Astrophysikalisches Numerikum

part02: simple F90

Peter H. Hauschildt

yeti@hs.uni-hamburg.de

Hamburger Sternwarte
Gojenbergsweg 112
21029 Hamburg

Topics

- simple Fortran90
  - loops
  - if statements
  - datatypes
  - numerics
  - I/O

simple $n!$ code

```
      program test
      implicit none
c
      integer n,result
      integer, external :: nfak
c
      do
       write(*,*) "enter n:"
       read(*,*) n
       result = nfak(n)
       write(*,*) 'n=',n,' n!=',result
      enddo
c
      stop
      end
```

simple $n!$ code

```
      integer function nfak(n)
      implicit none
      integer, intent(in) :: n
c
      integer i
c
      if(n .le. 1) then
       nfak = 1
       return
      else
       nfak = 1
       do i=2,n
        nfak = nfak*i
       enddo
      endif
      return
      end
```

variable declaration/assignment

```
      integer i
      real*8, dimension(1:100) :: vector
c
      i = 10
      i = i+5
      vector(i) = 10.d0
      vector(:) = 10.d0+10*vector(:)
```

- declaration defines type of named variable
- assignment sets variable to value

loops

```
do
 continue
enddo
do i=start,end,step
 continue
enddo
```

- infinite loop
- finite loop

if statement

```
if(....) then
 continue
elseif(....) then
 continue
else
 continue
endif
```

- `....`: logical expression
- example: `n .gt. 1`
- comparison operators: `.eq., .ne., .ge., .gt., .le., .lt., ...`

function/subroutines

```
      integer function nfak(n)
      implicit none
      integer, intent(in) :: n
      continue
      end function nfak
c
      subroutine test(n)
      implicit none
      integer, intent(inout) :: n
      continue
      end subroutine test
```

- function returns a single value of given datatype
- values can also be returned through arguments

datatypes

- `logical`: true/false boolean variable
- `character`: holds one ASCII character
- `character*`N: holds N ASCII characters (string)
- `integer`: limited range depends on "size":
- 2 bytes, 4 bytes, 8 bytes
- binary system coding

Floating point numbers

- several ways to represent real numbers on computers:
- *fixed point*: place radix point somewhere in the middle of the digits
- equivalent to using integers
- fixed point has a fixed window of representation
- $\rightarrow$ limits it from representing very large or very small numbers
- prone to a loss of precision when two large numbers are divided

`http://research.microsoft.com/~hollasch/cgindex/coding/ieeefloat.html`

Floating point numbers

- *rationals*: represent every number as the ratio of two integers
- *floating point*: represents reals in scientific notation
- decimal: $123.456 \rightarrow 1.23456 \times 10^2$
- hex:$123.abc \rightarrow 1.23abc \times 16^2$
- 'sliding window' of precision appropriate to the scale of the number
- can represent large and small numbers:
  - $1,000,000,000,000$
  - 0.0000000000000001

Storage Layout

	Sign	Exponent	Fraction	Bias
Single Precision	1 [31]	8 [30-23]	23 [22-00]	127
Double Precision	1 [63]	11 [62-52]	52 [51-00]	1023

- Sign Bit
- Exponent
  - bias is added to the actual exponent in order to recover the exponent
- Mantissa
  - implicit leading bit, fraction bits
  - binary $\rightarrow$ only possible non-zero digit is 1
  - mantissa has effectively 24 bits

Ranges of Floating-Point Numbers

- 32-bit integers $\rightarrow$ 32-bits of resolution
- Single-precision floating-point $\rightarrow$ only 24 bits
- $\rightarrow$ approximate 32-bit integer by effectively truncating from the lower end
- example:
  - $11110000\ 11001100\ 10101010\ 00001111$
    32-bit integer
  - $= +1.1110000\ 11001100\ 10101010 \times 2^{31}$
    as Single-Precision Float
  - $11110000\ 11001100\ 10101010\ 00000000$
    Corresponding Value

Ranges of Floating-Point Numbers

- range of positive floating point numbers:
  - normalized numbers (full precision of the mantissa)
  - denormalized numbers (portion of the fractions's precision)

Ranges of Floating-Point Numbers

	Denormalized	Normalized	Approximate Decimal
Single Precision	$\pm 2^{-149}$ to $(1-2^{-23})\times 2^{-126}$	$\pm 2^{-126}$ to $(2-2^{-23})\times 2^{127}$	$\pm \sim 10^{-44.85}$ to $\sim 10^{38.53}$
Double Precision	$\pm 2^{-1074}$ to $(1-2^{-52})\times 2^{-1022}$	$\pm 2^{-1022}$ to $(2-2^{-52})\times 2^{1023}$	$\pm \sim 10^{-323.3}$ to $\sim 10^{308.3}$

Ranges of Floating-Point Numbers

- five distinct numerical ranges that single-precision floating-point numbers are not able to represent:
  1. Negative numbers less than $-(2 \times 2^{23}) \times 2^{127}$ (negative overflow)
  2. Negative numbers greater than -2^{-149} (negative underflow)
  3. Zero
  4. Positive numbers less than 2^{-149} (positive underflow)
  5. Positive numbers greater than $(2 \times 2^{23}) \times 2^{127}$ (positive overflow)

Ranges of Floating-Point Numbers

	Binary	Decimal
Single	$\pm (2-2^{-23})^{127}$	$\sim \pm 10^{38.53}$
Double	$\pm (2-2^{-52})^{1023}$	$\sim \pm 10^{308.25}$

Special Values

- IEEE reserves exponent field values of all 0s and all 1s to denote special values in the floating-point scheme
- Zero
  - zero is not directly representable in the straight format, due to the assumption of a leading 1
  - special value denoted with an exponent field of zero and a fraction field of zero
  - note that -0 and $+0$ are distinct values, though they both compare as equal

Special Values

- Denormalized
  - exponent is all 0s, but the fraction is non-zero
  - does not have an assumed leading 1 before the binary point
  - represents a number $(-1)^s \times 0.f \times 2^{-126}$ where s is the sign bit and f is the fraction
  - double precision: $(-1)^s \times 0.f \times 2^{-1022}$
  - zero is a special type of denormalized number

Special Values

- Infinity
  - $+\infty$ and $-\infty$ are denoted with an exponent of all 1s and a fraction of all 0s
  - sign bit distinguishes between negative infinity and positive infinity
  - allows operations to continue past overflow situations
  - *Operations with infinite values are well defined in IEEE floating point arithmetic*

Special Values

- Not A Number (NaN)
  - used to represent a value that does not represent a real number
  - represented by a bit pattern with an exponent of all 1s and a non-zero fraction
  - two categories of NaN: NaNQ (Quiet NaN) and NaNS (Signalling NaN)
- NaNQ is a NaN with the most significant fraction bit set
- NaNQ's propagate freely through most arithmetic operations
- return from an operation when the result is not mathematically defined

Special Values

- NaNS is a NaN with the most significant fraction bit clear
- used to signal an exception when used in operations
- e.g., assign to uninitialized variables to trap premature usage

Special Operations

- any operation with a NaN yields a NaN result
- comparisons with NaNs are always *false*:
- NaNQ $==$ NaNQ $\rightarrow$ `false`

Special Operations

Operation	Result
n / ±Infinity	0
±Infinity x ±Infinity	±Infinity
±nonzero / 0	±Infinity
Infinity + Infinity	Infinity
±0 / ±0	*NaN*
Infinity – Infinity	*NaN*
±Infinity / ±Infinity	*NaN*
±Infinity x 0	*NaN*

IEEE 754 summary

Float Values (b = bias)

Sign	Exponent (e)	Fraction (f)	Value
0	00..00	00..00	+0
0	00..00	00..01 : 11..11	Positive Denormalized Real $0.f \times 2^{(-b+1)}$
0	00..01 : 11..10	XX..XX	Positive Normalized Real $1.f \times 2^{(e-b)}$
0	11..11	00..00	+Infinity
0	11..11	00..01 : 01..11	SNaN
0	11..11	10..00 : 11..11	QNaN
1	00..00	00..00	-0
1	00..00	00..01 : 11..11	Negative Denormalized Real $-0.f \times 2^{(-b+1)}$
1	00..01 : 11..10	XX..XX	Negative Normalized Real $-1.f \times 2^{(e-b)}$
1	11..11	00..00	-Infinity
1	11..11	00..01 : 01..11	SNaN
1	11..11	10..00 : 11.11	QNaN

fortran floating point

- `real, real*4`: single precision float
- `real*8, double precision`: double precision float
- IEEE 754 coding

I/O statements

```
      open(I,file="name",...)
c
      write(I,JJJ) string
      write(I,"(1x,a,....)") string
c
      read(I,JJJ) integer
      read(I,'(1x,a,....)') string
c
 JJJ  format(1x,a,...)
```

- `I`: "channel number", integer number or "`*`"
- `JJJ`: "format label", integer number of format statement

some format codes

- `Nx`: skip N characters
- `aN`: character I/O, N fields
- `iN`: integer I/O, N digits
- `lN`: logical I/O, N fields
- `esN.M`: floating point I/O, N fields, $M+1$ digits
- example: `es12.4` $\rightarrow$ `1.2345e+00`

problems

1. extend `Makefile` to compile/link $n!$ codes
2. write $n!$ functions with `integer`, `real` and `double precision` datatypes
3. test their range of validity
4. write a functions to compute $\log n!$
5. write code to generate tables of n, $n!$, $\log n!$