Introduction to Modern Fortran

KIND, Precision and COMPLEX

Nick Maclaren

nmm1@cam.ac.uk

March 2014

The Basic Problem

REAL must be same size as INTEGER
This is for historical reasons – ask if you care

32 bits allows integers of up to 2147483647
Usually plenty for individual array indices

But floating–point precision is only 6 digits
And its range is only $10^{-38} - 10^{+38}$

Index values are not exact in floating–point
And there are many, serious numerical problems

Example

```
REAL, DIMENSION(20000000) :: A
REAL :: X
X = SIZE(A)-1
PRINT *, X
```

Prints 20000000.0 – which is not right
That code needs only 80 MB to go wrong

See “How Computers Handle Numbers”
Mainly on the numerical aspects

Ordinary REAL Constants

These will often do what you expect

- But they will very often lose precision

```
0.0, 7.0, 0.25, 1.23, 1.23E12,
0.1, 1.0E−1, 3.141592653589793
```

Only the first three will do what you expect

- In old Fortran constructs, can cause chaos

E.g. as arguments to external libraries

KIND Values

You can get the KIND of any expression

KIND(var) is the KIND value of var
KIND(0.0) is the KIND value of REAL
KIND(0.0D0) is that of DOUBLE PRECISION
    This is described in a moment

Implementation–dependent integer values
    selecting the type (e.g. a specific REAL)

- Don’t use integer constants directly

SELECTED_REAL_KIND

You can request a minimum precision and range
Both are specified in decimal

SELECTED_REAL_KIND (Prec [, Range])

This gives at least Prec decimal places
and range $10^{-Range} - 10^{+Range}$

E.g. SELECTED_REAL_KIND(12)
at least 12 decimal places

Warning: Time Warp

Unfortunately, we need to define a module
We shall cover those quite a lot later

The one we shall define is trivial
Just use it, and don't worry about the details
Everything you need to know will be explained

Just compile it, but don't link it, using –c

```
nagfor –C=all –c double.f90
```

Using KIND (1)

You should write and compile a module

```
MODULE double
    INTEGER, PARAMETER :: DP =        &
        SELECTED_REAL_KIND(12)
END MODULE double
```

Immediately after every procedure statement
I.e. PROGRAM, SUBROUTINE or FUNCTION

```
USE double
IMPLICIT NONE
```

Using KIND (2)

Declaring variables etc. is easy

```
REAL(KIND=DP) :: a, b, c
REAL(KIND=DP), DIMENSION(10) :: x, y, z
```

Using constants is more tedious, but easy

```
0.0_DP, 7.0_DP, 0.25_DP, 1.23_DP, 1.23E12_DP,
0.1_DP, 1.0E−1_DP, 3.141592653589793_DP
```

That’s really all you need to know . . .

Using KIND (3)

Note that the above makes it trivial to change
ALL you need is to change the module

```
MODULE double
    INTEGER, PARAMETER :: DP = &
        SELECTED_REAL_KIND(15, 300)
END MODULE double
```

(15, 300) requires IEEE 754 double or better

Or even: SELECTED_REAL_KIND(25, 1000)

DOUBLE PRECISION (1)

- The best way to control precision

Most flexible, portable and future–proof
Advisable if you may want to use HECToR

All older (Fortran 77) code will do it differently
And quite a lot of programmers still do
The old method is fairly reliable, today

- You need to know about this, but avoid it

DOUBLE PRECISION (2)

DOUBLE PRECISION takes the space of 2 REALs
⇒ It need not be any more accurate, though

- Almost always, REAL is 32–bit IEEE 754

And DOUBLE PRECISION is 64–bit IEEE 754
Precision is 15 digits, range is $10^{-300} - 10^{+300}$

Main exception is Cray vector supercomputers
And when using compiler options to change precision

DOUBLE PRECISION (3)

You can use it just like REAL in declarations
Using KIND is more modern and compact

```
REAL(KIND=KIND(0.0D0)) :: a, b, c
```

Constants use D for the exponent – 1.23D12 or 0.0D0

```
REAL(KIND=KIND(0.0D0)) :: a, b, c
DOUBLE PRECISION, DIMENSION(10) :: x, y, z

0.0D0, 7.0D0, 0.25D0, 1.23D0, 1.23D12,
0.1D0, 1.0D–1, 3.14159265358979 3D0
```

Intrinsic Procedures

Almost all intrinsics 'just work' (i.e. are generic)
IMPLICIT NONE removes most common traps

- Avoid specific (old) names for procedures
AMAX0, DMIN1, DSQRT, FLOAT, IFIX etc.

- DPROD is also not generic – use a library

- Don't use the INTRINSIC statement

- Don't pass intrinsic functions as arguments

Type Conversion (1)

This is the main “gotcha” – you should use

```
REAL(KIND=DP) :: x
x = REAL(<integer expression>, KIND=DP)
```

Omitting the KIND=DP may lose precision

- With no warning from the compiler

Automatic conversion is actually safer!

```
x = <integer expression>
x = SQRT(<integer expression>+0.0_DP)
```

Type Conversion (2)

There is a legacy intrinsic function
If you are using explicit DOUBLE PRECISION

```
x = DBLE(<integer expression>)
```

All other "gotchas" are for COMPLEX

Warning

You will often see code like:

```
REAL*8 X, Y, Z
INTEGER*8 M, N
```

- Most of the Web and many books are wrong

A Fortran IV feature, NOT a standard one
'8' is NOT always the size in bytes

- I strongly recommend converting to KIND

Old Fortran Libraries

Be very careful with external libraries

- Make sure argument types are right

Automatic conversion does not happen
Not will you get a diagnostic (in general)

Any procedure with no explicit interface
I did say that using old Fortran was more painful

INTEGER KIND

You can choose different sizes of integer

```
INTEGER, PARAMETER :: big = &
        SELECTED_INT_KIND(12)
INTEGER(KIND=big) :: bignum
```

bignum can hold values of up to at least 10^{12}
Few users will need this – mainly for OpenMP

Some compilers may allocate smaller integers
E.g. by using SELECTED_INT_KIND(4)

CHARACTER KIND

It can be used to select the encoding
It is mainly a Fortran 2003 feature

Can select default, ASCII or ISO 10646
ISO 10646 is effectively Unicode
Useful for handling non–ASCII character sets

It is not covered in this course
Very few scientists want or use it

Complex Arithmetic

Fortran is the answer – what was the question?

Has always been supported, and well integrated

COMPLEX is a (real, imaginary) pair of REAL
It uses the same KIND as underlying reals

```
COMPLEX(KIND=DP) :: c
c = (1.23_DP,4.56_DP)
```

Full range of operations, intrinsic functions etc.

Example

```
COMPLEX(KIND=DP) :: c, d, e, f

c = (1.23_DP,4.56_DP)*CONJG(d)+SIN(f*g)
e = EXP(d+c/f)*ABS(LOG(e))
```

The functions are the complex forms
E.g. ABS is $\sqrt{re^2 + im^2}$
CONJG is complex conjugate, of course

Using COMPLEX really IS that simple!

Worst “Gotcha”

- Must specify KIND in conversion function

```
c = CMPLX(<X−expr>, KIND=DP)
c = CMPLX(<X−expr>, <Y−expr>, KIND=DP)
```

This will not work – KIND is default REAL
Usually with no warning from the compiler

```
c = CMPLX(0.1_DP,0.2_DP)
```

Conversion to REAL

```
REAL(KIND=DP) :: x
COMPLEX(KIND=DP) :: c
  . . . lots of statements . . .
x = x+c
c = 2.0_DP*x
```

Loses the imaginary part, without warning
Almost all modern languages do the same

A Warning for Old Code

```
C = DCMPLX(0.1_DP, 0.1_DP)
```

That is often seen in Fortran IV legacy code
It doesn't work in standard (modern) Fortran

- It will be caught by IMPLICIT NONE

Complex I/O

The form of I/O we have used is list–directed
COMPLEX does what you would expect

```
COMPLEX(KIND=DP) :: c = (1.23_DP,4.56_DP)
WRITE (*, *) C
```

Prints “(1.23,4.56)”
And similarly for input

There is some more on COMPLEX I/O later

Exceptions

Complex exceptions are mathematically hard

- Overflow often does what you won’t expect

Fortran, unfortunately, is no exception to this

See “How Computers Handle Numbers”

- Don’t cause them in the first place

- Use the techniques described to detect them