Appendix A

Fortran

A.1 Lineage

Fortran is one of the oldest programming languages still being used (and one of the oldest at all), see Fig. A.1.

However, while being backward-compatible to *Fortran 77*, the current versions *Fortran 90* and *Fortran 95* [1] are modern programming languages (more modern than e.g. *C*) and have not too much in common with the old versions of *Fortran* from the punch-card era — unless you insist on an outdated coding style.

In this course, we will actively use *F90/F95* (the differences are minor), while often comparing to *F77* for reference. Many codes and subroutines in computational physics are written in *F77*, so you should be able to read (and use) *F77* routines.

A.2 Basic language structure

Fortran is

not case sensitive: A variable `time` is the same as `Time`, `TIME` or even `tImE`

> → You cannot use `t` for time and `T` for temperature in the same program or subroutine — better use more descriptive names `time` and `Temp`

statically typed: Every variable has a data type the cannot change during program execution.

> → Even if you do not declare a variable, it will still have a type. Better control this and declare all variables.

[1] Henceforth, we will shortly call them *F77*, *F90* and *F95*; also we will not differentiate between *F90* and *F95* because the differences are small and irrelevant to us here.

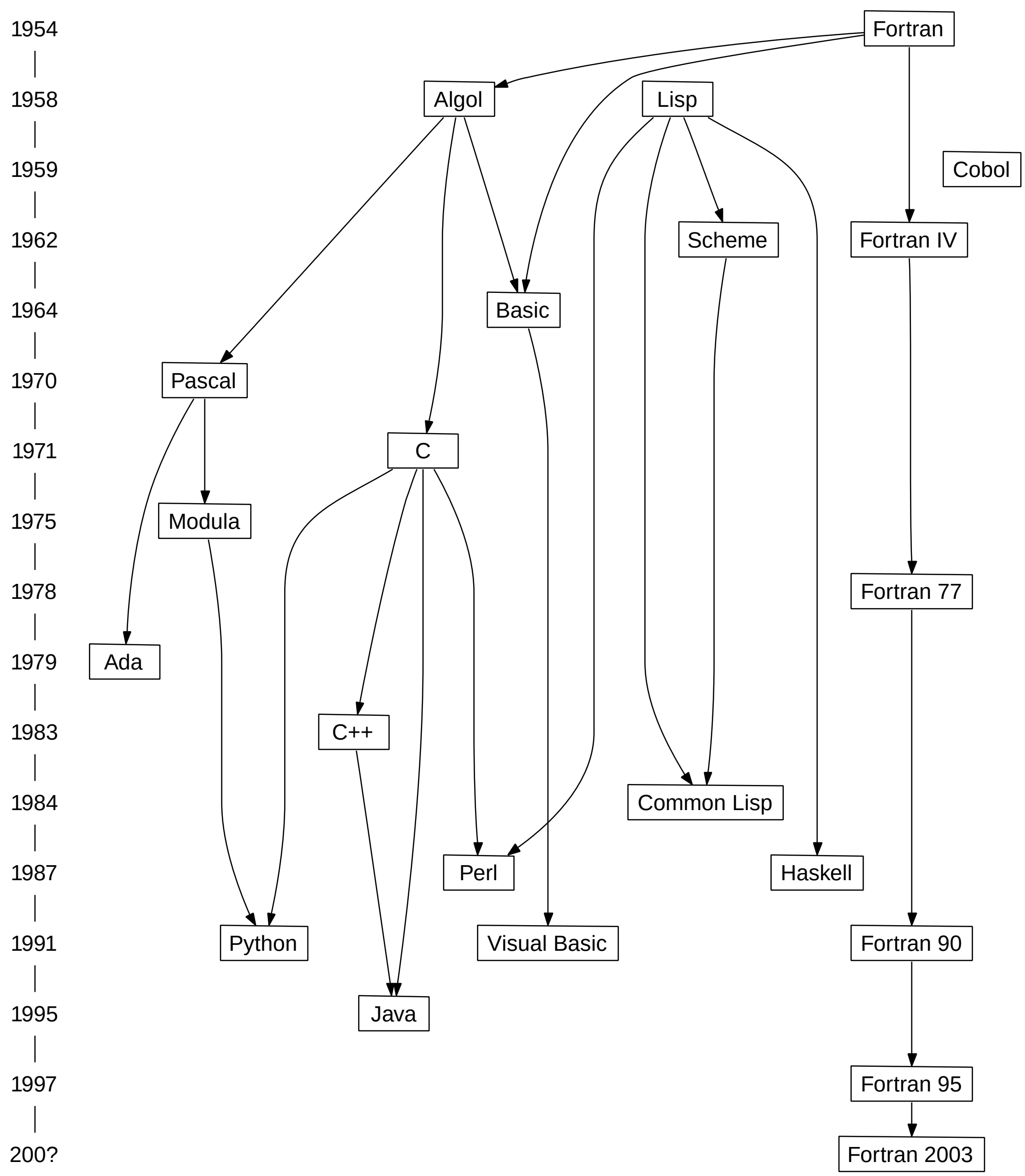

Figure A.1: Genealogy of some programming languages

call by reference: You can modify *any* argument of your functions and subroutines — this often happens inadvertently.

To protect yourself, use the '`intent`' statement ['`intent(in)`', '`intent(iout)`', and '`intent(inout)`', see § A.2.5 below].

line oriented: It will make a difference if you split a line or join two consecutive lines. While *F77* was also column oriented, *F90* has done away with this (apart from the limitation that lines must be shorter than 132 characters).

You can combine several lines with the '`;`' character.

A.2.1 Hello world example

Here is about the simplest *Fortran* program one can make up:

F77

```
! simple.f
!  A simple F77 program
␣␣␣␣␣␣program Hello
␣␣␣␣␣␣    print*, "Hello world"
␣␣␣␣␣␣end
```

F90

```
! simple.f90
! A simple F90 program
program Hello
    print*, "Hello world"
endprogram Hello
```

Note: By convention, *F77* program files have the suffix '`.f`', while *F90* or *F95* files have the suffix '`.f90`'. Many compilers implicitly assume this convention, so if you are trying to be original, you will encounter problems.[2]

Note: Fortran 77 requires all program text (anything apart from comments and labels) to start in the 7th column or later. A character in the first column of a line makes that line a comment. In the following, we will normally not highlight (and often not even show) the initial six columns any more.

Fortran 90 is no longer column oriented. Comments start with an exclamation mark and end at the end of line.

Note: If a line ends in the '`&`' character (which can be followed by whitespace), the following line is a *continuation line*, i.e. it continues the current line. For example,

[2] On the other hand, there is at least one silly compiler that needs to be told about these suffixes.

F77

```
␣␣␣␣␣␣print*, "Hello world, ",
␣␣␣␣␣&        "here I am, "
␣␣␣␣␣&        "and here is Pi: ",
␣␣␣␣␣&        4*atan(1.)
```

F90

```
print*, "Hello world, ",    &
        "here I am, "       &
        "and here is Pi: ", &
        4*atan(1.)
```

is just one command line. As you can see from the example, *F77* uses (an arbitrary non-blank character) the fifth column to mark continuation lines.

Note: The semicolon character ('&') key can be used to combine several short statements into one line:

F90

```
print*, "a"; print*, 'b'
if (x<0) then; y=x; else; y=-x; endif
```

A.2.2 Data types

Table A.1: Basic data types in *Fortran*

Type	*F77*	*F90*	Examples
character (1 byte)	`character`	`character`	`"a","; ",'"',"'"`
string (sequence of *N* characters)	`character*N`	`character(LEN=N)`	`"T'was brillig"`
logical (4 byte)	`logical`	`logical`	`.true., .false.`

integer (4 byte)	`integer`
	`integer*4`
		`integer`
		`integer*4`
		`integer(kind=...)`
			`0, -1,`
			`1234567890`

real (4 byte)	`real` `real*4`	`real` `real*4`	`0., -1.0, .5772176,` `6.67E-11`

double (8 byte)	`double precision`
	`real*8`
		`double precision`
		`real*8`
		`real(kind=...)`
			`0D0, -1.D,`
			`5.772176D-1,`
			`1.23D-128`

complex (4+4=8 byte)	`complex`	`complex` `complex(kind=...)`	`(.707, -.707),` `(0., 3.1415)`
complex (8+8=16 byte)	`complex`	`complex` `complex(kind=...)`	`(7.07D-1, -.707D0),` `(0.D, 3.1415D)`

Note: *Fortran* has inherited an implicit typing system: Unless declared otherwise, variables starting with a letter from `i` to `n` are of type *`integer`*, all other variables are *`real`*. This was very convenient in the punch-card era; nowadays, however, you should *always* declare the data type of all your variables, or you are asking for unnecessary trouble. Most *Fortran* compilers have a switch "`-u`" (or "`-Wimplicit`", "`-implicitnone`" or similar) that enforces explicit declaration of all variables. It is also good practise to put the line

```
implicit none
```

into all of your *Fortran* files.

Note: *Fortran 90* has a new way of choosing the data type that matches your requirements (number of digits, range). Here is a little example:

F90

```
integer, parameter :: digits=12, range=100
integer, parameter :: kr=selected_real_kind(digits,range)

integer, parameter :: irange=12
integer, parameter :: ki=selected_int_kind(irange)

! declare 3 vars with >= 12 digits and range at least 10^-100 to 10^100
real(KIND=kr)    :: x=3.1415926536_kr_12_100,y,z

! declare three integer vars with >= 12 digits
integer(KIND=ki) :: i,j,k
```

While this is an elegant approach (although in real life there are some drawbacks to this scheme), we will not use '`kind`' to specify data types in this course.

Type conversion

To convert data to a different type, use

int convert to integer (rounding towards 0)

nint convert to integer (nearest integer)

floor convert to integer (nearest integer $\leq x$)

ceiling convert to integer (nearest integer $\geq x$)

real convert to real

dble convert to double precision

cmplx convert to complex

Functions related to the number model

There are a number of useful functions that give you information about capabilities and features of the numbers you are using.

huge largest number that can be represented by the given data type ($\approx 3.4 \times 10^{38}$ for single precision floating-point numbers)

tiny smallest positive number that can be represented ($\approx 1.2 \times 10^{-38}$ for single precision floating-point numbers)

epsilon smallest positive number that makes a difference when added to 1. ($\approx 1.2 \times 10^{-7}$ for single precision floating-point numbers)

precision Number of decimals (≈ 7 for single precision floating-point numbers)

range Half range of decimal exponent (≈ 37 for single precision floating-point numbers, i.e. numbers between about 10^{-37} and 10^{37} can be represented)

nearest Nearest neighbour to argument x in positive or negative direction. '`nearest(10.,+1.) - 10.`' should give about `epsilon(1.)*10`.

These functions are useful e.g. when you want an iteration to give maximum accuracy at both single and double precision. If you make the threshold error a few `epsilon(x)`, the accuracy will automatically be adjusted depending on the data type of `x`.

A.2.3 Control structures

`if–then–else` and `select–case`

Short form:

```
if (condition) statement
```

Block form with else branch:

```
if (condition) then
    yes_block
else
    no_block
endif
```

Block form without else branch:

```
if (condition) then
    yes_block
endif
```

Examples

F90

```
if (x == 0) print*, 'Zero'

if (x < 0) then
    print*, 'Negative'
else
    print*, 'Non-negative'
endif

if (((x<0) .and. (y<0)) .or. ((x>0) .and. (y>0))) then
    print*, 'Equal signs'
endif

if ((x*y>=0) .and. .not. (x==0))
    arg = atan(y/x)
elseif (x==0) then
    arg = 0.5*pi*sign(1.,y)
else
    ! more to fix
endif
```

Notes: The following operators compare numbers:

F90 operator	:	'`==`'	'`/=`'	'`<`'	'`<=`'	'`>`'	'`>=`'
F77 operator	:	'`.eq.`'	'`.ne.`'	'`.lt.`'	'`.le.`'	'`.gt.`'	'`.ge.`'
Tests for	:	equality	inequality	$<$	$\leq$	$>$	$\geq$

Logical *and*, *or* and *negation* are represented by the operators '`.and.`', '`.or.`', and '`.not.`'.

To check several exclusive conditions, we can use

F90

```
if (condition1) then
    [...]
```

```
elseif (condition2)
    [...]
elseif (condition3)
    [...]
else
    [execute this if none of the conditions matched]
endif
```

If we are testing for certain values, it is more convenient to use the `select–case` statement:

F90

```
select case (i)
  case (0)
      print*, 'Zero'
  case (1:9)
      print*, 'Positive'
      print*, 'One digit only'
  case (11,13,17,19)
      print*, 'Two-digit prime'
  case default
      print*, 'Nothing special'
endselect
```

`do` loops

To count from 1 to 10, use

F77

```
      integer i

      do 123 i=1,10
          print*, 'i=', i
123   continue
```

F90

```
integer :: i

do i=1,10
    print*, 'i=', i
enddo
```

In F77, the number 123 is a *label* and is put in columns 2 to 5. The '`continue`' statement is a no-op command to attach the label to. In modern variants of *F77*, it can probably be replaced by '`enddo`'.

To count in steps of 3, use

F77

```
      integer i

      do 124 i=1,10,3
          print*, 'i=', i
124   enddo
```

F90

```
integer :: i

do i=1,10,3
    print*, 'i=', i
enddo
```

A *while* loop works like this:

F77

```
      i = 20
      do 126 while (i>10)
          print*, 'i=', i
          i = i-2+floor(sin(i*1.))
126   continue
```

F90

```
i = 20
do while (i>10)
    print*, 'i=', i
    i = i - 2 + floor(sin(i*1.))
enddo
```

All do loops can be left via '`exit`' and '`cycle`' (see § A.2.3 below). This can be used to build an *until* loop:

F90

```
do
    [...]
    if (condition) exit
enddo
```

Exiting control loops

A '`do`' loop can be exited or short-circuited using the '`exit`' and the '`cycle`' statement. While '`exit`' leaves the innermost loop (unless given a label, see below) and continues after the '`enddo`' command, '`cycle`' jumps back to the beginning of the loop and starts the next loop cycle (unless this was already the last one).

F90

```
prime = .true.
do i=2,floor(sqrt(1.*N))
  !
  ! Don't check even divisors > 2
  ! This is quite a stupid test (no gain in efficiency),
  ! but should work
  if (mod(i,2) == 0 .and. i > 2) then
    cycle
  endif
```

```
  !
  ! Check for other divisors
  if (mod(N,i) == 0) then
    print*, 'found divisor ', i
    prime = .false.
    exit
  endif
enddo

if (prime) then
  print*, N, 'is a prime'
else
  print*, N, 'is no prime'
endif
```

Named loops

You can attach a *name* to a loop to make it clearer what the '`cycle`', '`exit`', or '`enddo`' commands refer to. If you have nested loops, naming them allows you to chose which loop you want to '`exit`' or '`cycle`':

F90

```
outer: do i=1,ny
  inner: do k=1,nx
    [do something complicated]
    if (x<27) cycle outer
    [do something complicated]
    if (x>129) exit inner
    [do something complicated]
  enddo inner
enddo outer
```

Exiting the program

Use '`stop`' to exit the program:

F90

```
read(*,*) i
if (i == -1) STOP, "Read -1 -- exiting"

call sub(i)
[...]
```

A.2.4 Input and output

The simplest way of writing and reading is to use the default units and formats (see below) with 'print*' and 'read*':

F90

```
print*, 'Please give me a number:'
read*, x
print*, 'The result is ', sqrt(x**2+y**2), ' unless I am wrong'
```

If you want more control over how the data are formatted or where they are written from/to, use '('write) and 'read'. These commands normally take the form

```
read(unit,format) arg1, arg2, ... argN
write(unit,format) arg1, arg2, ... argN
```

The *unit* is a number that identifies a serial file or stream. By convention, '*' denotes *stdout* (standard output) for 'write' and *stdin* (standard input) for 'read'. As for the numerical unit numbers, 0 denotes *stderr*, 5 denotes *stdin*, and 6 denotes *stdout*.

The *format* allows to specify in detail how numbers or characters are printed. The default format * is guaranteed to print any printable number. If you specify your own format and the number of digits is too low to represent the variable to be printed, the corresponding filed will just print as '******' (or such), rather than becoming wider to accommodate the value (as *C* would).

F90

```
write(*,*) 'Please give me a number:'
read(*,*) x
write(*,*) 'The result is ', sqrt(x**2+y**2), ' unless I am wrong'
```

This does practically the same as the last example, because we have chosen the default unit and format.

Note that 'print*' and 'read*' are followed by a comma, while 'read()' and 'write()' are not.

Formats are strings (either variables declared with 'character(LEN=...)' or string constants) that have to be enclosed in brackets, e.g. '(I10)'.

Note: When using the 'E' or 'G' formatting code, you will want prepend '1p', or the numbers will look strange (0.271828183E1 instead of 2.71828183E0). *If you do this, don't*

Table A.2: Important formatting codes for (input and) output

Code	*Data type*	*Description*
`A`*w*	character	*w*: number of characters
`I`*w*	integer	*w*: total number of characters (digits + sign)
`F`*w.d*	float/double	*w*: total number of characters (sign + digits + decimal point) *d*: number of decimals after comma
`E`*w.d*	float/double	*nw*: total number of characters (sign + digits + decimal point + exponent with '`E`' and sign) *d*: number of significant digits
`D`*w.d*	float/double	basically like '`E`'
`G`*w.d*	float/double	like '`F`' if the width *w* accommodates *d* significant digits like '`E`' else
`L`*n.d*	logical	*w*: number of characters

forget to switch back with '`0p`' afterwards, or '`F`' formatting codes (in the same format line) will print their numbers multiplied by 10.

Example:

F90

```
real ::  e=2.71828183, pi=3.14159265359, three=3.
integer :: i=1234567
character(LEN=80) :: fmt1,fmt2,fmt3

print*, 'e=', e, ', pi=', pi

write(*,'(I10)') i
write(*,'(A5,I10)') 'i = ', i
write(*,'("i = ",I10)') i

write(*,'(F10.3)') e
write(*,'(A5,F10.3)') 'e = ', e
write(*,'("e = ",F10.3)') e

fmt1 = '("pi = ",F10.3))'
write(*,fmt1) pi

fmt2 = '("i =", I10, ",  (e, pi) =", 2(F10.4," "))'
write(*,fmt2) i, e, pi

fmt3 = '("i =", I10, ",  (e, pi) =", 2(1pG12.4," "),0p ", 3=", F10.4)'
write(*,fmt3) i, e*1e20, pi*1e20, three
```

Opening and closing files

In the simplest case, you do

F90

```
program Io_Simple

  real ::  e=2.71828183, pi=3.14159265359, three=3.
  integer :: i=1234567
  character(LEN=80) :: file='test.dat', fmt


  fmt = '(A6,F10.3)'

  open(1,FILE=file)            ! use unit 1 for this file

  write(1,fmt) 'pi = ',pi      ! write first record
  write(1,fmt) 'e = ',e        ! write second record
  write(1,*) 'i = ', i         ! third record using default format

  write(1,FMT=fmt,ADVANCE='NO') 'e = ', e ! start fourth record
  write(1,fmt,ADVANCE='NO') ', pi = ', pi ! start fourth record
  write(1,*) ', i = ', i       ! finish fourth record

  close(1)

endprogram Io_Simple
```

Note the 'ADVANCE='No'' keyword when you want to write without appending a newline (so you can continue that line in further write commands).

A.2.5 Functions

Functions return a value (and thus have a data type) and may have *side effects*, i.e. modify their arguments.

F90

```
real function log11(x)

    implicit none
    real :: x
    intent(in) :: x        ! prevent me from accidentally modifying x

    log11 = log(x)/log(11.)
```

```
endfunction
```

or

F90

```
function log17(x)

    implicit none
    real :: log17, x
    intent(in) :: x        ! prevent me from accidentally modifying x

    log17 = log(x)/log(17.)

endfunction
```

You can use another name for the return value, and you can return before the end of the block:

F90

```
function log17(x) result(res)

    implicit none
    real :: res, x
    intent(in) :: x        ! prevent me from accidentally modifying x

    if (x <= 0) then
        print*, 'Are you kidding me?'
        res = -huge(1.)
        return
    endif
    res = log(x)/log(17.)

endfunction
```

A.2.6 Using functions

Functions are essentially used like variables:

F90

```
y = log11(x)+sin(log17(x-3)**2)
```

If you have both the function definition and the program in one file, you can use `contains` to make the function an *internal function* of the program (or module):

F90

```
program Combined

    implicit none
    real :: x,y

    x = 5.
    y = log17(x)+sin(log17(x-3)**2)

    print*, 'x,y = ', x, y

contains  ! What follows are functions (in this case just one)
          ! and subroutines (in this case none) that are internal to
          ! this module.

    function log17(x)

        real :: log17, x
        intent(in) :: x     ! prevent me from accidentally modifying x

        log17 = log(x)/log(17.)

    endfunction log17

endprogram Combined
```

Note that the function block does not need an `implicit none` statement here, since the `implicit` statement of the `program` holds until the `endprogram`.

Alternatively, you can have the function definition outside the main program unit, but this is less convenient as you will have to declare the function type in the `program` block:

F90

```
function log17(x)

    implicit none
    real :: log17, x
    intent(in) :: x        ! prevent me from accidentally modifying x

    log17 = log(x)/log(17.)

endfunction log17
```

```
program Separate

    implicit none
    real :: x,y
    real :: log17             ! You _need_ to declare the type of
                              ! log17() here
    x = 5.
    y = log17(x)+sin(log17(x-3)**2)

    print*, 'x,y = ', x, y

endprogram Separate
```

A.2.7 Subroutines

Subroutines are similar to functions, but act only through their side effects.

They are used with the '`call`' statement.

F90

```
subroutine sanitize(x,y)
    !
    ! Make sure, x is non-negative and |y| not too large
    !
    implicit none
    real :: x,y
    intent(inout) :: x, y

    if (x < 0.) x = 0.
    if (abs(y) > 100.) y = 1e4/y

endsubroutine sanitize

program Test

    implicit none

    real :: a=-3.4, b=123.
    call sanitize(a,b)
    print*, 'a = ', a, ' , b = ', b
```

```
endprogram Test
```

A.2.8 Key words and optional arguments

Function and subroutine arguments can be accessed by order (as above) or by name (which allow you to change their order):

F90

```
call sanitize(Y=123., X=-3.4)
```

This makes some function calls much more transparent if you use descriptive names for the function arguments.

If you specify an argument to be '`optional`', it can be omitted when the function or subroutine is called. Use the '`present`' logical function to verify whether it was present in the call:

F90

```
subroutine sanitize(x,y,z)
    !
    ! Make sure, x is non-negative and |y| not too large
    !
    implicit none
    real :: x, y
    real, optional :: z
    intent(inout) :: x, y
    intent(in) :: z

    if (x < 0.) x = 0.
    if (abs(y) > 100.) y = 1e4/y
    if (present(z)) then
        x = x*z
        y = y/z
    endif

endsubroutine sanitize

program Test

    implicit none

    real :: a=-3.4, b=123., c=22.414
    call sanitize(a,b)
```

```
    print*, 'a = ', a, ' , b = ', b
    call sanitize(a,b,c)
    print*, 'a = ', a, ' , b = ', b

endprogram Test
```

A.3 Miscellaneous topics

A.3.1 Constants

The value of a *constant* can not be changed. To declare a constant, use the '`parameter`' keyword:

F90

```
integer, parameter :: N=17
real, parameter :: pi=4*atan(1.) ! only works with some compilers

real, dimension(N,N) :: a
```

As you see, you can use the constant N in the declaration of the array a. This would not (normally) work with a variable.

A.3.2 Strings

Strings are treated as character arrays and must have a length pre-specified. Many functions (in particular string comparison) ignore trailing space characters, which is almost always what you want.

You can concatenate strings using '//', trim trailing space with the '`trim`' function, and access substrings using array slice syntax (see below):

F90

```
character(LEN=80) :: name, first='Severus', last='Snape'

name = trim(first) // ' ' // trim(last)
print*, 'Full name: ', name
first = name(1:7)
last = name(9:)
print*, 'First name: ', first
print*, 'Last name: ', last
```

String functions

Some useful string functions are

repeat repeat a string: '`line = repeat("-", 70)`'

trim remove trailing whitespace from a string

len length of a string (including trailing whitespace)

trimlen length of a string excluding trailing whitespace

index, scan find characters or substrings within

A.3.3 Mathematical operators and functions

The operators '+', '−' '*' and '/' do what you expect (but see below). Exponentiation is represented by the '**' operator (using '^' will result in a compilation error).

One point to be wary of is that if both operands are integers, these operators will do *integer arithmetics*, which can sometimes be surprising. Compare the following:

F90

```
print*, "2/3 = ", 2/3, ", 123456789**2 = ", 123456789**2
```

will print

```
2/3 =  0 ,  123456789**2 =  -1757895751
```

while

F90

```
print*, "2./3 = ", 2./3, ", 1.23456789e8**2 = ", 1.23456789e8**2
```

prints

```
2./3. =  0.6666667 ,  1.23456789E8**2 =  1.524158E+16
```

Important mathematical functions

abs absolute value

sqrt square root

log, log10 natural and decadic logarithm

exp exponential function

sin, cos, tan trigonometric functions

asin, acos, atan cyclometric functions

atan2 'atan2(y,x)' gives the argument (phase angle) of the complex number $x + iy$.[3]

sinh, cosh, tanh hyperbolic functions

aimag imaginary part of complex number

conjg conjugate complex of complex number

mod, modulo remainder after division

sign copy sign: sign(x,y) returns $|x| \operatorname{sgn} y$

Random numbers

Fortran 90 has a built-in random number generator, which produces numbers x in the range $0 \le x < 1$. To get one random number, just call the subroutine `random_number()`:

F90

```
implicit none
real :: x

call random_number(x)
```

Most likely you will need more than one random number. The `random_number()` subroutine accepts an arbitrary floating-point array as argument and fills it completely with random numbers.

F90

```
program Rand

  implicit none
  real, dimension(5,5,5) :: x
  real                   :: mean, sigma2
  integer                :: ntot

  call random_number(x)         ! generate 5x5x5 random numbers

  ntot = size(x)
  mean   = sum(x)/ntot
  sigma2 = sum((x-mean)**2)/(ntot-1)

  print*, 'mean value        : ', mean        , &
          ' ideally: '          , 0.5
  print*, 'standard deviation: ', sqrt(sigma2), &
          ' ideally: '          , sqrt(1./12.)
```

[3] For some cases, this is the same as 'atan(y,x)' but that expression only covers the range $[-\pi/2, \pi/2]$ and fails if 're=0'

```
endprogram
```

If you want a reproducible sequence of "random" numbers, you can use the subroutine `random_seed()` to manipulate the *seed* of the generator.

A.3.4 Array syntax

Array syntax is very powerful feature of *F90*. It eliminates many loops which are difficult to read and provide ample opportunities for bugs or inefficiencies. Array syntax expresses *data parallelism*, i.e. the fact that one often applies the same operations to a whole array of data.

Compare the following codes in 'F77' and *F90*.

F77

```
      real a(4,5,6), b(4,5,6)
      real c(4,5,6)
      integer i1,i2,i3

      [initialize a and b]
      do 30 i3=1,6
          do 20 i2=1,5
              do 10 i1=1,4
                  c(i1,i2,i3) = a(i1,i2,i3) + b(i1,i2,i3)
10           continue
20       continue
30   continue
```

F90

```
real, dimension(4,5,6) :: a,b,c

[initialize a and b]
c = a + b
```

The *F90* version is much more compact (less opportunities for errors), does not require the variables *i1*, *i2*, and *i3*, and it is much closer to vector notation in mathematics, where you would normally write expressions like $\mathbf{C} = \mathbf{A} + \mathbf{B}$.

Note: All intrinsic arithmetic functions will act element-wise on arrays. So one could write

F90

```
c = cos(a)
b = exp(a)
c = c + 1.5 - sqrt(a**b)/atan(c)
```

For a matrix, 'exp(a)' will *not* be the matrix exponential you know from linear algebra, but simply the equivalent of

F77

```
      do 30 i3=1,6
          do 20 i2=1,5
              do 10 i1=1,4
                  b(i1,i2,i3) = exp(a(i1,i2,i3))
10           continue
20       continue
30   continue
```

Array slices

Often we do not want to access an array completely, but rather just a sub-block or line (e.g. a row or a column of a matrix). In *F90*, this is done using *array slices*, which use the ':' character to indicate an index range. For example, if *a* is a two-dimensional array (a matrix), 'a(1,:)' will refer to the first row, while 'a(:,3)' will refer to the third column. Similarly, 'a(2:4,:)' will refer to a matrix consisting of rows 2, 3, and 4, while 'a(1:2,5:8)' represents a two-dimensional submatrix formed by the intersection of rows 1 and 2 with columns 5, 6, 7, and 8. If you omit the end of the range, the range will count up to the largest index allowed, i.e. 'a(7:,:)' would be the same as 'a(7:199,:)' if a was declared as `real, dimension(199,15) :: a`.

F77

```
      real x(4,7,2)
      real y(4,2)
      integer i1,i2
      do 20 i2=1,2
          do 10 i1=1,4
            y(i1,i2) = x(i1,3,i2)
10       continue
          y(1,i1) = 2*x(2,2,:)
20   continue
```

F90

```
real, dimension(4,7,2) :: x
real, dimension(4,2)   :: y

y      =   x(:,3,:)
y(1,:) = 2*x(2,2,:)
```

Note: It is no accident that the outermost loop is over *i2* and the innermost over *i1*. Fortran stores the array *y* in memory in the order y(1,1), y(2,1), y(3,1), y(4,1), y(1,2), y(2,2), y(3,2), y(4,2), and for efficiency reasons, the innermost loop should always be over the index that is contiguous in memory, i.e. the first index.[4]

Array constructors

When we declare an array, we can initialize its values:

F90

```
real, dimension(3,3) :: zero=0.
real, dimension(3,3) :: unity = (/ (/ 1., 0., 0. /), &
                                   (/ 0., 1., 0. /), &
                                   (/ 0., 0., 1. /)  &
                                /)
```

Array functions

Some useful array functions:

sum sum all (or some) elements of an array

product multiply all (or some) elements of an array

all enquiry function returning true if the argument is true for *all* elements: '`if (all(vector>0)) print*, "positive"`'

any enquiry function returning true if the argument is true for *any of the* elements: '`if (any(vector<0)) print*, "someone is negative"`'

minval value of minimum element in array

maxval value of maximum element in array

shape shape (dimensionality) of an array

size size of an array (all dimensions or chosen one)

spread add dimensions by replication

transpose exchange dimensions

matmul matrix multiplication

dot_product dot product of two vectors

where (not really a function) brings '`if`' like decisions to array syntax

[4]In *C*, the contiguous index is the last index. This is why in *C*, one would use *i2* as innermost loop index:

```
for (i1=0; i1<4; i1++) {
    for (i2=0; i1<2; i1++) {
        y[i1,i2] = x[i1,3,i2];
    }
}
```

Note: There are also two functions `min()` and `max()` for calculating minimum and maximum of their arguments. If you think a bit about it, you will understand why both `min/maxval()` and `min/max` have a reason to exist. To calculate the maximum of x, y and z, you can do either

F90

```
big = max(x,y,z)
```

or

F90

```
big = maxval( (/x, y, z /) )
```

A.3.5 Assumed-shape arrays

In *F90*, you don't have to explicitly know the size of an array argument to a subroutine or function. The following example defines a function `cosh_1` of a 1-dimensional array argument `x` that will return an array of the same length as `x`.

F90

```
function cosh_1(x)

    implicit none
    real, dimension(:) :: x
    real, dimension(size(x,1)) :: cosh_1

    cosh_1 = 0.5*(exp(x)+exp(-x))

endfunction cosh_1
```

The argument `x` is a so-called *asumed-shape array*. The colon '`:`' stands for a dimension of unknown size; you do have to know the *shape* (dimensionality) of `x`, though. For two-dimensional `x`, the function would become

F90

```
function cosh_2(x)

    implicit none
    real, dimension(:,:) :: x
    real, dimension(size(x,1),size(x,2)) :: cosh_2

    cosh_1 = 0.5*(exp(x)+exp(-x))
```

```
endfunction cosh_1
```

Here is a more complex example (using in addition *assumed-length strings* and *optional arguments*) that prints out a matrix of arbitrary size with a given format.

F90

```
subroutine print_matrix(matx,fmt)
!
!  Print arbitrary-sized matrix MATX, optionally with given format FMT.
!  Usage:
!    call print_matx(matrix)
!    call print_matx(matrix, 'F12.3')
!

  integer                     :: i1, i2, n1, n2
  real, dimension(:,:)        :: matx
  character(LEN=*), optional  :: fmt
  character(LEN=256)          :: fmt1,linefmt

  n1 = size(matx,1)            ! get dimensions..
  n2 = size(matx,2)            ! of matrix

  !
  ! Construct format
  !
  if (present(fmt)) then
    fmt1 = fmt
  else
    fmt1 = '1pG12.4'           ! default format
  endif
  write(linefmt,'( "(", I4, "(", A10, ", "" ""))" )') n2, fmt1

  ! Debugging output; will print something like
  !     linefmt = <(    6(1pG12.4    , " "))>
  ! print*, 'linefmt = <', trim(linefmt), '>'

  do i1=1,n1
    write(*,linefmt) matx(i1,:)
  enddo

endsubroutine print_matrix
```

A.3.6 Allocatable arrays

Assumed-shape arrays can only be used in functions and subroutines. If your main program requires an array the dimensions of which are not known at compile-time (e.g. because they depend on user input), you can use *allocatable arrays*:

F90

```
program Alloc

  implicit none
  real, dimension(:,:), allocatable :: mtx   ! 2-dimensional array

  print*, 'Width of your square matrix?'
  read*, n
  allocate(mtx(n,n))

  ! Initialize the matrix, then
  call print_matrix(matx,fmt)
  ! do something else..

  deallocate(mtx)

  endprogram Alloc
```

A.3.7 Recursive functions/subroutines

For a function to call itself (directly, or via other functions), you have to declare it as '`recursive`':

F90

```
recursive function factorial(n) result(fact)

    implicit none
    integer, intent(in) :: n
    integer             :: fact

    if (n==0) then
        fact = 1
    else
        fact = factorial(n-1)*n
    endif
```

```
endfunction factorial
```

A.3.8 Modules and interfaces

A *module* is a container that can contain variables, functions and subroutines.

Another program unit gets access to these objects with the ‘`use`’ statement.

F90

```
module Hyper
!
!  A simple module for hyperbolic functions
!

    implicit none
    real :: e=2.718281828

contains

    real function cosh(x)
        real :: x
        cosh = 0.5*(exp(x)+exp(-x))
    endfunction cosh

endmodule Hyper

! ------------------------------------------------------------ !

program Super

    use Hyper

    implicit none
    real :: x

    x = 1.5
    print*, 'cosh(', x, ') = ', cosh(x)
    print*, 'e  = ', e

endprogram Super
```

The module and the main function will normally be in separate files (in that case, you would compile them with ‘`g95 hyper.f90 super_main.f90`’). But you can also have them in one single file; in this case, some compilers require that modules appear in the file before the program unit that uses them.

Modules can 'use' other modules and complicated codes often consist of a large number of modules.

Some techniques (e.g. overloading, see below) require that the program unit that uses a function (or subroutine) knows that function's (or subroutine's) *interface*. An interface for the 'cosh' function defined above would look like this

F90

```
interface
    real function cosh(x)
        real :: x
    endfunction cosh
endinterface
```

Obviously, writing interfaces is a tedious task, and even more so when a program is in flux, because the interface block would have to be updated each time the function or subroutine itself is considerably changed.

One advantage of modules is that they provide an automatic interface for all functions and subroutines they 'contain'. Thus, our `program Super` has automatically access to the interface of 'cosh' through the 'use Hyper' command.

A.3.9 Overloading

F90 allows overloading of functions and subroutines. As a real-life example, consider the following function that evaluates a polynomial for its argument x that can be a scalar or 1-dimensional array (in which case the result is a 1-d array, too).

F90

```
interface poly                ! Overload the 'poly' function
  module procedure poly_0
  module procedure poly_1
endinterface
!********************************************************
    function poly_0(coef, x)
!
!  Horner's scheme for polynomial evaluation.
!  Version for scalar.
!
      real, dimension(:) :: coef
      real :: x
```

```
      real :: poly_0
      integer :: Ncoef,i

      Ncoef = size(coef,1)

      poly_0 = coef(Ncoef)
      do i=Ncoef-1,1,-1
          poly_0 = poly_0*x+coef(i)
      enddo

    endfunction poly_0
!*******************************************************
    function poly_1(coef, x)
!
!  Horner's scheme for polynomial evaluation.
!  Version for 1-d array.
!
      real, dimension(:) :: coef
      real, dimension(:) :: x
      real, dimension(size(x,1)) :: poly_1
      integer :: Ncoef,i

      Ncoef = size(coef,1)

      poly_1 = coef(Ncoef)
      do i=Ncoef-1,1,-1
          poly_1 = poly_1*x+coef(i)
      enddo

    endfunction poly_1
!*******************************************************
```

A.3.10 Private functions

Data, functions and subroutine can be declared *private* to a module (or even another subroutine or function), which means they are inaccessible from outside, even by other program units that '`use`' the module. This can be useful for encapsulating data and to keep the namespace clean.

Overloading and private functions, together with user-defined data structures (which we have not covered here) allow *object-oriented* programming in *F90*.

A.4 Links

`http://www.techtutorials.info/fortran.html`: Collection of links to Fortran tutorials.

A.5 Appendix

A.5.1 Lab exercises

The following program has 1 syntax error and 1 run-time error.

```
subroutine phys381test( input, output1, output2 )
    implicit none
    integer, intent(in)::input
    integer, intent(out)::output1
    logical, intent(out)::output2
    integer::i
    if (input < 0 ) then
        output1 = 0
        output2 = .false.
    else
        output1 = 1
        i = 1
        while( i <= input )
            output1 = output1 * i
        end do
        output2 = .true.
    end if
end subroutine
```

Which line has the syntax error?

Write down the corrected statement to replace that line of statement.

The compiler error message is provided for your reference.

```
phys381test.f90:13.8:

        while( i <= input )
       1
Error: Unclassifiable statement at (1)
phys381test.f90:15.11:

        end do
          1
Error: Expecting END IF statement at (1)
```

A.5.2 Lab exercises

```
case1:
--------

# Simple Loop: Partial source code:

i = 0
x = 1
do while ( i < 5 )
   x = x+1
   i = i+1
end do
```

Questions: After the "end do" statement,

1. x = ? 2. i = ? 3. How many iterations does this loop construct have?

```
case2:
--------
# Loop nesting Loop: Partial
source code:

i = 1
x = 1
do while ( i > 0 .and. i < 5 )
     j = 0
    do while ( j < i )
       x = x+1
    end do
  i = i+1
end do
```

Questions: After the last "end do" statement,

1. x = ? 2. i = ? 3. j = ? 4. How many iterations does the outer loop construct have? 5. How many iterations does the last inner loop construct have?

A.5.3 Lab exercises

The study of fractal is related to chaos theory and the fact that inphysical systems (weather is one example) are inherently unpredictable on large time scales because small perturbations to the starting conditions will cause large changes over time.

The most famous of all fractal images is the Mandelbrot set. To generate the Mandelbrot set, we begin by considering a complex number, $c = x + yi$. We then apply the following algorithm:

- set $z = 0$ to start
- then repeatedly compute $z = z \times z + c$
- until $|z| > 2$ OR the number of iterations exceeds some threshold
- then output the number of iterations (let us call it n_c)

Write a program that performs the algorithm above and show that for:

(i) $c = 0.3 + 0.3i$ (or $x = 0.3$ and $y = 0.3$) the first 4 iterations give

1st iteration:	$z = 0.30 + 0.30i$	$\|z\| = 0.42$
2nd iteration:	$z = 0.30 + 0.48i$	$\|z\| = 0.57$
3rd iteration:	$z = 0.16 + 0.59i$	$\|z\| = 0.61$
4th iteration:	$z = -0.02 + 0.49i$	$\|z\| = 0.49$

In this case, z will remain bounded even after an infinite number of iterations.

(ii) $c = 0.5 + 1.0i$ (or $x = 0.5$ and $y = 1.0$) the first 5 iterations give

1st iteration:	$zz = 0.50 + 1.00i$	$\|z\| = 1.1$
2nd iteration:	$zz = -0.25 + 2.00i$	$\|z\| = 2.0$
3rd iteration:	$zz = -3.44 + 0.00i$	$\|z\| = 3.4$
4th iteration:	$zz = 12.32 + 1.00i$	$\|z\| = 12.4$
5th iteration:	$zz = 151.19 + 25.63i$	$\|z\| = 153.4$

and the size of z explodes toward infinite values. By the 10th iteration it will exceed the floating point range of most computers (*check this on your computer*). However, we can stop computing z once its absolute value (the complex absolute value or modulus is defined as the distance of a point in the complex plane from the origin, i.e. $|z| = |(x + yi)| = \sqrt{x \times x + y \times y}$) exceeds 2 because it can be shown that divergence is guaranteed at this point.

(iii) repeat this calulation for a series of different values of c (i.e. different x and y) and save into a file x, y and n_c. Use 1000 as an upper limit for the number of iterations.

(iv) You now have to plot the resulting output (n_c) as a function of the location of c on the complex plane. To do so, copy and paste the Gnuplot script file *gnuplot-fractal.gp*. The file is in Appendix H (**codes**) in the section containing the Gnuplot routines. You should then edit the file and modify the name of the data file to yours. Run the script by typing:

- gnuplot gnuplot-fractal.gp

A.5.4 Lab exercises

Write a program to assist in the design of a hydroelectric dam. Prompt the user for the height of the dam and for the number of cubic meters of water that are projected to flow from the top to the bottom of the dam each second. Predict how many megawatts (1 MW= 10^6 W) of power will be produced if 90% of the work done on the water by gravity is converted to electrical energy. Note that the mass of one cubic meter of water is 1000 kg.

Use 9.80 m/s^2 as the gravitational constant g. Be sure to use meaningful names for both the gravitational constant and the 90 % efficiency constant. For one run, use a height of 170m and flow of 1.30×10^3 m^3 s^{-1}. *The relevant formula (w=work, m=mass, g=gravity, h=height) is:* $w = mgh$.

Check your code: For a 170m dam and a flow of 1.30×10^3 m^3 s^{-1}, the dam can produce electrical power of 1949.220 Mega-watts.

A.5.5 Lab exercises

The brightness of a binary star varies as follows.

At time t = 0 its magnitude is 2.5, and it stays at this level until t = 0.9 days. Its magnitude is then determined by the formula $3.355 - \ln(1.352 + \cos(\pi \times (t - 0.9)/0.7))$ until t = 2.3 days. Its magnitude is then 2.5 until t = 4.4 days, and is then determined by the formula $3.598 - \ln(1.998 + cos(\pi \times (t - 4.4)/0.4))$ until t = 5.2 days. Its magnitude is then 2.5 until t = 6.4 days, after which the cycle repeats with a period of 6.4 days. Write a function which will input the value of the time t and output the brightness of the star at that time. Write a main program to print a graph of the brightness as a function of time in the interval t = 0 to t = 25.