Modern Fortran in Science and Technology

Release 1.0

Jonas Lindemann and Ola Dahlblom

Oct 11, 2021

CONTENTS:

1 Introduction 1

2 The Fortran Language 3
- 2.1 Program structure . . . 3
- 2.2 Variables . . . 5
- 2.3 Operators and expressions . . . 12
- 2.4 Numeric expressions . . . 13
- 2.5 Arrays and matrices . . . 15
- 2.6 Conditional statements . . . 23
- 2.7 Repetitive statements . . . 27
- 2.8 Built-in functions . . . 29
- 2.9 Program units and subroutines . . . 34
- 2.10 Input and output . . . 53
- 2.11 String manipulation . . . 68
- 2.12 Array features . . . 70
- 2.13 Pointers . . . 71
- 2.14 System functions . . . 72
- 2.15 Object-oriented programming . . . 73

3 Fortran and Python 85
- 3.1 Python extension modules . . . 85
- 3.2 Integrating Fortran in extension modules . . . 89
- 3.3 F2PY . . . 90

4 Managing Fortran projects 99
- 4.1 Make . . . 99
- 4.2 CMake . . . 109

5 Qt Creator for Fortran **117**
5.1 Starting Qt Creator . . . 118
5.2 Qt Creator main program modes . . . 118
5.3 Installing Fortran support in Qt Creator . . . 119
5.4 Creating a Fortran project . . . 119
5.5 Building the project . . . 121
5.6 Running the project . . . 124

6 Appendix 1 - Exercises **125**
6.1 Variables and data types . . . 125
6.2 Arrays and matrices . . . 126
6.3 Conditional statements . . . 127
6.4 Repetitive statements . . . 128
6.5 Built-in functions . . . 128
6.6 Program units and subroutines . . . 128
6.7 Input and output . . . 129
6.8 String manipulation . . . 130
6.9 Object-oriented programming . . . 130

7 Appendix 2 - Quick Fortran compilation guide **131**
7.1 Compiling single source Fortran programs . . . 131
7.2 Compiling multi-source Fortran programs . . . 132
7.3 Compiling with optimisation levels . . . 132
7.4 Compiling for debugging . . . 133
7.5 Compiling with more detailed code checking . . . 133
7.6 Compiling with runtime checks . . . 134

8 Indices and tables **135**

CHAPTER ONE

INTRODUCTION

This book is an introduction in programming with Fortran 95/2003/2008 in science and technology. The book also covers methods for integrating Fortran code with other programming languages both dynamic (Python) and compiled languages (C++). An introduction in using modern development enrvironments such as QtCreator/Eclipse/Photran, for debugging and development is also given.

CHAPTER

TWO

THE FORTRAN LANGUAGE

Fortran was the first high-level language and was developed in the fifties. The languages has since the developed through a number of standards Fortran IV (1966), Fortran 77, Fortran 90, Fortran 95, Fortran 2003, Fortran 2008 and the latest Fortran 2018. The advantages with standardised languages is that the code can be run on different computer architectures without modification. In every new standard the language has been extended with more modern language elements. To be compatible with previous standards older language elements are not removed. However, language elements that are considered bad or outdated can be removed after 2 standard revisions. As an example Fortran 90 is fully backwards compatible with Fortran 77, but in Fortran 95 some older language constructs where removed.

The following sections gives a short introduction to the modern Fortran language from Fortran 90 and above. The description is centered on the most important language features. A more thorough description of the language can be found in the book Modern Fortran Explained cite{metcalf00}

2.1 Program structure

Every Fortran-program must have a main program routine. From the main routine, other subroutines that make up the program are called. The syntax for a main program is:

```
program [program-name]
[specification statements]
[executable statements]
[contains]
```

(continues on next page)

(continued from previous page)

```
[subroutines]
end [program [program-name]]
```

From the syntax it can be seen that the only identifier that must be included in main program definition is **end**.

The syntax for a subroutine and functions are defined in the same way, but the **program** identifier is replaced with **subroutine** or **function**. A proper way of organizing subroutines is to place these in separat files or place the in modules (covered in upcoming sections). Subroutines can also be placed in the main program **contains**-section, which is the preferred method if all subroutines are placed in the same source file. The code below shows a simple example of a main program with a subroutine in Fortran.

```
program sample1

        integer, parameter :: dp=selected_real_kind(15,300)
        real(kind=dp) :: x, y
        real(kind=dp) :: k(20,20)

        x = 6.0_dp
        y = 0.25_dp

        write(*,*) x
        write(*,*) y
        write(*,*) dp

    call myproc(k)

        write(*,*) k(1,1)

contains

subroutine myproc(k)

        integer, parameter :: dp=selected_real_kind(15,300)
        real(kind=dp) :: k(20,20)

        k=0.0_dp
        k(1,1) = 42.0_dp

```

(continues on next page)

(continued from previous page)

```
        return

end subroutine myproc

end program sample1
```

The program source code can contain upper and lower case letters, numbers and special characters. However, it should be noted that Fortran does not differentiate between upper and lower case letters. The program source is written starting from the first position with one statement on each line. If a row is terminated with the charachter **&**, this indicates that the statement is continued on the the next line. All text placed after the character **!** is a comment and wont affect the function of the program. Even if the comments don't have any function in the program they are important for source code readability. This is especially important for future modification of the program. In addition to the source code form described above there is also the possibility of writing code in fixed form, as in Fortran 77 and earlier versions. In previous version of the Fortran standard this was the only source code form available.

2.2 Variables

Variables are named references to data stored in memory. When specifying variables in Fortran, the data type of the data must also be specified. This means that Fortran is a strongly typed language compared to Python where data types of variables can change during code execution. Python is a dynamically typed language.

By default Fortran assumes that variables starting with letters I to N are assumed to be integers all other variables are assumed to be floating point variables (real). This is also called the implict type rule and is considered bad practice in Fortran, but is default because many Fortran programs are still relying on this rule. Modern Fortran application should always disable the implict type rule by adding the following statement in each code unit:

```
implicit none
```

This is also described in more detail in the following sections.

2.2.1 Naming of variables

Variables in modern Fortran consists of 1 to 31 alphanumeric characters (letters except and , underscore and numbers). The first character of a variable name must be a letter. Allowable variable names can be:

```
a
a_thing
x1
mass
q123
time_of_flight
```

Variable names can consist of both upper case and lower case letters. It should be noted that **a** and **A** references the same variable. Invalid variables names can be:

```
1a      ! First character not a letter
a thing ! Contains a space character
_       ! Contains a non-alphanumeric character
```

2.2.2 Data types and declarations

There are 5 built-in data types in Fortran:

- integer, Integers.
- real, Floating point numbers.
- complex, Complex numbers.
- logical, Boolean values.
- character, Strings and characters.

The syntax for a variable declaration is:

```
type [[,attribute]... ::] entity-list
```

type defines the variable type and can beftype{integer}, ftype{real}, ftype{complex}, ftype{logical}, ftype{character}, or ftype{type}(type-name). ftype{attribute} defines additional special attributes or how the variable is to be used. The following examples shows some typical Fortran variable declarations.

```
integer :: a      ! Scalar integer variable
real    :: b      ! Scalar floating point variable
logical :: flag   ! boolean variable

real :: D(10)     ! Floating point array consisting of 10␣
↪elements
real :: K(20,20) ! Floating point array of 20x20 elements

integer, dimension(10) :: C      ! Integer array of 10␣
↪elements

character :: ch                  ! Character
character, dimension(60) :: chv ! Array of characters
character(len=80) :: line        ! Character string
character(len=80) :: lines(60)   ! Array of strings
```

Constants are declared by specifying an additional attribute, **parameter**. A declared constant can be used in following variable declarations. An example of use is shown in the following example.

```
integer, parameter :: A = 5 ! Integer constant
real :: C(A)                ! Floating point array where
                            ! the number of elements is
                            ! specified by A
```

The precision and size of the variable type can be specified by adding a parenthesis directly after the type declaration. The variables **A** and **B** in the following example are declared as floating point scalars with different precisions. The number in the parenthesis denotes for many architectures, how many bytes a floating point variable is represented with. This is however not standardised and should not be relied upon.

```
real(8) :: A
real(4) :: B
integer(4) :: I
```

To be able to choose the correct precision for a floating point variable, Fortran has a built in function **selected_real_kind** that returns the value to be used in the declaration with a given precision. This is illustrated in the following example.

```
integer, parameter :: dp = selected_real_kind(15,300)
real(kind=dp) :: X,Y
```

In this example the floating point variable should have at least 15 significant decimals and could represent numbers from 10:math:^*{-300}* to 10:math:^*{300}*. For several common architectures **selected_real_kind** will return the value 8. The advantage of using the above approach is that the precision of the floating point values can be specified in a architectural independent way. The precision constant can also be used when specifying numbers in variable assignments as the following example illustrate.

```
X = 6.0_dp
```

The importance of specifying the precision for assigning scalar values to variables is illustrated in the following example.

```
program constants

    implicit none

    integer, parameter :: dp = selected_real_kind(15,300)

    real(dp) :: pi1, pi2
    pi1 = 3.141592653589793
    pi2 = 3.141592653589793_dp

    write(*,*) 'pi1 = ', pi1
    write(*,*) 'pi2 = ', pi2

    stop

end program constants
```

The program gives the following results:

```
pi1 = 3.14159274101257
pi2 = 3.14159265358979
```

The scalar number assigned to the variable **pi1** is chosen by the compiler to be represented by the least number of bytes floating point precision, in this case ftype{real(4)}, which is shown in the output from the above program.

Variable declarations in Fortran always precedes the executable statements in the main program or in a subroutine. Declarations can also be placed directly after the **module** identifier in modules.

2.2.3 Implicit type rule

Variable do not have to be declared in Fortran. The default is that variables starting I, J,. . . , N are defined as ftype{integer} and variables starting with A, B,. . . ,H or O, P,. . . , Z are defined as ftype{real}. This kind of implicit variable declaration is not recommended as it can lead to programming errors when variables are misspelled. To avoid implicit variable declarations the following declaration can be placed first in a program or module:

```
implicit none
```

This statement forces the compiler to make sure that all variables are declared. If a variable is not declared the compilation is stopped with an error message. This is default for many other strongly typed languages such as, C, C++ and Java.

2.2.4 Assignment of variables

The syntax for scalar variable assignment is,

```
variable = expr
```

where **variable** denotes the variable to be assigned and **expr** the expression to be assigned. The following example assign the **a** variable the value 5.0 with the precision defined in the constant **ap**.

```
a = 5.0_dp
```

Assignment of boolean variables are done in the same way using the keywords, **.false.** and **.true.** indicating a true or false value. A boolean expression can also be used int the assignment. In the following example the variable, **flag**, is assigned the value **.false.**.

```
flag =.false.
```

Assignment of strings are illustrated in the following example.

```
character(40) :: first_name
character(40) :: last_name
character(20) :: company_name1
character(20) :: company_name2
```

(continues on next page)

(continued from previous page)

```
...

first_name = 'Jan'
last_name = "Johansson"
company_name1 = "McDonald's"
company_name2 = 'McDonald''s'
```

The first variable, **first_name**, is assigned the text ‘’Jan”, remaining characters in the string will be padded with spaces. A string is assigned using citation marks, ‘’ or apostrophes, ‘. This can be of help when apostrophes or citation marks is used in strings as shown in the assignemnt of the variables, **company_name1** och **company_name2**.

2.2.5 Defined and undefined variables

A variable in Fortran that has been assigned a value is considered to be defined and can be used safely. Variables that are not assigned values are said to be undefined and should not be used.

A program containing undefined variables will not fail compilation. Memory for undefined variables will be reserved and can be referenced. However, values of undefined variables are not automatically set to zero and references memory locations with unknown values. Often these variables will return garbage or random values. As rule always initialise variables to a default values or make sure they are assigned a value from another variable reference.

The following example shows an example of referencing defined and undefined variables.

```
program undef1

    implicit none

    integer, parameter :: dp = selected_real_kind(15,300)

    real(dp) :: a, b
    character(40) :: s1, s2

    a = 42.0_dp ! --- Defined
    s1 = 'My defined string'
```

(continues on next page)

(continued from previous page)

```fortran

    print*, a
    print*, b
    print*, s1
    print*, s2

end program undef1
```

This will print the following:

```
  42.000000000000000
  8.2890460584580950E-317
My defined string
```

The reason for the 8.28…E-317 value is that the variable reference **b** points to its given memory location, but no value has been assigned to this location and will contain whatever was in memory when the program was executed. Fortran will interpret the values at this location as a floating point value and present it as such. This illustrates why it is a good idea to initialise variables to 0.0_dp or make sure they will be assigned values from other variable references.

2.2.6 Derived datatypes

In certain cases it can be beneficial to create your own data types to handle the behavior of your program. To create new data types in Fortran you can create derived data types using the **type**-statement. Using this statement a new data type can be created by grouping existing Fortran data types into a new type. In the following example a data type of a particle is defined:

```fortran
type particle
    real(dp) :: x
    real(dp) :: y
    real(dp) :: z
    real(dp) :: m
end type particle
```

The new data type can now be used like any other data type in Fortran. To create a variable reference to a derived data type the **type** keyword precedes the name of the data type in the declaration. As in the following example:

```
type(particle) :: p0
```

The members of the derived data types are accessed using the %-operator. In the following example the members of the variable **p0** are assigned values:

```
p0 % x = 0.0_dp
p0 % y = 0.0_dp
p0 % z = 0.0_dp
p0 % m = 1.0_dp
```

Derived data types can contain multiple Fortran data types:

```
type particle
    real(dp)  :: x
    real(dp)  :: y
    real(dp)  :: z
    real(dp)  :: m
    logical   :: active
    integer   :: id
    character :: name(8)
end type particle
```

2.3 Operators and expressions

The following arithmetic operators are defined in Fortran:

Operator	Description
**	power to
*	multiplication
/	division
+	addition
-	subtraction

Parenthesis are used to specify the order of different operators. If no parenthesis are given in an expression operators are evaluated in the following order:

1. Operations with **
2. Operations with * or /
3. Operations with + or –

The following code illustrates operator precedence.

```
c = a+b/2 ! is equivalent to :math:`a+(b/2)`
c = (a+b)/2 ! in this case :math:`(a+b)` is evaluated and␣
↪then :math:`/` 2
```

Relational operators:

Operator	Description
< or .lt.	less than
<= or .le.	less than or equal to
> or .gt.	greater than
>= or .ge.	greater than or equal to
== or .eq.	equal to
/= or .ne.	not equal to

Logical operators:

Operator	Description
.and.	and
.or.	or
.not.	not

2.4 Numeric expressions

Numeric expressions in Fortran consist of operands of the built-in data types **integer**, fkeyw{real} or **complex**.

If the operands only consists of integer it is important to note that integer divisions are rounded towards 0. The following example illustrates this:

```
program expr1

    print*, 6/3
    print*, 8/3   ! 2.6666... rounded down to 2
    print*, -8/3  ! -2.666... rounded up to 2

end program expr1
```

Running the program will result in the following output:

```
2
2
-2
```

There are also some other things to be careful with when working with integer expressions. Consider the following example:

```fortran
program expr2

    print*, 2**3
    print*, 2**(-3)

end program expr2
```

When run will give the following output:

```
8
0
```

The reason for the 0 in the second expression is that **2**(-3)** is the same as 1/2**3, which will be truncated to 0 as an integer expression.

When mixing data types in expressions, weaker data types will be converted to the stronger type. The result of the expression will be of the stronger type. **real** is stronger than **integer**. Consider the following code:

```fortran
real(dp) :: a
integer :: i
real(dp) :: b

b = a * i
```

Here, the i variable reference will be converted to **real(dp)** when the expression is evaluated.

2.5 Arrays and matrices

In scientific and technical applications matrices and arrays are important concepts. As Fortran is a language mainly for technical computing, arrays and matrices play a vital role in the language.

Declaring arrays and matrices can be done in two ways. In the first method the dimensions are specified using the special attribute, **dimension**, after the data type declaration. The second method, the dimensions are specified by adding the dimensions directly after the variable name. The following code illustrate these methods of declaring arrays.

```
integer, parameter :: dp = selected_real_kind(15,300)
real(dp), dimension(20,20) :: K ! Matrix 20x20 elements
real(dp) :: fe(6) ! Array with 6 elements
```

The default starting index in arrays is 1. It is however possible to define custom indices in the declaration, as the following example shows.

```
real(ap) :: idx(-3:3)
```

This declares an array, **idx** with the indices [-3, -2, -1, 0, 1, 2, 3], which contains 7 elements.

2.5.1 Array assignment

Arrays are assigned values either by explicit indices or the entire array in a single statement. The following code assigned the variable, **K**, the value 5.0 at position row 5 and column 6.

```
K(5,6) = 5.0_dp
```

If the assignment had been written as

```
K = 5.0_dp
```

the entire array, **K**, would have been assigned the value 5.0. This is an efficient way of assigning entire arrays initial values.

Explicit values can be assigned to arrays in a single statement using the following assignment.

```
real(dp) :: v(5) ! Array with 5 elements
v = (/ 1.0_dp, 2.0_dp, 3.0_dp, 4.0_dp, 5.0_dp /)
```

This is equivalent to an assignment using the following statements.

```
v(1) = 1.0_dp
v(2) = 2.0_dp
v(3) = 3.0_dp
v(4) = 4.0_dp
v(5) = 5.0_dp
```

The number of elements in the list must be the same as the number of elements in the array variable.

Assignments to specific parts of arrays can be achieved by slicing. The following example illustrates this concept.

```
program slicing

    implicit none
    real :: A(4,4)
    real :: B(4)
    real :: C(4)

    B = A(2,:) ! Assigns B the values of row 2 in A
    C = A(:,1) ! Assigns C the values of column 1 in A

    stop

end program slicing
```

Using slicing rows or columns can be assigned in single statements as shown in the following code:

```
! Assign row 5 in matrix K the values 1, 2, 3, 4, 5

K(5,:) = (/ 1.0_dp, 2.0_dp, 3.0_dp, 4.0_dp, 5.0_dp /)

! Assign the array v the values 5, 4, 3, 2, 1

v = (/ 5.0_dp, 4.0_dp, 3.0_dp, 2.0_dp, 1.0_dp /)
```

2.5.2 Array expressions

In modern Fortran expressions can also be used on arrays. The operators will then apply element wise to the in the expression. For this to work the arrays must be of the same size. Array expression can also contain scalar values. These will be broadcast to the array elements. Consider the following arrays:

```fortran
real :: a(10,20), b(10,20), c(10,20)
real :: u(5), v(5)
```

The following expression will divide all values in **a** with the values in **b**.

```fortran
c = a/b
```

This is equivalent to:

```fortran
do i=1,10
    do j=1,20
        c(i,j) = a(i,j)/b(i,j)
    end do
end do
```

It is important to make sure the resulting variable on the left side of the assignment has the same size as the resulting array expression.

The following expression adds a scalar value to all elements in the array **v**:

```fortran
u = v + 1.0
```

This is equivalent to:

```fortran
do i=1,5
    u(i) = v(i) + 1.0
end do
```

It is also possible to use slicing to extract a ‘’slice” that can be used in an array expression:

```fortran
u = 5.0/v + a(1:5,5)
```

Which is equivalent to:

```
do i=1,5
    u(i) = 5.0/v(i) + a(i,5)
end do
```

2.5.3 Array storage

Memory allocation by the operating system is done in linear blocks of bytes. The operating system does not have the concept of multidimensional arrays. This is a concept introduced by the programming language, in this case Fortran, to make it easier for us to implement algorithms and access values stored in memory.

There are 2 conventions of storing 2D arrays in memory, by column and by row. Fortran as a convention stores arrays by column and C by row. The following figure~cite{array_storage} illustrates this concept:

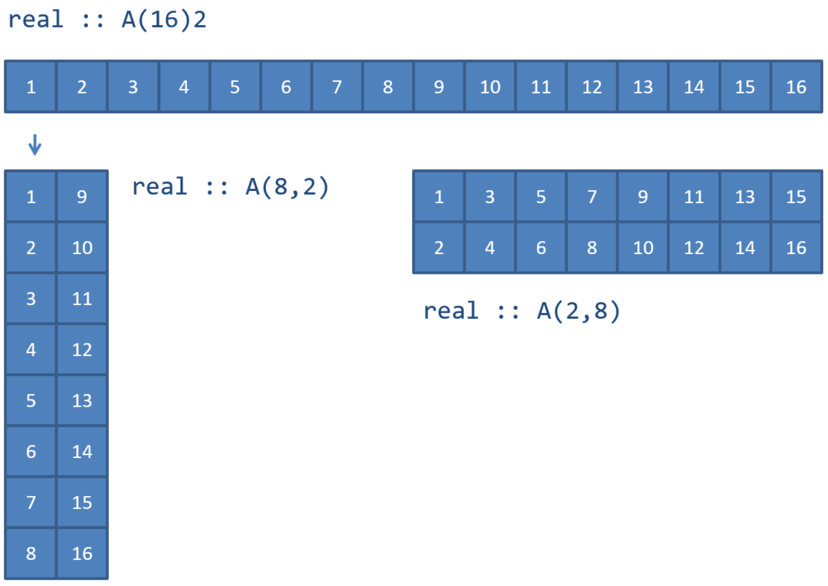


Fig. 2.1: Arrays in memory.

The storage of arrays in memory is especially important when calling libraries implemented in other languages, which usually stores arrays by row. The Python library NumPy by default stores all arrays using the C convention. Calling a Fortran subroutine with a pointer to these arrays will probably result in undefined behavior. NumPy supports column ordered arrays by supplying the array-constructor with the option pkeyw{order=F}.

2.5.4 Allocatable arrays

In Fortran 77 and earlier versions of the standard it was not possible to dynamically allocate memory during program execution. This capability is now available in Fortran 90 and later versions. To declare an array as dynamically allocatable, the attribute **allocatable** must be added to the array declaration. The dimensions are also replaced with a colon, :, indicating the number of dimensions in the declared variable. A typical allocatable array declaration is shown in the following example.

```
real, dimension(:,:), allocatable :: K
```

In this example the two-dimensional array, K, is defined as allocatable. To indicate that the array is two-dimensional is done by specifying **dimension(:,:)** in the variable attribute. To declare a one-dimensional array the code becomes:

```
real, dimension(:), allocatable :: f
```

Variables with the **allocatable** attribute can't be used until memory is allocated. Memory allocation is done using the **allocate** method. To allocate the variables, **K,f**, in the previous examples the following code is used.

```
allocate(K(20,20))
allocate(f(20))
```

When the allocated memory is no longer needed it can be deallocated using the command, **deallocate**, as the following code illustrates.

```
deallocate(K)
deallocate(f)
```

An important issue when using dynamically allocatable variable is to make sure the application does not ''leak''. ''Leaking'' is term used by applications that allocate memory during the execution and never deallocate used memory. If unchecked the application will use more and more resources and will eventually make the operating system start swapping and perhaps become also become unstable. A rule of thumb is that an **allocate** statement should always have corresponding **deallocate**.

2.5.5 Array subobjects

In many situations you want to work on smaller parts or slices of existing arrays. In Modern Fortran this can be accomplished by using the subobject feature. We will illustrate the concept of subobjects by an example. Consider the following declarations of an 2D- and 1D array:

```
use utils

real(dp) :: A(10,10)
real(dp) :: v(10)
```

To make sure we don't have any junk values in the arrays, we initialise these with random values between 1 and 0.

```
call init_rand()
call set_print_format(10, 4, 'F')

call rand_mat(A, 0.0_dp, 1.0_dp)
call rand_vec(v, 0.0_dp, 1.0_dp)
```

We print the arrays, so that we can see the structure:

```
call print_matrix(A, 'a')
print*, loc(A)

call print_vector(v, 'c')
print*, loc(v)
```

The print-statements are added to show the actual memory address, so we can see what happens when we create subobjects.

```
Matrix a ( 10 x  10) DP
------------------------------------------------------------
↪---------- ...
    0.7112    0.6538    0.9200    0.7415    0.7460    0.
↪4596    0.9778 ... 0.4111    0.0399    0.7354    0.9505  ␣
↪ 0.4428    0.7871    0.7390 ...
    0.9814    0.4057    0.4292    0.3406    0.6673    0.
↪2120    0.9523 ...
    0.1369    0.5396    0.4405    0.7577    0.9942    0.
↪7274    0.2653 ...
```

(continues on next page)

(continued from previous page)

```
    0.7179     0.9883     0.5549     0.6374     0.1337     0.
↪9464     0.7786 ...
    0.2915     0.8634     0.3962     0.8088     0.5708     0.
↪9827     0.8841 ...
    0.8294     0.8191     0.5452     0.2079     0.1126     0.
↪8199     0.2020 ...
    0.0706     0.6945     0.7744     0.3474     0.2566     0.
↪6155     0.0921 ...
    0.3754     0.7855     0.9828     0.3965     0.9551     0.
↪2119     0.2718 ...
    0.4026     0.5287     0.6198     0.9967     0.5866     0.
↪6598     0.4024 ...
------------------------------------------------------------
↪---------- ...
    140722079206560

Vector c ( 10) DP
------------------------------------------------------------
↪---------- ...
    0.9092     0.4976     0.6361     0.0355     0.7865     0.
↪2197     0.8824 ...
------------------------------------------------------------
↪---------- ...
    140722079206480
```

In the first example we are extracing the first column of the **A** array:

```
call print_vector(A(:,1))
print*, loc(A(:,1))
```

This will give us the following output:

```
Vector ( 10)
------------------------------------------------------------
↪---------- ...
    0.7112     0.4111     0.9814     0.1369     0.7179     0.
↪2915     0.8294 ...
------------------------------------------------------------
↪---------- ...
    140722079206560
```

From the output we can see that i seems to be the first column of the **A** array. We can also see that the memory location of the subobject is equivalent to the memory location of the **A**-array. This is due to the fact that memory for the **A**-array is stored column wise access a slice in this direction can be done directly without copying.

If we instead extract the first row of the **A**-array:

```
call print_vector(A(1,:))
print*, loc(A(1,:))
```

We get the following output:

```
Vector ( 10)
------------------------------------------------------------
↪----------- ...
    0.7112    0.6538    0.9200    0.7415    0.7460    0.
↪4596    0.9778 ...
------------------------------------------------------------
↪----------- ...
            10278912
```

Here we can see that the memory location is in a completely different location. This is due to the fact that the compiler needs to make a temporary copy to create this slice. This is important to think about especially if working with large array slices that are passed in subroutine calls. This can lead to crashes as these slices often are allocated on the stack.

Below illustrates some other examples of using array sub objects:

```
call print_matrix(A(1:2,1:2))

call print_matrix(A(1:6:2,1:6:2))
print*, loc(A(1:6:2,1:6:2))

call print_matrix(A(:,1:2))
print*, loc(A(:,1:2))
```

This gives the following output:

```
Matrix (  2 x   2) DP
--------------------
    0.1233    0.2606
```

(continues on next page)

(continued from previous page)

```
    0.8357    0.4479
--------------------
          27678720

Matrix (  3 x   3) DP
------------------------------
    0.1233    0.7812    0.1668
    0.4284    0.3687    0.6542
    0.2531    0.3314    0.3557
------------------------------
          27689200

Matrix ( 10 x   2) DP
--------------------
    0.1233    0.2606
    0.8357    0.4479
    0.4284    0.2254
    0.3541    0.3400
    0.2531    0.6490
    0.6420    0.5396
    0.5006    0.5849
    0.7185    0.3138
    0.8209    0.6203
    0.0232    0.9131
--------------------
    140728814257696
```

2.6 Conditional statements

One of the more important concepts in a programming language is the ability to execute code depending on certain conditions are fulfilled. Modern Fortran support this through the **if**- and the **select**-statement, which are described in this section.

The simplest form of if-statements in Fortran have the following syntax

```
if (scalar-logical-expr) then
    block
end if
```

where **scalar-logical-expr** is a boolean expression, that has to be evaluated as true, (**.true.**), for the statements in, **block**, to be executed. An extended version of the if-statement adds a **else**-block with the following syntax

```
if (scalar-logical-expr) then
    block1
else
    block2
end if
```

In this form the **block1** will be executed if **scalar-logical-expr** is evaluated as true, otherwise **block2** will be executed. A third form of if-statement contains one or more **else if**-statements with the following syntax:

```
if (scalar-logical-expr1) then
    block1
else if (scalar-logical-expr2) then
    block2
else
    block3
end if
```

In this form the **scalar-logical-expr1** is evaluated first. If this expression is true **block1** is executed, otherwise if **scalar-logical-expr2** evaluates as true **block2** is executed. If no other expressions are evaluated to true, **block3** is executed. An if-statement can contain several **else if**-blocks. The use of if-statements is illustrated in the following example:

```
program logic

        implicit none

        integer :: x
        logical :: flag

        ! Read an integer from standard input

        write(*,*) 'Enter an integer value.'
        read(*,*) x

        ! Correct value to the interval 0-1000

```

(continues on next page)

(continued from previous page)

```fortran
        flag = .FALSE.

        if (x>1000) then
                x = 1000
                flag = .TRUE.
        end if

        if (x<0) then
                x = 0
                flag = .TRUE.
        end if

        ! If flag is .TRUE. the input value
        ! has been corrected.

        if (flag) then
                write(*,'(a,I4)') 'Corrected value = ', x
        else
                write(*,'(a,I4)') 'Value = ', x
        end if

        stop

end program logic
```

Another conditional construct is the case-statement.

```fortran
select case (expression)
    case selector
        block
end select
```

In this statement the expression, **expression** is evaluated and the **case**-block with the corresponding **selector** is executed. To handle the case when no **case**-block corresponds to the **expr**, a **case**-block with the **default** keyword can be added. The syntax then becomes:

```fortran
select case (expr)
    case selector
        block
```

(continues on next page)

(continued from previous page)

```
    case default
        block
end select
```

Example of case-statement use is shown in the following example:

```
select case (display_mode)
    case (displacements)
        ...
    case (geometry)
        ...
end select
```

To handle the case when **display_mode** does not correspone to any of the alternatives the above code is modified to the following code.

```
select case (display_mode) case (displacements)
        ...
    case (geometry)
        ...
    case default
        ...
end select
```

The following program example illustrate how case-statements can be used.

```
program case_sample

    integer :: value

    write(*,*) 'Enter a value'
    read(*,*) value

    select case (value)
        case (:0)
            write(*,*) 'Greater than one.'
        case (1)
            write(*,*) 'Number one!'
        case (2:9)
            write(*,*) 'Between 2 and 9.'
        case (10)
```

(continues on next page)

(continued from previous page)

```
            write(*,*) 'Number 10!'
        case (11:41)
            write(*,*) 'Less than 42 but greater than 10.'
        case (42)
            write(*,*) 'Meaning of life or perhaps 6*7.'
        case (43:)
            write(*,*) 'Greater than 42.'
        case default
            write(*,*) 'This should never happen!'
    end select

    stop

end program case_sample
```

2.7 Repetitive statements

The most common repetitive statement in Fortran is the **do**-statement. The syntax is:

```
do variable = expr1, expr2 [,expr3]
    block
end do
```

variable is the control-variable of the loop. **expr1** is the starting value, **expr2** is the end value and **expr3** is the step interval. If the step interval is not given it is assumed to be 1. There are two ways of controlling the execution flow in a **do**-statement. The **exit** command terminates the loop and program execution is continued after the **do**-statement. The **cycle** command terminates the execution of the current block and continues execution with the next value of the control variable. The example below illustrates the use of a **do**-statement.

```
program loop_sample

    implicit none

    integer :: i

```

(continues on next page)

(continued from previous page)

```
    do i=1,20
        if (i>10) then
            write(*,*) 'Terminates do-statement.'
            exit
        else if (i<5) then
            write(*,*) 'Cycling to next value.'
            cycle
        end if
        write(*,*) i
    end do

    stop

end program loop_sample
```

The above program gives the following output:

```
Cycling to next value.
Cycling to next value.
Cycling to next value.
Cycling to next value.
5
6
7
8
9
10
Terminates do-statement.
```

Another repetitive statement available is the **do while**-statement. With this statement, the code block can execute until a certain condition is fulfilled. The syntax is:

```
do while (scalar-logical-expr)
    block
end do
```

The following code shows a simple **do while**-statement printing the function $f(x) = sin(x)$.

```
x = 0.0
do while x<1.05
    f = sin(x)
    x = x + 0.1
    write(*,*) x, f
end do
```

There are other repetitive statements such as **forall** and **where** covered int the array features sections.

2.8 Built-in functions

Fortran has a number of built-in functions covering a number of different areas. The following tables list a selection of these. For a more thorough description of the built-in function please see, Metcalf and Reid cite{metcalf00}.

2.8.1 Mathematical functions

Function	Description
acos(x)	Returns $\arccos(x)$
asin(x)	Returns $\arcsin(x)$
atan(x)	Returns $\arctan(x)$
atan2(y,x)	Returns $\arctan(\frac{y}{x})$ from $-\pi$ till math:*-pi*
cos(x)	Returns $\cos(x)$
cosh(x)	Returns $\cosh(x)$
exp(x)	Returns e^x
log(x)	Returns $\ln(x)$
log10(x)	Returns $\lg(x)$
sin(x)	Returns $\sin(x)$
sinh(x)	Returns $\sinh(x)$
sqrt(x)	Returns $\sqrt{x}$
tan(x)	Returns $\tan(x)$
tanh(x)	Returns $\tanh(x)$

2.8.2 Miscellaneous conversion functions

Function	Description
abs(a)	Returns absolute value of a
aint(a)	Truncates a floating point value
int(a)	Converts a floating point value to an integer
nint(a)	Rounds a floating point value to the nearest integer
real(a)	Converts an integer to a floating point value
max(a1,a2[,a3,...])	Returns the maximum value of two or more values
min(a1,a2[,a3,...])	Returns the minimum value of two or more values

2.8.3 Vector and matrix functions

Function	Description
dot_product(u, v)	Returns the scalar product of $u \cdot v$
matmul(A, B)	Matrix multiplication. The result must have the same for as $\mathbf{AB}$
transpose(C)	Returns the transpose $\mathbf{C}^T$. Elementet C_{ij}^T motsvarar C_{ji}

2.8.4 Array functions

Function	Description
all(mask)	Returns true of all elements in the logical array **mask** are true. For example all(A>0)! returns true if all elements in **A** are greater than 0.
any(mask)	Returns true if any of the elements in **mask** are true.
count(mask)	Returns the number of elements in **mask** that are true.
maxval(array)	Returns the maximum value of the elements in the array **array**.
minval(array)	Returns the minimum value of the elements in the array **array**.
product(array)	Returns the product of the elements in the array **array**.
sum(array)	Returns the sum of elements in the array **array**.

Most built-in functions and operators in Fortran support arrays. The following example shows how functions and operators support operations on arrays.

```fortran
real, dimension(20,20) :: A, B, C

C = A/B ! Division :math:`C_{ij}=A_{ij}/B_{ij}`

C = sqrt(A) ! Square root :math:`C_{ij}=\sqrt{A_{ij}}`
```

The following example shows how a stiffness matrix for a bar element easily can be created using these functions and operators. The Matrix $\mathbf{K}_e$ is defined as follows

$$\mathbf{K}_e = (\mathbf{G}^T\mathbf{K}_{el})\mathbf{G}$$

The G^T is returned by using the Fortran function **transpose** and the matrix multiplications are performed with **matmul**. The matrices $\mathbf{K}_{el}$ and $\mathbf{G}$ are defined as

$$\mathbf{K}_{el} = \frac{EA}{L}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

and

$$\mathbf{G} = \begin{bmatrix} n_x & n_y & n_z & 0 & 0 & 0 \\ 0 & 0 & 0 & n_x & n_y & n_z \end{bmatrix}$$

Length and directional cosines are defined as

$$L = \sqrt{(x_2-x_1)^2 + (y_2-y_1)^2 + (z_2-z_1)^2}$$

$$n_x = \frac{x_2-x_1}{L}, \quad n_y = \frac{y_2-y_1}{L}, \quad n_z = \frac{z_2-z_1}{L}$$

In the example the input parameters are assigned the following values:

$$x_1 = 0, \; y_1 = 0, \; z_1 = 0$$
$$x_2 = 1, \; y_2 = 1, \; z_2 = 1$$
$$E = 1, A = 1$$

```fortran
program function_sample

        implicit none
```

(continues on next page)

(continued from previous page)

```

        integer, parameter :: dp = selected_real_kind(15,
↪300)

        integer :: i, j

        real(dp) :: x1, x2, y1, y2, z1, z2
        real(dp) :: nx, ny, nz
        real(dp) :: L, E, A
        real(dp) :: Kel(2,2)
        real(dp) :: Ke(6,6)
        real(dp) :: G(2,6)

        ! Initiate scalar values

        E = 1.0_dp
        A = 1.0_dp
        x1 = 0.0_dp
        x2 = 1.0_dp
        y1 = 0.0_dp
        y2 = 1.0_dp
        z1 = 0.0_dp
        z2 = 1.0_dp

        ! Calcuate directional cosines

        L = sqrt( (x2-x1)**2 + (y2-y1)**2 + (z2-z1)**2 )
        nx = (x2-x1)/L
        ny = (y2-y1)/L
        nz = (z2-z1)/L

        ! Calucate local stiffness matrix

        Kel(1,:) = (/  1.0_dp , -1.0_dp  /)
        Kel(2,:) = (/ -1.0_dp,   1.0_dp  /)

        Kel = Kel * (E*A/L)

        G(1,:) = (/ nx, ny, nz, 0.0_dp, 0.0_dp, 0.0_dp /)
        G(2,:) = (/ 0.0_dp, 0.0_dp, 0.0_dp, nx, ny, nz /)
```

(continues on next page)

(continued from previous page)

```

        ! Calculate transformed stiffness matrix

        Ke = matmul(matmul(transpose(G),Kel),G)

        ! Print matrix

        do i=1,6
                write(*,'(6G10.3)') (Ke(i,j), j=1,6)
        end do

        stop

end program function_sample
```

The program produces the following output

```
0.1925    0.1925    0.1925    -.1925    -.1925    -.1925
0.1925    0.1925    0.1925    -.1925    -.1925    -.1925
0.1925    0.1925    0.1925    -.1925    -.1925    -.1925
-.1925    -.1925    -.1925    0.1925    0.1925    0.1925
-.1925    -.1925    -.1925    0.1925    0.1925    0.1925
-.1925    -.1925    -.1925    0.1925    0.1925    0.1925
```

For a more thorough description of matrix handling in Fortran 90/95, see Metcalf and Reid cite{metcalf00}

2.8.5 Elemental procedures

2.8.6 Vector and matrix functions

2.8.7 Reduction routines

2.8.8 Information functions

2.9 Program units and subroutines

2.9.1 Subprograms

A subroutine in Fortran 90/95 has the following syntax

```
subroutine subroutine-name[([dummy-argument-list])]
    [argument-declaration]
    ...
    return
end subroutine [subroutine-name]
```

All variables in Fortran program are passed to subroutines as references to the actual variables. Modifying a parameter in a subroutine will modify the values of variables in the calling subroutine or program. To be able to use the variables in the argument list they must be declared in the subroutine. This is done right after the subroutine declaration. When a subroutine is finished control is returned to the calling routine or program using the **return**-command. Several return statements can exist in subroutine to return control to the calling routine or program. This is illustrated in the following example.

```
subroutine myproc(a,B,C)
    implicit none
    integer :: a
    real, dimension(a,*) :: B
    real, dimension(a) :: C
    .
    .
    .
    return
end subroutine
```

A subroutine is called using the **call** statement. The above subroutine is called with the following code.

```
call myproc(a,B,C)
```

It should be noted that the names used for variables are local to each respective subroutine. Names of variables passed as arguments does not need to have the same name in the calling and called subroutines. It is the order of the arguments that determines how the variables are referenced from the calling subroutine.

In the previous example illustrates how to make the subroutines independent of problem size. The dimensions of the arrays are passed using the **a** parameter instead of using constant values. The last index of an array does not have to specified, indicated with a *, as it is not needed to determine the address to array element.

2.9.2 Functions

Functions are subroutines with a return value, and can be used in different kinds of expressions. The syntax is

```
type function function-name([dummy-argument-list])
    [argument-declaration]
    ...
    function-name = return-value
    ...
    return
end function function-name
```

The following code shows a simple function definition returning the value of $sin(x)$

```
real function f(x)
    real :: x
    f=sin(x)
    return
end function f
```

The return value defined by assigning the name of the function a value. As seen in the previous example. The function is called by giving the name of the function and the associated function arguments.

```
a = f(y)
```

The following example illustrates how to use subroutines to assign an element matrix for a three-dimensional bar element. The example also shows how dynamic memory allocation can be used to allocate matrices. See also the example in section XX

```
program subroutine_sample

    integer, parameter :: dp = &
    selected_real_kind(15,300)

    real(dp) :: ex(2), ey(2), ez(2), ep(2)
    real(dp), allocatable :: Ke(:,:)

    ep(1) = 1.0_dp
    ep(2) = 1.0_dp
    ex(1) = 0.0_dp
    ex(2) = 1.0_dp
    ey(1) = 0.0_dp
    ey(2) = 1.0_dp
    ez(1) = 0.0_dp
    ez(2) = 1.0_dp

    allocate(Ke(6,6))

    call bar3e(ex,ey,ez,ep,Ke)
    call writeMatrix(Ke)

    deallocate(Ke)

    stop

end program subroutine_sample

subroutine bar3e(ex,ey,ez,ep,Ke)

    implicit none

    integer, parameter :: dp = &
    selected_real_kind(15,300)
```

(continues on next page)

(continued from previous page)

```

    real(dp) :: ex(2), ey(2), ez(2), ep(2)
    real(dp) :: Ke(6,6)

    real(dp) :: nxx, nyx, nzx
    real(dp) :: L, E, A
    real(dp) :: Kel(2,2)
    real(dp) :: G(2,6)

    ! Calculate directional cosines

    L = sqrt( (ex(2)-ex(1))**2 + (ey(2)-ey(1))**2 +  &
    (ez(2)-ez(1))**2 )
    nxx = (ex(2)-ex(1))/L
    nyx = (ey(2)-ey(1))/L
    nzx = (ez(2)-ez(1))/L

    ! Calculate local stiffness matrix

    Kel(1,:) = (/  1.0_dp , -1.0_dp  /)
    Kel(2,:) = (/ -1.0_dp,   1.0_dp  /)

    Kel = Kel * (ep(1)*ep(2)/L)

    G(1,:) = (/ nxx, nyx, nzx, 0.0_dp, 0.0_dp, 0.0_dp /)
    G(2,:) = (/ 0.0_dp, 0.0_dp, 0.0_dp, nxx, nyx, nzx /)

    ! Calculate transformed stiffness matrix

    Ke = matmul(matmul(transpose(G),Kel),G)

    return

end subroutine bar3e

subroutine writeMatrix(A)

    integer, parameter :: dp = &
    selected_real_kind(15,300)

```

(continues on next page)

(continued from previous page)

```fortran
    real(dp) :: A(6,6)

    ! Print matrix

    do i=1,6
        write(*,'(6G10.4)') (A(i,j), j=1,6)
    end do

    return

end subroutine writeMatrix
```

The program gives the following output.

```
0.1925    0.1925    0.1925    -.1925    -.1925    -.1925
0.1925    0.1925    0.1925    -.1925    -.1925    -.1925
0.1925    0.1925    0.1925    -.1925    -.1925    -.1925
-.1925    -.1925    -.1925    0.1925    0.1925    0.1925
-.1925    -.1925    -.1925    0.1925    0.1925    0.1925
-.1925    -.1925    -.1925    0.1925    0.1925    0.1925
```

2.9.3 Keyword and optional arguments

Sometimes when implementing subroutines the number of arguments can grow, making the usage of the unnecessary complicated. To solve this wrapper subroutines could be written providing default parameters for main subroutines. This has the drawback of additional maintenance of the wrapper subroutines when the main subroutine is changed. Fortran 2003 provides a solution to this using keyword and optional arguments. An additional parameter attribute, **optional**, can be specified when declaring subroutine parameters. In the following example the, **c** is declared optinal and does not need to be given when the routine is called.

```fortran
subroutine dostuff(A, b, c)

    real    :: A(10,10)
    integer :: b
    integer, optional :: c
    ...
```

The **dostuff** routine can be called in 2 ways:

```
call dostuff(A, b)     ! c is omitted as it is optional
call dostuff(A, b, c)
```

If a routine is called without optional parameters the routine has to be able to determine if a parameter used this can be done using a special function, **present(…)**. This functions returns **.true.** if given parameter is present in the call to the subroutine.

In addition of having optional parameters, subroutine parameters can also be specified by parameter name or keyword. In the following example all these techniques are employed when implementing the **order_icecream** subroutine. This routine only has one required argument, **number**. The other parameters are optional as indicated by the, **optional** in the parameter declaration.

```
program optional_arguments

        implicit none

        call order_icecream(2)
        call order_icecream(2, 1)
        call order_icecream(4, 4, 2)
        call order_icecream(4, topping=3)

contains

subroutine order_icecream(number, flavor, topping)

        integer :: number
        integer, optional :: flavor
        integer, optional :: topping

        print *, number, 'icecreams ordered.'

        if (present(flavor)) then
                print *, 'Flavor is ', flavor
        else
                print *, 'No flavor was given.'
        end if

        if (present(topping)) then
```

(continues on next page)

(continued from previous page)

```fortran
                print *, 'Topping is ', topping
        else
                print *, 'No topping was given.'
        end if

end subroutine order_icecream

end program optional_arguments
```

In the last call the **topping** keyword is used to specify the last optional argument, but leaving the **flavor** parameter undefined.

2.9.4 Procedure arguments

An efficient feature that exists in many other languages is the ability to pass subroutines as arguments to subroutines. This can provide efficient ways to provide algorithms with user provided functions to be used within the algorithm. As an example, a function can be input to a numeric differentiation algorithm as a subroutine parameter. It is now possible to do this in Fortran 2003.

To implement a subroutine that takes a function as an input parameter, the function definition has to be declared in the subroutine parameter declaration using an **interface** block.

```fortran
real function integrate(a, b, func)

    real :: a, b

    interface
        real function func(x)
            real, intent(in) :: x
        end function func
    end interface

    ...
```

The routine can then be called by providing a function with the same interface as input to the function:

```fortran
real function myfunc(x)

    real :: x

    myfunc = sin(x)**2

end function myfunc
```

Calling the **integrate** function then becomes:

```fortran
area = integrate(0.0, 1.0, myfunc)
```

```fortran
module utils

    use mf_datatypes

        implicit none

contains

real(dp) function myfunc(x)
    real(dp), intent(in) :: x

        myfunc = sin(x)

end function myfunc

subroutine tabulate(startInterval, endInterval, step, func)
        real(8), intent(in) :: startInterval, endInterval,␣
↪step
        real(8) :: x

        interface
                real(8) function func(x)
                        real(8), intent(in) :: x
                end function func
        end interface

        x = startInterval

        do while (x<endInterval)
```

(continues on next page)

(continued from previous page)

```fortran
                print *, x, func(x)
                x = x + step
        end do

        return
end subroutine tabulate

end module utils
```

```fortran
program procedures_as_arguments

    use mf_datatypes
        use utils

        implicit none

    call tabulate(0.0_dp, 3.14_dp, 0.1_dp, myfunc)

end program procedures_as_arguments
```

2.9.5 Modules

When programs become larger, they often need to be split into more manageable parts. In other languages this is often achieved using include files or packages. In Fortran 77, no such functionality exists. Source files can be grouped in files, but no standard way of including specific libraries of subroutines or function exists in the language. The C preprocessor is often used to include code from libraries in Fortran, but is not standardised in the language itself.

In Fortran 90 the concept of modules was introduced. A Fortran 90 module can contain both variables, parameters and subroutines. This makes it possible to divide programs into well defined modules which are more easily maintained. The syntax for a module is similar to that of how a main program in Fortran is defined.

```fortran
module module-name
    [specification-stmts]
contains
    module-subprograms]
```

(continues on next page)

(continued from previous page)

```
end module [module-name]]
```

The block **specification-stmts** defines the variables that are available for programs or subroutines using the module. In the block, **module--sub-programs**, subroutines in the module are declared. A module can contain only variables or only subroutines or both. One use of this, is to declare variables common to several modules i a separate module. Modules are also a good way to divide a program into logical and coherent parts. Variables and functions in a module can be made private to a module, hiding them for routines using the module. The keywords **public** and **private** can be used to control the access to a variable or a function. In the following code the variable, **a**, is hidden from subroutines or programs using this module. The variable, **b**, is however visible. When nothing is specified in the variable declaration, the variable is assumed to be public.

```
module mymodule

    integer, private :: a
    integer :: b
    ...
```

The ability to hide variables in modules enables the developer to hide the implementation details of a module, reducing the risk of accidental modification variables and use of subroutines used in the implementation.

To access the routines and variables in a module the **use** statement is used. This makes all the public variables and subroutines available in programs and other modules. In the following example illustrate how the subroutines use in the previous examples are placed in a module, **truss**, and used from a main program.

```
module truss

    use mf_datatypes
    use mf_utils

    ! Public variable declarations

        ! Variables that are visible for other programs
        ! and modules

        ! Private variables declarations

```

(continues on next page)

(continued from previous page)

```fortran
contains

subroutine bar3e(ex,ey,ez,ep,Ke)

        implicit none


        real(dp) :: ex(2), ey(2), ez(2), ep(2)
        real(dp) :: Ke(6,6)

        real(dp) :: nxx, nyx, nzx
        real(dp) :: L, E, A
        real(dp) :: Kel(2,2)
        real(dp) :: G(2,6)

        ! Calculate directional cosines

        L = sqrt( (ex(2)-ex(1))**2 + (ey(2)-ey(1))**2 &
                + (ez(2)-ez(1))**2 )

        nxx = (ex(2)-ex(1))/L
        nyx = (ey(2)-ey(1))/L
        nzx = (ez(2)-ez(1))/L

        ! Calculate local stiffness matrix

        Kel(1,:) = (/  1.0_dp , -1.0_dp  /)
        Kel(2,:) = (/ -1.0_dp,   1.0_dp  /)

        Kel = Kel * (ep(1)*ep(2)/L)

        G(1,:) = (/ nxx, nyx, nzx, &
                0.0_dp, 0.0_dp, 0.0_dp /)
        G(2,:) = (/ 0.0_dp, 0.0_dp, 0.0_dp, &
                nxx, nyx, nzx /)

        ! Calculate transformed stiffness matrix

        Ke = matmul(matmul(transpose(G),Kel),G)

```

(continues on next page)

(continued from previous page)

```fortran
        return

end subroutine bar3e

end module truss
```

Main program using the **truss** module.

```fortran
program module_sample

    use mf_datatypes
    use mf_utils
        use truss

        implicit none

        real(dp) :: ex(2), ey(2), ez(2), ep(2)
        real(dp), allocatable :: Ke(:,:)

        ep(1) = 1.0_dp
        ep(2) = 1.0_dp
        ex(1) = 0.0_dp
        ex(2) = 1.0_dp
        ey(1) = 0.0_dp
        ey(2) = 1.0_dp
        ez(1) = 0.0_dp
        ez(2) = 1.0_dp

        allocate(Ke(6,6))

        call bar3e(ex,ey,ez,ep,Ke)
    call print_matrix(Ke)

        deallocate(Ke)

        stop

end program module_sample
```

Please note that the declaration of **ap** in the **truss** module is used to define the precision of the variables in the main program.

2.9.6 Public and private attributes

When implementing modules, some of the routines and variables are only used the implementation of the module. That is, some of the variables and subroutines should not be accessible for the user of the module. To control access to variables and subroutines the attributes **private** and **public** can be used in the declaration of variables and subroutines. A variable can be declared private by adding the keyword **private** to the attribute list in the declaration as shown in the following example:

```
real, private :: a
```

If no **private** attribute is given the variable is by default declared as **public**. If a private variable is access from another module a main program will generate a compiler error.

To declare a subroutine or function as private it has to be declared as such in the specification part of the module, that is before the **contains**-keyword. In the following example illustrates the concept.

```
module mymodule

    private myprivatesub

contains

    subroutine myprivatesub

        print *, 'This subroutine can only be called from␣
↪within the module.'

    end subroutine myprivatesub

    subroutine mypublicsub

        print *, 'This subroutine can be called from other␣
↪modules.'

    end subroutine mypublicsub

end module mymodule
```

In this example, **myprivatesub**, can only be called from within the module.

Calling it from another module or main program will result in a compiler error. **myprivatesub** is not declared as private in the specification part and hence can be called from all other modules.

2.9.7 Overloading

As Fortran is a strongly typed language, supporting multiple data types in a single subroutine is not possible and requires separate unique subroutines declarations. To simplify module use and enable a module user to call a routine with different data types, Fortran 90 supports the concept of overloading. Using overloading the compiler can decide which routine to call depending on the datatype used. However, this requires a special declaration in the module specification.

To illustrate this, a function, **func**, is implemented that can take either a floating point parameter or an integer parameter. To implement this function, an interface declaration for **func** is added in the module specification part:

```
module overloaded

    interface func
        module procedure ifunc, rfunc
    end interface
    ...
```

This tells the compiler to map the **func**-function to the functions **ifunc** or **rfunc**, depending on the datatype used when the function is called. **ifunc** or **rfunc** are implemented as normal functions as shown below:

```
    ...

contains

integer function ifunc(x)

    integer, intent(in) :: x
    ifunc = x * 42

end function ifunc

real function rfunc(x)

```

(continues on next page)

(continued from previous page)

```
    real, intent(in) :: x
    rfunc = x / 42.0

end function rfunc

end module overloaded
```

The **func**-function can now be called using either floating point values or integer values illustrated below in the following example:

```
program overloading

    use special

    integer :: a = 42
    real :: b = 42.0

    a = func(a)
    b = func(b)

    print *, a
    print *, b

end program overloading
```

Running this program produce the following output:

```
$ ./overloading
        1764
1.0000000000000000
```

This means that **ifunc** is called in the first call to **func** and **rfunc** is called in the second call to **func**.

2.9.8 operator overloading

In many modern languages such as C++ and Python, the operators can be overloaded to support expressions for user implemented data types. This is also possible in Fortran. To illustrate how this is achieved, a **vector_operations**-module is implemented, enabling addition of vectors using the + operator.

First, a vector type is defined in our module **vector_operations**. This is the actual data type that will be used in the expressions to be evaluated.

```
module vector_operations

    type vector
        real :: components(3)
    end type vector
    ...
```

Next, an interface for overloading the + operator is defined. The interface tells the compiler which function to call when it encounters an expression with our vector data type. In this example the **vector_plus_vector**-function will be called for the + operator.

```
...
interface operator(+)
    module procedure vector_plus_vector
end interface
...
```

In the final step the actual function for adding vectors is implemented. This functions needs to have two input parameters for the vectors to be added in the operation. It also needs to return a **vector** data type.

```
...
contains

type(vector) function vector_plus_vector(v1, v2)

    type(vector), intent(in) :: v1, v2
    vector_plus_vector%components = v1%components + v2
↪%components

end function vector_plus_vector
```

(continues on next page)

(continued from previous page)

```fortran

end module vector_operations
```

The new data type together with the defined + operator can now be used to implement compact expressions for vector algebra as illustrated in the following code:

```fortran
program operator_overloading

    use vector_operations

    type(vector) :: v1
    type(vector) :: v2
    type(vector) :: v

    v1%components = (/1.0, 0.0, 0.0/)
    v2%components = (/0.0, 1.0, 0.0/)

    v = v1 + v2

    print *, v

end program operator_overloading
```

Running the code will produced the expected output:

```
$ ./opoverload
  1.00000000000000000        1.00000000000000000        0.
↪00000000000000000
```

Operators for -, * and / can be implemented using the same technique.

2.9.9 Allocatable dummy arguments

```fortran
program allocatable_dummy

    implicit none

    real, allocatable :: A(:,:)
```

(continues on next page)

(continued from previous page)

```fortran

    call createArray(A)

    print *, size(A,1), size(A,2)

    deallocate(A)

contains

subroutine createArray(A)

    real, allocatable, intent(out) :: A(:,:)

    allocate(A(20,20))

end subroutine createArray

end program allocatable_dummy
```

2.9.10 Allocatable functions

```fortran
program allocatable_function

    implicit none

    real :: A(20)

    A = create_vector(20)
    print *, size(A,1)

contains

function create_vector(n)

    real, allocatable, dimension(:) :: create_vector
    integer, intent(in) :: n

    allocate(create_vector(n))
```

(continues on next page)

(continued from previous page)

```fortran

end function create_vector

end program allocatable_function
```

2.9.11 Submodules (2003)

```fortran
module points
        type point
                real :: x, y
        end type point

        interface
                real module function point_dist(a, b)
                        type(point), intent(in) :: a, b
                end function point_dist
        end interface
end module points

```

```fortran
submodule (points) points_a
contains

real module function point_dist(a,b)
        type(point), intent(in) :: a, b
        point_dist = sqrt((a%x-b%x)**2+(a%y-b%y)**2)
end function point_dist

end submodule points_a
```

```fortran
program submodules

    use points

    implicit none

end program submodules
```

2.10 Input and output

Input and output to and from different devices, such as screen, keyboard and files are accomplished using the commands **read** and **write**. The syntax for these commands are:

```
read(u, fmt) [list]
write(u, fmt) [list]
```

u is the device that is used for reading or writing. If a star (*) is used as a device, standard output and standard input are used (screen, keyboard or pipes).

fmt is a string describing how variables should be read or written. This is often important when writing results to text files, to make it more easily readable. If a star (*) is used a so called free format is used, no special formatting is used. The format string consists of one or more format specifiers, which have the general form:

```
[repeat-count] format-descriptor w[.m]
```

where **repeat-count** is the number of variables that this format applies to. **format-descriptor** defines the type of format specifier. **w** defined the width of the output field and **m** is the number of significant numbers or decimals in the output. The following example outputs some numbers using different format specifiers and table~ref{table:formatkoder} show the most commonly used format specifiers.

```
program formatting

        implicit none

        integer, parameter :: dp = &
                selected_real_kind(15,300)

        write(*,'(A15)') '123456789012345'
        write(*,'(G15.4)') 5.675789_dp
        write(*,'(G15.4)') 0.0675789_dp
        write(*,'(E15.4)') 0.675779_dp
        write(*,'(F15.4)') 0.675779_dp
        write(*,*)         0.675779_dp
        write(*,'(I15)')   156
        write(*,*) 156
```

(continues on next page)

(continued from previous page)

```

        stop

end program formatting
```

The program produces the following output:

```
123456789012345
    5.676
    0.6758E-01
    0.6758E+00
        0.6758
0.675779000000000
            156
        156
```

begin{table} begin{center} begin{tabular}{|l|l|} hline Kod & Beskrivning \ hline E & Scientific notation. Values are converted to the format "-d.dddE+ddd". \ F & Decimal notation. Values are converted to the format "-d ddd.ddd…". \ G & Generic notation. Values are converted to the format -ddd.ddd or -d.dddE+ddd \ I & Integers. \ A & Strings \ TRn & Move n positions right \ Tn & Continue at position n \ hline end{tabular} end{center} caption{Formatting codes in fkeyw{read}/**write**} label{table:formatkoder} end{table}

During output a invisible cursor is moved from left to right. The format specifiers TR:math:n and T:math:n are used to move this cursor. TR:math:n moves the cursor n positions to the right from the previous position. T:math:n places the cursor at position n. Figure~ref{fig:format_positiong} shows how this can be used in a write-statement.

Fig. 2.2: Positioning of output in Fortran 90/95

The output routines in Fortran was originally intended to be used on row printers where the first character was a control character. The effect of this is that the default behavior of these routines is that output always starts at the second position. On modern computers this is not an issue, and the first character can be used for printing. To print from the first character, the format specifier **T1** can be used to position the cursor at the first position. The following code writes ''Hej hopp!" starting from the first position.

```
write(*,'(T1,A)') 'Hej hopp!'
```

A more thorough description of the available format specifiers in Fortran is given in Metcalf and Reid~cite{metcalf00}.

2.10.1 Reading and Writing from files

The input and output routines can also be used to write data to and from files. This is accomplished by associating a file in the file system with a file unit number, and using this number in the **read** and **write** statements to direct the input and output to the correct files. A file is associated, opened, with a unit number using an **open**-statement. When operations on the file is finished it is closed using the **close**-statement.

The file unit number is an integer usually between 1 and 99. On many systems the file unit number 5 is the keyboard and unit 6 the screen display. It is therefore recommended to avoid using these numbers in file operations.

In the **open**-statement the properties of the opened files are given, such as if the file already exists, how the file is accessed (reading or writing) and the filename used in the filesystem.

An example of reading and writing file is given in the following example.

```
program sample2

    use mf_datatypes

        implicit none

    real(dp), allocatable :: infield(:,:)
    real(dp), allocatable :: rowsum(:)
        integer :: rows, i, j

        !  File unit numbers

        integer, parameter :: infile = 15
        integer, parameter :: outfile = 16

        !  Allocate matrices

```

(continues on next page)

(continued from previous page)

```fortran
        rows=5
        allocate(infield(3,rows))
        allocate(rowsum(rows))

        !  Open the file 'indata.dat' for reading

        open(unit=infile,file='indata.dat',&
                access='sequential',&
                action='read')

        !  Open the file 'utdata.dat' for writing

        open(unit=outfile,file='utdata.dat',&
                access='sequential',&
                action='write')

        !  Read input from file

        do i=1,rows
                read(infile,*) (infield(j,i),j=1,3)
                rowsum(i)=&
                        infield(1,i)+infield(2,i)+infield(3,
↪i)
                write(outfile,*) rowsum(i)
        end do

        !  Close files

        close(infile)
        close(outfile)

        !  Free used memory

        deallocate(infield)
        deallocate(rowsum)

        stop

end program sample2
```

In this example, 2 files are opened, ffname{indata.dat} and ffname{utdata.dat} with **open**-statements. Using the **read**-statement five rows with 3 numbers on each row are read from the file ffname{indata.dat}. The sum of each row is calculated and is written using **write**-statements to the file ffname{utdata.dat}. Finally the files are closed using the **close**-statements.

2.10.2 Dynamic format codes

One problem that arises when writing formatted output, is how to handle output of data in which the number of columns is unknown at compile time. To solve this, a special technique using strings as file units can be employed. To illustrate this technique we implement a subroutine **writeArray**, which takes an array of any size as input and tries to print it nicely. First we declare the module subroutine and extract the size of the incoming array:

```fortran
subroutine writeArray(A)

    real(8), dimension(:,:) :: A
    integer :: rows, cols, i, j
    character(255) :: fmt

    rows = size(A,1)
    cols = size(A,2)

    ...
```

Next, we use a **write**-statement, that instead of a file unit number takes the string, **fmt**, and uses it as an output file. In the **write**-statement, we write out the needed format code for printing the incoming array, which is then stored in the string **fmt**.

```fortran
...
write(fmt, '(A,I1,A)') '(',cols, 'G8.3)'
...
```

The generated format code can now be used when printing the incoming array **A**.

```fortran
    ...
    do i=1,rows
        print fmt, (A(i,j), j=1,cols)
    end do
```

(continues on next page)

(continued from previous page)

```
    return

end subroutine writeArray
```

In the following main program, the implemented **writeArray** subroutine is used to print a 6 by 6 matrix.

```
program dynamic_fcodes

    use array_utils

    real(8) :: A(6,6)

    A = 42.0_8

    call writeArray(A)

end program dynamic_fcodes
```

The resulting formatted output is shown below:

```
$ ./dynamic_fcodes
42.0    42.0    42.0    42.0    42.0    42.0
42.0    42.0    42.0    42.0    42.0    42.0
42.0    42.0    42.0    42.0    42.0    42.0
42.0    42.0    42.0    42.0    42.0    42.0
42.0    42.0    42.0    42.0    42.0    42.0
42.0    42.0    42.0    42.0    42.0    42.0
```

2.10.3 Namelist I/O

The standard way of writing or reading text files in Fortran is using list directed I/O. This means specifying a list of variables to be read or written using the **read**- and **write**-statements. Fortran will automatically handle the conversion of datatypes to and from a text based format. A more flexible way of handling text file I/O is using namelists. Namelists can be considered as named list of variables to be used for reading or writing. In this scheme, variables can be read and written to files using names. To write variables and data using this technique, variables must be listed using the special **namelist** statement as shown below:

```
integer :: no_of_eggs, litres_of_milk, kilos_of_butter,␣
↪list(5)
namelist /food/ no_of_eggs, litres_of_milk, kilos_of_butter,
↪ list
```

Here a namelist, **food**, is defined consisting of the specified variables. Variables in a namelist can be of any type. To write the variables to a file, the **nml**-keyword can be used in the **read**- and **write**-statements to specify which namelist that should be used.

The namelist in the text file starts with the & character followed by the namelist-name then the namelist variable pairs are listed separated by commas. The namelist is ended with a single /. The following example shows 2 namelist entries in a text file:

```
&food litres_of_milk=5, no_of_eggs=12, kilos_of_butter=42,␣
↪list=1,2,3,4,5 /
&food litres_of_milk=6, no_of_eggs=24, kilos_of_butter=84,␣
↪list=2,3,4,5,6 /
```

Multiple namelist entries can be read from an opened file. The following code shows how 2 namelist entries of the type **food** are read from an opened file:

```
open(unit=ir, file='food.txt', status='old')
read(ir, nml=food)
print *, no_of_eggs, litres_of_milk, kilos_of_butter
read(ir, nml=food)
print *, no_of_eggs, litres_of_milk, kilos_of_butter
close(unit=ir)
```

Running this code produces the following output:

```
12            5           42
24            6           84
```

fmode

Writing using namelist I/O is done in the same way as reading. The following code shows how the same namelist variables are written to a namelist:

```
open(unit=iw, file='food2.txt', status='new')
write(iw, nml=food)
close(unit=iw)
```

The contents of the written file, **food2.txt**, is shown below:

```
&FOOD
NO_OF_EGGS=          24,
LITRES_OF_MILK=          6,
KILOS_OF_BUTTER=         84,
LIST=          2,           3,           4,           5,        ␣
↪    6,

/
```

Pleas notice that Fortran allways uses uppercase for variable names in the written file.

```
program namelist_io

        implicit none

        integer, parameter :: ir = 15
        integer, parameter :: iw = 16
        integer :: no_of_eggs, litres_of_milk, kilos_of_
↪butter, list(5)
        namelist /food/ no_of_eggs, litres_of_milk, kilos_
↪of_butter, list

        list = 0

        open(unit=ir, file='food.txt', status='old')
        read(ir, nml=food)
        print *, no_of_eggs, litres_of_milk, kilos_of_butter
        read(ir, nml=food)
        print *, no_of_eggs, litres_of_milk, kilos_of_butter
        close(unit=ir)

        open(unit=iw, file='food2.txt', status='new')
        write(iw, nml=food)
        close(unit=iw)

end program namelist_io
```

2.10.4 Unformatted I/O

In the previous sections data was read and written in human readable text format. For larger data structures this can be very inefficient. To solve this Fortran can also write data in its native binary format directly to disk. This can save space and can also be read and written much faster to disk. However, the binary format is not standardised and differs between different hardware platforms, preventing files to be used on different hardware.

Reading and writing binary data is done using the same **read**- and wr**write**-ite statements as before, but without the formatting options. Writing an array to disk in binary form can be done with just one simple statement:

```fortran
real :: A(100)
...
write(iw) A
```

Reading the same array back from disk is just as easy, using the **read**-statement.

```fortran
real :: A(100)
...
read(ir) A
```

It is also possible to write several variables to disk using multiple **write**- statements.

```fortran
real :: A(100), B(200)
...
write(iw) A
write(iw) B
```

However, it is important to note that data has to be read back in the same order it is was written. So the code for reading the data back becomes:

```fortran
real :: A(100), B(200)
...
read(ir) A
read(ir) B
```

To enable reading and writing unformatted I/O files (binary files) the keyword **form='unformatted'** must be added to the **open**-statement.

```fortran
real :: A(100), B(200)
...
open(unit=ir, file='arrays.dat', form='unformatted')
read(ir) A
read(ir) B
close(ir)
```

The concept of unformatted I/O is illustrated in a larger example. In this example an array of the derived datatype **particle** is created, initialised and then saved to disk as unformatted I/O. After saving the data to disk it is read back using unformatted I/O and printed on standard output. The listing is shown below:

```fortran
program unformatted_io_2

    implicit none

    ! ---- Define some program constants

    integer, parameter :: iw = 15
    integer, parameter :: ir = 16
    integer, parameter :: nParticles = 1000

    ! ---- Define particle data type

    type particle
            real :: position(3)
            real :: velocity(3)
            real :: mass
    end type particle

    ! ---- Program variables

    integer :: i

    ! ---- Allocatable array of particles

    type(particle), allocatable :: particles(:)

    allocate(particles(nParticles))

    ! ---- Initialise particle array
```

(continues on next page)

(continued from previous page)

```fortran

    do i=1,nParticles
        particles(i)%position = 0.0
        particles(i)%velocity = 0.0
        particles(i)%mass = 1.0
    end do

    ! ---- Write all particles to disk

    open(unit=iw, file='particles.dat', form='unformatted', status='replace')
    write(unit=iw) particles
    close(unit=iw)

    ! ---- Deallocate particles array

    deallocate(particles)

    ! ---- Allocate new array and read back data from disk

    allocate(particles(nParticles))

    open(unit=ir, file='particles.dat', form='unformatted')
    read(unit=ir) particles
    close(unit=ir)

    ! ---- Print contents of array

    print*, particles

    ! ---- Deallocate array

    deallocate(particles)


end program unformatted_io_2
```

The code produced the following output when run:

```
$ ./unformatted_io_2
```

(continues on next page)

(continued from previous page)

```
0.00000000        0.00000000        0.00000000        0.
↪00000000        0.00000000        0.00000000        1.
↪00000000        0.00000000        0.00000000        0.
↪00000000        0.00000000        0.00000000        0.
↪00000000        1.00000000        0.00000000        0.
↪00000000        0.00000000        0.00000000        0.
↪00000000        0.00000000        1.00000000        0.
↪00000000        0.00000000        0.00000000        0.
↪00000000        0.00000000        0.00000000        1.
↪00000000        0.00000000        0.00000000        0.
↪00000000        0.00000000 ....
```

Which means that the data was read correctly back from disk.

2.10.5 Direct access files

A variant of unformatted I/O is direct access files. One problem with unformatted I/O is that files have to be read and written sequentially. This make it inefficient if you would like to access certain parts of the file randomly. To solve this problem Fortran provides direct access file format. In this format the file is divided in several equally spaced data records. These records can be read randomly back from a single file. It can be compared to a datbase file with data records.

To create a direct access file consisting of records of the following derived data type,

```
type account
    character(len=40) :: account_holder
    real :: balance
end type account
```

the size of the data record has to be calculated. This can be done using the **inquire**-function. This assigns a variable the record size of the data type, which is shown in the following listing:

```
type(account) :: account
integer :: recordSize
...
inquire(iolength=recordSize) account
```

The **recordSize** variable can now be used when we create a direct access file using the **open**-statement:

```fortran
open(unit=iw, file='accounts.dat', access='direct',␣
↪recl=recordSize, status='replace')
```

Writing the records is accomplished using the normal **write**-statement with an added **rec**-option for the record position to be written.

```fortran
write(iw, rec=1) account
```

It is possible to write to any record position when writing record. Reading record is done using the **read**-statements using the **rec**-option.

When reading or writing to direct access files there is an invisible cursor or pointer pointing to the current record. It is possible to manipulate this cursor using the **rewind**- and **backspace**-statements. The **rewind**-statement moves the pointer to the first record in the file. the **backspace**-statement moves the pointer one record back in the file, these operations are illustrated in the following figure.

Fig. 2.3: **rewind**- and **backspace**-statements

It is also possible to truncate a direct access file at a given position using the **endfile**-statement, as illustrated in the following figure. All records after the current record will be truncated.

Fig. 2.4: Truncating a file using the **endfile**-statement

The following code shows a complete example, writing to records to a direct access file:

```fortran
program unformatted_io

    implicit none

    integer, parameter :: iw = 15
    type account
    character(len=40) :: account_holder
    real :: balance
    end type account
```

(continues on next page)

(continued from previous page)

```fortran

    type(account) :: accountA
    type(account) :: accountB
    integer :: recordSize

    inquire(iolength=recordSize) accountA

    print *, 'Record size =',recordSize

    accountA%account_holder = 'Olle'
    accountA%balance = 400

    accountB%account_holder = 'Janne'
    accountB%balance = 800

    open(unit=iw, file='bank.dat', access='direct',␣
↪recl=recordSize, status='replace')
    write(iw, rec=1) accountA
    write(iw, rec=2) accountB
    close(unit=iw)

end program unformatted_io
```

2.10.6 Error handling in I/O operations

One problem with reading and writing files is that errors can occur on a system level, such as unavailable disk space, file system problems and non-existant files. If not handled, the program will crash in an unexpected way and prevent proper clean up code to be run. In Fortran I/O errors are handled using the **err**-option in all I/O related functions. Using the **err**-option a label can be defined where the execution continues when an I/O error occurs.

In the following code a file is opened for reading. Error handling code is added for each I/O operation, providing a different label and response for all error conditions.

```fortran
program error_handling

    implicit none
```

(continues on next page)

(continued from previous page)

```fortran

    integer, parameter :: ir = 15
    integer :: a

    open(unit=ir, file='test.txt', status='old', err=101)
    read(ir,*,err=102) a
    close(unit=ir,err=103)

    stop

101 print *, 'An error occured when opening the file.'

    stop

102 print *, 'An error occured when reading the file.'

    stop

103 print *, 'An error occured when closing the file.'

    stop

end program error_handling
```

Figure~ref{fig:error_handling} shows the flow of the program.

Fig. 2.5: Error handling i Fortran I/O operations

It is also possible to determine the reason for ther error by using the **iostat** option. A variable is associated with the the **iostat**-option and when an error occurs the variable will be assigned an with an error code. The following code shows an example of how this option can be used:

```fortran
integer :: ierr
...
open(..., iostat=ierr)
...
read(unit=xx, iostat=ierr)
...
close(unit=xx, iostat=ierr)
```

If no error was encountered the associated variable with be assigned an error code of 0. Other codes can be:

- -2, End of record condition occurs in non-advancing I/O.
- -1, End of file condition.
- >1, Standardised list of fortran error codes.

When the **iostat**-option is used all default error messages will be suppressed and code execution will continue. The best use of this option is combined with the **err**-option to provide clear error messages to users as show in the following code example:

```fortran
subroutine read_from_file(...)
    ...
    integer :: ierr
    ...
    read(unit=xx, err=101, iostat=ierr)
    ...
    return

101 print*, 'Error ', ierr, ' reading file.'
    return

end subroutine
```

2.11 String manipulation

There are several ways of manipulating strings in Fortran. Strings can be concatenated with the operator, //, as shown in the following example:

```fortran
c1 = 'Hej '
c2 = 'hopp!'
c = c1 // c2 ! = 'Hej hopp!'
```

Fortran does not have dynamic strings, so the size of the resulting string must be large enough for the concatenated string.

Substrings can be extracted using a syntax similar to the syntax used when indexing arrays.

```fortran
c3 = c(5:8) ! Contains the string 'hopp'
```

A common task in many codes is the conversion of numbers to and from strings. Fortran does not have any explicit functions these type of conversions, instead the the **read** and **write** statements can be used together with strings to accomplish the same thing. By replacing the file unit number with a character string variable, the string can be read from and written to using **read** and **write** statements.

To convert a floating point value to a string the following code can be used.

```fortran
character(255) :: mystring
real(8) :: myvalue
value = 42.0
write(mystring,'(G15.4)') value
! mystring now contains '      5.676'
```

To convert a value contained in string to a floating point value the read-statement is used.

```fortran
character(255) :: mystring
real(8) :: myvalue
mystring = '42.0'
read(mystring,*) myvalue
! myvalue now contains 42.0
```

A more complete example is shown in the following listing:

```fortran
program strings2

        implicit none

        integer :: i
        character(20) :: c

        c = '5'
        read(c,'(I5)') i
        write(*,*) i

        i = 42
        write(c,'(I5)') i
        write(*,*) c

```

(continues on next page)

(continued from previous page)

```
        stop

end program strings2
```

The program produces the following output.

```
5
42
```

2.12 Array features

2.12.1 Forall- and Where-statements

Fortran has added a number of new loop-statements. The **forall**-statement has been added to optimise nested loops for execution on multiprocessor machines. The syntax is:

```
forall (index = lower:upper [,index = lower:upper])
    [body]
end forall
```

The following example shows how a **do**-statement can be replaced with a **forall**-statement.

```
do i=1,n
    do j=1,m
        A(i,j)=i+j
    end do
end do

! Is equivalent with

forall(i=1:n, j=1:m)
    A(i,j)=i+j
end forall
```

Another statement optimised for multiprocessor architectures is the **where**-statement. With this statement conditional operations on an array can be achieved efficiently. The syntax comes in two versions.

```
where (logical-array-expr)
    array-assignments
end where
```

and

```
where (logical-array-expr)
    array-assignments
else where
    array-assignments
end where
```

The usage of the **where**-statement is best illustrated with an example.

```
where (A>1)
    B = 0
else where
    B = A
end where
```

In this example two arrays with the same size are used in the **where**-statement. In this case the values in the **B** array are assigned 0 when an element in the A array is larger than 1 otherwise the element in **B** is assigned the same value as in the **A** array.

2.12.2 Elemental procedures

2.13 Pointers

```
program pointers

        implicit none

        integer, allocatable, dimension(:,:), target :: A
        integer, dimension(:,:), pointer :: B, C

        allocate(A(20,20))

        B => A
```

(continues on next page)

(continued from previous page)

```

        print *, size(B,1), size(B,2)

        call createArray(C)

        print *, size(C,1), size(C,2)

        deallocate(C)

        B => null()

        B(1,1) = 0 ! Dangerous!

contains

subroutine createArray(D)

        integer, dimension(:,:), pointer :: D

        allocate(D(10,10))

end subroutine createArray

end program pointers
```

2.14 System functions

2.14.1 C Interoperability

```
program c_interop

        use iso_c_binding

        implicit none

        integer(c_int) :: a
        real(c_float) :: b
```

(continues on next page)

(continued from previous page)

```fortran
        real(c_double) :: c

        a = 42
        b = 42.0_c_float
        c = 84.0_c_double

        print *, a, b, c

end program c_interop
```

2.14.2 Access to computing environment

2.15 Object-oriented programming

In procedural programming, data and subroutines are treated separately. Subroutines operate on provided data structures and variables. In object-oriented programming data and subroutines are combined into objects. Objects in numerical computing can be be different matrix types, particles, vectors or solvers. The major benefits are that the actual data structures used in the implementation of an object can be hidden from the user of the object, enabling the developer of an object to improve the implementation without affecting users of the objects (encapsulation). Another important feature of object-oriented programming is the ability to inherit and extend functionality of objects (inheritance). This enables user of object and developers to extend and modify functionality of existing objects, relying on functionality of the parent object.

In Fortran 2003 object-oriented features where added to the language, making Fortran almost as feature rich as other more recent languages. Most modern Fortran compilers today support the object-oriented features added in the 2003 standard, enabling developers to implement truly object-oriented numerical applications in Fortran.

The functionality and data structures of objects are defined in classes in most programming object-oriented languages. Classes can be seen as templates for objects. When an object is to be created the class is used as the template for the new object. Created objects are also called instances of a class.

In Fortran the object-oriented features are implemented by extending the derived datatype concepts of fortran 90. A derived datatype now has a **contains**-section in which the procedures of the objects are specified. Derived datatypes are now also by definition objects or instances of the derived type.

To illustrate the concepts, a simple particle object is defined as an object in Fortran. The particle object is defined as a derived data type in a module, particles. To eliminate any name clashes when creating new objects of this type, the derived type is given the name, **particle_class**. This is also fits the object-oriented model of derived type being equivalent to classes. The initial class definition then becomes:

```
module particles

    implicit none

    type particle_class
        real :: pos(3)
        real :: vel(3)
    end type particle_class

end module particles
```

The definition currently corresponds exactly to a derived data type in Fortran and can be used as such as well. To create instance of the **particle_class** is equivalent to creating a variable of a specified derived data type:

```
use particles
...
type(particle_class) :: particle
```

To access the variables of the instance the % operator is used. In the following examplen the **pos**-variable is assigned the coordinate $(0, 0, 0)$.

```
particle % pos = (/ 0.0, 0.0, 0.0 /)
```

However, accessing the instance variables goes against the principles of object-oriented programming, where one of the more important aspects is data encapsulation and hiding the internal workings of the objects. How do we use the new Fortran features to prevent the need to access the data structures directly? The first aspect to cover is initialisation of the instance variables. To do this, a method, **init**, will be added to our class. First, a **contains**-section with a procedure specification for the **init** subroutine. As it is not allowed to have duplicate subroutine names in a module we assign an actual implementation subroutine using the => operator. Please note that no parameters lists are specified in this declaration. Secondly, to distinguish the member variables from other variables in the class implementation, as well as preventing name collisions with the names of the access methods, the member variables are prefixed with **m_**. The complete class then becomes:

```fortran
module particles

    implicit none

    type particle_class
        real :: m_pos(3)
        real :: m_vel(3)
    contains
        procedure :: init => particle_init
    end type particle_class
    ...
```

This declaration states that the class, **particle_class**, has a member subroutine with the name, **init**, defined later in the source code by the subroutine **particle_init**. To complete the class definition, the subroutine **particle_init** have to be added to the **particles** module.

All member subroutines can take a first dummy argument with containing a reference to the actual instance. This will be used to enable us to do our initialisation on the actual data structures of the instance. The **init**-subroutine of our particle class then becomes:

```fortran
module particles

...

contains

subroutine particle_init(this)

    class(particle_class) :: this

    this % m_pos = (/0.0, 0.0, 0.0/)
    this % m_vel = (/0.0, 0.0, 0.0/)

end subroutine particle_init

```

(continues on next page)

(continued from previous page)

```
end module particles
```

In the previous example the **particle_init**-subroutine has a dummy variable, **this**, which is used to access the actual instance variables of the class. This variable is automatically passed to the routine by the compiler. It is now possible to initialise our newly created instance without accessing the member variables directly. The code to create a new object and initialise it data structure then becomes:

```
type(particle_class) :: particle

call particle % init
```

The dummy argument, **init**, can be left out of the call to **init**. The same concept of passing the instance variable as a argument in the class definition can also be found in Python, where a special variable, **self** is used in the member subroutines. Please also note that we call the method with the **init** and not the actual implemented subroutine, **particle_init**. This enables us to have the same class interface in several classes.

2.15.1 Access methods

The class now has the ability to initialise its data variables. However, we don't have any ways of accessing the variables without accessing them directly. To solve this we have to add special methods for accessing internal class variables. First, methods for assigning the position and velocity of the particle is added in the class declaration:

```
...
type particle_class
    real :: m_pos(3)
    real :: m_vel(3)
contains
    procedure :: init => particle_init
    procedure :: set_position => particle_set_position |\
↪hladded|
    procedure :: set_velocity => particle_set_velocity |\
↪hladded|
end type particle_class
...
```

When this has been done the implementations of these subroutines are added in the **contains**-section of the **parcticles**-modules:

```
contains

...

subroutine particle_set_position(this, x, y ,z)

    class(particle_class) :: this
    real :: x, y, z

    this % m_pos = (/x, y, z/)

end subroutine particle_set_position

subroutine particle_set_velocity(this, vx, vy ,vz)

    class(particle_class) :: this
    real :: vx, vy, vz

    this % m_vel = (/vx, vy, vz/)

end subroutine particle_set_velocity

end module particles
```

In the same way as in the **init**-subroutine, **this**, is used to access the member variables of the class instance.

It is now possible to assign values to our instances without directly accessing the internal member variables as shown in the following code:

```
call particle % set_position(1.0, 1.0, 1.0)
call particle % set_velocity(1.0, 1.0, 1.0)
```

To retrieve values from the instance, 2 additional subroutines are needed, **get_position** and **get_velocity** are added to the class definition and implementation.

```
module particles

```

(continues on next page)

(continued from previous page)

```
    implicit none

    type particle_class
        real :: m_pos(3)
        real :: m_vel(3)
    contains
        procedure :: init
        procedure :: set_position => particle_set_position
        procedure :: set_velocity => particle_set_velocity
        procedure :: get_position => particle_get_position␣
↪|\hladded|
        procedure :: get_velocity => particle_get_velocity␣
↪|\hladded|
    end type particle_class

contains

...

subroutine particle_get_position(this, x, y ,z)

    class(particle_class) :: this
    real, intent(out) :: x, y, z

    x = this % m_pos(1)
    y = this % m_pos(2)
    z = this % m_pos(3)

end subroutine particle_get_position

subroutine particle_get_velocity(this, vx, vy ,vz)

    class(particle_class) :: this
    real, intent(out) :: vx, vy, vz

    vx = this % m_vel(1)
    vy = this % m_vel(2)
    vz = this % m_vel(3)

end subroutine particle_get_velocity
```

(continues on next page)

(continued from previous page)

```fortran
...
```

It is now possible to access the instance variables using these subroutines as shown in the following example:

```fortran
real :: x, y, z
...
call particle % get_position(x, y, z)
```

It is also possible to use functions to access the instance variables, as shown in the following code:

```fortran
real :: x, y, z
...
x = particle % x()
```

x() is a Fortran function member of **particle_class**.

2.15.2 Pretty printing

Other functionalities that could be integrated into the class is the ability to pretty print its state variables. Consider the following subroutine:

```fortran
subroutine particle_print(this)

    class(particle_class) :: this

    print*, 'Particle position'
    print*, '-----------------'
    write(*, '(3G10.3)') this % m_pos(1), this % m_pos(2),␣
↪this % m_pos(3)

    print*, ''

    print*, 'Particle velocity'
    print*, '-----------------'
    write(*, '(3G10.3)') this % m_vel(1), this % m_vel(2),␣
↪this % m_vel(3)

```

(continues on next page)

(continued from previous page)

```
end subroutine particle_print
```

This subroutine will enable a user of the class to easily print the instance variables in a nice formatted way without actually accessing the instance variables:

```
call particle % print
```

This code would give the following output:

```
Particle position
-----------------
1.00      1.00      1.00

Particle velocity
-----------------
1.00      1.00      1.00
```

An extension of this could be to provide subroutines for reading and writing the object instance to a file.

2.15.3 Restricting access

In the previous code examples the internal state variables where encapsulated using access methods. However, it is still possible for a user of the instance to access the member variable. This could let to users of the instance modifying the variables directly either willingly or by mistake, which could lead complicated bugs. To solve this Fortran enables the classes to mark these instance variable as private preventing access to them. Adding a **private**-directive in the class declaration hides these variables from the users of the instance, as shown in the following code:

```
type particle_class
private |\hladded|
    real :: m_pos(3)
    real :: m_vel(3)
contains
```

Assigning these variables as shown in the following example,

```
particle % m_pos(1) = 0.0
```

will produce the following compilation error:

```
    particle % m_pos(1) = 0.0
                1
Error: Component 'm_pos' at (1) is a PRIVATE component of
↪'particle_class'
\end{fortrancodeenv}
```

This means that using the **private** in class declaration effectively will prevent any users mistakenly accessing the private instance variables of any classes.

The **private** before the variable declaration will make all variables private. It is also possible to selectively make variables public or private by removing the private declaration and adding the variable attribute **private** to the variable declaration as below:

```
type particle_class
    real, private :: m_pos(3)
    real :: m_vel(3)
```

In this code **m_pos** is private and **m_vel** is public as all variables are public by default.

In a similar way it is possible to prevent access to member subroutines or functions using the **private**- or **public**-attributes on for each subroutine declaration as in the following example:

```
type particle_class
private
    real :: m_pos(3)
    real :: m_vel(3)
contains
    procedure :: init => particle_init
    procedure :: set_position => particle_set_position
    procedure :: set_velocity => particle_set_velocity
    procedure :: get_position => particle_get_position
    procedure :: get_velocity => particle_get_velocity
    procedure :: print => particle_print
    procedure, private :: setup => particle_setup |\hladded|
end type particle_class
```

Here a private method **setup** has been added that can only be called for member subroutines of the **particle**-class.

2.15.4 Extending existing classes

Classes in Fortran can be extended using the special attribute **extends** in its type definition. In the following example the previous **particle_class** is extended by the **sphere_particle_class** to handle a spherical particle.

```
type, extends(particle_class) :: sphere_particle_class
private
    real :: m_radius
contains
    procedure :: set_radius => sphere_set_radius
    procedure :: get_radius => sphere_get_radius
end type sphere_particle_class
```

In the type definition a private member variable, **m_radius** and its access methods, **set_radius** and **get_radius** are added. The access methods are similar to the access method of the **particle_class**.

```
subroutine sphere_set_radius(this, r)

    class(sphere_particle_class) :: this
    real :: r

    this % m_radius = r

end subroutine sphere_set_radius

real function sphere_get_radius(this)

    class(sphere_particle_class) :: this

    get_radius = this % m_radius

end function sphere_get_radius
```

The above type definition, will override the existing **init**-routine from the **particle_class**. This means that the existing initialisation routine will not be called. To solve this, the **sphere_init**-routine needs to call the existing **particle_init**-routine from the extended class. The **sphere_init** initalisation routine is shown below:

```
subroutine sphere_init(this)

    class(sphere_particle_class) :: this

    ! --- Calling inherited init routine.

    call this % particle_class % init()

    this % m_radius = 1.0

end subroutine sphere_init
```

The format of a call to a inherited routine is:

```
call [instance reference] \% [type definition name] \%␣
↪[routine name]
```

CHAPTER

THREE

FORTRAN AND PYTHON

When developing numerical codes today, it is more and more important to be able to combine benefits form many programming languages. One language that have gained popularity in numerical computing i Python. Python is a powerful dynamic scripting language, which is easy to use and combined with the Scipy toolkit it can provide a environment very similar to MATLAB. Python also supports the development of a multitude of different kinds of applications ranging from graphical user interfaces to web interfaces and web services.

By combining Fortran and Python, the performance of Fortran can be combined with the easy of use and flexibility of Python. This chapter describes a method for combining these languages using a special tool, f2py.

3.1 Python extension modules

In addition to implement Python modules in Python, modules can also be implemented in C using a special API, which usually is found in the libpython library. To illustrate how an extension module is developed, a simple sum function will be implemented in an extension module, calcualtions, using the Python extension API.

A typical Python extension module requires 3 parts:

- Exported functions defined using the Python extension API.
- Module function table declaring all exported functions in the module.
- Module initialisation function for initialising the module and function table.

An exported function must be declared in a format that can be understood by Python. Our exported sum function is declared as shown in the following code:

```
static PyObject*
sum(PyObject *self, PyObject *args)
```

The left side declares the return values from the function. This declaration must always be there even if the function does not return anything. Next, the sum function is declared. All exported functions have the same arguments. is a pointer to the module instance that the function belongs to. The second argument is a special Python object containing the arguments that the function is called with.

Next, the input arguments must be parsed. This is done using the -function in the Python API. This function parses the input arguments, , for the required parameters. If no match is found the function returns NULL, which will trigger an exception in the Python interpreter. If a match is found the function will assign values to provided C-variables. Argument parsing for our sum function is shown in the following code:

```
// C variables that will contain input values

double a;
double b;

// Parse input arguments

if (!PyArg_ParseTuple(args, "dd", &a, &b))
    return NULL;
```

First, variables, and , are declared for storing the actual input arguments. The -function takes the input parameter, args, and processes this according to a signature string describing the required Python arguments. Our function sum takes two double values as input arguments, the signature string for this is ”dd”.

Now we have all input data, so now we do our actual computation in C:

```
double c = a + b;
```

To be able to use the computed value in Python it has to be converted to a PyObject. This can be done using the function . This function is similar to the -function as it also uses the signature string to define what Python-datatypes to create. This is used in the last part of the -function to return a Python-datatype.

```
return Py_BuildValue("d", c);
```

The complete sum function then becomes:

```
static PyObject*
sum(PyObject *self, PyObject *args)
{
    double a;
    double b;

    // Parse input arguments

    if (!PyArg_ParseTuple(args, "dd", &a, &b))
        return NULL;

    // Do our computation

    double c = a + b;

    // Return the results

    return Py_BuildValue("d", c);
}
```

To be able to compile this function as an extension module, a function table and module initialisation have to be added. The additional code required is shown below:

```
// Module function table.

static PyMethodDef
module_functions[] = {
    { "sum", sum, METH_VARARGS, "Calculate sum." },
    { NULL }
};

// Module initialisation

void
initcext(void)
{
```

(continues on next page)

(continued from previous page)

```
    Py_InitModule3("cext", module_functions, "A minimal␣
↪module.");
}
```

To build the extension module, the module in NumPy is used. The following is used to build the extension module:

```
from numpy.distutils.core import setup, Extension

setup(
    ext_modules = [
        Extension("cext",
            sources=["calculations_c.c"]),
    ]
)
```

Building the module from the command line is then done using the following command:

```
> python setup.py build
running build
running build_ext
building 'calculations' extension
gcc -fno-strict-aliasing -I/Users/lindemann/anaconda/
↪include -arch x86_64 -DNDEBUG -g -fwrapv -O3 -Wall -
↪Wstrict-prototypes -I/Users/lindemann/anaconda/include/
↪python2.7 -c calculations.c -o build/temp.macosx-10.5-x86_
↪64-2.7/calculations.o
gcc -bundle -undefined dynamic_lookup -L/Users/lindemann/
↪anaconda/lib -arch x86_64 -arch x86_64 build/temp.macosx-
↪10.5-x86_64-2.7/calculations.o -L/Users/lindemann/
↪anaconda/lib -o build/lib.macosx-10.5-x86_64-2.7/
↪calculations.so
```

The following example shows how the module can be used like any other module in Python:

```
>>> import cext
>>> dir(cext)
['__doc__', '__file__', '__name__', '__package__', 'sum']
>>> s = cext.sum(2.0, 3.0)
```

(continues on next page)

(continued from previous page)

```
>>> print s
5.0
>>>
```

3.2 Integrating Fortran in extension modules

To integrate Fortran in a Python extension module, requires us to compile and link Fortran code into the extension module. To illustrate this, the example in the previous section will be modified to call a fortran subroutine to perform the computation. To link a Fortran routine with a C, the calling convention in Fortran must be adapted to C. In the following example the , and is used to define a Fortran routine that uses the C calling convention and C datatypes to make the linking easier:

```
subroutine forsum(a, b, c) bind(C, name='forsum')

    use iso_c_binding

    real(c_double), value :: a, b
    real(c_double)        :: c

    c = a + b

end subroutine forsum
```

The code in the Python extension module is now updated to call the Fortran routine as shown below:

```
static PyObject*
sum(PyObject *self, PyObject *args)
{
    double a;
    double b;
    double c;

    // Parse input arguments

    if (!PyArg_ParseTuple(args, "dd", &a, &b))
```

(continues on next page)

(continued from previous page)

```
        return NULL;

    // Do our computation

    forsum(a, b, &c);

    // Return the results

    return Py_BuildValue("d", c);
}
```

The reason for the & operator is to pass the -variable as a reference to the Fortran routine.

To build the modified extension module, the Fortran routine must be compiled separately and then provided as a -file to the script:

```
from numpy.distutils.core import setup, Extension

setup(
    ext_modules = [
        Extension("fext",
            sources=["fext.c"],
            extra_objects=["forsum.o"])
    ]
)
```

It is also possible to transfer matrices between Fortran and Python. However, it requires even more complicated binding code. Instead of doing this by hand, special tools can be used to automatically generate the binding code for us as well as enabling us to use NumPy arrays to transfer matrices between Fortran and Python in an efficient way.

3.3 F2PY

F2PY is a tool developed by Pearu Peterson that parses Fortran code, generates Python wrapper code and compiles it as a Python extension module. F2PY automatically create wrapper code for Fortran arrays, so that NumPy arrays can be passed directly to the generated functions.

To illustrate the process of generating an extension module with F2PY the following simple Fortran routine will be wrapped as a module:

```fortran
subroutine simple(a,b,c)

    real, intent(in) :: a, b
    real, intent(out) :: c

    c = a + b

end subroutine simple
```

To be able to use F2PY effectively it is important that the -attribute is used on the subroutine arguments. If not specified, F2PY, will treat all subroutine parameters as input-variables and no output parameters can be passed back to the the calling Python routine.

To create a Python module from the source code we execute the -command on the command line as show below:

```
> f2py -m fortmod -c simple.f90
...
3n535b8krwsz88vl8bm0000gn/T/tmp5STblc/src.macosx-10.5-x86_
↪64-2.7/fortranobject.o /var/folders/w6/
↪1zqjp3n535b8krwsz88vl8bm0000gn/T/tmp5STblc/simple.o -L/
↪opt/local/lib/gcc49/gcc/x86_64-apple-darwin14/4.9.1 -L/
↪Users/lindemann/anaconda/lib -lgfortran -o ./fortmod.so
Removing build directory /var/folders/w6/1zq...
```

In the build directory there should now be a or a depending on the platform used.

The new module is loaded and used as shown in the following example:

```python
>>> import fortmod
>>> print fortmod.simple(2.0, 3.0)
5.0
```

F2PY will automatically generate built-in documentation in the module. To display this documentation the property is used, as shown in the following example:

```python
>>> print fortmod.__doc__
This module 'fortmod' is auto-generated with f2py␣
↪(version:2).
```

(continues on next page)

(continued from previous page)

```
Functions:
  c = simple(a,b)
.
>>> print fortmod.simple.__doc__
c = simple(a,b)

Wrapper for ``simple``.

Parameters
----------
a : input float
b : input float

Returns
-------
c : float
```

As show above, F2PY generates documentation both for the generated module as well as for the individual functions.

Already now it is clear that using F2PY is significantly easier that hand-coding Python wrappers for Fortran. F2PY takes care of all the steps.

3.3.1 Passing arrays

F2PY will automatically handle conversion of NumPy arrays when calling a Fortran extension module. However, it is important to note that NumPy by default uses C ordered arrays. These will be automatically converted to Fortran ordered arrays. For smaller arrays the overhead is not so large, but for large arrays the overhead can be significant. To avoid the automatic conversion, NumPy arrays should be created with the option in the array constructor, as shown in the following example:

```
A = ones((10,10), 'f', order='F')
```

Using this option will pass the allocated memory for the NumPy array directly to the Fortran routine without conversion.

3.3.2 A more complete example - Matrix multiplication

To illustrate the use of arrays in a Fortran extension module we create a Fortran subroutine that takes two input arrays and returns the matrix multiplication of these two arrays, The first version of the function is shown below:

```fortran
! A[r,s] * B[s,t] = C[r,t]
subroutine matrix_multiply(A,r,s,B,t,C)
    integer :: r, s, t
    real, intent(in) :: A(r,s)
    real, intent(in) :: B(s,t)
    real, intent(out) :: C(r,t)

    C = matmul(A,B)
end subroutine matrix_multiply
```

Input variables define the sizes of the incoming matrices. We use the Fortran attributes and to tell F2PY what should be treated as an input variable or an output variable. Creating a Fortran extension module with F2PY on the above routine produces the following corresponding Python routine (from the generated documentation):

```
c = matrix_multiply(a,b,[r,s,t])

Wrapper for ``matrix_multiply``.

Parameters
----------
a : input rank-2 array('f') with bounds (r,s)
b : input rank-2 array('f') with bounds (s,t)

Other Parameters
----------------
r : input int, optional
    Default: shape(a,0)
s : input int, optional
    Default: shape(a,1)
t : input int, optional
    Default: shape(b,1)

Returns
-------
```

(continues on next page)

(continued from previous page)

```
c : rank-2 array('f') with bounds (r,t)
```

We can see in the documentation that the syntax of the Python routine is:

```
c = matrix_multiply(a,b,[r,s,t])
```

The Fortran output argument, is returned on the left side and the input arguments, are input parameters to the Fortran routine. Please note that the size input parameters will be provided by the generated function and are not required when calling the routine from Python.

The created extension module can be uses from Python as shown in the following code:

```
from numpy import *
from fortmod import *

A = ones((6,6), 'f', order='F') * 10.0
B = ones((6,6), 'f', order='F') * 20.0

C = matrix_multiply(A, B)

print C
```

Output from the Python code is:

```
[[ 1200.  1200.  1200.  1200.  1200.  1200.]
 [ 1200.  1200.  1200.  1200.  1200.  1200.]
 [ 1200.  1200.  1200.  1200.  1200.  1200.]
 [ 1200.  1200.  1200.  1200.  1200.  1200.]
 [ 1200.  1200.  1200.  1200.  1200.  1200.]
 [ 1200.  1200.  1200.  1200.  1200.  1200.]]
```

Output variables, , from Fortran will be automatically created. It is not possible to reference data in an already existing array as shown in the following example:

```
A = ones((6,6), 'f', order='F') * 10.0
B = ones((6,6), 'f', order='F') * 20.0
C = zeros((6,6), 'f', order='F')

print "id of C before multiply =",id(C)
```

(continues on next page)

(continued from previous page)

```
C = matrix_multiply(A, B)

print "id of C after multiply =",id(C)
```

In this example, an array is created before the call to our Fortran routine. The id or memory location is queried using the and displayed before and after the call. The output is:

```
id of C before multiply = 4299985824
id of C after multiply = 4340070160
```

The array is apparently overwritten. This is due to how the Python language is designed. An euqality operator will replace the reference to the first instance with a new instance. The next section covers how to pass variables that can be modified by Fortran.

3.3.3 Matrix mulitplication with modifiable output variables

If the Fortran extension module should be able to modify the contents of the incoming arrays, the attribute must be used. This tells F2PY to generate code that handles this. Our modified matrix multiplication subroutine then becomes:

```
! A[r,s] * B[s,t] = C[r,t]
subroutine matrix_multiply2(A,r,s,B,t,C)
    integer :: r, s, t
    real, intent(in) :: A(r,s)
    real, intent(in) :: B(s,t)
    real, intent(inout) :: C(r,t)

    C = matmul(A,B)
end subroutine matrix_multiply2
```

The only difference is the attribute on the array declaration. However, the generated Python routine is quite different:

```
matrix_multiply2(a,b,c,[r,s,t])

Wrapper for ``matrix_multiply2``.
```

(continues on next page)

(continued from previous page)

```

Parameters
----------
a : input rank-2 array('f') with bounds (r,s)
b : input rank-2 array('f') with bounds (s,t)
c : in/output rank-2 array('f') with bounds (r,t)

Other Parameters
----------------
r : input int, optional
    Default: shape(a,0)
s : input int, optional
    Default: shape(a,1)
t : input int, optional
    Default: shape(b,1)
```

Now all input parameters are given on the right side. Now it is possible to directly modify the variable in the Fortran code and pass any changes back to Python, without copying the data. The memory address of the array is the same as used by the NumPy array in the Python code. The following code shows how to use the modified Fortran extension:

```
A = ones((6,6), 'f', order='F') * 10.0
B = ones((6,6), 'f', order='F') * 20.0
C = zeros((6,6), 'f', order='F')

print "id of C before multiply =",id(C)

matrix_multiply2(A, B, C)

print "id of C after multiply =",id(C)

print C
```

For this code to work it is now required to create the array, , before calling the Fortran extension. This is due to the fact that the memory area for the array needs to exist before the call as the pointer to the array is passed directly to the Fortran code. The output of the Python code is shown below:

```
id of C before multiply = 4302082976
```

(continues on next page)

(continued from previous page)

```
id of C after multiply = 4302082976
[[ 1200.  1200.  1200.  1200.  1200.  1200.]
 [ 1200.  1200.  1200.  1200.  1200.  1200.]
 [ 1200.  1200.  1200.  1200.  1200.  1200.]
 [ 1200.  1200.  1200.  1200.  1200.  1200.]
 [ 1200.  1200.  1200.  1200.  1200.  1200.]
 [ 1200.  1200.  1200.  1200.  1200.  1200.]]
```

From the output, we can see that the memory of the array is the same before and after the call to the Fortran extension module.

CHAPTER

FOUR

MANAGING FORTRAN PROJECTS

When projects get larger, compiling and maintenance issues can become quite complex. Many projects also needs to be able to build on different platforms and operating environments. Using simple shell scripts often work when the projects are small. However, shell scripts often lack the ability to handle compilation dependencies between source files, requiring a complete rebuild of the project for each modification. Tools such as CMake and Make can solve many of these problems efficiently.

4.1 Make

Make is a tool that can build software according to special rules defined in a makefile. Make automatically handles dependencies between source files and only rebuilds parts of the software that are affected by the change.

A makefile consists of a series of rules, dependencies and actions. The general syntax for a makefile is:

target: [dependencies]
system command

A simple rule to compress a file, myfile.txt, is shown below:

```
myfile.gz: myfile.txt
    cat myfile.txt | gzip > myfile.gz
```

In this example a rule, , is created to compress the file. The rule depends on the file, . The action for compressing the file is shown in the second row. When you run the make command in the directory the following output is shown:

```
$ ls
Makefile    myfile.txt
$ make
cat myfile.txt | gzip > myfile.gz
$ ls
Makefile    myfile.gz   myfile.txt
```

If make is run again it will recognise that the has not been changed and not execute the action again.

```
$ make
make: 'myfile.gz' is up to date.
```

If the is changed, make will recognise this and run the specified action again.

```
$ touch myfile.txt
$ make
cat myfile.txt | gzip > myfile.gz
$
```

4.1.1 Compiling code with make

Using these rules a build system for compiling source code can be implemented. To compile a simple Fortran application the are some steps that needs to be done:

1. Compile the source files to an object files (.o).
2. Link the object files to an executable.

For a single source file 2 rules are needed. One rule for compiling the source file to an object file and one rule for linking the object file to an executable. A simple makefile for a single source file application is shown below:

```
myprog: myprog.o
    gfortran myprog.o -o myprog

myprog.o: myprog.f90
    gfortran -c myprog.f90
```

In the above example the executable, , depends on the object file . The second rule defines how the object file is created from the source file, , which is also listed as a dependency for the rule. Running make on this makefile produces the following output:

```
$ ls
Makefile    myprog.f90
$ make
gfortran -c myprog.f90
gfortran myprog.o -o myprog
```

Make first creates the object file as this is a dependency for the creating the executable. Next, the executable, , is created by using the gfortran compiler to create an executable from the object file. When running make again, make will check for modifications and only execute actions if necessary.

Using make on a single source file is perhaps not the most useful thing. However, when compiling multiple files using make becomes more useful. To extend our above example to multiple source files we add the needed dependencies to the rule for building the executable, . We also need an additional rule for building our additional sourcefile, .

```
myprog: myprog.o mymodule.o
    gfortran myprog.o mymodule.o -o myprog

myprog.o: myprog.f90
    gfortran -c myprog.f90

mymodule.o: mymodule.f90
    gfortran -c mymodule.f90
```

The interesting happens when the file is updated:

```
$ touch mymodule.f90
$ make
gfortran -c mymodule.f90
gfortran myprog.o mymodule.o -o myprog
```

Make detects the change in the file and only compiles this file. As the was not updated the existing object file can be reused. This is why it is a good idea to use make in large projects. Modifying a single source file in a large application will only rebuild what is needed to satisfy the dependencies.

4.1.2 Fortran 90 Module dependencies

One problem compiling Fortran 90 code and modules is module dependencies. When compiling a module the compiler creates -files which can be compared to automatically generated header files in C. When compiling a module which uses another module the used module must be compiled first, so that the -file is available for the compiler.

In the following exaple we have a module, , which uses . If we update the previous makefile we get the following makefile:

```
myprog: module_main.o module_truss.o
    gfortran module_main.o module_truss.o -o myprog

module_main.o: module_main.f90
    gfortran -c module_main.f90

module_truss.o: module_truss.f90
    gfortran -c module_truss.f90
```

Running make produces the following output:

```
$ make
module_main.f90:3.5:

 use truss
     1
Fatal Error: Can't open module file 'truss.mod' for reading␣
↪at (1): No such file or directory
make: *** [module_main.o] Error 1
```

The compiler complains that it is missing the -file, , to be able to compile main module. To solve this an additional dependency, , is added to the build rule. This means that to build the file the file must first be build. The updated make file is shown below:

```
myprog: module_main.o module_truss.o
    gfortran module_main.o module_truss.o -o myprog

module_main.o: module_main.f90 module_truss.o
    gfortran -c module_main.f90

```

(continues on next page)

(continued from previous page)

```
module_truss.o: module_truss.f90
    gfortran -c module_truss.f90
```

Running make again will produce the desired results:

```
$ make
gfortran -c module_truss.f90
gfortran -c module_main.f90
gfortran module_main.o module_truss.o -o myprog
```

From the above output it can be seen that make figures out the dependencies and builds the first which produces the needed which is needed when compiling the file.

4.1.3 Using variables in make

To specify explicit commands in the make file rules can make the makefiles difficult to maintain. Too solve this, make supports variables in the same way as in normal bash-scripts. To use the value of a variable in the makefile, the name of the variable is enclosed in . In the following example, the variable, , is used to specify which compiler that is going to be used. The compiler flags are specified in the variable and the name of the application binary is specified in the variable. In this example a special clean rule has been added to clean all build files generated when compiling the application. In the rule the is used to make the rule more generic.

```
FC=gfortran
FFLAGS=-c
EXECUTABLE=myprog

$(EXECUTABLE): myprog.o mymodule.o
    $(FC) myprog.o mymodule.o -o myprog

myprog.o: myprog.f90
    $(FC) $(FFLAGS) myprog.f90

mymodule.o: mymodule.f90
    $(FC) $(FFLAGS) mymodule.f90
```

(continues on next page)

(continued from previous page)

```
clean:
    rm -rf *.o *.mod $(EXECUTABLE)
```

Running make produces the desired result, but with a more flexible make file.

```
$ make
gfortran -c myprog.f90
gfortran -c mymodule.f90
gfortran myprog.o mymodule.o -o myprog
```

4.1.4 Internal macros

To create even more generic makefiles and rules, make also has some useful internal macros that can be used. The most important internal macros are:

`$@`	The target of the current rule executed.
`$^`	Name of all prerequisites
`$<`	Name of the first prerequisite

Using `$^` and `$@` a more generic build rule for linking our application can be created

```
FC=gfortran
FFLAGS=-c
EXECUTABLE=myprog

$(EXECUTABLE): myprog.o mymodule.o
    $(FC) $^ -o $@
...
```

Here, `$^`, is used to list all prerequisites for this build, . The `$@`, denotes the current target as the output file for the compiler, in this case or .

The rules for compiling source code can also be updated in a similar way:

```
...
myprog.o: myprog.f90
    $(FC) $(FFLAGS) $< -o $@
...
```

Here the $< variable denotes the first prerequisite, . The target macro, $@, is also used to define the outputfile for the compiler.

There are several more internal macros that can be used in makefiles. For more information please see the GNU Make documenation **:raw-latex:`\cite{gnumake12}`**.

4.1.5 Suffix rules

If a project consists of a larger number of source files, a large number of rules must be written. Make, solves this by implementing so called explicit rules. These rules can be regarded as a recipy for how to go from one extension, to another . A explicit rule for compiling a Fortran source file to an object file then becomes:

```
FC=gfortran
FFLAGS=-c
EXECUTABLE=myprog

...

.f90.o:
    $(FC) $(FFLAGS) $< -o $@
```

This rule eliminates all the compilation rules used in the previous sections and makes the makefile more compact. To make the explicit rules work for compiling Fortran code, make needs to now which suffixes are used for Fortran source code. This is done with the special rule . The following example shows the completed makefile with the suffix rule:

```
FC=gfortran
FFLAGS=-c
EXECUTABLE=myprog

$(EXECUTABLE): myprog.o mymodule.o
    $(FC) $^ -o $@

.f90.o:
    $(FC) $(FFLAGS) $< -o $@

clean:
```

(continues on next page)

(continued from previous page)

```
    rm -rf *.o *.mod $(EXECUTABLE)

.SUFFIXES: .f90 .f03 .f .F
```

4.1.6 Wildcard expansion and substitution

Some times it can be beneficial to create lists of files by using wildcards. To do this in make, the `$(wildcard ...)` function can be used. To create a list of f90 source files the following assignment can be used:

```
F90_FILES := $(wildcard *.f90)
```

Please note the `:=` assignment operator used in conjunction with make function calls.

When we have a list of source files, a list of object-files can easily be created by using the function. This uses patterns to substitute the file suffixes from .f90 to .o. The assignment statement then becomes:

```
OBJECTS := $(patsubst %.f90, %.o, $(F90_FILES))
```

The rule to link all object files into an executable then becomes:

```
$(EXECUTABLE): $(OBJECTS)
    $(FC) $^ -o $@
```

This a much more generic rule, which can be reused for other projects without any change.

4.1.7 Pattern rules

The suffix rules defined in the previous section are provided by GNU make for compatibility with older makefiles. The recommended way of implementing suffix rules is using so called pattern rules.

A pattern rules specifies a ”Recipe” for a rule that can handle multiple targets of a specific type. Using the `%` operator in the target specification to match filenames for which the generic rule will apply. A rule to compile Fortran source code to object files is written using pattern rules as follows:

```
%.o: %.f90
    $(FC) $(FFLAGS) $< -o $@
```

This defines a recipe for make how to create an object-file from a .f90 source file. This rule is implicitly used when make encounters an object-file (implicit pattern rule).

The completed makefile with wildcards and pattern rules is shown below:

```
FC=gfortran
FFLAGS=-c
EXECUTABLE=myprog

F90_FILES := $(wildcard *.f90)
OBJECTS := $(patsubst %.f90, %.o, $(F90_FILES))

$(EXECUTABLE): $(OBJECTS)
    $(FC) $^ -o $@

%.o: %.f90
    $(FC) $(FFLAGS) $< -o $@

clean:
    rm -rf *.o *.mod $(EXECUTABLE)
```

Please note that when using pattern rules the is not needed.

Even if the described makefile automatically can compile all source files, dependencies between Fortran 90 modules are not handled. The easiest way of handling module dependencies are to explicitly express these dependencies in the make file. To illustrate this, consider the following example:

myprog.f90 Main fortran program. Uses the mymodule module located in the mymodule.f90 source file.

mymodule.f90 Module mymodule. Uses the myutils module in the myutils.f90 source file.

myutils.f90 Module myutils. Self contained module without dependencies.

To build this example, we need to build myutils.f90 first as the mymodule.f90 needs the myutils.mod file created when myutils.f90 is compiled. To enable this dependency an additional rule is added to our make file:

```
mymodule.o: myutils.o
```

This tells make that the object-file mymodule.o depends on myutils.o and makes sure that it will be built first. If we update the makefile in the previous section to handle this it becomes:

```
FC = gfortran
FFLAGS = -c
EXECUTABLE = myprog

F90_FILES := $(wildcard *.f90)
OBJECTS := $(patsubst %.f90, %.o, $(F90_FILES))

$(EXECUTABLE): $(OBJECTS) $(MODFILES)
    $(FC) $^ -o $@

mymodule.o: myutils.o

%.o %.mod: %.f90
    $(FC) $(FFLAGS) $< -o $@

clean:
    rm -rf *.o *.mod $(EXECUTABLE)
```

When executing this makefile with make, myutils.f90, will be the first target to be built.

```
$ make
gfortran -c myutils.f90 -o myutils.o
gfortran -c mymodule.f90 -o mymodule.o
gfortran -c myprog.f90 -o myprog.o
gfortran mymodule.o myprog.o myutils.o -o myprog
```

For more advanced make file use, the CMake tool is a better tool. CMake is covered in the next section.

4.2 CMake

When projects become large the time needed for maintaining the build system increases. This is often due to the fact that different OS environments needs to be handled in different ways and this has to be included in the makefile. CMake is a tool that can generate targeted makefiles and project files for most existing development environments. CMake works by parsing special files, CMakeLists.txt, and generating the needed makefiles and project files.

4.2.1 Compiling code with cmake

To use CMake, a CMakeLists.txt file has to be created. This is a normal text files with special CMake statements in it. Usually this files starts with a . This prevents the CMakeLists.txt file to be used by a too old cmake. The first actual statement is usually -function defining the name of the project.

```
cmake_minimum_required(VERSION 2.6)
project(simple)
```

The name of the project is not the same as the executable but is used when generating project files for development environments.

CMake by default does not support Fortran, so a special function, -function is used to enable this:

```
enable_language(Fortran)
```

To create an executable the -function is used. This command takes an executable name as the first argument and a list of source files.

```
add_executable(simple myprog.f90)
```

The completed CMakeLists.txt file then becomes:

```
cmake_minimum_required(VERSION 2.6)
project(simple)
enable_language(Fortran)

add_executable(simple myprog.f90)
```

Now when we have a CMakeLists.txt file it is possible to run in the same directory to create the needed makefiles to build the project:

```
$ ls
CMakeLists.txt  myprog.f90
$ cmake .
-- The C compiler identification is GNU 4.2.1
-- The CXX compiler identification is Clang 4.0.0
-- Checking whether C compiler has -isysroot
-- Checking whether C compiler has -isysroot - yes
-- Checking whether C compiler supports OSX deployment␣
↪target flag
-- Checking whether C compiler supports OSX deployment␣
↪target flag - yes
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- The Fortran compiler identification is GNU
-- Check for working Fortran compiler: /opt/local/bin/
↪gfortran
-- Check for working Fortran compiler: /opt/local/bin/
↪gfortran  -- works
-- Detecting Fortran compiler ABI info
-- Detecting Fortran compiler ABI info - done
-- Checking whether /opt/local/bin/gfortran supports␣
↪Fortran 90
-- Checking whether /opt/local/bin/gfortran supports␣
↪Fortran 90 -- yes
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/.../simple
$ ls
CMakeCache.txt      CMakeLists.txt      cmake_install.cmake
CMakeFiles      Makefile        myprog.f90
```

As show in the above output, cmake, has generated a lot of files one of them being a normal makefile. To build the project, the normal make command can be used.

```
$ make
Scanning dependencies of target simple
[100%] Building Fortran object CMakeFiles/simple.dir/myprog.
↪f90.o
Linking Fortran executable simple
[100%] Built target simple
```

CMake generates a lot of files when run. Which can make the source tree quite cluttered. The recommended way of running CMake is to create a separate build directory and generate the build files in this directory. This is done in the following example:

```
$ mkdir build
$ cd build
$ cmake ..
-- The C compiler identification is GNU 4.2.1
.
.
-- Generating done
-- Build files have been written to: /Users/.../simple/build
```

Make is then run in this directory as before. In this approach it is easy to remove the build files by removing the build directory.

4.2.2 Building debug and release versions

By default CMake generates build files for compiling debug versions of an applicaiton. That is using no optimisation and with debug symbols. Controlling the build type can be done by assigning the variable to either or when executing CMake. Variables can be set on the command line by using the switch -D as shown in the following example:

```
$ cmake -D CMAKE_BUILD_TYPE=Release ..
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/lindemann/
↪Development/progsci_book/source/cmake_examples/simple/
↪build
```

4.2.3 Adding library dependencies

In the previous examples the binaries have been built without any library dependencies. To add link dependencies, the can be used. To add the libraries, and as dependencies of the executable, the CMakeList.txt becomes:

```
cmake_minimum_required(VERSION 2.6)
project(simple)
enable_language(Fortran)

add_executable(simple myprog.f90)
target_link_libraries(simple blas m)
```

To show what switches that are actually used when building the executable, the , is set to . This will show the actual commands used during the build.

```
$ mkdir build
$ cd build/
$ cmake -D CMAKE_VERBOSE_MAKEFILE=ON ..
-- The C compiler identification is GNU 4.2.1
-- The CXX compiler identification is Clang 4.0.0
...
-- Generating done
-- Build files have been written ...
$ make
...
/opt/local/bin/gfortran [...]/mymodule.f90.o  -o multiple  -
↪lblas -lm
...
```

Which shows that the libraries have been added to the actual compilation command.

4.2.4 Variables and conditional builds

Often when compiling code under different platforms, special flags and commands have to be used. CMake supports conditional statements in the CMakeLists.txt files to handle these cases. To test for a Unix-build the following if statement can be used:

```
if (UNIX)
    message("This is a Unix build.")
endif (UNIX)
```

is predefined variable that is true when building on Unix-type system. When running CMake on a Unix-type system will print ”This is a Unix build.” on the console.

CMake also has an else-statement. The following code, creates a build target and adds different build options depending on the platform used:

```
if (UNIX)
    add_executable(multiple myprog.f90 mymodule.f90)
    target_link_libraries(multiple blas m)
else (UNIX)
    if (WIN32)
        add_executable(multiple myprog.f90 mymodule.f90)
        target_link_libraries(multiple blas32)
    else (WIN32)
        message("Not supported configuration.")
    endif (WIN32)
endif (UNIX)
```

It is also possible to use variables in CMake. Variables can be both strings and lists of strings. A variable is created by using the -function. The following example shows how a simple string variable is created:

```
set(MYVAR "Hello, world!")
```

To use the actual value of a variable, it has to be preceded by a $ enclosed by curly brackets as shown in the following example:

```
set(MYVAR "Hello, world!")
message(${MYVAR})
```

This will print the contents of the variable, . If not enclosed it will print the name of the variable.

Variables can also be lists of values which can be iterated over. Creating a list is also done using the -function, as shown in this example:

```
set(MYLIST a b c)
message(${MYLIST})
```

Here, , containing 3 strings. The -function will concatenate the items in the list and the resulting output of running cmake will be:

```
$ cmake ..
abc
-- Configuring done
-- Generating done
-- Build files have been written to: ...
```

Using a list variable it is also possible to do an iteration using a -statement, which the following example shows:

```
set(MYLIST a b c)
foreach(i ${MYLIST})
    message(${i})
endforeach(i)
```

Running this using CMake produced the following output:

```
$ cmake ..
a
b
c
-- Configuring done
-- Generating done
-- Build files have been written to: ...
```

4.2.5 Controlling optimisation options

Optimisation options can differ between compilers. To control the optimisation options in CMake, conditional builds using if-statements can be used. First, the used compiler needs to be queried. The path to the actual compiler is stored in the . To create an if-statement on the compiler the compiler command must be extracted from the compiler path. This can be accomplished using the

```
get_filename_component (Fortran_COMPILER_NAME ${CMAKE_
↪Fortran_COMPILER} NAME)
```

This command extracts the filename component of the path and stores it in the variable . Next, an if-statement has to implemented that queries for different compilers. A string comparison can be done using the operator in CMake. Compilation flags for CMake are stored in for release mode flags and for debug flags.

An example fo this king of conditional compilation statement is shown below (from **:raw-latex:`\cite{cmakecond12}`**):

```
if (Fortran_COMPILER_NAME STREQUAL "gfortran")
  set (CMAKE_Fortran_FLAGS_RELEASE "-funroll-all-loops -fno-
↪f2c -O3")
  set (CMAKE_Fortran_FLAGS_DEBUG   "-fno-f2c -O0 -g")
elseif (Fortran_COMPILER_NAME STREQUAL "ifort")
  set (CMAKE_Fortran_FLAGS_RELEASE "-f77rtl -O3")
  set (CMAKE_Fortran_FLAGS_DEBUG   "-f77rtl -O0 -g")
elseif (Fortran_COMPILER_NAME STREQUAL "g77")
  set (CMAKE_Fortran_FLAGS_RELEASE "-funroll-all-loops -fno-
↪f2c -O3 -m32")
  set (CMAKE_Fortran_FLAGS_DEBUG   "-fno-f2c -O0 -g -m32")
else (Fortran_COMPILER_NAME STREQUAL "gfortran")
  message ("No optimized Fortran compiler flags are known,␣
↪we just try -O2...")
  set (CMAKE_Fortran_FLAGS_RELEASE "-O2")
  set (CMAKE_Fortran_FLAGS_DEBUG   "-O0 -g")
endif (Fortran_COMPILER_NAME STREQUAL "gfortran")
```

4.2.6 Generating project files for development environments

CMake is not limited to generating makefiles, it can also generate project files for a number of graphical development environments. Supported generators in CMake can be listed by running the -command without parameters. The following list is produced on a Mac OS X based machine:

Generators

The following generators are available on this platform: Unix Makefiles = Generates standard UNIX makefiles. Xcode = Generate Xcode project files. CodeBlocks - Unix Makefiles = Generates CodeBlocks project files. Eclipse CDT4 - Unix Makefiles = Generates Eclipse CDT 4.0 project files. KDevelop3 = Generates KDevelop 3 project files. KDevelop3 - Unix Makefiles = Generates KDevelop 3 project files.

This lists covers most common development environments for Mac OS X. When running on a Windows machine, generators for Visual Studio and other development environments for that platform will be available as well.

To generate build files for a different generator the -switch is used. In the following example build files for the Eclipse-environment are generated.

```
$ mkdir build_eclipse
$ cd build_eclipse/
$ cmake -G "Eclipse CDT4 - Unix Makefiles" ../multiple/
-- The C compiler identification is GNU 4.2.1
-- The CXX compiler identification is Clang 4.0.0
-- Could not determine Eclipse version, assuming at least 3.
↪6 (Helios). Adjust CMAKE_ECLIPSE_VERSION if this is wrong.
...
-- Generating done
-- Build files have been written to: ...
$ ls -la
total 112
drwxr-xr-x   8 lindemann  staff    272 Aug 29 20:07 .
drwxr-xr-x  13 lindemann  staff    442 Aug 29 20:06 ..
-rw-r--r--   1 lindemann  staff  14343 Aug 29 20:07 .
↪cproject
-rw-r--r--   1 lindemann  staff   5527 Aug 29 20:07 .project
-rw-r--r--   1 lindemann  staff  17808 Aug 29 20:07␣
↪CMakeCache.txt
drwxr-xr-x  21 lindemann  staff    714 Aug 29 20:07␣
↪CMakeFiles
-rw-r--r--   1 lindemann  staff   4770 Aug 29 20:07 Makefile
-rw-r--r--   1 lindemann  staff   1562 Aug 29 20:07 cmake_
↪install.cmake
```

When generation is completed this directory can be added to a Eclipse workspace as a project.

Please note that in the above example we are using a build directory not located in the source tree. This is the recommended way for an Eclipse based project.

CHAPTER

FIVE

QT CREATOR FOR FORTRAN

Qt Creator is a integrated development environment, IDE, for C++ and Qt, but can be easily adapted as a development environment for Fortran using plugins provided together with this book.

The user interface of Qt Creator resembles the one found in commercial alternatives such as Microsoft Visual Studio or Inte Visual Fortran. This chapter gives a short introduction on how to get started with this development enviroment.

To be able to use the Fortran Project templates and compile Fortran code in Qt Creator, require the following pre-requisites:

- A working Fortran compiler available in the search path. On Windows this is best achieved by installing the MinGW packages when installing Qt Creator. On Mac OS X the gfortran compiler is available in the MacPorts or Brew distributions.
- CMake installed and available as a command line tools. On Windows CMake can be installed when installing the Qt Creator environment. On Max OS X these tools are provided withe the MacPorts and Brew distributions.
- Fortran Templates for Qt Creator which can be downloaded here. These templates work for all platforms.

5.1 Starting Qt Creator

Starting Qt Creator can be done from the Start-menu on Windows and from the application launcher on Mac OS X. When Qt Creator has been launched the main windows is shown in Fig. 5.1.

Fig. 5.1: Qt Creator main window

On the left side of the window, 2 toolbars are shown. The top toolbar controls the main program modes of Qt Creator. The lower toolbar is the project build toolbar, which controls how the projects are built and run.

5.2 Qt Creator main program modes

There are 7 program modes in Qt Creator, controlling the workflow of the development environment. The different modes are listed below:

- **Welcome mode** - Shows a welcome screen, providing shortcuts for many of the common operations of the development environment.
- **Edit mode** - This is probarbly the most used mode of the development environment. This mode provides access to the files within a project as well as an source code editor supporting most languages.
- **Design mode** - In this mode user interfaces for Qt can be designed. This mode will not be used for Fortran development.
- **Debug mode** - This mode will be activated when the application is run in debug mode, for interactive debugging your Fortran application.
- **Project mode** - This mode provides access to settings that applies to the current project.
- **Analyse mode** - Provides access to profiling tools. This mode will not be used in this book.
- **Help mode** - Provides access to the online documenation provided by the development environment.

Switching between modes are in many cases done automatically. Debugging an application will automatically switch to debug mode. Creating or opening a project will automatically switch to edit mode.

5.3 Installing Fortran support in Qt Creator

As Qt Creator is not used by default for Fortran development, additional support for Fortran must be added to Qt Creator. We will use the templates and highlighting from the qtcreator-fortran project.

5.3.1 Installing project and highlighting templates

Go to the user configuration directory of Qt Creator

```
cd ~/.config/QtProject/qtcreator
```

Download this archive:

```
wget https://github.com/jonaslindemann/qtcreator-fortran/
↪archive/master.zip
```

Unzip the archive:

```
unzip master.zip
```

Restart Qt Creator.

A video on how to install the extensions are also available on YouTube.

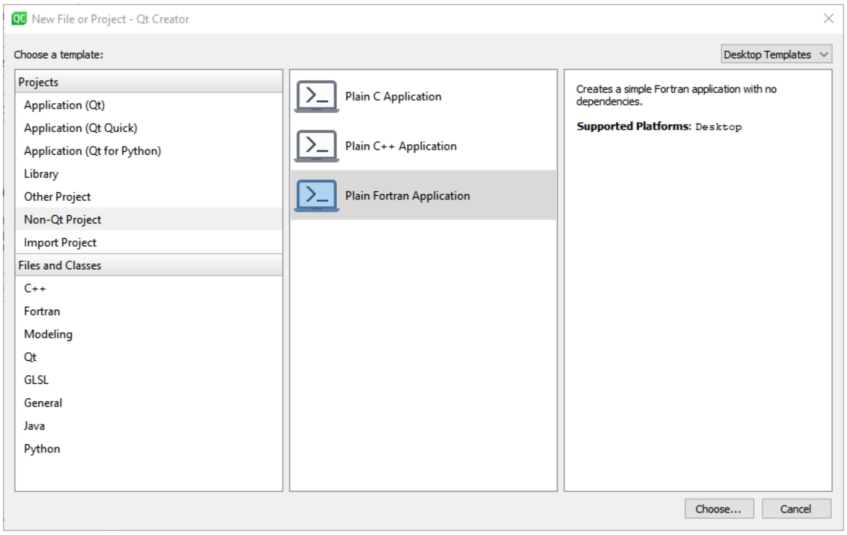
5.4 Creating a Fortran project

To use Qt Creator as a development environment for Fortran, a project has to be created. A project defines, which files that are required for building the program as well as any required settings. By default Qt Creator uses its own custom project format, but can also handle CMake based project, which is also what the installed Fortran plugins use.

To create a new project, select **File/New file or project…*** from the menu. This brings up the project/file creation window as shown in Fig. 5.2.

Next, select **Non-Qt Project** and select the **Plain Fortran application** and click the **Choose…** Button as shown in Fig. 5.3.

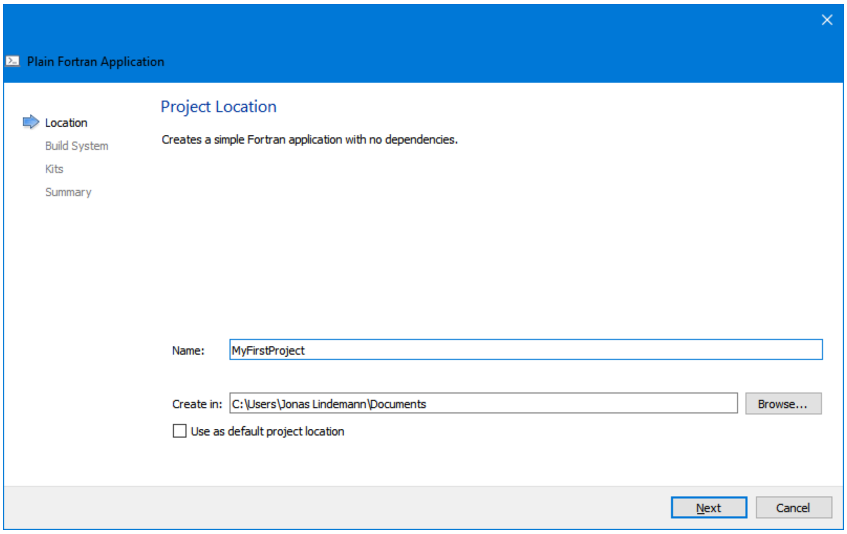
Enter a name and directory for your Fortran application and click **Next**. Next select the build system, CMake, in the next step shown in Fig. 5.4.



Creates a simple Fortran application with no dependencies.

Supported Platforms: `Desktop`

Fig. 5.2: Qt Creator new project/file selection window.



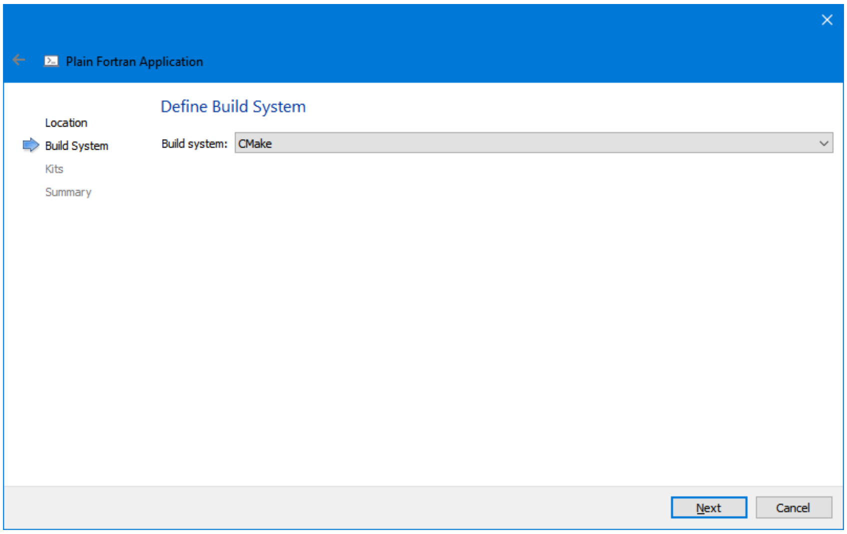
Creates a simple Fortran application with no dependencies.

Name: MyFirstProject

Create in: C:\Users\Jonas Lindemann\Documents

☐ Use as default project location

Fig. 5.3: Choosing a project name.



Define Build System

Build system: CMake

Fig. 5.4: Select build system.

Click **Next** to continue. In the next step we need to select which compiler toolchain to use. In Qt Creator a toolchain is called a kit. Select one of the presented kits as shown in Fig. 5.5.

Click **Next** to continue. In the next step a versioning system can be selected, but this can be skipped, see Fig. 5.6.

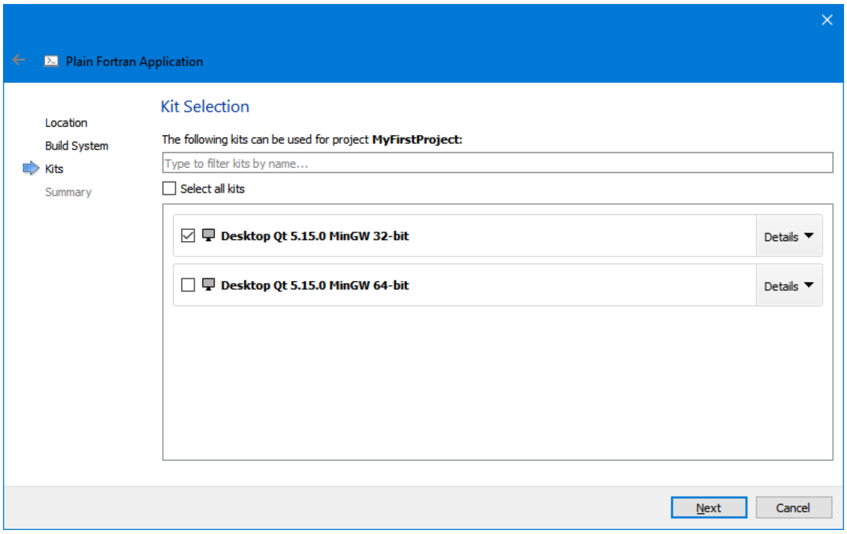
Click **Finish** to finish the project creation. If all worked a new CMake Fortran project should have been created as shown in Fig. 5.7

5.5 Building the project

Building the project is done either by selecting **Build/Build [Project name]** from the menu or using the build button in the bottom left of the window as shown in Fig. 5.8.

The results of the build process is shown in the **Compile output** pane as shown in Fig. 5.9.

Errors during the build will also be shown in the **Compile output** pane. An example of this is shown in Fig. 5.10.



The following kits can be used for project **MyFirstProject:**

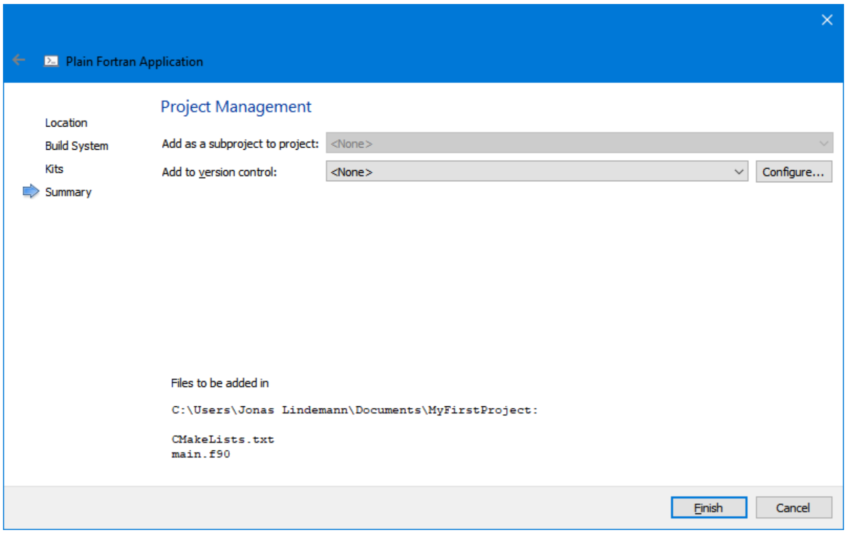
Type to filter kits by name...

☐ Select all kits

☑ **Desktop Qt 5.15.0 MinGW 32-bit** Details

☐ **Desktop Qt 5.15.0 MinGW 64-bit** Details

Fig. 5.5: Select compiler toolchain (Kit).



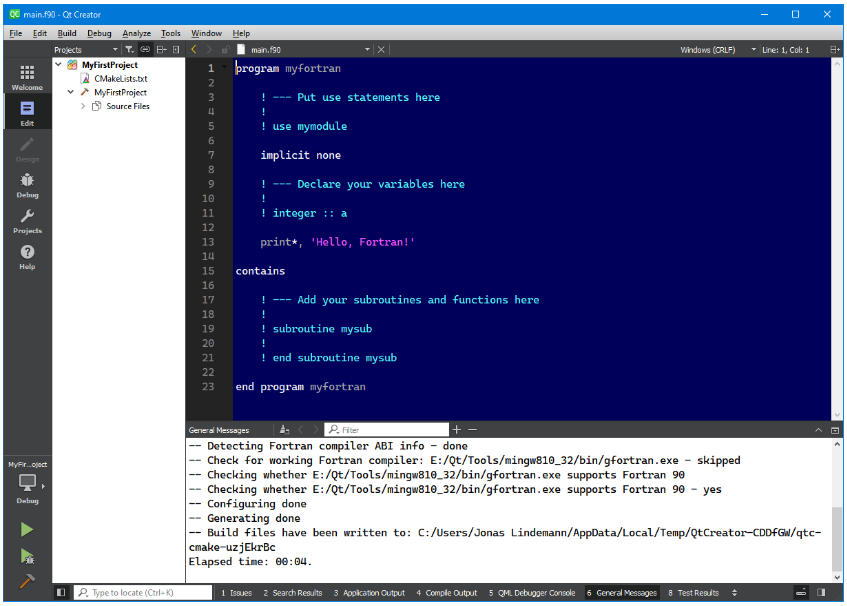
Add as a subproject to project: <None>

Add to version control: <None> Configure...

Files to be added in

```
C:\Users\Jonas Lindemann\Documents\MyFirstProject:

CMakeLists.txt
main.f90
```

Fig. 5.6: Select version control system.

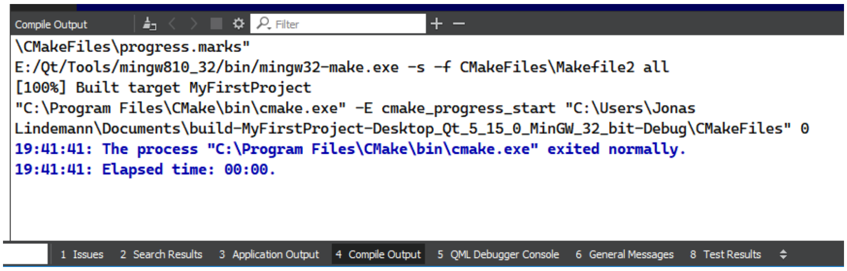


```
program myfortran

    ! --- Put use statements here
    !
    ! use mymodule

    implicit none

    ! --- Declare your variables here
    !
    ! integer :: a

    print*, 'Hello, Fortran!'

contains

    ! --- Add your subroutines and functions here
    !
    ! subroutine mysub
    !
    ! end subroutine mysub

end program myfortran
```

```
-- Detecting Fortran compiler ABI info - done
-- Check for working Fortran compiler: E:/Qt/Tools/mingw810_32/bin/gfortran.exe - skipped
-- Checking whether E:/Qt/Tools/mingw810_32/bin/gfortran.exe supports Fortran 90
-- Checking whether E:/Qt/Tools/mingw810_32/bin/gfortran.exe supports Fortran 90 - yes
-- Configuring done
-- Generating done
-- Build files have been written to: C:/Users/Jonas Lindemann/AppData/Local/Temp/QtCreator-CDDfGW/qtc-
cmake-uzjEkrBc
Elapsed time: 00:04.
```

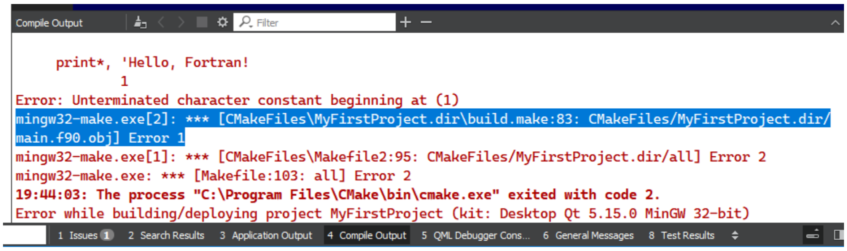
Fig. 5.7: New Fortran project created.

Fig. 5.8: Building a Fortran application.



```
\CMakeFiles\progress.marks"
E:/Qt/Tools/mingw810_32/bin/mingw32-make.exe -s -f CMakeFiles\Makefile2 all
[100%] Built target MyFirstProject
"C:\Program Files\CMake\bin\cmake.exe" -E cmake_progress_start "C:\Users\Jonas
Lindemann\Documents\build-MyFirstProject-Desktop_Qt_5_15_0_MinGW_32_bit-Debug\CMakeFiles" 0
19:41:41: The process "C:\Program Files\CMake\bin\cmake.exe" exited normally.
19:41:41: Elapsed time: 00:00.
```

Fig. 5.9: Output from application compilation.



```
    print*, 'Hello, Fortran!
            1
Error: Unterminated character constant beginning at (1)
mingw32-make.exe[2]: *** [CMakeFiles\MyFirstProject.dir\build.make:83: CMakeFiles/MyFirstProject.dir/
main.f90.obj] Error 1
mingw32-make.exe[1]: *** [CMakeFiles\Makefile2:95: CMakeFiles/MyFirstProject.dir/all] Error 2
mingw32-make.exe: *** [Makefile:103: all] Error 2
19:44:03: The process "C:\Program Files\CMake\bin\cmake.exe" exited with code 2.
Error while building/deploying project MyFirstProject (kit: Desktop Qt 5.15.0 MinGW 32-bit)
```

Fig. 5.10: Compilation error.

5.6 Running the project

Running the finished application can be done by selecting . This will run the first target in the project. If the CMakeLists.txt file contains more targets (add_executable), the selected executable to run can be selected by clicking on the ”Terminal” icon in the lower left toolbar. This brings up a menu in which you can select the target to run as shown in the following figure:

CHAPTER

SIX

APPENDIX 1 - EXERCISES

6.1 Variables and data types

1-1 Which of the following names can be used as Fortran variable names?

a) number_of_stars

b) fortran_is_a_nice_language_to_use

c) 2001_a_space_odyssey

d) more$_money

1-2 Declare the following variables in Fortran: 2 scalar integers a, and b, 3 floating point scalars c, d and e, 2 character strings infile and outfile and a logical variable f.

1-3 Declare a floating point variable *a* that can represent values between 10^{-150} and 10^{150} with 14 significatn numbers.

1-4 What is printed by the following program?

```
program precision

    implicit none

    integer, parameter :: ap = &
    selected_real_kind(15,300)

    real(ap) :: a, b

    a = 1.234567890123456
```

(continues on next page)

(continued from previous page)

```
    b = 1.234567890123456_ap

    if (a==b) then
        write(*,*) 'Values are equal.'
    else
        write(*,*) 'Values are different.'
    endif

    stop

end program precision
```

6.2 Arrays and matrices

2-1 Declare a $[3 \times 3]$ floating point array, **Ke**, and an 3 element integer array, **f**.

2-2 Declare an integer array, **idx**, with the indices [0, 1, 2, 3, 4, 5, 6, 7]

2-3 Give the following assignments:

a) Floating point array, **A**, is assigned the value 5.0 at (2,3).

b) Integer matrix, **C**, is assigned the value 0 at row 2.

2-4 Write a program declaring a floating point matrix, **I**, with the dimensions [10×10] and initialise it with the identity matrix.

2-5 Declare an allocatable 2-dimensional floating point array and a 1-dimensional floating point vector. Also show programstatements how memory for these variables are allocated and deallocated.

2-6 Create a subroutine, **identity**, initialising a *arbitrary* twodimensionl to the identity matrix. Write a program illustrating the use of the subroutine.

2-7 Create a program that:

a) Defines an array to have 100 elements;

b) assigns to the elements the values 1,2,3, . . . , 100;

c) reads two integer values in the range 1 to 100;

d) reverses the order of the elements of the array in the range specified by the two values.

2-8 Given the array declaration

```fortran
real, dimension(50,20) :: a
```

write array sections representing

i) the first row of a;

ii) the last column of a;

iii) every second element in each row and column;

iv) as for (iii) in reverse order in both dimensions;

2-9 Complete excercise **2.8** using array syntax instead of do constructs.

2-10 Create a derived data type for a particle in a particle system. The particle should have the following attributes:

- position
- velocity
- mass

Create an allocatable array with 1000 particles. Initialise all particles with the position (0,0,0) and a random velocity and a mass of 1.0.

6.3 Conditional statements

3-1 Give the following if-statements:

a) If the value of the variable, **i**, is greater than 100 print 'i is greater than 100!'

b) If the value of the logical variable, **extra_filling**, is true print 'Extra filling is ordered.', otherwise print 'No extra filling.'.

3-2 Give a case-statment for the variable, **a**, printing 'a is 1' when a is 1, 'a is between 2 and 20' for values between 1 and 20 and prints 'a is not between 1 and 20' for all other values.

6.4 Repetitive statements

4-1 Write a program consisting of a do-statement 1 to 20 with the control variable, i. For values, i, between 1 till 5, the value of i is printed, otherwise 'i > 5' is printed. The loop is to be terminated when i equals 15.

6.5 Built-in functions

5-1 Give the following expressions in Fortran:

a) $\frac{1}{\sqrt{2}}$

b) $e^x \sin^2 x$

c) $\sqrt{a^2 + b^2}$

d) $|x - y|$

5-2 Give the following matrix and vector expressions in Fortran. Also give appropriate array declarations:

a) $\mathbf{AB}$

b) $\mathbf{A^T A}$

c) $\mathbf{ABC}$

d) $\mathbf{a} \cdot \mathbf{b}$

5-3 Show expressions in Fortran calculating maximum, mininmum, sum and product of the elements of an array.

6.6 Program units and subroutines

6-1 Implement a function returning the value of the the following expression:

$e^x \sin^2 x$

6-2 A subroutine receives as an argument an array of values, x, and the number of elements in x, n. If the mean and variance of the values in x are estimated by

$$mean = \frac{1}{n} \sum_{i=1}^{n} x(i)$$

and

$$variance = \frac{1}{n-1}\sum_{i=1}^{n}(x(i) - mean)^2$$

Write a subroutine which returns these calculated values as arguments. The subroutine check for invalid values of n (<=1). Write a main program that illustrates the use of this subroutine.

6-3 Create a module, **statistics**, containing the functions in 6-2. Change the program in 6-2 to use this module. The module is placed in a separate file, and the main program in **main.f90**.

6.7 Input and output

7-1 Write a program which reads a value, x, and calculates and prints the corresponding value x/(1.+x). The case x=-1 shoud produce an error message and be followed by an attempt to read a new value of x.

7-2 Write a program listing $f(x) = \sin x$ from -1.0 to 1.0 inintervals of 0.1. The output from the program should have the following format:

```
         11111111112222222222223
123456789012345678901234567890
 x      f(x)
-1.000 -0.841
-0.900 -0.783
-0.800 -0.717
-0.700 -0.644
-0.600 -0.565
-0.500 -0.479
-0.400 -0.389
-0.300 -0.296
-0.200 -0.199
-0.100 -0.100
 0.000  0.000
 0.100  0.100
 0.200  0.199
 0.300  0.296
 0.400  0.389
 0.500  0.479
```

(continues on next page)

(continued from previous page)

```
  0.600  0.565
  0.700  0.644
  0.800  0.717
  0.900  0.783
  1.000  0.841
```

7-3 Write a program calculating the total length of a piecewise linear curve. The curve is defined in a textfile line.dat.

The file has the following structure:

```
{number of points n in the file}
{x-coordinate point 1} {y-coordinate point 1}
{x-coordinate point 2} {y-coordinate point 2}
.
.
{x-coordinate point n} {y-coordinate point n}
```

The program must not contain any limitations regarding the number of points in the number of points in the curve read from the file.

6.8 String manipulation

8-1 Declare 3 strings, **c1**, **c2** and **c3** containing the words 'Fortran', 'is' och 'fun'. Merge these into a new string, **c4**, making a complete sentence.

8-2 Write a function converting a string into a floating point value. Write a program illustrating the use of the function. |

6.9 Object-oriented programming

9-1 Implement a derived datatype for a vector and use operator overloading to implement common vector operations such as adding, subtracting and multiplication (cross-product).

CHAPTER

SEVEN

APPENDIX 2 - QUICK FORTRAN COMPILATION GUIDE

This chapters is a quick guide on how to compile simple Fortran application needed for the exercises in this book. The examples uses the gfortran compiler.

7.1 Compiling single source Fortran programs

To compile a simple Fortran 90 source file into an executable, execute the - command with the source file as the only argument:

```
$ ls
myprog.f90
$ gfortran myprog.f90
$ ls
a.out       myprog.f90
```

This produces an executable called , which can be executed using the following command:

```
$ ./a.out
 Hello, World!
```

By default, the name of the executable will be , which is not always the best name for an executable. To tell the compiler to name the executable to something more meaningful the command line switch, , can be used as shown in the following example:

```
$ gfortran myprog.f90 -o myprog
$ ls
myprog      myprog.f90
$ ./myprog
 Hello, World!
```

7.2 Compiling multi-source Fortran programs

Often a Fortran application consists of multiple source files. To compile multiple source files, gfortran supports adding additional source files as parameters on the command line as shown below:

```
$ gfortran mymodule.f90 myprog.f90 -o myprog
$ ls
mymodule.f90    mymodule.mod    myprog      myprog.f90
$ ./myprog
 Hello, World!
```

However, the order of the source files are important. If a module depends on a another module the dependent module needs to be built first as the first module uses a special -file generated during the compilation.

7.3 Compiling with optimisation levels

By default, gfortran, does not optimise the generated code in anyway. To increase performance optimisation options must be specified. There are 3 levels default optimisation available in the , and compiler command line options.

- O1 - Tries to reduce code size and execution time, but without increasing compilation time.
- O2 - Adds allmost all optimisation that does not increase the code size. No loop unrolling is done.
- O3 - Applies all optimisation options even those that increases code size. Loop unrolling is done.

To compile with optimisation just add the above options as the first option to the compiler command as shown in the following example:

```
$ gfortran -O3 mymodule.f90 myprog.f90 -o myprog
```

7.4 Compiling for debugging

To debug a compiled executable using gdb or a graphical debugger, the executable needs to have debugging information included in the binary. By default gfortran does not add any debugging information in the executable. To tell the compiler to include this in the binary, the -switch must be used. The following commands show how a debug enabled executable is built:

```
$ gfortran -g mymodule.f90 myprog.f90 -o myprog
```

It is possible to add debug information to an optimised code as well. However, the execution path of an optimised executable is not always obvious. It is also possible that certain variables have been eliminated by the optimisation options of the compiler.

7.5 Compiling with more detailed code checking

Gfortran by default does not report all code issues in the source code. The level of code checks can be increased by using the or switches. The option warns of the use of extensions to the used Fortran standard (by default Fortran 95). Can also be used together with the switches, , and to check for extension to other Fortran standards. In the below example the code is compiled with the option:

```
$ gfortran -pedantic mymodule.f90 myprog.f90 -o myprog
myprog.f90:3.1:

 implicit none
 1
Warning: Nonconforming tab character at (1)
myprog.f90:5.1:

 print*, 'Hello, World!'
 1
Warning: Nonconforming tab character at (1)
```

In the above example, gfortran complains about the use of a tab-character in the source files. The tab-character is not part of the Fortran standard.

The switch tells the compiler to check for code practices that should be avoided. The example below shows how it can be used:

```
$ gfortran -Wall mymodule.f90 myprog.f90 -o myprog
myprog.f90:3.1:

 implicit none
 1
Warning: Nonconforming tab character at (1)
myprog.f90:5.1:

 print*, 'Hello, World!'
 1
Warning: Nonconforming tab character at (1)
```

7.6 Compiling with runtime checks

Some application errors can not be detected at compile time. To check for these errors, gfortran can add checks in the executable for these. To enable a certain check the -switch can be used to enable specific checks. Common checks are:

- Accessing array elements outside its bounds.
- Modification of loop variables.
- Memory allocation and deallocation.
- Runtime checks for pointer handling.
- Check for when an array-temporary has to be created for pass-
  ing an

  argument.
- Add all available runtime checks.

It is important to remove these checks in the final code as they add an additional overhead in the execution speed.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search