intermediate procedure when we go to compute the fluxes for the Euler equations. In the x direction the flux has the form

$$\texttt{xFlux} = \big(\rho u\,,\,\rho u^2 + p\,,\,\rho uv\,,\,\rho uw\,,\,u(E+p)\big)$$

where we see the necessity of knowing the pressure p.

Pure Subroutine Example

```
PURE SUBROUTINE ConsToPrim(prim,cons)
! Transformation from conservative variables to primitive variables
   USE Constants,ONLY: RP,gamma ! gamma is the adiabatic coefficient
   IMPLICIT NONE
   REAL(KIND=RP),INTENT(IN)  :: cons(5)
   REAL(KIND=RP),INTENT(OUT) :: prim(5)
! Local variables
   REAL(KIND=RP) :: sRho
!
```

```
   sRho=1.0_RP/cons(1)
! rho
   prim(1)=cons(1)
! vels = momenta/rho
   prim(2:4)=cons(2:4)*sRho
! pressure
   prim(5)=(gamma-1.0_RP)*(cons(5)-0.5_RP*SUM(cons(2:4)*prim(2:4)))
!
   RETURN
END SUBROUTINE ConsToPrim
```

7.8 Associate Construct

This is, by far, the newest feature of Fortran discussed in these notes. The ASSOCIATE construct was introduced with the release of `Fortran2003`. It offers a way to assign simple abbreviated expressions to more complicated statements. Further, it does this "masking" of an expression to an intermediate variable ***without*** the need to declare a new variable (provided it is contained inside an ASSOCIATE block). This can be used to make the code more readable, which is particularly important when collaborating on a large coding project. It also offers some flexibility to "rename" variables in a FUNCTION/SUBROUTINE for a code snippet without the need to declare a bunch of local variables.

associateExample.f90

```
PROGRAM associateExample
   USE Constants, ONLY: RP
   IMPLICIT NONE
   REAL(KIND=RP) :: x,y,a,temp
!
   x    = 0.5_RP
   y    = 0.85_RP
   a    = 1.2_RP
   temp = 6.28_RP
! begin the associate block where we mask to new variable names
   PRINT*
   ASSOCIATE(radius => SQRT(x*x+y*y), rho => temp)
      PRINT*,radius+a,radius-a,rho
      rho = rho + 9.43_RP
   END ASSOCIATE
   PRINT*
   PRINT*,temp
!
END PROGRAM associateExample
```

Note that after the end of the ASSOCIATE construct, any changes made within the construct to the value of the associating entity (in this case `rho`) that associates with `temp` is reflected. We verify this behavior with the following output.

associateExample.f90 - Commands and Output

```
gfortran associateExample.f90 -o Associate
./Associate

2.1861541461658009      -0.2138458538341990        6.2800000000000000

15.710000000000000
```

Chapter 8: Some Debugging Practices

At this point we are familiar with Fortran, many of its features, and know how to create and organize a coding project. However, one aspect of computer programming for scientific computing has not been discussed and that is ***debugging***. We have the knowledge of how to write a FUNCTION, SUBROUTINE, or PROGRAM. Also, getting the source code to compile is aided by the compiler because it will throw errors and/or warnings to the terminal to guide you (usually even stating line numbers in particular routines where it detects a problem). But, having source code compile and produce an executable is only a small part of the work for scientific computing. The immediate question arises: Is the data produced by the program correct? Does it solve the problem we want in the expected way? Even further back from examining the output of the program. Does the program ***actually run***? Does the code terminate due a segmentation fault? Does it produce `NaN`? Does it run indefinitely for some reason?

Once the code is compiling a new aspect of computer programming takes over and we must ***debug*** the code. Bugs are very common when writing numerical algorithms and they occur for programmers of any skill or experience level. It is only the nature of the bugs that change as you gain more experience coding. In this Chapter we will try to highlight some of the common mistakes or bugs that plague scientific computing programs written in Fortran. We will also try to offer advice, compiler flags, and tools to track down and remedy common bugs.

8.1 Useful `gfortran` Compiler Flags for Debugging

There are many useful `gfortran` command line options to tell the compiler to include additional information inside the compiled program useful for debugging or alter the behavior of the compiled program to help detect bugs.

- `-g`: Generates debugging information that is usable in the GNU Debugger (GDB). It is possible to include even more debugging information with the `-g3` flag.
- `-fbacktrace`: Specifies that if a program crashes, a backtrace should be produced when possible that shows what function(s) or subroutine(s) were called/active at the time of the error.
- `-fbounds-check`: Add a check that the array index is within bounds of the array every time an array element is accessed. ***This substantially slows down a program using it***, but is a very useful option to locate bugs related to arrays and memory access. Without this flag, an illegal array access will produce either a subtle erorr that might not be apparent until later in the program execution ***or*** will cause an immediate segmentation fault with very little information about the cause of the error.
- `-ffpe-trap=zero,overflow,underflow`: This flag tells the Fortran compiler to *trap* the listed floating point errors (fpe). Including *zero* on the list means that if the program attempts to divide by zero it will simply terminate rather than setting a result to `+Infinity` or `NaN` and continuing. Similarly, if *overflow* is on the list the program will halt if it tries to store a number larger than can possibly be stored in a given variable type, e.g. the largest a REAL(KIND=RP) can be is approximately `1.79E+308`.

  Trapping *underflow* halts the computation if a number is too small because the exponent is a very large negative number. For a REAL(KIND=RP) can be is approximately `2.23E-308`. If we do not trap underflow values, then they will be set to 0. Although this is generally the correct thing to do, computing with such small number likely indicates a bug of some sort in the program. Thus, it is useful to trap them.

In addition to debugging compiler flags there are also flags to throw warnings during compilation. To be clear these are warnings about section of code that are "allowed" but their syntax or use are potentially questionable. These sections of code might be correct, but the warnings will often identify bugs before the program is even run.

- `-Wall`: Short for ***warn about all***. This flag tells `gfortran` to generate warnings about many common sources of bugs, such as having a FUNCTION or SUBROUTINE with the same nae as an INTRINSIC one, or passing the same variable as an INTENT(IN) and an INTENT(OUT) argument of the same SUBROUTINE. Despite this compiler flag's name, this does not turn on all possible warning options.
- `-Wextra`: In conjunction with `-Wall`, gives warnings about even more potential problems. In particular, this flag warns about SUBROUTINE arguments that are never used, which is almost always a bug.

- `-Wconversion`: Warns about implicit conversion between data types. For example, say we want a double precision variable `sqrt3` to hold an accurate value for the square root of 3. In the code we might accidentally write `sqrt3 = SQRT(3.)`. Because 3. is a single precision value, the single-precision SQRT function will be used by the PROGRAM, and the value of variable `sqrt3` will possess single precision accuracy (even though it is declared as a double precision variable). `-Wconversion` will generate a warning here, because the single precision result of SQRT is implicitly converted into a double precision value.

- `-pedantic`: Generate warnings about language features that are supported by `gfortran` but are not part of the official `Fortran95` standard. This is particularly useful if you want ensure your code will compile and run with ***any*** `Fortran95` compiler.

8.2 Examples of Debugging

Here we create some small examples to demonstrate some of the practices we use to debug Fortran programs. One of the first things to check in your code is the calls to any FUNCTION or SUBROUTINE. Are the arguments passed in the correct order such that the routine will compute what is expected? It sounds silly, but this type of bug can be quite common. Especially if you are calling a routine that you didn't write (due to a collaborative project) or you wrote a long time ago. This type of bug can be very tricky to find. This is because compiler flags will not notice argument ordering issues, as long as the variable types and/or dimensions all match correctly then the compiler will not throw any errors.

A main take away for debugging is ***output information and data during the execution*** either to the screen or to a file! It is incredibly difficult to just stare at a piece of code and parse what all is happening and what could be going wrong. It is much easier to track values through printing. To illustrate this consider the following scenario:

1. The code compiles and runs.
2. You look at the data produced at the end of the program and see that values have become `NaN`.

In a perfect world this wouldn't happen, because a code that produces `NaN` has crashed in someway but was allowed to continue to run. This wastes computing resources. How do we track down the offending routine that causes a `NaN`. A compiler flag to trap overflow might work, but not always. For example, a `NaN` can occur when you take SQRT or LOG of a negative number and try to store it in a REAL variable rather than a COMPLEX variable. Instead what we can do is intermittently PRINT values to the screen. This will help identify where in the PROGRAM the value of a variable becomes undefined.

To be more specific, we examine a problem involving array indexing and how it can lead to errors. We also demonstrate how the compiler flags discussed above help to prevent such array access inconsistencies. Let's look at a dummy PROGRAM that attempts to access memory in a DO loop that is not defined.

arrayBug.f90

```fortran
PROGRAM arrayBug
   USE Constants
   IMPLICIT NONE
   INTEGER                     :: i
   REAL(KIND=RP),DIMENSION(10) :: Vec
! Fill the array Vec with values
   DO i = 1,10
      Vec(i) = i
   END DO! i
... other code could happen
! Print the values to screen
   DO i = 0,10
      PRINT*,'i=',i,Vec(i)
   END DO! i
END PROGRAM arrayBug
```

We see that the second loop is trying to access data at `Vec(0)`, which does not exist. What happens if we compile and run this program normally? Sometimes this will cause a segmentation fault, but more often the code will compile and run producing something like arrayBug.f90 - Standard Run

```
gfortran arrayBug.f90
./a.out
 i=           0   1.4729579099465083E-319
 i=           1   1.0000000000000000
 i=           2   2.0000000000000000
 i=           3   3.0000000000000000
 i=           4   4.0000000000000000
 i=           5   5.0000000000000000
 i=           6   6.0000000000000000
 i=           7   7.0000000000000000
 i=           8   8.0000000000000000
 i=           9   9.0000000000000000
 i=          10   10.000000000000000
```

We see that the code ran but when it went to access the non-existent `Vec(0)` it produced a kind of "garbage" value. Now we compile the program with the `-fbounds-check` flag to see how it helps detect such array bound inconsistencies.

arrayBug.f90 - Array Check Run

```
gfortran -fbounds-check arrayBug.f90
./a.out
At line 13 of file arrayBug.f90
Fortran runtime error: Index '0' of dimension 1 of array 'vec' below lower bound of 1
```

Great! The compiler found that we had accidentally tried to access an index in the one dimensional array `Vec` that was outside of its index range. Note that the flag `-ffpe-trap=underflow` ***would not*** catch this bug because we didn't operate on the non-existent value "stored" in `Vec(0)`. This was purely an array indexing issue.

While we are discussing memory a useful tool called `valgrind` is available to track the memory usage of PROGRAM. It is particularly useful to identify ***memory leaks***. These occur when a routine ALLOCATEs memory but does not DEALLOCATE it. This memory is then never released back to the heap and is ***forever*** unavailable to the program until termination. If memory leaks are severe enough (or occur often enough) the PROGRAM will eat through all available RAM and the it will crash. A simple example of a memory leak is found below.

memoryLeak.f90

```
SUBROUTINE memoryLeak(N)
   USE Constants,ONLY: RP
   IMPLICIT NONE
   INEGER,INTENT(IN) :: N
! local variables
   REAL(KIND=RP),ALLOCATABLE,DIMENSION(:,:,:) :: A
!
   ALLOCATE(A(0:N,0:N,0:N))
! some computations
!
   RETURN
END SUBROUTINE memoryLeak
```

Once the PROGRAM leaves this SUBROUTINE the memory allocated for `A` will be unavailable. Further, if this SUBROUTINE is called a second time it will, again, ALLOCATE memory which, in turn, will be lost once we leave the current routine. This more concretely illustrates the issue that if we do this enough and we will run out of heap memory! We can fix this memory leak by adding a corresponding DEALLOCATE statement before we leave the routine. This ensures that the allocated memory remains within the "scope" of the current routine and is not lost somewhere in the ether.

memoryLeakFix.f90

```
SUBROUTINE memoryLeakFix(N)
   USE Constants,ONLY: RP
   IMPLICIT NONE
   INEGER,INTENT(IN) :: N
! local variables
   REAL(KIND=RP),ALLOCATABLE,DIMENSION(:,:,:) :: A
!
   ALLOCATE(A(0:N,0:N,0:N))
! some computations
   DEALLOCATE(A)
!
   RETURN
END SUBROUTINE memoryLeakFix
```

As a final example we investigate how assigning the INTENT of a variable incorrectly can lead to bugs. In this case, say we have a SUBROUTINE where information should be passed into the routine, operated upon, and then passed back to the calling routine.

intentBug.f90

```
SUBROUTINE intentBug(work_array,N)
   USE Constants,ONLY: RP
   IMPLICIT NONE
   INTEGER                      ,INTENT(IN)  :: N
   REAL(KIND=RP),DIMENSION(0:N),INTENT(OUT) :: work_array
!  local variables
   INTEGER :: i
!
   DO i = 0,N
      work_array(i) = 2.0_RP*work_array(i) - 1.4_RP
   END DO! i
!
   RETURN
END SUBROUTINE intentBug
```

This bug is more subtle than the previous ones discussed. It gets into the details of how Fortran and its INTENT functionality work. This code compiles and runs. However, the `work_array` is assigned INTENT(OUT). In this case, it is a static array of size $N + 1$ but when the memory is set aside Fortran does not give it any data from the calling routine (because of the given intent). The Fortran compiler sees INTENT(OUT) and decides that it only has to pass data *back* to the calling routine. ***This is the issue***. The SUBROUTINE relies on data that should be given in the `work_array`, but due to the INTENT(OUT) the `work_array` is filled with "garbage" data initially. This pollutes the data produced by the SUBROUTINE. The fix is to change to INTENT(INOUT) which tells the compiler to pass data into the routine as well.

intentBugFix.f90

```
SUBROUTINE intentBugFix(work_array,N)
   USE Constants,ONLY: RP
   IMPLICIT NONE
   INTEGER                      ,INTENT(IN)    :: N
   REAL(KIND=RP),DIMENSION(0:N),INTENT(INOUT) :: work_array
!  local variables
   INTEGER :: i
!
   DO i = 0,N
      work_array(i) = 2.0_RP*work_array(i) - 1.4_RP
   END DO! i
!
   RETURN
END SUBROUTINE intentBugFix
```

Chapter 9: Putting It All Together

Now we have some proficiency in Fortran and can tackle a general programming project. That is the purpose of this Chapter. We will walkthrough a larger project that will pull from many aspects of the previous Chapters. In particular, we want to design a program that is well structured and divided into digestible pieces. This paradigm makes the code easier to read, easier to debug, and easier to reuse parts of it in other projects. The problem at hand with come from the numerical approximation of a partial differential equation (PDE) in one spatial dimension. The numerical scheme itself will be kept quite simple as the purpose here is to teach good practices in Fortran coding (not necessarily to build up knowledge of numerical PDE solvers).

9.1 Problem description

We wish to approximate the solution of the linear advection equation

$$u_t + f_x = u_t + au_x = 0,$$

where subscripts represent partial derivatives in time or space, u is the solution, and $f = au$ is the flux function. We assume, without loss of generality, that the wave speed $a > 0$ and is a constant. We solve the PDE on a one dimensional domain $[0, 1]$. We assume periodic boundary conditions

$$u(0, t) = u(1, t).$$

The initial value problem is fully prescribed once we consider an initial condition for the solution

$$u(x, 0) = u_0(x).$$

The analytical solution of the linear advection equation is given by the initial condition translated by an amount proportional to the wave speed and the time that has passed, i.e.,

$$u^{ex}(x, t) = u_0(x - at).$$

In practice, having an analytical solution is useful to verify theoretical properties of your numerical scheme, such as the order of accuracy or the convergence rate.

9.2 Discretization of the Problem

We forgo presenting the theory and motivation of how we arrive at the complete discretization of the linear advection equation and simply state its discrete version. We choose to use ***Forward Euler*** for the discretization in time and a biased (or "upwind") discertization of the flux derivative in space. The superscript n denotes the time level of the approximation and the subscript j represents the spatial discretization. The discrete version of the PDE is given by

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{\Delta x}\left(f_j^n - f_{j-1}^n\right), \quad j = 1, \ldots, N.$$

Here, it is assumed the domain is $[0, 1]$ which gives

$$\Delta x = \frac{1}{N}.$$

One way to specify the time step (that is ***very ad hoc***, there are much better ways to do this) is by fixing a number of time steps and dividing the final time, i.e,

$$\Delta t = \frac{T}{\#_{\text{steps}}}.$$

The discretization is kept general in terms of the flux to increase the portability of any routines we write.

9.3 Designing the Project

Now we have been handed a discretization of the linear advection equation and tasked with creating a Fortran program to numerically solve the problem. Before we tackle this task and start writing code we should take some time and think about the program's structure. What parts of the algorithm depends on another? How should I read in data? How should I export data? In a given time step, what are the tasks to prepare the solution for advancement to the next time step? Thinking about these questions we can organize some of our thoughts:

- Every routine will need a precision parameters from the `Constants.f90` MODULE.
- Computing the flux at a given node x_j requires the solution at the current time at that note, u_j^n, as well as the wave speed a.
- We need to set the periodicity of the problem through the fluxes.
- Once the fluxes are all computed we can determine the flux difference.
- From the flux difference we can update the solution at a given node, advancing in time.
- We can put this process into a time loop that advances until the final time T.
- During the time integration we can export data files to plot the solution as it evolves over time.
- A driving program is necessary to determine the final time, the number of time steps, the number of nodes N.
- The driver can also manage memory for the solution array as well as the initial condition.

These obviously aren't all the aspects organizing and implementing the coding project. But, it is a good start such that we can construct our program, filling in any gaps along the way.

9.4 Implementing the Project

We have a certain skeleton of what we need to implement this project and solve the linear advection equation numerically. First, we need a version of the `Constants.f90` module. In this case, we also include the wave speed a.

`Constants.f90`

```fortran
MODULE Constants
   IMPLICIT NONE
   INTEGER      ,PARAMETER :: RP = SELECTED_REAL_KIND(15)
   REAL(KIND=RP),PARAMETER :: pi = 4.0_RP*ATAN(1.0_RP)
   REAL(KIND=RP),PARAMETER :: a  = 1.0_RP
END MODULE Constants
```

Next, we create a set of routines that operate on the fluxes. This includes a routine to precompute and ***fill*** the flux with values, another routine that computes the flux difference needed by the PDE approximation, and a routine to set the initial condition of the problem. We do this to maintain flexibility of the code if we wanted to solve for a different flux function.

`fluxRoutines.f90`

```fortran
MODULE fluxRoutines
   IMPLICIT NONE
! Nothing to declare
CONTAINS
!
   SUBROUTINE fillFlux(u,f,N)
      USE Constants
      IMPLICIT NONE
      INTEGER                     ,INTENT(IN)  :: N
      REAL(KIND=RP),DIMENSION(N)  ,INTENT(IN)  :: u
      REAL(KIND=RP),DIMENSION(0:N),INTENT(OUT) :: f
!  Local variables
      INTEGER :: i
!
      f = 0.0_RP
      DO i = 1,N
         f(i) = a*u(i)
      END DO! i
! set periodic boundary conditions
      f(0) = f(N)
!
      RETURN
   END SUBROUTINE fillFlux
!
!////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE fluxDifference(f,df,N)
      USE Constants,ONLY: RP
      IMPLICIT NONE
      INTEGER                      ,INTENT(IN)  :: N
      REAL(KIND=RP),DIMENSION(0:N),INTENT(IN)  :: f
      REAL(KIND=RP),DIMENSION(N)  ,INTENT(OUT) :: df
!  Local variables
      INTEGER :: i
!
      df = 0.0_RP
      DO i = 1,N
         df(i) = f(i) - f(i-1)
      END DO! i
!
      RETURN
   END SUBROUTINE fluxDifference
!
!////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE InitialCondition(x,u,N)
      USE Constants,ONLY: RP,pi
      IMPLICIT NONE
      INTEGER, INTENT(IN)                    :: N
      REAL(KIND=RP),DIMENSION(N),INTENT(IN)  :: x
      REAL(KIND=RP),DIMENSION(N),INTENT(OUT) :: u
!  Just an example. Can use different functions
      u = SIN(2.0_RP*pi*x) + 1.0_RP
!
      RETURN
   END SUBROUTINE initialCondition
END MODULE fluxRoutines
```

We create a routine for plotting the solution to a file that can be visualized with `VisIt`.

Plotter.f90

```
MODULE Plotter
   IMPLICIT NONE
! Nothing to declare
CONTAINS
```

```
   SUBROUTINE ExportToTecplot_1D(x,u,fUnit,N)
!  File writer routine to make movies
      USE Constants,ONLY: RP
      IMPLICIT NONE
      INTEGER                   ,INTENT(IN) :: N,fUnit
      REAL(KIND=RP),DIMENSION(N),INTENT(IN) :: x
      REAL(KIND=RP),DIMENSION(N),INTENT(IN) :: u
!  Local variables
      INTEGER :: j
!
      WRITE(fUnit,*)'#u'
      DO j = 1,N
         WRITE(fUnit,*)x(j),u(j)
      END DO! j
!
      RETURN
   END SUBROUTINE ExportToTecplot_1D
END MODULE Plotter
```

Now we have the functionality in place to compute the fluxes and the flux difference as well as plot the solution to a file. Thus, we are prepared to step the solution forward in time. To do so, we create a routine that steps the solution forward a single time step. Moreover, we write an integration routine that takes the solution up to the final time T. During the time integration we also put in logic to output solution files on a fixed interval stride depending on the number of time steps the user wants to use. These files will be saved in a folder called `./Movies` with sequentially written files. To output these movie files in slightly cumbersome because of how we have to track the current number of files written and convert that into a file name. However, it is invaluable to be able to output solution snapshots and examine the approximate solution over time.

timeIntegration.f90

```
MODULE TimeIntegration
   IMPLICIT NONE
! Nothing to declare
CONTAINS
!
   SUBROUTINE Integrate(x,u,T,dt,dx,N)
      USE Constants,ONLY: RP
      USE Plotter  ,ONLY: ExportToTecplot_1D
      IMPLICIT NONE
      INTEGER                   ,INTENT(IN)    :: N
      REAL(KIND=RP)             ,INTENT(IN)    :: dx,T,dt
      REAL(KIND=RP),DIMENSION(N),INTENT(INOUT) :: u,x
!  Local variables
      INTEGER           :: num_steps,j,fUnit,m
      REAL(KIND=RP)     :: local_t
      CHARACTER(LEN=16) :: fName   = "Movies/UXX.curve"
      CHARACTER(LEN=2)  :: numChar
      m         = 0
      local_t   = 0.0_RP
      num_steps = T/dt
!  Time integration loop
      DO j = 0,num_steps-1
         local_t = (j+1)*dt
         CALL StepByEuler(u,dx,dt,N)
         IF (MODULO(j,50).EQ.0) THEN
!  Print solution every 50 time steps for movies
```

```
            m = m + 1
            WRITE(numChar,'(i2)')m
            IF (m.GE.10) THEN
               fName(9:10) = numChar
            ELSE
               fName(9:9)    = "0"
               fName(10:10)  = numChar(2:2)
            END IF
            OPEN(UNIT=11,FILE=fName)
            CALL ExportToTecplot_1D(x,u,11,N)
            CLOSE(11)
         END IF
      END DO! j
!
      RETURN
   END SUBROUTINE Integrate
!
!//////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE StepByEuler(u,dx,dt,N)
      USE Constants   ,ONLY: RP
      USE fluxRoutines,ONLY: fillFlux,fluxDifference
      IMPLICIT NONE
      INTEGER                   ,INTENT(IN)    :: N
      REAL(KIND=RP)             ,INTENT(IN)    :: dx,dt
      REAL(KIND=RP),DIMENSION(N),INTENT(INOUT) :: u
!  Local variables
      INTEGER                      :: i
      REAL(KIND=RP),DIMENSION(0:N) :: f
      REAL(KIND=RP),DIMENSION(N)   :: fdiff
!
      CALL fillFlux(u,f,N)
      CALL fluxDifference(f,fdiff,N)
      DO i = 1,N
         u(i) = u(i) - dt*fdiff(i)/dx
      END DO! i
!
      RETURN
   END SUBROUTINE StepByEuler
END MODULE TimeIntegration
```

The next step in the project is to write a driving program that will allocate the memory, perform the time integration up to the final time T while writing intermediate solution files, and then deallocate memory. However, we need a way to input this run time information into the program. We can read in from the terminal, but as we discussed before this becomes cumbersome if we want to run the linear advection solver for many different configurations, resolutions, or final times. Therefore, before we write the driver we put together some file reading routines to get information from a file `input.dat`. We assume the input file has the following form.

input.dat

```
number of nodes = 250
time steps      = 2000
final time      = 0.5
```

This is helpful because we can (somewhat) label what value is being read into the program. The file reading routines will be designed to detect an `=` operator and get the numerical value to the right.

fileReading.f90

```fortran
MODULE FileReadingRoutines
   USE Constants,ONLY:RP
   IMPLICIT NONE
! Nothing to declare
CONTAINS
!
   FUNCTION GetRealValue(inputLine) RESULT(realNum)
!  Read the value of a real number declared after an = sign in a inputLine
      CHARACTER(LEN=*) :: inputLine
      REAL(KIND=RP)    :: realNum
      INTEGER          :: strLen,leq
!
      leq    = INDEX(inputLine,'=')
      strLen = LEN_TRIM(inputLine)
      READ(inputLine(leq+1:strLen),*)realNum
!
      END FUNCTION GetRealValue
!
!///////////////////////////////////////////////////////////////////////////
!
   FUNCTION GetIntValue(inputLine) RESULT(intNum)
!  Read the value of an integer number declared after an = sign in an inputLine
      CHARACTER(LEN=*) :: inputLine
      INTEGER          :: intNum,strLen,leq
!
      leq    = INDEX(inputLine,'=')
      strLen = LEN_TRIM(inputLine)
      READ(inputLine(leq+1:strLen),*)intNum
!
   END FUNCTION GetIntValue
END MODULE FileReadingRoutines
```

Finally, we are prepared to create the driver for the program.

fileReading.f90

```fortran
PROGRAM SolveLinearAdvection
   USE FileReadingRoutines
   USE Constants      ,ONLY: RP
   USE FluxRoutines   ,ONLY: InitialCondition
   USE TimeIntegration,ONLY: Integrate
   IMPLICIT NONE
   INTEGER                                :: N,tsteps,i
   REAL(KIND=RP)                          :: dx,T,dt
   REAL(KIND=RP),ALLOCATABLE,DIMENSION(:) :: u,x
   CHARACTER(LEN=132)                     :: inputLine
!
!  Open the input file
   OPEN(UNIT=24601,FILE='input.dat')
!  Read the number of nodes for the approximation
   READ(24601,'(A132)')inputLine
   N = GetIntValue(inputLine)
!  Read the number of time steps
   READ(24601,'(A132)')inputLine
   tsteps = GetIntValue(inputLine)
```

```
!  Read the final time
   READ(24601,'(A132)')inputLine
   T = GetRealValue(inputLine)
!  Close the input file
   CLOSE(33)
!
!  Allocate memory
   ALLOCATE(u(N),x(N))
!  Compute delta t
   dt  = T/tsteps
!  Compute delta x assuming interval is [0,1]
   dx  = 1.0_RP/N
!  Fill the nodes where the solution is stored
   DO i = 1,N
      x(i) = dx*i
   END DO! i
!  Initialize the solution
   CALL InitialCondition(x,uu,N)
!  Solve the PDE, internally this prints the solution
   CALL Integrate(x,uu,T,dt,dx,N)
!  Deallocate memory
   DEALLOCATE(u,x)
END PROGRAM SolveLinearAdvection
```

9.5 Compiling the Project

Despite the relative simplicity of this project, that is the update to the solution from one time level to the next, it contains many source files. So, the compilation of this project can be somewhat unwieldy by hand. Therefore, we provide a `Makefile` to build the source of the project to create an executable.

linAdv Makefile

```
F90 = /usr/local/bin/gfortran
FFLAGS = -Ofast
# Object Files for build
OBJS = \
Constants.o \
FileReading.o \
FluxRoutines.o \
Plotter.o \
SolveLinearAdvection.o \
TimeIntegration.o \

linAdv : $(OBJS)
         $F90  -o $@ $(OBJS)
# Object dependencies and compilation
Constants.o : ./Constants.f90
        $(F90) -c $(FFLAGS) $(INCLUDES) -o $@ ./Constants.f90

FileReading.o : ./FileReading.f90 \
Constants.o
        $(F90) -c $(FFLAGS) $(INCLUDES) -o $@ ./FileReading.f90

FluxRoutines.o : ./FluxRoutines.f90 \
Constants.o
        $(F90) -c $(FFLAGS) $(INCLUDES) -o $@ ./FluxRoutines.f90
```

```
Plotter.o : ./Plotter.f90 \
Constants.o
	$(F90) -c $(FFLAGS) $(INCLUDES) -o $@ ./Plotter.f90

SolveLinearAdvection.o : ./SolveLinearAdvection.f90 \
FluxRoutines.o \
FileReading.o \
Constants.o \
TimeIntegration.o
	$(F90) -c $(FFLAGS) $(INCLUDES) -o $@ ./SolveLinearAdvection.f90

TimeIntegration.o : ./TimeIntegration.f90 \
Plotter.o \
Constants.o \
FluxRoutines.o
	$(F90) -c $(FFLAGS) $(INCLUDES) -o $@ ./TimeIntegration.f90
# Utility targets
.PHONY: clean
clean:
	rm *.o *.mod
	rm linAdv
```

Once you compile the project it is ready to run! You can adjust the parameters in `input.dat` to change the spatial or temporal resolution. Also, you can run the program for a longer time to see the effect of numerical dissipation on the approximate solution. This Chapter was just an exercise in coding together a small numerical project in Fortran. Obviously, there is much more to investigate if the algorithms are implemented correctly. This would include testing the convergence rates, which should be first order in both space and time based on the building blocks of the numerical algorithm. We could also add functionality to plot the exact solution as well, because it is known to just be the shifted initial condition for the linear advection equation.

For completeness we include a plot in Figure 2 of the computed solution against the exact solution for the parameters $N = 300$, $\#_{\text{steps}} = 2001$, $T = 1.0$ and initial condition $u_0(x) = \sin(2\pi x) + 1$. Note that there is noticeable numerical dissipation effects in the approximate solution compared to the exact solution.

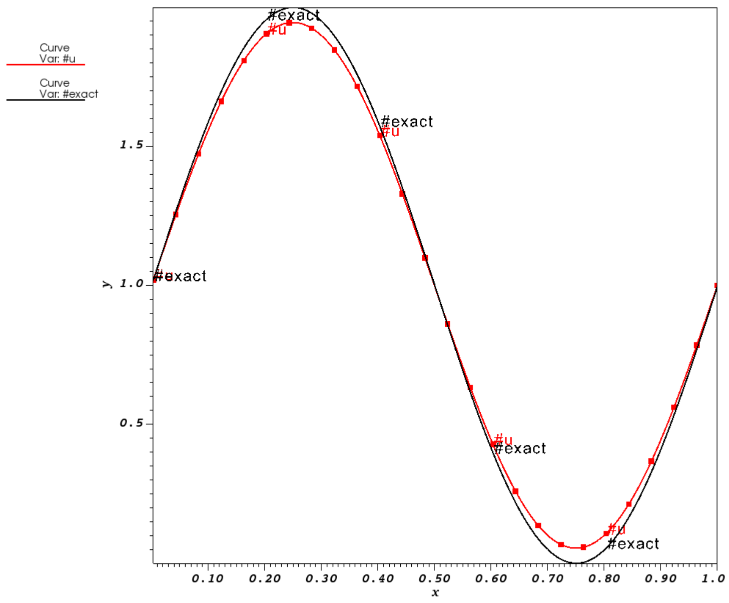

Figure 2: Approximate and exact solution of the linear advection equation.

Chapter 10: OOP for Discontinuous Galerkin (DG) Method

We've presented Fortran as an option for an object oriented programming language. We'll now examine some very particular examples of how Fortran can be used to implement a discontinuous Galerkin spectral element method (DGSEM) in one spatial dimension. We will introduce a couple derived TYPE variables to act as classes that make the structure of the program simpler. Note, this is ***one*** way to organize a 1D DGSEM code, but it is not the ***only*** way. Know that this is a general overview and these will only be code snippets, rather than a full, working implementation. We will outline three classes that could be used in a one dimensional DG implementation:

- Nodal DG Storage
- DG Element
- DG Mesh

We note that the discussion follows closely the object oriented DG structure introduced in the book by David Kopriva, "*Implementing spectral methods for partial differential equations: Algorithms for scientists and engineers*." Springer Science & Business Media, 2009.

After the discussion of the DG classes we will give implementations for outputting the DG solution to a file for plotting. We give provide working code (for one, two, and three dimensions) to create TecPlot files, that can be plotted in `VisIt`. We also provide examples of using `matplotlib` to create convergence plots.

10.1 Nodal DG Storage Class

Let's start with a class which precomputes and stores the necessary information for the DG first derivative approximation. We make no assumptions about whether the strong or weak form of the DG approximation is used, so we will store the

- order of the approximation, N.
- Lagrange interpolating polynomial basis evaluated at the endpoints, $\ell_j(\pm 1)$.
- quadrature weights, w_j. Alternatively, for computational efficiency, one could store the inverse of the quadrature weights $1/\omega_j$. This is because division is more expensive than multiplication, so if we only perform the division once and multiply thereafter we will see a slight performance boost.
- polynomial derivative matrix, $D_{nj} = \ell_j'(\xi_n)$.
- negative transpose of the polynomial derivative matrix scaled by the quadrature weights, $\hat{D}_{jn} = -\frac{D_{nj}w_n}{w_j}$.

In the psuedocode for *NodalDGStorageClass* (Alg. 7) we group the data and procedures together. The constructor will allocate all necessary memory and precompute the data. The destructor will deallocate the memory (thus destroying any data that memory contained).

Algorithm 7: *NodalDGStorageClass*: Precomputes and stores derivative matrices, quadrature weights, etc.

```
Class NodalDGStorage
    Data:
        $N$, $\{D\}_{i,j=0}^{N}$, $\left\{\hat{D}_{i,j}\right\}_{i,j=0}^{N}$, $\{\ell_j(-1)\}_{j=0}^{N}$, $\{\ell_j(1)\}_{j=0}^{N}$, $\{w_j\}_{j=0}^{N}$
    Procedures:
        ConstructNodalDGStorage($N$)
        DestructNodalDGStorage()
End Class NodalDGStorage
```

Next, we give an implementation of the *NodalDGStorageClass* in form of a MODULE. The constructor in this case builds a DG approximation space using the Legendre-Gauss nodes and weights. To do so we USE specific routines from modules for interpolation and quadrature. We don't explicit form these modules, but complete pseudocode can be found in the book by Kopriva. If you want to change to a different set of quadrature points then you invoke a CALL to a different SUBROUTINE. Note that this implementation combines many of the Fortran topics covered previously, most notably dynamic arrays. We define a NodalDGStorage TYPE and treat it as an object. Recall that to access members of an object you use the % command.

NodalDGStorageClass.f90

```
MODULE NodalDGStorageClass
   USE Constants
   IMPLICIT NONE
!
   TYPE NodalDGStorage
      INTEGER                                  :: N
      REAL(KIND=RP),ALLOCATABLE,DIMENSION(:)   :: l_at_minus_one
      REAL(KIND=RP),ALLOCATABLE,DIMENSION(:)   :: l_at_one
      REAL(KIND=RP),ALLOCATABLE,DIMENSION(:)   :: weights
      REAL(KIND=RP),ALLOCATABLE,DIMENSION(:,:) :: D
      REAL(KIND=RP),ALLOCATABLE,DIMENSION(:,:) :: D_hat
   END TYPE NodalDGStorage
!
CONTAINS
!
   SUBROUTINE ConstructNodalDGStorage(this,N)
      USE Interpolation,ONLY: BarycentricWeights,LagrangeInterpolatingPolynomial, &
                              PolynomialDerivativeMatrix
! Note the continuation marker in Fortran & that allows a longer line to be broken apart
      USE Quadrature   ,ONLY: LegendreGaussNodesAndWeights
      IMPLICIT NONE
      INTEGER             ,INTENT(IN)  :: N
      TYPE(NodalDGStorage),INTENT(OUT) :: this
!  Local variables
      REAL(KIND=RP),DIMENSION(0:N)  :: x,BaryWeights  ! nodes and weights of interpolation
      INTEGER                       :: i,j

      ALLOCATE(this%l_at_minus_one(0:N),this%l_at_one(0:N),this%weights(0:N))
      ALLOCATE(this%D(0:N,0:N),this%D_hat(0:N,0:N))
      this%l_at_minus_one = 0.0_RP
      this%l_at_one       = 0.0_RP
      this%weights        = 0.0_RP
      this%D              = 0.0_RP
      this%D_hat          = 0.0_RP
      this%N              = N
!
      CALL LegendreGaussNodesAndWeights(x,this%weights,N)
      CALL BarycentricWeights(x,BaryWeights,N)
      CALL LagrangeInterpolatingPolynomial(x,BaryWeights,this%l_at_minus_one,-1.0_RP,N)
      CALL LagrangeInterpolatingPolynomial(x,BaryWeights,this%l_at_one      , 1.0_RP,N)
      CALL PolynomialDerivativeMatrix(this%D,x,N)
      DO j = 0,N
         DO i = 0,N
            this%D_hat(i,j) = -this%D(j,i)*(this%weights(j)/this%weights(i))
         END DO! i
      END DO! j
```

```
!
      RETURN
   END SUBROUTINE ConstructNodalDGStorage
!
!//////////////////////////////////////////////////////////////////////
!
   SUBROUTINE DestructNodalDGStorage(this)
      IMPLICIT NONE
      TYPE(NodalDGStorage) :: this

      this%N = NONE ! NONE defined as -1
      DEALLOCATE(this%l_at_minus_one,this%l_at_one,this%weights,this%D_hat,this%D)
   END SUBROUTINE DestructNodalDGStorage
!
END MODULE NodalDGStorageClass
```

10.2 DG Element Class

Inherent in the DGSEM approximation is that the domain of interest is broken into subdomains (called elements) and the solution on each element is taken to be a polynomial of degree N. It will make the workflow of an implementation easier to follow if it can match this element construct. Therefore, we create the *DGElementClass*.

The *DGElementClass*, whose data and procedures are gathered in Alg. 8, stores the element's geometry data and solution. The geometry data for a one dimensional element is the left and right boundary locations and the length. Other data stored in an element includes the numerical fluxes, $\mathbf{F}^{*,L/R}$, on the left and the right of the element and the interpolated values of the solution, $\mathbf{Q}^{L/R}$, from which to compute the numerical flux. For convenience, we also have the element store the number of equations. Finally, we store the solution and time derivative on each element.

To compute the weak form, one dimensional spatial derivative on the k^{th} element

$$\dot{\mathbf{Q}}_j^k = -\frac{2}{\Delta x_k}\left\{\frac{\ell_j(1)}{w_j}\mathbf{F}^*\left(\mathbf{Q}^k(1),\mathbf{Q}^{k+1}(-1),\hat{x}\right) - \frac{\ell_j(-1)}{w_j}\mathbf{F}^*\left(\mathbf{Q}^{k-1}(1),\mathbf{Q}^k(-1),-\hat{x}\right) + \sum_{m=0}^{N}\mathbf{F}_m\hat{D}_{jm}\right\},$$

where $j = 0,\ldots,N$ and $k = 1,\ldots,K$ we need access to the data stored in the *NodalDGStorageClass*. Thus, we pass an instance of the *NodalDGStorageClass*, called *dG*, to the derivative computation implemented in the procedure *SystemDGDerivative*.

Algorithm 8: *DGElementClass*: Stores the solution, time derivative, etc. on a given subdomain.

```
Class DGElement
   Data:
      $nEqn,\ \Delta x,\ x_L,\ x_R,\ \left\{\mathbf{F}^{*,L}\right\}_{n=1}^{nEqn},\left\{\mathbf{F}^{*,R}\right\}_{n=1}^{nEqn},\left\{\mathbf{Q}^{L}\right\}_{n=1}^{nEqn},\left\{\mathbf{Q}^{R}\right\}_{n=1}^{nEqn},\left\{\mathbf{Q}_{j,n}\right\}_{j=0;n=1}^{N;nEqn},\left\{\dot{\mathbf{Q}}_{j,n}\right\}_{j=0;n=1}^{N;nEqn}$
   Procedures:
      ConstructDGElement($x_L$,$x_R$,$nEqn$)
      LocalTimeDerivative()
      SystemDGDerivative($dG$)
      DestructDGElement()
End Class DGElement
```

Below we give an implementation of the *DGElementClass*. There are lots of moving parts to a DG implementation, so it is helpful to break it apart into manageable chunks by way of a MODULE. We note that in Fortran a line can only be 132 characters long. So, if it turns out a FUNCTION or SUBROUTINE call exceeds this limit we use the continuation operator & to extend a single line across multiple lines. We show an example in `DGElementClass.f90`.

DGElementClass.f90

```
MODULE DGElementClass
   USE NodalDGStorageClass
   IMPLICIT NONE
!
   TYPE DGElement
      REAL(KIND=RP)                             :: delta_x,xL,xR
      INTEGER                                   :: nEqn
      REAL(KIND=RP),ALLOCATABLE,DIMENSION(:)    :: QR,QL,FstarR,FstarL
      REAL(KIND=RP),ALLOCATABLE,DIMENSION(:,:) :: Q,Q_dot
   END TYPE DGElement
!
CONTAINS
!
   SUBROUTINE ConstructDGElement(this,nEqn,xL,xR,N)
      IMPLICIT NONE
      INTEGER        ,INTENT(IN)    :: N
      TYPE(DGElement),INTENT(INOUT) :: this
      REAL(KIND=RP)  ,INTENT(IN)    :: xL,xR
      INTEGER        ,INTENT(IN)    :: nEqn
!
      ALLOCATE(this%QR(nEqn),this%QL(nEqn),this%FstarR(nEqn),this%FstarL(nEqn))
      ALLOCATE(this%Q(0:N,nEqn),this%Q_dot(0:N,nEqn))
      this%nEqn    = nEqn
      this%xL      = xL
      this%xR      = xR
      this%delta_x = xR - xL
      this%QR      = 0.0_RP
      this%QL      = 0.0_RP
      this%FstarR  = 0.0_RP
      this%FstarL  = 0.0_RP
      this%G       = 0.0_RP
      this%Q       = 0.0_RP
      this%Q_dot   = 0.0_RP
!
      RETURN
   END SUBROUTINE ConstructDGElement
!
!//////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE LocalTimeDerivative(this,dG)
      IMPLICIT NONE
      TYPE(DGElement)     ,INTENT(INOUT) :: this
      TYPE(NodalDGStorage),INTENT(IN)    :: dG
!  Local variables
      INTEGER                                     :: j,N,nEqn
      REAL(KIND=RP),DIMENSION(0:dG%N,this%nEqn) :: F,F_prime
!
      N    = dg%N
      nEqn = this%nEqn
      DO j = 0,N
         CALL Flux(this%Q(j,:),F(j,:),nEqn)
      END DO! j
!  the & is used as a continuation operator if a line in Fortran becomes too long
      CALL SystemDGDerivative(this%FstarR,this%FstarL,F,F_prime,dG%D_hat,dG%weights,&
                              & dG%l_at_one,dG%l_at_minus_one,nEqn,N)
!  Scale by the inverse of the Jacobian
```

```
      this%Q_dot = (-2.0_RP/this%delta_x)*F_prime
!
      RETURN
   END SUBROUTINE LocalTimeDerivative
!
!//////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE SystemDGDerivative(FR,FL,F,Fprime,Dhat,weights,l_one,l_minus_one,nEqn,N)
      IMPLICIT NONE
      INTEGER                           ,INTENT(IN)  :: nEqn,N
      REAL(KIND=RP),DIMENSION(nEqn)     ,INTENT(IN)  :: FR,FL
      REAL(KIND=RP),DIMENSION(0:N,nEqn),INTENT(IN)  :: F
      REAL(KIND=RP),DIMENSION(0:N,nEqn),INTENT(OUT) :: Fprime
      REAL(KIND=RP),DIMENSION(0:N,0:N)  ,INTENT(IN)  :: Dhat
      REAL(KIND=RP),DIMENSION(0:N)      ,INTENT(IN)  :: weights,l_one,l_minus_one
!  Local variables
      INTEGER :: i,j
!
      F_prime = 0.0_RP
!  Volume terms
      DO i = 1,nEqn
         CALL MxVDerivative(Fprime(:,i),F(:,i),Dhat,N)
      END DO! i
!  Surface terms
      DO j = 0,N
         DO i = 1,nEqn
            Fprime(j,i) = Fprime(j,i) + (FR(i)*l_one(j) + FL(i)*l_minus_one(j))/weights(j)
         END DO! i
      END DO! j
!
      RETURN
   END SUBROUTINE SystemDGDerivative
!
!//////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE DestructElement(this)
      IMPLICIT NONE
      TYPE(DGElement),INTENT(INOUT) :: this
!
      this%nEqn    = NONE
      this%xL      = 0.0_RP
      this%xR      = 0.0_RP
      this%delta_x = 0.0_RP
      DEALLOCATE(this%QR,this%QL,this%FstarR,this%FstarL,this%Q,this%Q_dot)
!
      RETURN
   END SUBROUTINE DestructElement
!
END MODULE DGElementClass
```

10.3 DG Mesh Class

We manage the data for the approximation on the entire domain at the mesh level. The *DGMeshClass*, described in Alg. 9, stores the number of elements K, the elements themselves, an instance of *NodalDGStorage*, and the connections between the elements. The *sharedNodePointers* store pointers to the elements on the left and the right of an interface and simplify the computation of the numerical flux. We assume here that $x_R > x_L$ so that the $\mathbf{Q}^L$ and $\mathbf{Q}^R$ arrays are on the left and right of the elements. That way we do not have to store information in the *sharedNodePointers* to distinguish between which corresponds to the left and which to the right. For a two or three dimensional implementations, we would have to be more general.

The constructor for the mesh class creates the elements and connections. The constructor will take the number of elements and the location of the element boundaries as input. It constructs an instance of the *NodalDGStorage* class and uses the element boundary information to construct the elements. The element connections are constructed next. Since there is essentially no difference between a physical boundary and an element boundary, the limits on the p^k array include the endpoints. At the physical boundaries we set the neighboring element to a defined constant NONE. Later, we can test for being on a boundary by checking to see if one of the elements equals NONE.

The global time derivative procedure performs four basic operations. First, it interpolates the solutions to the boundaries on each of the elements. It then computes the physical boundary values by way of a procedure *ExternalState* whose implementation is problem dependent. We pass a parameter with defined values of LEFT or RIGHT to the procedure so that different boundary conditions can be applied at the left and right boundaries. For full generality, we also pass the boundary value of the solution to allow for the implementation of reflecting boundary conditions. Then the numerical fluxes are computed for each element boundary point and sent to the appropriate element. Again, we have assumed that the elements are laid out left to right. Finally, each element computes its local time derivative values.

Algorithm 9: *DGMeshClass*: Stores the elements and DG solution space for the global approximation.

```
Class DGMesh
    Data:
      $K$ // # of elements
      $\{e_k\}_{k=1}^{K}$ // Elements
      $\{p^k\}_{k=0}^{K}$ // sharedNodePointers
    Procedures:
      ConstructDGMesh($K$,$N$,$\{x_k\}_{n=0}^{K}$)
      GlobalTimeDerivative(t)
      DestructDGMesh()
End Class DGMesh
```

Finally, we present an implementation of the *DGMeshClass*. Note the next step to an approximation would be to send the computational mesh to an explicit time integration routine (like Runge-Kutta or Adams-Bashforth) to complete the solution of the PDE numerically.

DGMeshClass.f90

```fortran
MODULE DGMeshClass
   USE DGElementClass
   IMPLICIT NONE
!
   TYPE NodePointers
      INTEGER :: eLeft,eRight
   END TYPE NodePointers
!
   TYPE DGMesh
      INTEGER                                      :: K
      TYPE(NodalDGStorage)                         :: dg
      TYPE(DGElement)   ,ALLOCATABLE,DIMENSION(:) :: e
      TYPE(NodePointers),ALLOCATABLE,DIMENSION(:) :: p
   END TYPE DGMesh
!
CONTAINS
!
```

```
   SUBROUTINE ConstructMesh1D(this,x_nodes,K,N,nEqn)
      IMPLICIT NONE
      TYPE(DGMesh)                ,INTENT(INOUT) :: this
      INTEGER                     ,INTENT(IN)    :: K,N,nEqn
      REAL(KIND=RP),DIMENSION(0:K),INTENT(IN)    :: x_nodes
!  Local variables
      INTEGER :: i
!
      this%K = K
      ALLOCATE(this%e(K),this%p(0:K))
      CALL ConstructNodalDGStorage(this%dg,N)
      DO i = 1,K
         CALL ConstructDGElement(this%e(i),nEqn,x_nodes(i-1),x_nodes(i),N)
      END DO! i
      DO i = 1,K-1
         this%p(i)%eLeft  = i
         this%p(i)%eRight = i+1
      END DO! i
      this%p(0)%eLeft  = NONE
      this%p(0)%eRight = 1
      this%p(K)%eLeft  = K
      this%p(K)%eRight = NONE
!
      RETURN
   END SUBROUTINE ConstructMesh1D
!
!//////////////////////////////////////////////////////////////////////////
!
   SUBROUTINE GlobalTimeDerivative(this,t)
      IMPLICIT NONE
      TYPE(DGMesh) ,INTENT(INOUT)            :: this
      REAL(KIND=RP),INTENT(IN)               :: t
!  Local variables
      REAL(KIND=RP),DIMENSION(this%e(1)%nEqn) :: QL_ext,QR_ext,F
      REAL(KIND=RP)                          :: x_temp
      INTEGER                                :: i,j,idL,idR,nEqn,N
!
      N    = this%dg%N
      nEqn = this%e(1)%nEqn
      DO j = 1,this%K
!  Interpolate solution to the boundary
         DO i = 1,nEqn
            CALL InterpolateToBoundary(this%e(j)%Q(:,i),this%dg%l_at_minus_one, &
                                       & this%e(j)%QL(i),N)
            CALL InterpolateToBoundary(this%e(j)%Q(:,i),this%dg%l_at_one,this%e(j)%QR(i),N)
         END DO! i
      END DO! j
!  Impose boundary conditions
      j = this%p(0)%eRight
      CALL AffineMap(-1.0_RP,this%e(j)%xR,this%e(j)%xL,x_temp)
      CALL ExternalState(QL_ext,this%e(j)%QL,x_temp,t,LEFT,nEqn)
      j = this%p(this%K)%eLeft
      CALL AffineMap(1.0_RP,this%e(j)%xR,this%e(j)%xL,x_temp)
      CALL ExternalState(QR_ext,this%e(j)%QR,x_temp,t,RIGHT,nEqn)
```

```
!  Solve the Riemann problem to get numerical flux at the interface
      DO j = 0,this%K
         idL = this%p(j)%eLeft
         idR = this%p(j)%eRight
         IF (idL .EQ. NONE) THEN
            CALL RiemannSolver(QL_ext,this%e(idR)%QL,this%e(idR)%FstarL,-1,this%e(idR)%nEqn)
         ELSE IF (idR .EQ. NONE) THEN
            CALL RiemannSolver(this%e(idL)%QR,QR_ext,this%e(idL)%FstarR, 1,this%e(idL)%nEqn)
         ELSE
            CALL RiemannSolver(this%e(idL)%QR,this%e(idR)%QL,F,1,this%e(idR)%nEqn)
            this%e(idR)%FstarL = -F
            this%e(idL)%FstarR =  F
         END IF
      END DO! j
!  Compute the time derivative on each element
      DO j = 1,this%K
         CALL LocalTimeDerivative(this%e(j),this%dg)
      END DO! j
!
      RETURN
   END SUBROUTINE GlobalTimeDerivative
!
!///////////////////////////////////////////////////////////////////////
!
   SUBROUTINE DestructDGMesh(this)
      TYPE(DGMesh),INTENT(INOUT) :: this
!  Local variables
      INTEGER :: i

      DO i = 1,this%K
         CALL DestructDGElement(this%e(i))
      END DO! i
      CALL DestructNodalDGStorage(this%dg)
      DEALLOCATE(this%e,this%p)
      this%K = NONE
      RETURN
   END SUBROUTINE DestructDGMesh
!
END MODULE DGMeshClass
```

10.4 Output Solution to a File

There is no native plotter in Fortran. You cannot simply type `plot` to visualize the computed solution of a given PDE. Instead you have to output the solution to a file and then use another piece of software to plot it. Somewhat frustratingly, the format of the output files becomes important for the plotter. Different plotters can only read certain types of files.

To visualize the computed solution we will focus on TecPlot type files, which use the extension `.tec`, and can be plotted using the software `VisIt`. For convergence plots we will demonstrate how to use `matplotlib`, a package for Python, for which we can use much simpler `.dat` files.

10.4.1 Plotting Routines for `VisIt`

We forgo pseudocode in this section and only provide working implementations procedures to output the solution for plotting in proper TecPlot format. The routine assumes you provide a file associated with the INTEGER fUnit.

Note that all these routines are designed to print a single component of the solution (e.g. the density ρ in the Euler equations). It is possible to output multiple pieces of solution data into the same .tec file. We simply have to provide appropriate labels for the data.

An important thing to remember in VisIt is that it has "smart" file grouping. What this means is, if you have a series of files representing the solution at different times, say $t = 0$ to $t = 1$ with $\Delta t = 0.1$, you have 11 files. If you want to create a movie of the solution you simply name these files something like Movie00.tec to Movie11.tec or rho00.tec to rho11.tec. VisIt automatically recognizes that the files have the same name and are ordered by number. It then open the files simultaneously and allow you to play the movie of the solution. It is also very easy to save movies so you can play them later at presentations, impressing everybody.

First, we show a one dimensional output routine for solution data. In this case we use the .curve proprietary file extension. You pass the mesh x and solution u both of size $N + 1$. We use the pound sign, #, to name the solution. It may be useful to name the curve $N = 3, K = 128$ to indicate the parameters of the computation. To do this you include a #N=3,K=128 at the beginning of the file. We provide a one dimensional VisIt plot in Fig. 3.

Print 1D Solution to File

```
SUBROUTINE ExportToTecplot_1D(x,u,N,fUnit,exact)
!  File writer for one dimensional solution vector
   USE Constants,ONLY: RP
   IMPLICIT NONE
   INTEGER                     ,INTENT(IN)          :: N,fUnit
   REAL(KIND=RP),DIMENSION(0:N),INTENT(IN)          :: x,u
   REAL(KIND=RP),DIMENSION(0:N),INTENT(IN),OPTIONAL :: exact
!  Local variables
   INTEGER :: j
!
   WRITE(fUnit,*)'#N=3,K=128'
   DO j = 0,N
      WRITE(fUnit,*)x(j),u(j)
   END DO! j
!  if the exact solution is available, output that as well
   IF (PRESENT(exact)) THEN
      WRITE(fUnit,*)'#exact'
      DO j = 0,N
         WRITE(fUnit,*)x(j),exact(j)
      END DO! j
   END IF
!
   RETURN
END SUBROUTINE ExportToTecplot_1D
```

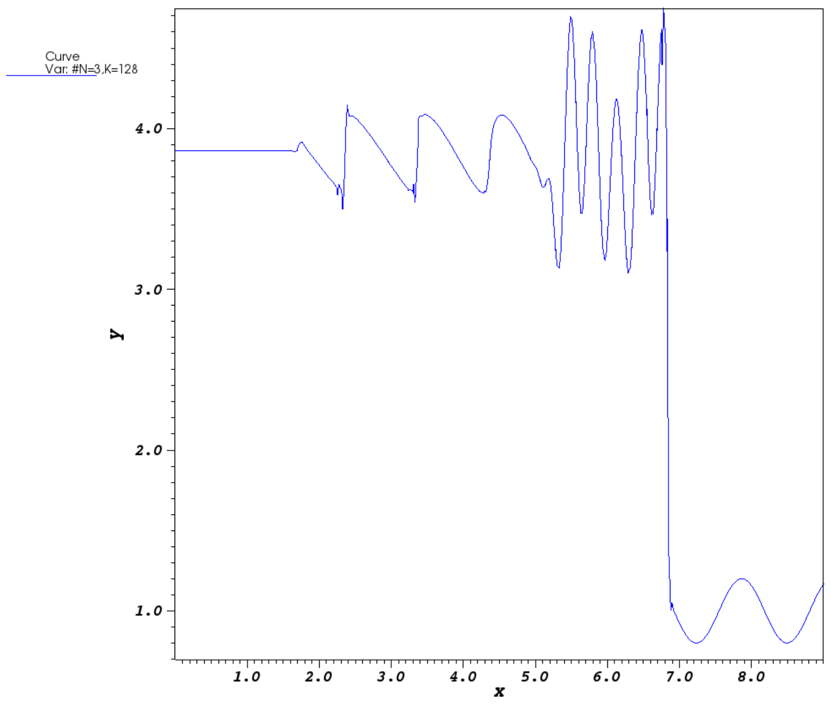

Figure 3: Sine-shock interaction at $T = 1.8$.

Next we provide a two dimensional procedure to output the solution to a file. Again, we assume there is a file name associated with the INTEGER fUnit. You pass the SUBROUTINE the points in each direction x and y and the solution u each of size $K \times (N+1) \times (N+1)$ where K is the number of quadrilateral elements. The CHARACTER solutionFile lets you name the solution. For example solutionFile = 'p' for the pressure. The structure of the two dimension `.tec` file is a little more complicated. Basically, you have to tell `VisIt` how the solution is sliced in the $x - y$ direction. This structure is automatically incorporated in the implementation provided. We provide an example of a two dimensional `VisIt` plot in Fig. 4.

Print 2D Solution to File

```
SUBROUTINE ExportToTecplot_2D(x,y,u,N,K,fUnit,solutionFile)
   USE Constants,ONLY: RP
   IMPLICIT NONE
   INTEGER                            ,INTENT(IN) :: N,K,fUnit
   CHARACTER(LEN=*)                   ,INTENT(IN) :: solutionFile
   REAL(KIND=RP),DIMENSION(K,0:N,0:N),INTENT(IN) :: x,y,u
!  Local variables
   INTEGER :: i,j,l
!
   WRITE(fUnit,*)'TITLE = "',solutionFile,'solution.tec"'
   WRITE(fUnit,*)'VARIABLES = "x","y","',solutionFile,'"'
!
   DO l = 1,K
      WRITE(fUnit,*)"ZONE I =",N+1,",J=",N+1,",F=POINT"
      DO j = 0,N
         DO i = 0,N
            WRITE(fUnit,*)x(l,i,j),y(l,i,j),u(l,i,j)
         END DO! i
      END DO! j
   END DO! l
!
   RETURN
END SUBROUTINE ExportToTecplot_2D
```

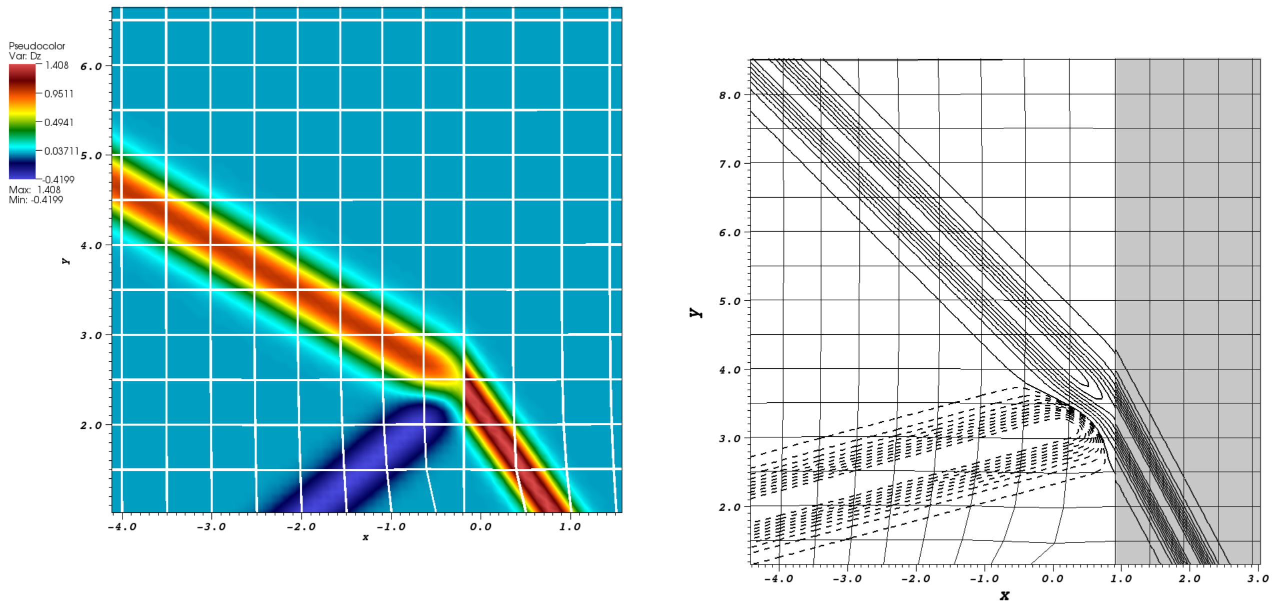

Figure 4: Pseudocolor and contour plot of reflection and transmission of the D_z component of TE Gaussian plane wave from a moving dielectric interface. White or black borders indicate element boundaries.

An alternative implementation of the two dimensional procedure is provided. This version of the export routine is taken from an implementation of the shallow water equations where the quantities of interest are water height h and velocity components u, v in the x, y directions. However, it is written in a general enough fashion that this routine can be adapted to another set of equations of interest.

Print 2D Solution to File (Multicomponent Version)

```
SUBROUTINE ExportAllToTecplot_2D(x,y,u,N,K,fUnit,solutionFile,nEqn)
   USE Constants,ONLY: RP
   IMPLICIT NONE
! here nEqn is the number of equations
   INTEGER                                  ,INTENT(IN) :: N,K,fUnit,nEqn
   CHARACTER(LEN=*)                         ,INTENT(IN) :: solutionFile
   REAL(KIND=RP),DIMENSION(K,0:N,0:N)       ,INTENT(IN) :: x,y
   REAL(KIND=RP),DIMENSION(K,0:N,0:N,nEqn),INTENT(IN) :: u
!  Local variables
   INTEGER :: i,j,l
!
   WRITE(fUnit,*)'TITLE = "',solutionFile,'solution.tec"'
   WRITE(fUnit,*)'VARIABLES = "x","y","h","u","v"'
!
   DO l = 1,K
      WRITE(fUnit,*)"ZONE I =",N+1,",J=",N+1,",F=POINT"
      DO j = 0,N
         DO i = 0,N
```

```
            WRITE(fUnit,*)x(l,i,j),y(l,i,j),u(l,i,j,1),u(l,i,j,2),u(l,i,j,3)
         END DO! i
      END DO! j
   END DO! l
!
   RETURN
END SUBROUTINE ExportAllToTecplot_2D
```

Finally, we provide a three dimensional procedure to output the solution to a file. Again, fUnit indicates the filename, The SUBROUTINE accepts the points in each direction x, y, z and the solution u each of size $K \times (N+1) \times (N+1) \times (N+1)$ where K is the number of hexahedral elements. The CHARACTER solutionFile lets you name the solution. This output function assumes the approximation is of the same order, $N+1$, in each direction. We provide an example of a three dimensional VisIt plot in Fig. 5. We format the output of the 3D data to print four real numbers with thirteen digits, five of which are after the decimal point.

Print 3D Solution to File

```
SUBROUTINE ExportToTecplot_3D(x,y,z,u,N,K,fUnit,solutionFile)
   USE Constants,ONLY: RP
   IMPLICIT NONE
   INTEGER                                ,INTENT(IN) :: N,K,fUnit
   CHARACTER(LEN=*)                       ,INTENT(IN) :: solutionFile
   REAL(KIND=RP),DIMENSION(1:K,0:N,0:N,0:N),INTENT(IN) :: x,y,z,u
!  Local variables
   INTEGER            :: i,j,h,p
   CHARACTER(LEN=32) :: valuesFMT
!  Format the output of real numbers
   valuesFMT = "(4E13.5)"
!
   WRITE(fUnit,*)'TITLE = "',solutionFile,'solution.tec"'
   WRITE(fUnit,*)'VARIABLES = "x","y","z","',solutionFile,'"'
!
   DO h = 1,K
      WRITE(fUnit,*)"ZONE I =",N+1,",J=",N+1,",K=",N+1,",F=POINT"
      DO i = 0,N
         DO j = 0,N
            DO p = 0,N
               WRITE(fUnit,valuesFMT)x(h,i,j,p),y(h,i,j,p),z(h,i,j,p),u(h,i,j,p)
            END DO! p
         END DO! i
      END DO! j
   END DO! h
!
   RETURN
END SUBROUTINE ExportToTecplot_3D
```

10.4.2 Plotting Routines for matplotlib

We use matplotlib to create convergence plots. The reason is that matplatlib is a little more general, and we don't need any special format for the data finals. So, for example, all you need to create a spectral convergence plot is a list of the polynomial order N and the error (I usually use the $\mathcal{L}_\infty$ error because it is easy to calculate). We only need a few Python commands to create these plots. Again, we simply provide implementations that demonstrate how to use matplotlib. Note that in python the # symbol is the comment command. One nice feature is that matplotlib can load LaTeX packages such that you can put mathematical symbols in axis labels and figure legends. Also, this means that the fonts used in a figure will match the fonts of the larger LaTeX document, which is a nice consistency.

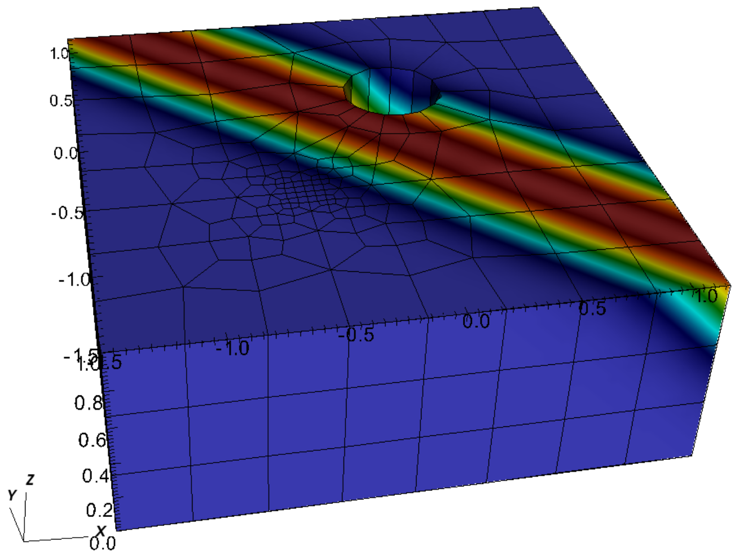

Figure 5: Pseudocolor plot of propagation of a pressure wave across a 3D moving mesh. Black lines indicate element boundaries.

First, we'll create a semilog spectral convergence plot. This demonstrates the exponential convergence of the DGSEM approximation. All we need to assume is that we have a file `specConv.dat` which contains the polynomial order and error information. The Python code to create the plot is given below. We show the resulting convergence plot in Fig. 6.

Spectral Convergence Plot

```
#!/usr/bin/env python

from math import log10
import matplotlib
matplotlib.use('Agg')
from matplotlib import rc
from numpy import *
from matplotlib.pyplot import *

rc('text', usetex=True)
rc('font', family='serif')
rc('font', size=10)
matplotlib.rcParams['text.latex.preamble']=[r"\usepackage{amsmath}"]

fig = figure(1, figsize=(5,5))
ax = fig.add_subplot(111)
xlabel('$N$',fontsize=14)
ylabel(r'$L_{\infty}$ Error',fontsize=14)
ax.xaxis.set_minor_locator( MultipleLocator(1) )
ax.yaxis.set_minor_locator( MultipleLocator(1) )
ax.axhline(color="black")
ax.axvline(color="black")

data = loadtxt('specConv.dat')
error1 = array([[data[i][0],data[i][1]] for i in range(len(data))])
error2 = array([[data[i][0],data[i][2]] for i in range(len(data))])
ax.plot(error1[:,0], error1[:,1],'*-',markersize=7,label='$\Delta t = 1/2000$')
ax.plot(error1[:,0], error2[:,1],'o-',markersize=5,label='$\Delta t = 1/4000$')
```

```
semilogy()
axis([3,17,5e-9,2e-2])

legend = ax.legend(numpoints=1,loc='upper right')
setp(legend.get_texts(), fontsize=10)

tight_layout(1)

savefig('specConv.pdf')
```

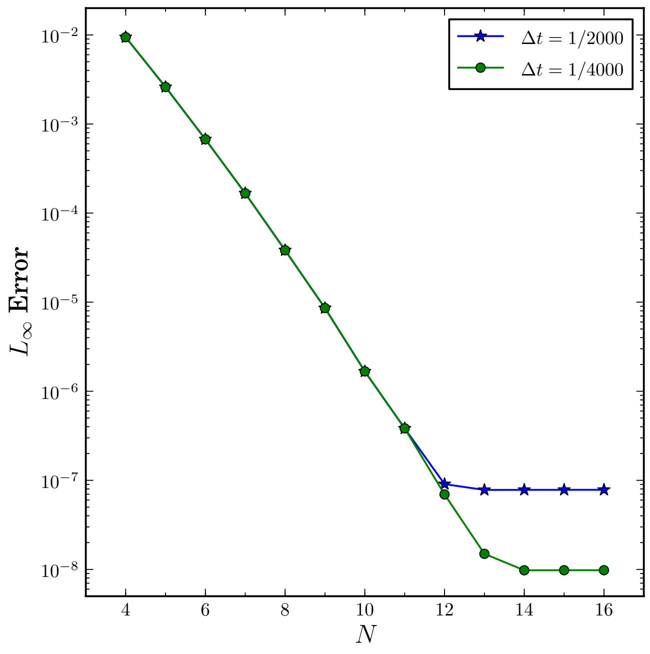

Figure 6: Semilog plot displays spectral convergence.

We also give an example of code to create a log-log time convergence plot. We add a triangle to indicate the slope of the line which conveniently shows the correct temporal order. We assume that there is a file containing the temporal error `timeConv.dat`. We present the example of the log-log temporal convergence plot in Fig. 7.

Temporal Convergence Plot

```
#!/usr/bin/env python

from math import log10
import matplotlib
matplotlib.use('Agg')
from matplotlib import rc
from numpy import *
from matplotlib.pyplot import *

rc('text', usetex=True)
rc('font', family='serif')
rc('font', size=10)
matplotlib.rcParams['text.latex.preamble']=[r"\usepackage{amsmath}"]

fig = figure(1, figsize=(5,5))
ax = fig.add_subplot(111)
```

```
xlabel(r'$\log_{10}(\Delta t)$',fontsize=14)
ylabel(r'$\log_{10}(L_{\infty}\textrm{ Error})$',fontsize=14)
ax.axhline(color="black")
ax.axvline(color="black")

triangle_points = [[(-3.098),(-6.175)],[(-3.098),(-6.55)],[(-3.222),(-6.55)]]
triangle = Polygon(triangle_points,fill=False)
x_mid = (triangle_points[2][0] + triangle_points[0][0])/2.0
y_mid = ((triangle_points[0][1] + triangle_points[1][1])/2.0)
vertical_label_pos = (triangle_points[0][0],y_mid)
horizontal_label_pos = (x_mid,triangle_points[1][1])
ax.add_patch(triangle)
ax.annotate('$3$',xy=vertical_label_pos,xytext=(+5,-1.5),textcoords='offset points',
            ha='center',va='center')
ax.annotate('$1$',xy=horizontal_label_pos,xytext=(0,-5),textcoords='offset points',
            ha='center',va='center')

data = loadtxt('timeConv.dat')
error1 = array([[data[i][2],data[i][3]] for i in range(len(data))])
ax.plot(error1[:,0], error1[:,1],'o-',markersize=5)
axis([(-3.31),(-3),(-6.6),(-6.09)])

tight_layout(1)

savefig('timeConv.pdf')
```

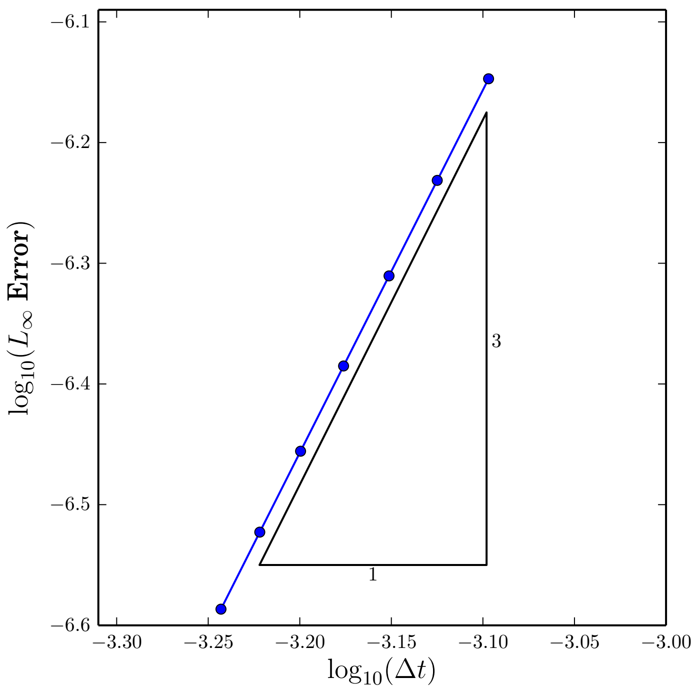

Figure 7: Loglog plot of the temporal convergence. We see the correct convergence rate for third order Runge-Kutta.

Chapter 11: Source Code Management with `git`

Now that we know how to write and organize computer programs in Fortran, which are possibly object oriented, a practical question arises. How do we keep track of alterations to our source code? Is it possible to rewind to a previously known working version of a program? The answer to both of these questions is yes, through a source code management tool. We will cover a commonplace and popular tool known as `git`, but other options are avaiable like CVS, SVN, or Mercurial. All of the options for source code management require a bit of background knowledge, but once we know a few basic commands we can save ourselves a lot time and/or heartache as we change programming projects.

What is the purpose of managing source code? A short answer is it let's us edit and alter working source code without fear. We can always rewind a project to a previous working version if our edits "break" the source code, i.e., edits to a master version of the code may cause the program not to compile or could introduce bug(s). A longer answer is that source code management offers a structured way to organize and document changes we make to our programming projects. Also, source code management tools assist in the creation of large, collaborative programming projects. We can have *several people* working independently on a shared master version of the source code. In all these ways a code management tool makes a groups source code easier to read and we can explain what changes were made to the code and when.

This is a quick introduction to the basic functionality of `git`. To learn about the more advanced `git` commands and structures there is always Google. However, so you don't have to wade through a bunch of search results. A thorough, free book that outlines `git`'s functionality is located at

```
www.git-tower.com/learn/ebook/command-line/
```

The source code tool `git` comes preinstalled on Mac devices. To install `git` on a Linux system you can use the command

```
sudo apt-get install git
```

and on Windows follow the instructions on the following webpage

```
http://msysgit.github.io
```

Source code management through `git` will introduce a repository of (hidden) files. To construct a repository we point the terminal to the file containing our source files and type the command

```
git init
```

The first thing you should do you after `git` is installed is set your user name and e-mail address. This is important because every `git` commit uses this information, and it's immutably baked into the commits you pass around. This is especially important in remote repositories to keep track of who has made what changes to pieces of code.

```
git config --global user.name "Inigo Montoya"
git config --global user.email inigo.montoya@example.se
```

You can change other settings as well using the `config` option. You can always check your `git` settings for a certain repository with the command

```
git config --list
```

Another important preliminary part of the `git` repository is creating a `.gitignore` file. In this file, you can tell the repository to ignore certain files that it should not concern itself with tracking changes. Example of such files would be compiled code files like those with the `.o` or `.mod` extensions. Also, you will not track changes to output files like those with a `.tec` extension. This is because these files all change frequently when recompiling or running the code and tracking them does not make sense, particularly for a collaborative coding project. Once we are inside the folder that contains the `git` repository we create a `.gitignore` file with

```
touch .gitignore
```

Note that the file name `.gitignore` is ***case sensitive*** and the name of the file matters. The `git` repository looks for a file with this specific name and will not stage or commit files that it is told to ignore. The `.gitignore` is a hidden file that can easily be edited in the terminal using programs like `pico`, `nano`, or `vim`. Here is an example of the possible `.gitignore` contents

Example .gitignore

```
# Add any directories or files you don't want to be tracked by git version control

# Ignore any files with these extensions
*.o
*.mod
*.tec

# Ignore an entire folder (always append a slash)
plotFiles/
```

Now that we have a repository we can add source files to it using

```
git add fileName
```

However, once we add a file we have not ***committed*** the changes into the repository. We have simply told `git` to stage the file and have it ready to commit. Next we tell `git` to commit changes for a given file using the command

```
git commit fileName
```

If we want to commit all files that have been changed (without even adding them) we can use the blanket command

```
git commit -a
```

This command will prompt for a comment to be added, which can be added in a `pico` type environment. To add the comment at the command line we can use the `-m` command,

```
git commit -a -m "Commit message"
```

We can experiment will new source code by introducing ***branches***. A branch allows us to keep multiple copies of the source code in the repository. The command

```
git branch -b 'experimental'
```

will create a new branch off of the "master" containing new source code. If we mess things up too much on the experimental branch we can rewind to an earlier committed version (or to the "master" branch). If we type

```
git branch
```

it will tell us which branch we are on. With the `git checkout [branch_name]` command we can move between branches of the repository.

Once we are sure that the code on an 'experimental' branch is debugged how do we add it back to the main branch of source code? This is accomplished through the ***merge*** command. You can merge any branch into your current branch with the `git` merge command. To include changes on the "experimental" branch on the "master" branch, you can merge in the "experimental" branch.

```
git merge experimental
```

This may cause conflicts, which `git` will identify, that the user must resolve. Once any conflict issues are resolved the user must add the offending files once again and the commit like before.

If we really mess up the source code on an experimental branch we can rewind to a previous version. To do so we will reassign where the head (`$HEAD`) of the `git` repository points. This can also undo an unwise merge that was made from another branch. This includes the commands `git revert` and `git reset`. The difference is `revert` goes back to a previous version, but keeps a copy. While `reset` will erase earlier versions on the branch. For example, if we want to rewind to a previous version of code and discard all changes we say

```
git checkout master
git reset --hard [previous working version]
```

There are many other command and subtleties to using `git`, a Google search will reveal as much. But this will get you started with using local repositories.

11.1 Remote Repositories

It is common for a research group to collaborate on a large coding project. As such, the group needs a convenient way to manipulate code, keep track of changes and versions, all while identifying who made certain changes to the source code. This can be achieved using a ***remote repository***. There are many freely available hosting websites to handle a remote `git` repository. Some popular ones are

- `http://www.github.com`: Freely create public repositories that anyone can access. Also offers some documentation capabilities for the source code. Example found at `http://github.com/project-fluxo/fluxo`.
- `http://www.gitlab.com`: Freely create private repositories for your workgroup to branch and experiment. Tied to `github` so it is easy to merge into the public repository and "go live" with source code updates.
- `http://www.bitbucket.org`: Can create private or public repositories. For small groups/projects (5 people or fewer) it is free. For larger groups and more storage space there is a monthly fee.

You can think of the remote repository as a "Master" copy of the source code for the research group. Each member of the group still has a local repository of files that they work with. But committed changes to the code must be pushed to (and pulled from) the remote repository to maintain the latest versions. Much of the functionality of `git` for remote repositories remains the same as it was for a local repository, but for a couple added steps.

First, to initialize a local copy of the remote repository we must make a `clone`. There are different protocols that can be used to copy the remote repository. We provide an example using the SSH transfer protocol (as it is fairly common)

```
git clone user@server:/origin.git
```

We can manage our remote repositories using the `remote` command. To see which remote servers you have configured, you can run the `git remote` command. It lists the short names of each remote handle you havve specified. If you havve cloned your repository, you should at least see *origin*.

Once we have a clone of the repository locally, we operate on and edit files in the same way as we described earlier. We still stage and commit changes to the local repository. But how do we send any of the committed changes back to the remote repository? For this we use the `push` command. In general, you push the changes of whatever branch you are working onto a branch in the remote repository. The general command is

```
git push [alias] [branch]
```

where *alias* identifies the name of the remote repository and *branch* identifies which branch to push onto. So, for example, we can push changes onto the master branch

```
git push origin master
```

Pretty easy. This will automatically merge your committed changes into the master branch. Now if someone clones that repository they will receive your newly pushed master branch. However, it is possible to just push your branch into the repository and a merge can take place later. Instead, we can push a branch you've committed, let's call it palm, into the remote repository using

```
git push origin palm
```

Now when people clone or fetch from that repository, they will see a "palm" branch as well.

What about getting the newest files from the remote repository? We have two, slightly different options to achieve this there is the `fetch` command or the `pull` command. The difference is subtle. If we use the commend

```
git fetch origin [branch]
```

we will download new branches and data from a remote repository, but this command ***does not merge any changes***. The new files will show bookmarks to where each branch was on the remote repository when you synchronized, but it is up to you to inspect the new changes and merge them into your local repository. In contrast, if we used the command

```
git pull origin [branch]
```

we will effectively run a `fetch` command immediately followed by a `merge` command of the branch on the remote repository that is tracked by the branch we are currently working on.

Either option for obtaining the newest data from the remote repository will work, and it boils down to personal preference. The `pull` command is quick and easy, but you relinquish some control over how the merges to your branches occur. Some people find this "magic" updating off-putting. Using `fetch` and then performing the merges yourself can be tedious and time consuming, but it has the advantage that you know exactly what merges are being made and what changes occur in your local repository.

The last major issue you can run into with the remote repository is pushing to remote branches where someone else has pushed changes in the meantime. As an example let's consider two developers, named Luke and Leia, who are working from the remote repository Hoth. If Luke and Leia clone the repository at the same time, both do commits, but then she pushes and then Luke tries to push, `git` will, by default, not allow Luke to overwrite Leia's changes. What happens when Luke's push is initiated is the remote repository, basically, runs `git log` on the branch Luke tries to push in order to make sure that the repository can see the current tip of the server's branch in Luke's push history. If the current tip on the server is missing in Luke's history, it concludes that Luke is out of date and rejects his push. What Luke must do is first ***fetch and merge*** the changes made by Leia's push. Then Luke can push again, though laborious this process makes sure Luke takes all of Leia's changes to the project into account.

Always remember that anytime you merge changes into a branch there may be conflicts you need to resolve in your local repository. It can frustrating when you first start using `git` or any other source code management tool since there is a little bit of a learning curve. However, once you get used to it, source code management becomes indispensable in keeping a stable, working version of a large coding project (whether it is just you or a collaborative effort).

As a closing statement on source code management. ***This is a coding practice not specific to Fortran!*** The `git` utility simply offers a systematic tool to track changes made to files or documents over time. Due to the prevalence and availability of online hosting, remote `git` repositories offer easy dissemination of project files across a workgroup. The focus of this Chapter was to discuss `git` in the context of Fortran, but we can just as easily use `git` if we are collaborating on a coding project in `C++`, `CUDA` or even a LaTeX document.