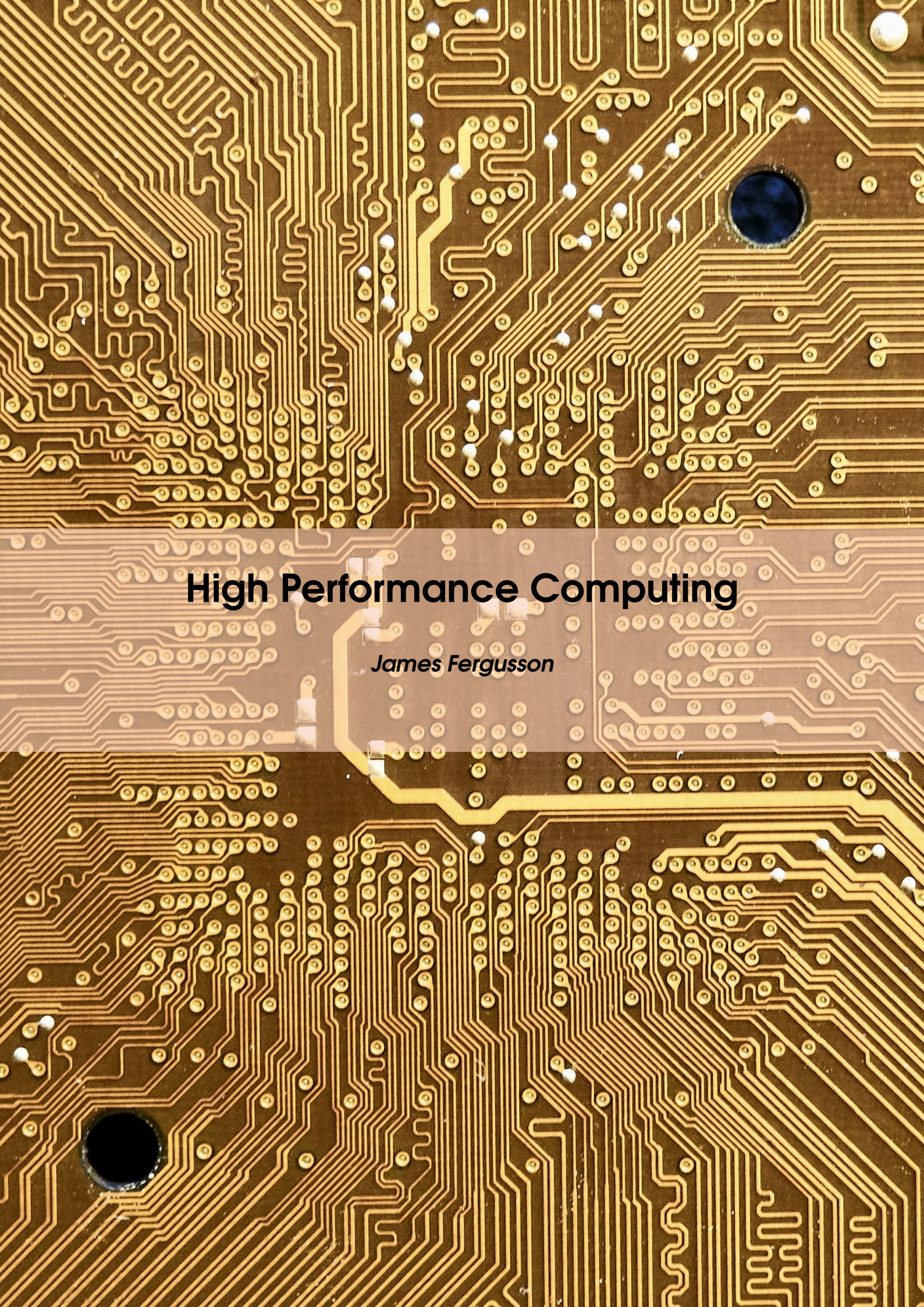


High Performance Computing

James Fergusson

Contents

Preamble

Details:

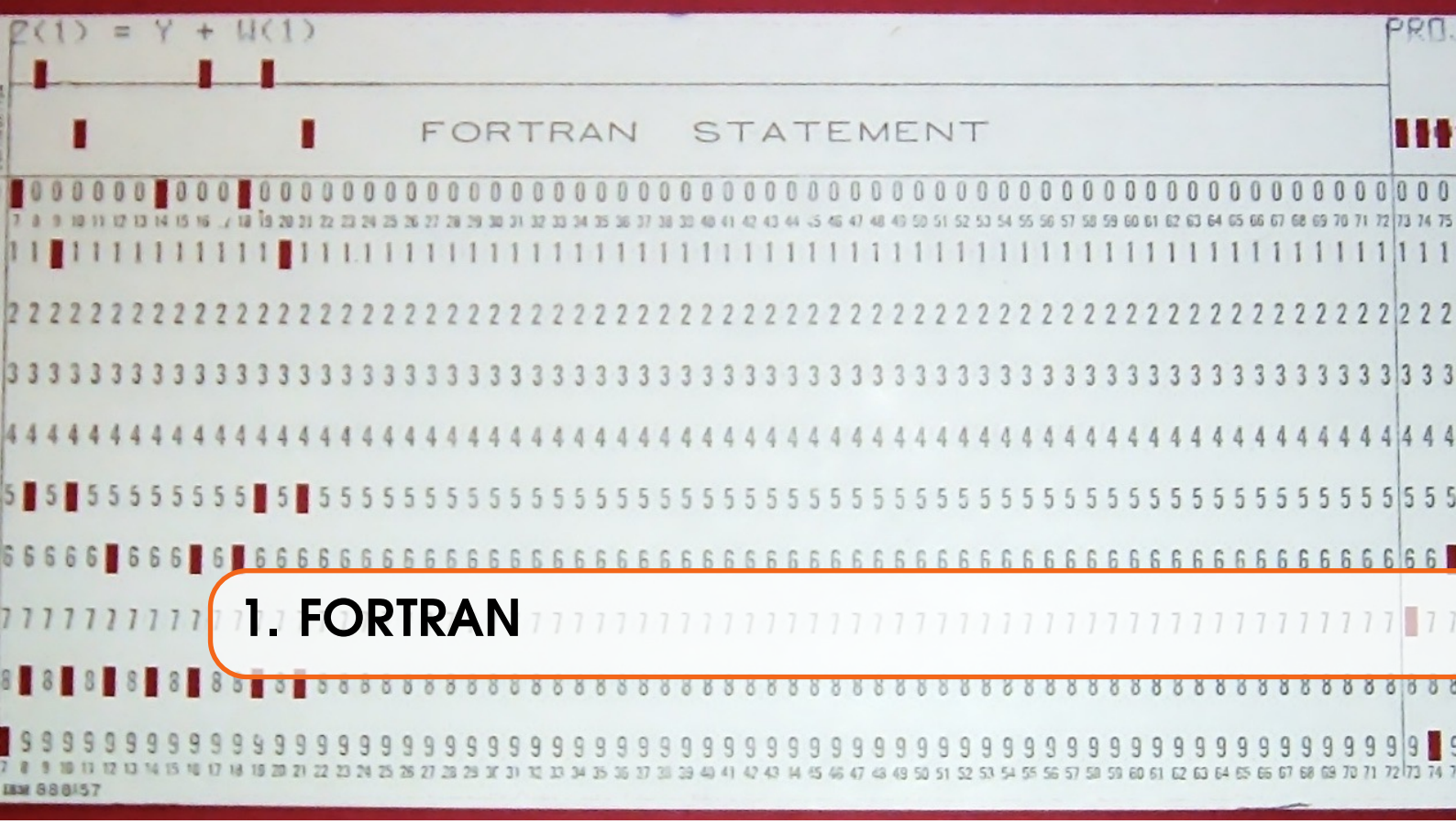
Dr James Fergusson

Room: B1:12

J.Fergusson@DAMTP.cam.ac.uk

Textbooks:

??



1. FORTRAN

1.1 Background

FORTRAN is one of the oldest coding languages. It was first originally developed in the 50s and was designed to work with punch cards which influences the standards created for it. So why would be want to learn about such an old fashioned language, surely we should just switch to more modern languages that are widely used like C++. Well the reason is that FORTRAN was created specificly for numerical computation (FORTRAN is a abbrevication of FORmular TRANslation) when memory and compute were very limited. This means that there is a large pool of legacy code for standard numerical tasks all of which is written in FORTRAN. In fact many standard libraries used in C++, Python and other languages are wrapped FORTRAN code (for example numpy). It is also likely that you will need to interact with FORTRAN codes in academia as many numerical codes used for scientific tasks use it. For example, while C++ dominates software development current statistics for ARCHER2 (a key supercomputer for scientific research) use are:

1. FORTRAN - 81.1%
2. C++ - 14.9%
3. Python - 2.1%
4. Other - 1.9%

so FORTRAN is still dominant in academic circles.

1.2 FORTRAN77

The first FORTRAN standard was released in 1966 but wasn't widely addopted until its revision in 1977. This standard, called FORTRAN77, became very widely used, and persisted for 13 years. It is now obselete for new development but there is a large set of legacy code still used so you may come across it occasionally. It is quite tricky as it has several features which are significantly different to modern coding languages. We will breifly review its quirks so you can recognise it when you see it (and understand why all of your edits make it crash)

Firstly, it was based on punchcards so had some very specific requirement for formating. Each line could only be 80 characters long (as this was the width of a punchcard). Each line reserved

specific purpose for each column:

1. Column 1 : Any character here indidcates the line is a comment (normal to use "C" or "*")
2. Column 2-5 : Label space, used for "GO TO" commands and "FORMAT" specifiers
3. Column 6 : Any character here indicates the line is a continuation of the previous line (normal to use "+")
4. Column 7-72 : Programming Statements
5. Column 73-80 : Sequence Number used to reorder punchcards if they were dropped

So the first issue with working with FORTRAN77 is you have to spend a bit of time counting spaces. If you acidentily put characters in the first spaces you line is ignored as a comment, in the first 5 the initial charaters are interpreted as a label. Anything in space 6 appends this line to the one above. Anything that comes after character 72 is ignored.

Here is a simple Fortran77 program (it is typical to use the extension .f or .for for FORTRAN77):

Snippets/F77/HelloWorld.f

```
1         PROGRAM main
2 c       Simple hello world program
3         WRITE(*,*) 'Hello World'
4         STOP
5         END
```

We should note the following points:

1. You start your code with the command `PROGRAM` and end them with `END`.
2. To write putput to the terminal you use `WRITE(*,*)` where the first * denotes the terminal and the second denotes upspecifed formating.
3. You have to `STOP` the program before it `END`s

To compile and run this code we use the `gfortran` compiler with the commands:

```
1 $ gfortran HelloWorld.f -o helloworld.exe
2 $ ./helloworld.exe
3 Hello World
4 $
```

Due to the limited row length, indentation is sometimes skipped which can make it hard to read. Also FORTRAN ignores all white space, which some programmes use to optimise line lenghts making this even worse.

Here are the intrinsic variables for FORTRAN77:

Snippets/F77/Intrinsics.f

```
1  PROGRAM main
2  IMPLICIT NONE
3
4  INTEGER i,j,k
5  REAL x,y,z1,z2,z3,z4
6  DOUBLE PRECISION d1, d2
7  COMPLEX c1
8  LOGICAL bool1, bool2, bool3, bool4
9  CHARACTER str*8
10 REAL arr(4,5,6)
11
12 i = 2
13 j = 3
```

```
14 x = 2.0
15 y = 3.0
16 c1 = (1,3)
17 str = 'hello'
18
19 z1 = x+y
20 z2 = x*y
21 z3 = y/x
22 z4 = y**x
23 WRITE(*,*) z1,z2,z3,z4
24
25 bool1 = i .LT. j
26 bool2 = i .GE. j
27 bool3 = i .EQ. j
28 bool4 = i .NE. j
29 WRITE(*,*) bool1,bool2,bool3,bool4
30
31 STOP
32 END
```

Firstly I have put the command `INTRINSIC NONE` (technicly not part of the 77 standard but widely supported in 77 compilers) which switched off a standard FORTRAN feature which is that all undefined variables that start with a letter i-n are automaticaly assigned as `INTEGER` and all others are assigned as `REAL`. All variable names (this includes functions and subroutines) can only be 6 characters long and must start with a letter. This is why in numerical packages you often see names like: `POTRF` for Choleski decomposition in the LAPACK numerical library (see more examples here: https://icl.utk.edu/ mgates3/docs/lapack.html).

In FORTRAN77 we only have one integer, which is 32 bit, but we have both 32 (`REAL`) and 64 (`DOUBLE PRECISION`) bit floating point numbers. We have boolians `LOGICAL` and strings with length defined by the *num part. Conveniently we have complex numbers as an intrinsic data type unlike other languages. Arrays are intrinsic, but not dynamic, in FORTRAN77. You can create multi-dimension arrays (up to 7 dimensions) simply by listing the dimension sizes in brackets when declaring them. Finally all declaration (ie assigning variables) must happen *before* any statements (code that does stuff with them). You cannot add variables halfway though some code. This means that you will ivariabley spend alot of time scrolling up and down the code when working on it.

Control flow in FORTAN77 is more limited intrinsicly but can be exteneded to any case by using `GOTO` instructions (which can make yor code very hard to read)

Snippets/F77/Controlflow.f

```
1       PROGRAM main
2       IMPLICIT NONE
3
4       INTEGER i,j,k
5       INTEGER n,m
6
7       n=0
8       m=1
9
10      IF (n .EQ. m) THEN
11          WRITE(*,*) "same"
12      ELSE IF (n .GT. m)
13          WRITE(*,*) "larger"
14      ELSE
```

```
15          WRITE(*,*) "smaller"
16      END IF
17
18      DO 99 i=1,10
19          n = n+i
20  99  CONTINUE
21      WRITE(*,*) n
22
23  88  IF(m .GT. 10) THEN
24          WRITE(*,*) m
25          m = m+1
26          GOTO 88
27      END IF
28      WRITE(*,*) m
29
30      STOP
31      END
```

Here we see the `IF` and `DO` structures plus a `IF`+`GOTO` which defines a 'while' loop. Finally, as FORTRAN does not care about spaces or case (technicaly the FORTRAN standard assumes all uppercase but most compilers don't care) `END IF` `GO TO` are identical to `endif` and `goto`. I personally always write FORTRAN instructions in caps, and variables in lower case, as I think it looks better but this choice is personal.

Fortran77 allows two subprogrammes, `FUNCTION` and `SUBROUTINE`. The first is a normal function $f(x,y,...)$ and the second takes multiple parameters and modifies them.

Snippets/F77/Subprograms.f

```
1  PROGRAM main
2  IMPLICIT NONE
3
4  INTEGER i,j,k
5  INTEGER add
6
7  i=1, j=2
8  k = add(i,j)
9  WRITE(*,*) k
10
11 CALL rotate(i,j,k)
12 WRITE(*,*) i,j,k
13
14 STOP
15 END
16
17 INTEGER FUNCTION add(i,j)
18 IMPLICIT NONE
19 INTEGER i,j
20 add = i+j
21 RETURN
22 END
23
24 SUBROUTINE rotate(i,j,k)
25 IMPLICIT NONE
26 INTEGER i,j,k,n
27 n=i
28 i=j
29 j=k
```

```
30 k=n
31 RETURN
32 END
```

Here the routines are just listed at the end after the main PROGRAM statements. If the code gets too long you can place subprograms in seperate files and add them with the statment INCLUDE filename.f which simply cuts and pastes the code from the file to this location in the main program. Key points are that the function name needs to have a type in both the main program and its declaration, also the name is the variable that you return. Subroutines operate on all variables passed to them and require a CALL statement. Remember that implicit types apply to these too if IMPLICIT NONE is not set, and the 6 character limit still applies to names.

There is quite a bit more I could put here but as you should only use FORTRAN77 if you really have to we will move onto the much more common, and modern, FORTRAN90 standard which is the most common version used today

1.3 FORTRAN90

Fortran90 was a major revision of the standard. So much so that the two standards, while backwards compatable, cannot be used in the same file with each other so a small amount of care must be taken if you wish to combine them.

The main differences between FORTRAN90 and FORTAN77 are:

1. Dropped the static formatting requirement and adopted the flexible format that all modern codes follow. Now variable names can be 31 characters long and lines 132 characters long. Comments start with, lines are continued by putting & at the end of the lihne to be continued and lower case letters are formaly allowed (but FORTRAN is still case insensitive).
2. Introduced MODULEs to replace the INCLUDE method for functions and subroutines.
3. Allowed dynamic array allocation and introduced intrinsic array functions.
4. New control structues like DO WHILE to avoid the need for GOTO and no longer require labels.

Let us look at the intrinsics again as there are some subtilties. We still have the 5 standard intrinsic types: INTEGER, REAL, COMPLEX, LOGICAL, and CHARACTER but we can control the precision of each with KIND

Snippets/F90/Intrinsics.f90

```
1  PROGRAM main
2      IMPLICIT NONE
3
4      INTEGER      :: i
5      REAL         :: x
6      COMPLEX      :: c
7      LOGICAL      :: bool
8      CHARACTER    :: a
9      CHARACTER(len=180) :: string
10
11     ! values can be assigned at creation (could do this in 77 but it was odd)
12     REAL             :: y=3.5
13     REAL, PARAMETER :: pi=3.1415927
14
15     ! change allocation size
16     INTEGER(KIND=2) :: int_short
17     INTEGER(KIND=4) :: int_normal
18     INTEGER(KIND=8) :: int_long  ! not always available
```

```
19
20      REAL(KIND=4) :: real_normal
21      REAL(KIND=8) :: real_long ! also DOUBLE PRECISION
22      REAL(KIND=16) :: real_quad ! not always available
23
24      COMPLEX(KIND=4) :: cmpx_normal
25      COMPLEX(KIND=8) :: cmpx_long
26
27  END PROGRAM
```

Arrays are where FORTRAN90 shines over C++. They are an intrinsic type and come with multiple tools for their manipulation. Python users will find this all quite familiar, numpy is a lightly wrapped version of FORTRANs native array capabilities so most of what you can do in python is available here.

Snippets/F90/arrays.f90

```
1  PROGRAM main
2      IMPLICIT NONE
3
4      ! Fixed size at compile
5      INTEGER i,j,k
6      INTEGER                  :: vector1(5)
7      INTEGER                  :: array1(5,5)
8      INTEGER, DIMENSION(5,5) :: array2
9      ! Dynamic allocation
10     INTEGER, ALLOCATABLE                    :: vector2(:)
11     INTEGER, ALLOCATABLE                    :: array3(:,:)
12     INTEGER, DIMENSION(:,:), ALLOCATABLE    :: array4
13     INTEGER, POINTER                        :: array5(:,:)
14     INTEGER, DIMENSION(:,:), POINTER        :: array6
15
16     ALLOCATE(array3(5,5))
17     IF (ALLOCATED(array3)) THEN
18         WRITE(*,*) 'Array 3 allocated'
19         WRITE(*,*) "array size", SIZE(array3,1), " X ", SIZE(array3,2)
20     END IF
21
22     ALLOCATE(array4(-5:3,0:4))  ! indicies can range from anything to anything else,
       ↪  default is 3 -> 1,2,3
23
24     ALLOCATE(array5(5,5))
25     ALLOCATE(array6(3,3))
26
27     array1 = 0 ! whole array assignment
28     array1(2,2) = 7 ! element assignment
29     vector1 = (/1,2,3,4,5/) ! all elements in one go
30     vector1 = (/(i**2,i=1,5)/) ! via a constructor rule
31     array1(1,:) = 5 ! row assignment
32     array1(3,1:5:2) = 3 ! assignment with step size -> (3,1), (3,3), (3,5) = 3
33
34     array2 = 5*array1 + 6 ! supports whole array element-wise calcuations (most
       ↪  functions too!)
35
36     array5 = 1
37     array6 = array5(2:4,2:4) ! assiging from sub-array
38
```

```
39      ! Intrinsic array operations
40
41      i = SUM(array1) ! add all elements together
42      j = PRODUCT(array1) ! multiply all elements together
43      k = MAXVAL(array1) ! find the maximum
44      k = MAXVAL(array1, MASK = (MOD(array1,2)>0)) ! find the maximum odd number
45      vector1 = MAXVAL(array1, DIM=1) ! find the maximum in each row
46      i = SIZE(SHAPE(array1))
47      ALLOCATE(vector2(i))
48      vector2 = MAXLOC(array1) ! find the location of the maximum
49
50      array5 = MATMUL(array1,array2)
51      array1 = TRANSPOSE(array1)
52
53      !  Cool syntax for conditional array modification
54      WHERE (array1/=0)
55          array1 = -array1
56      ELSEWHERE
57          array1 = 5
58      END WHERE
59
60      ! NEED TO DEALLOCATE ALLOCATABLE ARRAYS AFTER USE!!
61      DEALLOCATE(array3)
62
63  END PROGRAM
```

Some points to note are that arrays are allocatable, but this means that you need to deallocate them once you are finished to conserve memory. The pointer array is so you can pass them to functions in unallocated form. Constructors only work on 1D arrays so you need to use `RESHAPE` statement to make multi-dimension arrays from them. Not in the example above are functions like `ALL` which applies conditions to the entire array, and `SPREAD` (which is the command which allows broadcasting in python to work) copies one array to multiple dimensions of a higher dimenson array.

I/O in Fortran is handled via `READ` and `WRITE` statments. Format statments have to be specified in advance using `FORMAT` and a label which can be used in `READ` and `WRITE` statments. Files are opened and closed with labels so you can specify which file you are using.

Snippets/F90/io.f90

```
1  PROGRAM main
2      IMPLICIT NONE
3
4      INTEGER :: i
5      REAL :: x
6      INTEGER :: fileunit
7      CHARACTER(180) :: filename
8      REAL, DIMENSION(:,:), ALLOCATABLE :: array1
9
10     fileunit = 99
11
12     i = 4000
13     x = 2.45e3
14     ALLOCATE(array1(10,10))
15     array1 = 5.67
16     filename = "test.out"
17
```

```
18      90101 FORMAT('The output is: ',I5,'   ',E13.5E3)
19
20      OPEN(UNIT=fileunit, FILE=filename)
21      WRITE(UNIT=fileunit,FMT=90101) i,x
22      CLOSE(UNIT=fileunit)
23
24      ! for arrays use this:
25      OPEN(UNIT=fileunit, FILE=filename, STATUS='REPLACE', FORM='UNFORMATTED',
        ↪  ACCESS='STREAM')
26      WRITE(fileunit,*) array1
27      CLOSE(fileunit)
28
29      OPEN(UNIT=fileunit, FILE=filename, STATUS='OLD', FORM='UNFORMATTED',
        ↪  ACCESS='STREAM')
30      READ(fileunit,*) array1
31      CLOSE(fileunit)
32
33  END PROGRAM
```

The other major addiition to FORTRAN90 was the inclusions of MODULEs. This allows us ot make packages of functions and subroutines that can be used everywhere in the code. Let's create a couple of modules as an example

Snippets/F90/module1.f90

```
1  MODULE keep_count
2      IMPLICIT NONE
3
4      INTEGER, SAVE, PRIVATE :: count
5
6      PUBLIC get_count, increment
7
8      CONTAINS
9
10         FUNCTION get_count()
11             IMPLICIT NONE
12             INTEGER get_count
13             get_count = count
14         END FUNCTION get_count
15
16         SUBROUTINE increment(amount)
17             IMPLICIT NONE
18             INTEGER, INTENT(IN) :: amount
19             count = count+amount
20         END SUBROUTINE increment
21
22         SUBROUTINE reset()
23             IMPLICIT NONE
24             count = 0
25         END SUBROUTINE reset
26
27 END MODULE keep_count
```

Snippets/F90/module2.f90

```
1  MODULE constants
2      IMPLICIT NONE
3
4      SAVE
5      DOUBLE PRECISION, PARAMETER :: pi=3.1415927
6      DOUBLE PRECISION, PARAMETER :: e=2.71828
7      DOUBLE PRECISION, PARAMETER :: G=6.67430e-11
8      DOUBLE PRECISION, PARAMETER :: c=299792458
9      DOUBLE PRECISION, PARAMETER :: h=6.62607015e-34
10
11 END MODULE constants
```

Snippets/F90/usemodule.f90

```
1  PROGRAM main
2
3      USE keep_count
4      USE constants, ONLY : speed_of_light => c
5
6      IMPLICIT NONE
7
8      INTEGER :: i
9
10     CALL reset()
11
12     DO i=10,99
13         IF (MOD(i,7)==0) THEN
14             CALL increment(1)
15         END IF
16     END DO
17
18     WRITE(*,*) 'number of 2 digit numbers divisible by 7 is: ', get_count()
19     WRITE(*,*) 'speed of light is: ', speed_of_light
20
21 END PROGRAM
```

Looking at these we see that to use a module we just add it with the `USE` command at the begining and we can use the `ONLY` statment to specify which portion of the module we want to use and `=>` to rename them to avoid local name clashes. Variables declared above the `CONTAINS` statement are global to the functions and subroutines in the module and can be accessed in the main programme, unless they are specified `PRIVATE` which hides them from the main programme. All functions and subroutines must come after the contains statement. The `SAVE` statement ensures that the value of parameters are not forgoton when the module goes out of scope i.e. it could be used by routines in two different modules and we want the value to persist for all (doing this allows us to define a “global" variable in a way that is protected as it can only be accessed via the module). The other key point is that all dummy variables names in module subroutines statments should have the extra qualifier `INTENT` which can be one of `IN`, `OUT`, `INOUT`. The compiler uses these to optimise your code but they are not required. `IN` variables should be ones that the subroutine uses but does not modify, `OUT` are ones that the subroutine modifies for return without reference to their input value and `INOUT` are variables whose value is used as input and is modified for return. All other variables internal to the subroutine are private and do not require statments.

We have data structures in FORTRAN90 (which are the equivelent of structs in C). These are defined as new `TYPE`s:

Snippets/F90/types.f90

```
PROGRAM main
    IMPLICIT NONE

    TYPE :: my_type
        INTEGER :: moons
        REAL :: coord_x, coord_y
        CHARACTER(30) :: name
    END TYPE my_type

    TYPE(my_type) :: planets(8)
    REAL :: vector1(8)
    REAL :: x

    planets(1)%moons = 0
    planets(1)%name = "Mercury"
    planets(1)%coord_x = 58e6
    planets(1)%coord_y = 0e0

    planets(3) = (/1,152e6,0e0,"Earth"/)

    vector1 = planets%coord_x
    x = SUM(planets%moons)

END PROGRAM
```

We can overload operators for types we create using modules as follows:

Snippets/F90/overload.f90

```
MODULE overload
    IMPLICIT NONE

    TYPE fraction
        INTEGER :: numerator
        INTEGER :: denominator
    END TYPE fraction

    INTERFACE OPERATOR (+)
        MODULE PROCEDURE frac_add
    END INTERFACE

    CONTAINS

    SUBROUTINE reduce(frac1)
        IMPLICIT NONE
        TYPE(fraction), INTENT(INOUT) :: frac1
        INTEGER :: a,b

        a = frac1%numerator
        b = frac1%denominator

        DO WHILE (b/=0)
            a = b
            b = MOD(a,b)
        END DO

        frac1%numerator = frac1%numerator/a
```

```
29          frac1%denominator = frac1%denominator/a
30
31      END SUBROUTINE reduce
32
33      FUNCTION frac_add(frac1,frac2)
34          IMPLICIT NONE
35          TYPE(fraction), INTENT(IN) :: frac1, frac2 ! needed for overloading
36          TYPE(fraction) :: frac_add
37
38          frac_add%numerator = frac1%numerator * frac2%denominator + frac2%numerator
            ↪  * frac1%denominator
39          frac_add%denominator = frac1%denominator * frac2%denominator
40
41          CALL reduce(frac_add)
42
43      END FUNCTION frac_add
44
45  END MODULE overload
```

Modules can be compiled by the simple command we used previously

```
1 $ gfortran module1.f90 module2.f90 runmodules.f90 -o executable.exe
2 $ ./executable.exe
```

The key part is that the modules are specified in order of their dependancy. Modules deoendancies myst be strictly hierarchical, so you cannot have module1 using module2 and module2 using module1, and compiled in order from bottom to top. Compilers execute the files in order that they recieve them so putting lower modules forit ensures that the .mod files exist for later modules/programs that use them.

2. Compilation

2.1 Makefiles

Complitation of code is a bit fiddly to do from the command line every time, especially once the project becoems large with multiple source files. This has been solved by using programmes that handle the building of projects automatically. The simplest of these is to create a `Makefile`

Makefiles consist of a list of rules, usually for updating files when files they depend on change, but they can be used more generally. The rules take the form (note we need tab rather than 4 spaces for indentation):

```
target ... : dependancy ...
  commands
  ...
  ...
```

```
test :
  echo "Hello!"
```

```
test1 : test2
  echo "Hello two!"

test2 :
  echo "Hello one!"
```

This is OK but a bit verbose. Just like when writing python we are better to use variables to make the code simpler to understand. In make files variables are created with = and := signs. The values are accessed by `$(var)`. The first assignment = is implicit, which means that it doesn't expand the rhs immediately, the second := is explicit, in that it does expand it before assignment. The difference can be seen in the examples:

```
var1 = $(var2)
var2 = "hello"
echo $(var1)

var3 = "hello"
var3 := $(var3)
```

Here we don't expand var1 until we get to echo $(var1) so it doesn't matter that var2 isn't defined when we assign var1. With := this would matter as we would try to expand $(var2) when creating var1 and it wouldn't exist. Conversely for var3 if the second assignment was implicit this would create an infinite loop that can't be expanded. Here := works fine as we would expand it before assignment.

There is also ?= which assigns the variable only if it has not previously been assigned and += which will add another element to a list (which is specified just by spaces between variable names), ie:

```
var1 = one two three
var1 += four
```

We can also create pattern specific variables using:

1. % – This will match any non-empty string and can be used in any string object, but only once.
2. $@ – The filename of the target
3. $< – The first filename of the dependency
4. $^ – The filenames of all the dependencies

This allows us to set up generic rules for all files of a specific type, like object files which are always created from their c files, ie:

```
# compilers, flags and libraries
CC = gcc
CFLAGS := -g -O3 -xHost

# library packages you are useing
LIBS :=

# Directories for code objects, and librarys
OBJDIR := Objects
SRCDIR := Code
BINDIR := .

# source file(s) without suffix
CFILES = file1 file2 file3

PROGRAMS = program1 program2

#This says don't look for a file called program1 or program2
.PHONY := $(PROGRAMS)

program1 : $(CFILES:%=$(OBJDIR)/%.o) $(mainfile1:%=$(OBJDIR)/%.o)
  $(CC) $(CFLAGS) -I$(SRCDIR) $^ -o $(BINDIR)/$@ $(LIBS)

program2 : $(CFILES:%=$(OBJDIR)/%.o) $(mainfile2:%=$(OBJDIR)/%.o)
  $(CC) $(CFLAGS) -I$(SRCDIR) $^ -o $(BINDIR)/$@ $(LIBS)

# everything that ends in '.o' should be made from the same file with '.c' instead
$(OBJDIR)/%.o: $(SRCDIR)/%.c
  $(CC) $(CFLAGS) -I$(SRCDIR) -c $< -o $@

clean :
  rm $(OBJDIR)*.o
```

`Makefiles` can be a pain to create as the syntax is pretty unreadable and you generally don't create them enough to get really familiar with the process. Typically you get one that works then just cut and paste it everywhere you work then debug the inevitible issues that come up. This will allow you to automate code building and is generally ok for most small projects. However, if you move to a new system the `Makefile` will need to be updated with the correct compilers, flags and libraries etc...

2.2 CMake

The creation of `Makefiles` is sufficently difficult that there exist tools to automate the process for you. The industry standard approach is `CMake` which will generate `Makefiles` automatically from some small script files. The real benefit of using tools like `CMake` are that you can use them to automate many other build managment tasks like installing and building libraries or fetching them from git repositories, running testing frameworks and using continuous integration, and installing compiled executables.

The other main benefit is that it makes your code portable so you can jump between platforms and it will automatically discover the local compiler and libraries needed for building. It ends up working a bit like a 'package manager' for C++, like conda is for python, where it looks after all the system dependencies of your project for you and even allows you to change the generator used (`Make` by default, but there are many others) and to easily add things like parallel builds for large projects.

You will need to install `CMake` onto your system (`brew install cmake` or similar on Linux

or Windows). Once you have done this we can see a simple cmake build file for a "Hello World" program:

Snippets/CMake/HelloWorld/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.15...3.26)

project(
    HelloWorld
    VERSION 1.0
    LANGUAGES CXX
)

add_executable(hello main.cpp)
```

The cmake build file must always be called `CMakeLists.txt` and exist in the project directory (wherever you would normally initialise `git`). The first line in it must always be the command `cmake_minimum_required()` which specified the minimum verson of cmake required to build the project. This is needed so as cmake commands have evolved significantly since its first release in 2000. While all scripts are backwards compatible we will be wanting to use many features form "modern" cmake which is generally anything from 3.13 on. In the above we have specified a range from 3.15 to 3.26. Next you need to give your project a name and optionally a version and language. Finally we need to specify an executable we want to build and which cpp file contains the main program to build from.

The project can be built with:

```
$ cmake -S . -B build
$ cmake --build build
```

The first builds the project from the `CMakeLists.txt` in the source directory "." and builds the project in the directory "build" (which it will create if it does not exist). The second uses the build to create the executable in the build directory.

We can add libraries simply with the following commands:

Snippets/CMake/Library/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.15...3.26)

project(library_example
    VERSION 1.0
    LANGUAGES CXX
)

# This would work, but wouldn't build a library:
# add_executable(test main.cpp harmonic.cpp)
# Instead do this:

add_library(harmonic_lib
    STATIC
    harmonic.cpp
    harmonic.hpp
)

add_executable(test main.cpp)
```

```
19
20 target_link_libraries(test PRIVATE harmonic_lib)
```

Here we add the libraries with `add_libraries()` and then describe the dependance graph with the command `target_link_libraries()`. The format of cmake commands are all (target OPTIONS dependancies..) which mimics that of `Make`. We can list multiple (same level) dependancies on the same line but heiracical depencanceis should be input on multiple lines, e.g.

```
1 target_link_libraries(executable PRIVATE lib1)
2 target_link_libraries(lib1 PRIVATE lib2)
3 target_link_libraries(lib2 PRIVATE lib3)
```

We used two options to help us create our libaraies, `STATIC` and `PRIVATE`. the first is the type of library we are creating which can be `STATIC`, `SHARED`, and `MODULE`. The difference is `STATIC` are liked at compile time, `SHARED` are linked at runtime. The first will copy the library code add put it in your executable, the other is more efficent for libraries that multiple executables use as then we only need one copy. A simple rule for this is to use `STATIC` for libraries you built that only one file uses and `SHARED` for libraries that you build that multiple files use. `MODULE` is for shared libaraies that you don't link to, but instead load at runtime with a commnad like `dlopen()`. The options for linking are `PUBLIC`, `PRIVATE`, and `INTERFACE`. `PUBLIC` is for when things that link to the target need to be able to know about the dependant, `PRIVATE` is for when they don't and `INTERFACE` is for when we need to know header but not detail, for example when using header libraries, you need to specify them as `INTERFACE`. It is possible to make lib1 depend on lib2 AND lib2 depend on lib1 and cmake still work but if this happens in your code, you should have a long hard think about what you were trying to acheive.

The simple examples above are not really how you should structure your projects. It is much better to make then hieracical with the code in seperate folders. Despite what you will read in most tutorials on cmake, there is no standard way to do this (and everyone complains about it). You should just aim for something helps make you code more understandable by clearly reflecting its purpose and structure. The key difference is that you have to put `CMakeLists.txt` files in each folder. I have included my personal favourite approach as an example but you can choose others. I like to keep .ccp and .hpp files together which makes sense for most personal projects. If you are designing libraries to be used by others then you will want to split off the public .hpp files into an `include` directory. I have all code in a folder called `src` and libraries in subfolders off that. I place all binaries created in a folder called `bin`. I put the cmake commands for building libraries in the library sub directories and cmake commands for building executables in the src directory. The project folder then just have general cmake commands to set up the project. You should add both bin and build to git.ignore files. This format is fairly simple to understand but has the weakness (strength?) of being strictly heirarcical.

Cmake can do lots more. It can manage the use of external libraries like MPI or OpenMP using the `find_package()`:

```
find_package(OpenMP)
if(OpenMP_CXX_FOUND)
  target_link_libraries(MyTarget PUBLIC OpenMP::OpenMP_CXX)
endif()

find_package(MPI REQUIRED)
target_link_libraries(MyTarget PUBLIC MPI::MPI_CXX)
```

Eternal code can be included in you project with `fetch_content()`:

```
  FetchContent_Declare(contentname
      GIT_REPOSITORY https://github.com/google/content.git
      GIT_TAG        703bd9caab50b139428cea1aaff9974ebee5742e
  )
  FetchContent_MakeAvailable(contentname)
```

And, as we will see later, cmake can manage continuous integration tasks

2.3 Helper Tools

There are several useful helper tools you should use for your code to make it easier for others to use and understand. We will show the several of them Here

Doxygen

Doxygen is a tool that creates documentation for your code automatically from the comments inside it for a wide variety of languages. First you have to install it (`brew install doxygen` or download binaries for Linux/Windows). Then you just need to create a configiration file in the project directory using

```
$ doxygen -g doxygen.config
```

This creates a long configuration file which deinfed doxygen's behaviour. To begin with you can ignore almost all of it other than changing the following:

```
PROJECT_NAME           = "Doxygen Test"
OUTPUT_DIRECTORY       = docs
EXTRACT_ALL            = YES
EXTRACT_PRIVATE        = YES
INPUT                  = src
RECURSIVE              = YES
```

Documentation is then generated by the simple command

```
$ doxygen doxygen.config
```

Which will create documentation in the `OUTPUT_DIRECTORY` in both html and latex format, with the latter having a MAKEFILE to compile the document. This will be almost blank for the example projects as I ahve not added documentation to them. This can be fixed easily by adding some comments above (or inside) each function. This is done easiest by using the Doxygen extension to VSCode. Then lyou can add comment bolcks aboce each function by typing `/**` then hitting enter which will create blocks like:

```
 /**
 * @brief
 *
 * @param x
 * @param y
 * @return
 */
double harmonic::harm_add(double x, double y)
```

To which you can add a description of the function after `@brief` and a description of each parameter after `@param`. Play with this and see how the documentation is updated.

Doxygen is very easy to add to a project being build by `CMake` with the command `doxygen_add_docs`. We will jump straight to the general case where we define a cmake module file to automate this. We need to create a directory `cmake` and include add the following file, `Doxygen.cmake`, to it:

```
function(Doxygen input output)
    find_package(Doxygen)

    if (NOT DOXYGEN_FOUND)
        add_custom_target(doxygen COMMAND false
            COMMENT "Doxygen not found!!"
        )
        return()
    endif()

    set(DOXYGEN_GENERATE_HTML YES)
    set(DOXYGEN_HTML_OUTPUT ${PROJECT_SOURCE_DIR}/${output}/html)
    set(DOXYGEN_GENERATE_LATEX YES)
    set(DOXYGEN_LATEX_OUTPUT ${PROJECT_SOURCE_DIR}/${output}/latex)
    set(DOXYGEN_EXTRACT_ALL YES)
    set(DOXYGEN_EXTRACT_PRIVATE YES)
    set(DOXYGEN_RECURSIVE YES)
    set(DOXYGEN_QUIET YES)

    doxygen_add_docs(doxygen
        ${PROJECT_SOURCE_DIR}/${input}
        COMMENT "Generating documentation"
    )

endfunction()
```

We then update our `CMakeLists.txt` to include the new function:

```
cmake_minimum_required(VERSION 3.15...3.26)

project(folder_example
  VERSION 1.0
  LANGUAGES CXX
)

list(APPEND CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake")

set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${PROJECT_SOURCE_DIR}/bin)

include(Doxygen)
Doxygen(src docs)

add_subdirectory(src)
```

And documentation is created on the command line by specifiing the target `doxygen`

```
cmake --build build -t doxygen
```

Enforcing style

You should also add tools that help enforce formatting for your code to ensure a consistent style, and some static analysis to catch simple programming errors. `clang-format` does the former, and `clang-tidy` does the latter. These are some of the most poular tools but there are many alternatives like: `cpplint` and `cppcheck`. You can also combine these tools with other static checkers like `include-what-you-use` which removes unnessecary `include` statements.

clang-format

To use `clang_format` you just call it on the command line for whatever files you want to format:

```
lang-format -i --style=Google filename1.cpp
```

Where the `-i` means 'inplace', so re-format the file and put output to original filename, and `--style` defined the formatting style you want to use from LLVM, Google, Chromium, Mozilla, Webkit, Microsoft, GNU, custom where custom is defined by a file you create. We can automate this in `cmake` by adding a funcion like we did before. We need to include a `ClangFormat.cmake` file:

```
function(ClangFormat target directory)
    find_program(CLANG-FORMAT_PATH clang-format REQUIRED)

    # define a list of suffixes for files ot be considered
    set(EXPRESSION h hpp hh c cc cxx cpp)
    # prepend the directory to the list
    list(TRANSFORM EXPRESSION PREPEND "${directory}/*.")

    # create list of files to format recuresivly and following symboloc links
    file(GLOB_RECURSE SOURCE_FILES FOLLOW_SYMLINKS
        LIST_DIRECTORIES false ${EXPRESSION}
    )

    # Add command to run clang-format on list of files before build
    add_custom_command(TARGET ${target} PRE_BUILD COMMAND
        ${CLANG-FORMAT_PATH} -i --style=Google ${SOURCE_FILES}
    )
endfunction()
```

Then we need to add the following lines to the `CMakeLists.txt` in the `project` and `src` directorys:

```
# in project one
include(Doxygen)
include(ClangFormat)

# in src one:
add_executable(main main.cpp)
ClangFormat(main .)
```

clang-tidy

Clang tidy also works on the command line, just like clang-format, on a file by file basis:

```
$ clang-tidy filename1.cpp -options...
```

We will again automate it with `cmake` is a similar fashion. First we create a function in a `ClangTidy.cmake` file:

```
function(ClangTidy target)
    find_program(CLANG-TIDY_PATH clang-tidy REQUIRED)

    set_target_properties(${target}
        PROPERTIES CXX_CLANG_TIDY
        "${CLANG-TIDY_PATH};-checks=clang-analyzer-*,
    bugprone-*,cppcoreguidelines-*,modernize-*,readability-*"
    )
endfunction()
```

analyzer and bugprone are pretty good the next threee are a bit opinionated so can be neglected. Then we include it and add the function to the relevent `CMakeLists.txt` files (it's not recursive on libraries so you have to add it everywhere):

```
# in project one
include(Doxygen)
include(ClangFormat)
include(ClangTidy)

# in src one:
add_executable(main main.cpp)
ClangTidy(main)
ClangFormat(main .)

# in src/harmonic one:
add_library(harmonic_lib
...
)
ClangTidy(harmonic_lib)
```

You have two other options you can add, `--warnings-as-errors=*` which will cause the build to fail if there are any warnings (for those who want to be hardcore about it) and `--fix` which will make automated fixes for issues it finds (beware this is a little buggy)

You should also consider building debug versions or your project with:

```
$ cmake -S . -B build/debug -DCMAKE_BUILD_TYPE=Debug
$ cmake -S . -B build/release -DCMAKE_BUILD_TYPE=Release
$ cmake --build build/debug
$ cmake --build build/release
```

Testing

The mainenance of code over time is a very challenging task. In the begining everything is easy, particularly if you are working on a project solo. However as time passes and the code evolves and expands or new developers are added it becomes an increasingly difficult task to know if or where bugs are being introduced to the project. Some may not become apparent until long after they are created and some may only appear for specfific circumstances.

The industry response to this is to introduce unit testing and to make it automatic. This means that the project will automatically check that it is behaving correctly after each new addition to catch problems before they make it into the main branch. This is well worth implementing in your own projects, even if you are the only author and the scope is small. It is impossible to predict when, or where you will want to reuse parts of thie project in the future or what expansions you will want to do. Debugging your code will use considerably more of your time, and is much more difficult, that writing it.

`Cmake` has a built in utility for running and managing the automated testing of your code, `CTest`. We will skip over how to create manual testing frameworks by hand as you should never do this and it's pretty ugly and jump straight to using one of the most common testing frameworks, `GoogleTest` (there are many others but google will do the job fine).

To add testing to out project we just add a directory `test` and add the following `CMakeLists.txt` file to it which downloads googletest and adds it to our project:

```
include(FetchContent)

FetchContent_Declare(
    googletest
    GIT_REPOSITORY https://github.com/google/googletest.git
    GIT_TAG v1.13.0
)

# For Windows: Prevent overriding the parent project's compiler/linker settings
set(gtest\_force\_shared\_crt ON CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(googletest)
```

Now we just need to add out tests which we do by writing some code that uses our librarys:

```
#include <gtest/gtest.h>

#include "../src/harmonic/harmonic.hpp"

TEST(harmonic, basic_operations){

    EXPECT_EQ(harmonic::harm_add(20e0, 5e0), 4e0) << "Harmonic add failed basic
    ↪ operation";
    EXPECT_EQ(harmonic::harm_add(5e0, 20e0), harmonic::harm_add(20e0, 4e0)) <<
    ↪ "Harmonic add does not commute";

}
```

I strongly recoment that you call your testing files "test_library.cpp" so it is obvious what they are testing. Using the googletest framework we add tests with the command `TEST` followed by the name of the testing suite followed then the name of the test, here `harmonic` for the suite and `basic_operations` for the test itself. The tests are created using the two commands `EXPECT_*` and `ASSERT_*` with the difference being that the second abandons the rest of the test if it fails while expects continues. You would use `ASSERT_*` for something like checking the size of a vector before a loop which checks the entries because if the size is wrong the loop wouldn't make sense.

The types of test allowed can be seen here: https://google.github.io/googletest/reference/assertions.html. There are all the normal ones you would expect. Note for floating point arithmitic you have `EXPECT_FLOAT_EQ`, `EXPECT_DOUBLE_EQ` and `EXPECT_NEAR`. There is also `EXPECT_THAT` which can be used with a wide variety of matcher functions listed here: https://google.github.io/googletest/reference/matchers.h

Once we have created the test code we have to add it to the `CMakeLists.txt` in the testing folder

```
....

add_executable(test_harm test_harmonic.cpp)
target_link_libraries(test_harm
  PRIVATE harmonic_lib
  GTest::gtest_main
)

include(GoogleTest)
gtest_discover_tests(test_harm)
```

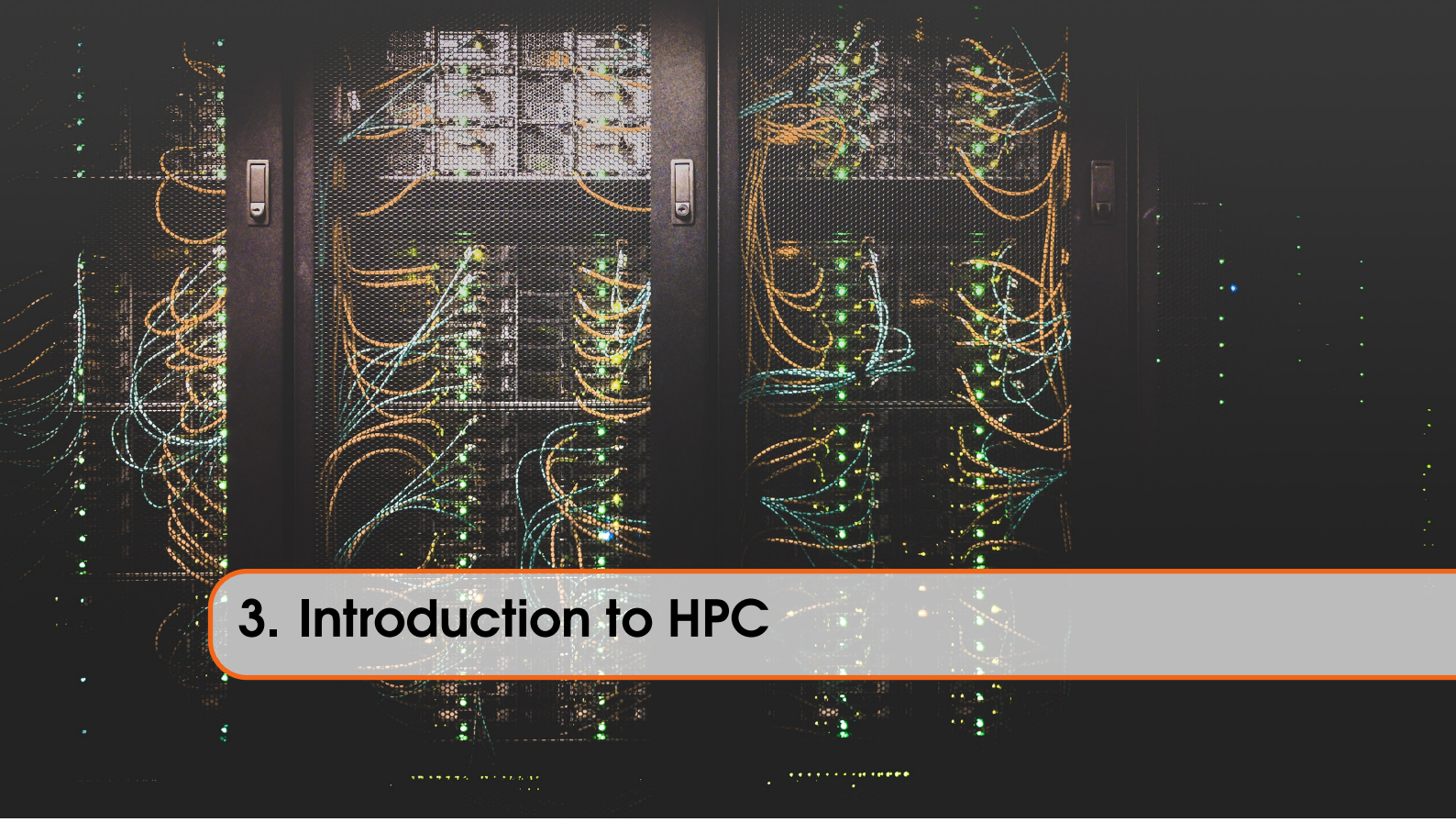
Now we just build the project normally:

```
$ cmake -S . -B build
$ cmake --build build
```

And we can run the tests with:

```
$ ctest --test-dir build
```

Which will antomatically discover all our tests and run them. Specific tests can be run with the /bash[-N] or `-R` options where the first takes an exact list and the second matches a regexp.



3. Introduction to HPC

Background

High Performance Computing for us means remote computing resources which are an order of magniture more "powerful/performant/productive?" than your local machine. They can come in the form of: servers, clusters, supercomputers or accelerators.

Supercomputing is becoming an increasinly important topic in scientific research. Its growth is being driven by the exponential increase in data across all scientific research areas and also the availabilty of ever more powerful computing resources. In 2009 this was labeled the "Fourth Paradigm" of Data-Intensive Scientific Discovery (First three are: Direct Observation, Theoretical Modeling, Numerical Computation). With the explosion of Machine Learning techniques over the last decade we are creating the Fifth Paradigm of AI4Science where scientific discuvery is driven by Machine Learning approaches. Both of the fourth and fifth paragigms are only possible on HPC resources.

The main use cases for scientific HPC are:

1. Computation of theoretical predictions
2. Simulation of physical systems
3. Data Analysis (sometimes refered to as HPDA)
4. Training ML models

Scientists who want to be effective in these areas increaingly need to master the task of writing efficent scalable parallel code. While typically scientists are HPC users who leave the complex task of creating and maintaining HPC systems to specalist HPC managers, in order to get best performance scientists need to gain an understanding of how HPC systems work. This has led to the new role of Research Software Engineer (RSE) who is a typically a scientist who has tranitioned to becoming an expert on the implementation and optimisation of scientfic codes on HPC systems.

The key descriptor in HPC is performance but what do we mean by this? The simple answer is the abililty to do lots of Floating Point Operations Per Second (FLOPS). This is not actually a very useful benchmark for us. As scientists we are much more intersted in productivity, not resource usage. We should aim for the maximum performance per unit of programming activity. The performance of our code will depend on many aspects of the architecture: memory size, latency,

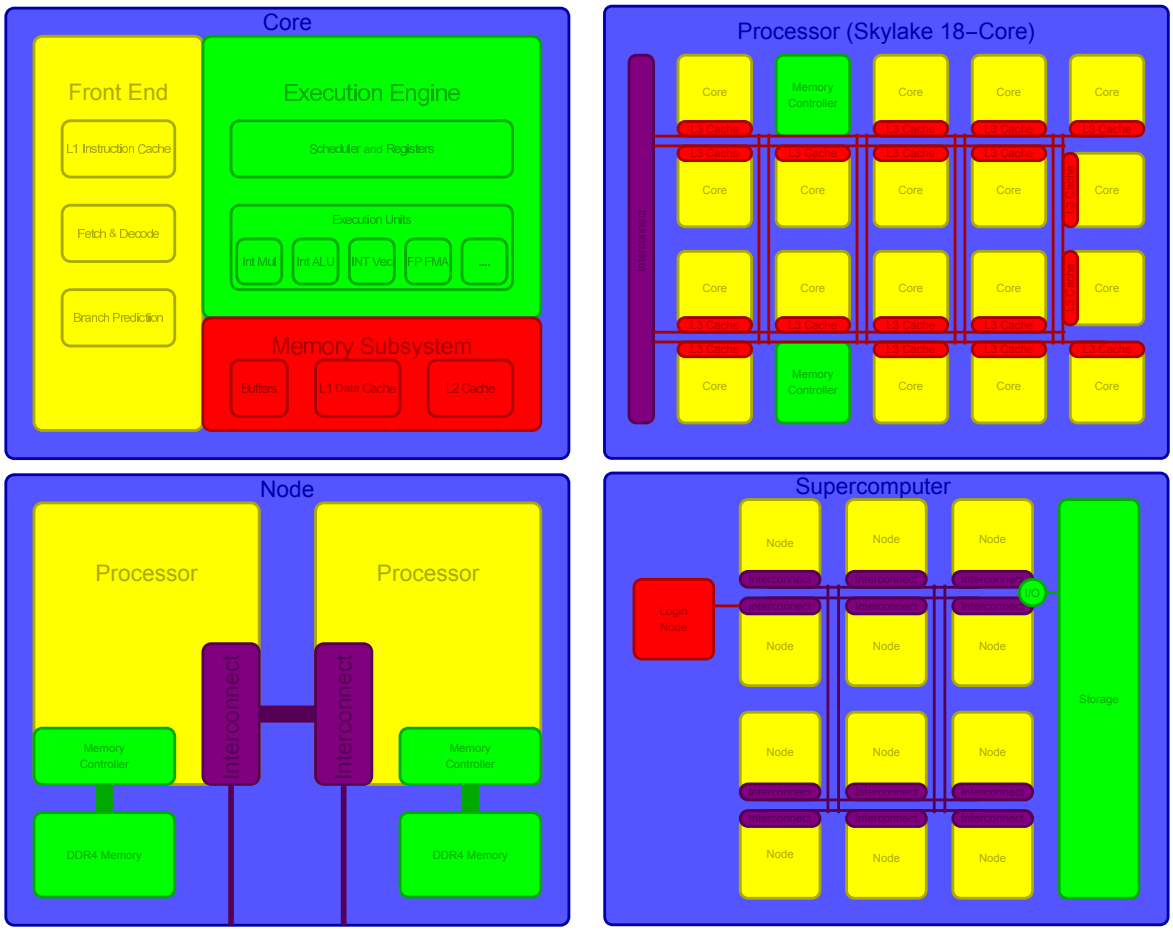


Table 3.1: Diagrams of the various layers of HPC architectures

bandwith and locality; interconnect latency and bandwidth; accelerator access patterns; ease of programming paradigms. Supercomputer rankings are caluated for dense linear algerbra problems solved using a specific library which may not be relevent to your application. We should also not chase performance at the expense of producing results. There is no point spending months optimising a code we will only use for a short period. Most gains are from finding optimal algorythims that work for the architectures available to you.

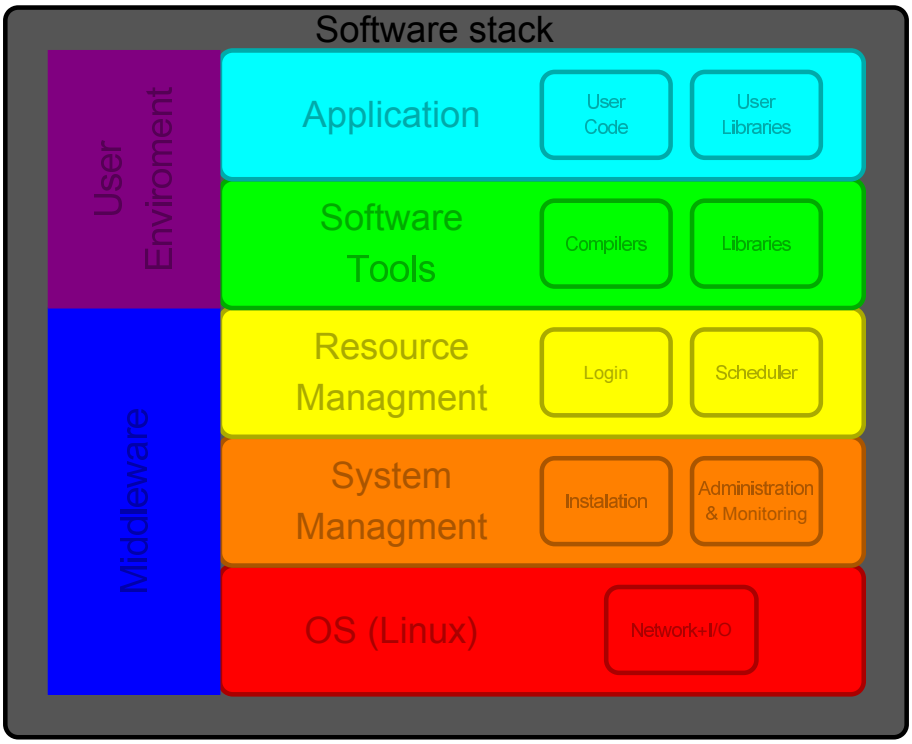
HPC dictionary

HPC resources are almost always remote to the machine you are working on. They are composed of the following pieces of hardware:

1. Nodes - which will contain processors (CPU/GPU/TPUs etc...), local memory and network cards. They are just like individual computers.
2. Interconnect - network which links the nodes together
3. Storage - Some kind of large disk space
4. Login Node - Special dedicated node that handles access to the machine.

Each node can have multiple XPU (X=C,G,T,...) devices so are described as "multiprocessor" (PU = Processing Unit). Each XPU will consist of multiple "cores" (smalles unit of computing). Generally processor=XPU=socket but sometimes processor=core (annoying!!). Please refer to the diagrams in 3.1 for a visual representation.

Supercomputers will also have the following software stack (please refer to **??**):



HPC software stack

1. OS - 99.999% of the time this will be Linux
2. System Mangment - for HPC manager to control: users accounts, installation of libraries and software, and monitoring the machine
3. Resource Managment - Software to control access to, and use of, the hardware amongst users. Key in this is the scheduler which determines the jobs that can run on what nodes and when.
4. Software tools - pre-installed software for users. Includes compilers and common scientific libraries.
5. Application Layer - Consists of user installed libraries, scripts, and compiled execuatbles.

First 3 are the HPC system middleware and you cannot control them. The last 2 are controlable by the user and define the runtime enviroment for the application.

Access

HPC systems will almost alwyas be accessed via ssh from the terminal (see setting up ssh config etc... in research computing lecture). Virtually all interactions with HPC will happen via the command line in the ssh terminal. Sometimes IDEs can manage some if this for you but you must know how to do all tasks from the command line if required. This is where you will control you enviroment and submit you jobs.

Scheduler

In order to manage a large number of users on a single system we need to have a scheduler which says who can use what resources, when and for how long. The scheduler will have queues set up which will have specific limits in terms of number of nodes requestable, maximum memory used and maximum walltime. If you violate these limits you job will either not run, or be terminated by the scheduler. You will submit jobs to queues using a terminal command to pass the scheduler a job script, which described what resources you require, how many processes and threads to create, and for how long the job should run. Try to be accurate (+ some contingency). If you submit jobs from more resources than you need they will take longer to get through the queue and will

waste resources others could use. Schedulers will also assign users a priority (how this is done is endlessly configureable - see informatoin pages for each specific system you use for more detail) which will typically decrease when you use the system and increase when you don't to fairly share the resources around the users.

Common schedulers are:

1. Slurm
2. Moab/Tourqe
3. PBS
4. Cobalt

We will look at Slurm as that is what is on Fawcett, all schedulers work in a similar way so knowing one is enough to allow you to adapt to any other. You can alternativly watch the following: Slurm Intro

A scheduler will have two parts that you will need to understand: a set of terminal commands to use to control and query your jobs and see the current status of queues; a format for job submission scripts to specify what resources you need and what executables to run.

Job scripts generally begin like this:

```
#!/bin/bash
#SBATCH --job-name=testjob                  # Job name
#SBATCH --mail-type=END,FAIL                # Mail events (NONE, BEGIN, END, FAIL,
↪  ALL)
#SBATCH --mail-user=email@something.ac.uk   # Where to send mail
#SBATCH --ntasks=8                          # Number of MPI ranks
#SBATCH --cpus-per-task=4                   # Number of cores per MPI rank
#SBATCH --nodes=2                           # Number of nodes
#SBATCH --ntasks-per-node=4                 # How many tasks on each node
#SBATCH --ntasks-per-socket=2               # How many tasks on each CPU or socket
#SBATCH --mem-per-cpu=100mb                 # Memory per core
#SBATCH --time=00:05:00                     # Time limit hrs:min:sec
#SBATCH --output=testjob.log                # Standard output and error log

module load ...
srun  {commands+exectable}

```

Common Slurm commands:

1. `sbatch` - for submission `$ sbatch myscript.sh`
2. `scancel` - obvious `$ scancel <jobid>`
3. `squeue/sinfo/sstat/sacct` - view information on queues / nodes and partitions / running job statistics / recent job statistics
4. `scontrol` - detailed reporting on jobs or nodes eg: `$ scontrol show jobid -dd <jobid>` but also controling submitted jobs eg: `$ scontrol hold <jobid>`, `$ scontrol release <jobid>`, and `$ scontrol requeue <jobid>`
5. `sprio` - view job priorities (or squeue -P)
6. `sview` - graphical display of system and jobs.
7. `salloc` - request allocation for interactive use (may not be disabled).
8. `srun` - request allocation for running an application (may not be disabled).

The most important are `sbatch` for submitting jobs to run on the system, and `squeue` to see its status.

Libraries

Mathematical libraries are controled by the command `module` and are loaded by the user either in the job script or the terminal that submitts the job (whose enviroment is used for the job at runtime).

Common libraries are:

1. BLAS/LAPACK/SCALAPAK - Linear Algebra
2. FFTW - Fast Fourier Transform
3. PETSC - ODE/PDE solver
4. Tensorflow - Machine Learning

Why should you use libraries? Simply because they will be much faster and mor accurate than anything you will write. They are highly optimised and well tested, they will use sophisticated methods to reduce scaling and the libraries will have been build to optimise use of the available computational resources. They will often include thread level parallelisim for free and, due to their ubiquity, they are highly portable.

The most common one you will use is `BLAS` which stands for Basic Linear Algebra Subprograms. It is written in FORTRAN77 (so short names and FORTRAN style array access). If you are using C/CPP then you can instead use CBLAS which has a C-style interface. BLAS contains simple vector-vector, vector-matrix and matrix-matrix operations (coresponding roughly to BLAS1, BLAS2, BLAS3). LAPACK adds more complex operations like linear solvers eignevalue decomposition and singular value decomposition. ScaLAPACK extends this to distributed memory and parallel operation.

You will need to look at each package to see what it does and how to use it. These libraries are also wrapped into different packages, eg OpenBLAS (just BLAS/LAPACK) and intel's MKL library (which includes BLAS/LAPACK/ScaLAPAK, FFTW and PDE solvers plus DNN). The MKL is optimised to run on intel chips so performance is worse on other vendors hardware (there is an undocumented env variable to optimise MKL for AMD chips which is `export MKL_DEBUG_CPU_TYPE=5` but I've never tried this) OpenBLAS is hardware agnostic so is more portable but contains fewer libraries. AMD have their own version, AOCL (BLAS/LAPACK + FFTW plus RNG and crypto). There are many other libraries you can use but are less likely to be generally available eg NAG, GSL.

Library managment is controled by the `module` package. Common commands are:

1. `module avail`
2. `module list`
3. `module load`
4. `module unload`

As we stated before, you will have to load modules either to your enviroment via the command line after logging in, via your .bashrc file, or you can keep your login enviroment clean and add load the modules in the submission script itself. The last is best practice but the former makes sure the modules definitley load before submission.