Today:
- Fortran dynamic memory allocation
- Array operations
- Computer storage
- Binary representation
- Floating point
- Exceptions

Friday:
- Computer arithmetic
- Fortran subroutines and functions

Read: Class notes and references.
Often a program needs to be written to handle arrays whose size is not known until the program is running.

Fortran 77 approaches:

- Allocate arrays large enough for any application,
- Use “work arrays” that are partitioned into pieces.

We will look at some examples from LAPACK since you will probably see this in other software!
Often a program needs to be written to handle arrays whose size is not known until the program is running.

Fortran 77 approaches:

- Allocate arrays large enough for any application,
- Use “work arrays” that are partitioned into pieces.

We will look at some examples from LAPACK since you will probably see this in other software!

The good news:

**Fortran 90 allows dynamic memory allocation.**
Memory allocation

real(kind=8) dimension(:), allocatable :: x
real(kind=8) dimension(:,,:), allocatable :: a

allocate(x(10))
allocate(a(30,10))

! use arrays

deallocate(x)
deallocate(a)
If you might run out of memory, better to do:

```fortran
real(kind=8), dimension(:,:), allocatable :: a
allocate(a(30000,10000), stat=allocated_error)
if (allocated_error /= 0) then
    print *, "Insufficient memory"
    stop
endif
```
Array operations in Fortran

Fortran 90 supports some operations on arrays...

! $CLASSHG/codes/fortran/vectorops.f90
program vectorops
  implicit none
  real(kind=8), dimension(3) :: x, y

  x = (/10.,20.,30./) ! initialize
  y = (/100.,400.,900./)

  print *, "x = "
  print *, x

  print *, "x**2 + y = "
  print *, x**2 + y ! componentwise
Array operations in Fortran

! $CLASSHG/codes/fortran/vectorops.f90
! continued...

print *, "x*y = "
print *, x*y  ! = (x(1)y(1), x(2)y(2), ...)

print *, "sqrt(y) = "
print *, sqrt(y)  ! componentwise

print *, "dot_product(x,y) = "
print *, dot_product(x,y)  ! scalar product

end program vectorops
Array operations in Fortran — Matrices

! $CLASSHG/codes/fortran/arrayops.f90
program arrayops
  implicit none
  real(kind=8), dimension(3,2) :: a
  ...
  ! create a as 3x2 array:
  A = reshape((/1,2,3,4,5,6/), (/3,2/))

Note:
  • Fortran is case insensitive: \( A = a \)
  • Reshape fills array by columns, so

\[
A = \begin{bmatrix}
  1 & 4 \\
  2 & 5 \\
  3 & 6
\end{bmatrix}.
\]
real(kind=8), dimension(3,2) :: a
real(kind=8), dimension(2,3) :: b
real(kind=8), dimension(3,3) :: c
integer :: i

print *, "a = ", a

do i=1,3
    print *, a(i,:)
enddo

b = transpose(a) ! 2x3 array

c = matmul(a, b) ! 3x3 matrix product
real(kind=8), dimension(3,2) :: a
real(kind=8), dimension(2) :: x
real(kind=8), dimension(3) :: y

x = (/5,6/)
y = matmul(a,x) ! matrix-vector product
print *, "x = ",x
print *, "y = ",y
Linear systems in Fortran

There is no equivalent of the Matlab backslash operator for solving a linear system $Ax = b \quad (b = A\backslash b)$

Must call a library subroutine to solve a system.

Later we will see how to use LAPACK for this.

Note: Under the hood, Matlab calls LAPACK too!
Computer memory

Memory is subdivided into bytes, consisting of 8 bits each.

One byte can hold $2^8 = 256$ distinct numbers:

- 00000000 = 0
- 00000001 = 1
- 00000010 = 2
- \ldots
- 11111111 = 255

Might represent integers, characters, colors, etc.

Usually programs involve integers and real numbers that require more than 1 byte to store.

Often 4 bytes (32 bits) or 8 bytes (64 bits) used for each.
Integers

To store integers, need one bit for the sign (+ or −)
In one byte this would leave 7 bits for binary digits.

Two-complements representation used:

\[
\begin{align*}
10000000 & = -128 \\
10000001 & = -127 \\
10000010 & = -126 \\
\ldots & \\
11111110 & = -2 \\
11111111 & = -1 \\
00000000 & = 0 \\
00000001 & = 1 \\
00000010 & = 2 \\
\ldots & \\
01111111 & = 127 \\
\end{align*}
\]

Advantage: Binary addition works directly.
Integers are typically stored in 4 bytes (32 bits). Values between roughly $-2^{31}$ and $2^{31}$ can be stored.

In Python, larger integers can be stored and will automatically be stored using more bytes.

Note: special software for arithmetic, may be slower!

```python
>>> 2**30
1073741824
```

```python
>>> 2**100
1267650600228229401496703205376L
```

Note L on end!
Integer overflow in gfortran

! $CLASSHG/codes/fortran/integers.f90
program integers
  implicit none
  integer :: i,j

  i = 2**30
  print *, "i = ",i

  j = 4 * i
  print *, "j = ",j
end program integers

produces the following:

  i =  1073741824
  j =      0  This is wrong!
32-bit vs. 64-bit architecture

Each byte in memory has an address, which is an integer. On 32-bit machines, registers can only store

\[ 2^{32} = 4294967296 \approx 4 \text{ billion} \]

distinct addresses \( \implies \) at most 4GB of memory can be addressed.

Newer machines often have more, leading to the need for 64-bit architectures (8 bytes for addresses).

**Note:** Integers might still be stored in 4 bytes, for example.
Fixed point notation

Use, e.g. 64 bits for a real number but always assume $N$ bits in integer part and $M$ bits in fractional part.

Analog in decimal arithmetic, e.g.:
- 5 digits for integer part and
- 6 digits in fractional part

Could represent, e.g.:

00003.141592
00000.000314
31415.926535

Disadvantages:
- Precision depends on size of number
- Often many wasted bits (leading 0’s)
- Limited range; often scientific problems involve very large or small numbers.
Floating point real numbers

Base 10 scientific notation:

\[ 0.2345 \times 10^{-18} = 0.00000000000000002345 \]

Mantissa: 0.2345, Exponent: -18
Floating point real numbers

**Base 10 scientific notation:**

\[
0.2345 \times 10^{-18} = 0.2345 \times 10^{-18} = 0.000000000000000002345
\]

**Mantissa:** 0.2345,  \hspace{1cm} **Exponent:** -18

**Binary floating point numbers:**

**Example:** **Mantissa:** 0.101101,  \hspace{1cm} **Exponent:** -11011  means:

\[
0.101101 = 1(2^{-1}) + 0(2^{-2}) + 1(2^{-3}) + 1(2^{-4}) + 0(2^{-5}) + 1(2^{-6})
\]
\[
= 0.703125  \text{ (base 10)}
\]
\[
-11011 = -1(2^{4}) + 1(2^{3}) + 0(2^{2}) + 1(2^{1}) + 1(2^{0}) = -27 \text{ (base 10)}
\]

So the number is

\[
0.703125 \times 2^{-27} \approx 5.2386894822120667 \times 10^{-9}
\]
Floating point real numbers

Fortran:

real (kind=4): 4 bytes
   This used to be standard single precision real

real (kind=8): 8 bytes
   This used to be called double precision real

Python float datatype is 8 bytes.

8 bytes = 64 bits,

53 bits for mantissa and 11 bits for exponent (64 bits = 8 bytes).

We can store 52 binary bits of precision.

\[ 2^{-52} \approx 2.2 \times 10^{-16} \implies \text{roughly 15 digits of precision.} \]
Floating point real numbers

Since \(2^{-52} \approx 2.2 \times 10^{-16}\) this corresponds to roughly 15 digits of precision.

For example:

```python
>>> from numpy import pi
>>> pi
3.1415926535897931

>>> 1000 * pi
3141.5926535897929
```

Note: storage and arithmetic is done in base 2
Converted to base 10 only when printed!
8 bytes floats: 64 bits for each real number with
53 bits for mantissa and
11 bits for exponent.

Exponents range between $-1022$ and $1023$, so magnitude of
real number must be less than $N_{\text{max}} \approx 2^{1023} \approx 1.8 \times 10^{308}$.

If an operation gives a number outside this range we get an
overflow exception.

Or perhaps a special value representing “infinity”.

Real overflow

! $CLASSHG/codes/fortran/reals.f90

program reals
  implicit none
  real (kind=8) :: x,y,z
  x = 1.d308
  print *, "x = ",x
  y = 10.d0 * x
  print *, "y = ",y
  z = y / 10.d0
  print *, "z = ",z
end program reals

x = 1.000000000000000E+308
y = +Infinity
z = +Infinity
Exponents range between $-1022$ and $1023$. Smallest nonzero real number is about

$$N_{\text{min}} = 2^{-1022} \approx 2.2 \times 10^{-308} \text{ if we insist it be normalized (i.e. no leading zeros).}$$

Can represent even smaller numbers by using **gradual underflow**, and **subnormal numbers** e.g.,

$$0.000005 \times 10^{-308} = 5.0 \times 10^{-314}$$

With 16 digits, can go down to about $10^{-324}$ in this manner.
Real underflow

$CLASSHG/codes/fortran/underflow.f90

program underflow
    implicit none
    real (kind=8) :: x

    x = 1.d-308
    print *, "x = ", x

    do while (x > 0.d0)
        x = x / 10.d0
        print *, "x = ", x
    enddo
end program underflow
Gradual underflow $\Rightarrow$ less precision for smaller $x$

\[
\begin{align*}
x &= 9.999999999999999E-309 \\
x &= 1.000000000000002E-309 \\
x &= 9.999999999999969E-311 \\
x &= 9.999999999999475E-312 \\
x &= 9.999999999984653E-313 \\
x &= 1.000000000013287E-313 \\
x &= 9.999999999638807E-315 \\
x &= 9.999999984816838E-316 \\
x &= 9.999999836597144E-317 \\
x &= 9.999997366268915E-318 \\
x &= 9.99987484955998E-319 \\
x &= 9.99888671826830E-320 \\
x &= 9.99888671826830E-321 \\
x &= 9.80126045993180E-322 \\
x &= 9.881312916824931E-323 \\
x &= 9.881312916824931E-324 \\
x &= 0.000000000000000
\end{align*}
\]
Not-a-Number (NaN)

Some arithmetic operations give undefined results.

The result of such an operation is often replaced by a special value representing $\text{NaN}$.

**Examples:**

\[
\frac{0}{0} = \text{NaN} \\
0 \times \text{Infinity} = \text{NaN}
\]
Not-a-Number (NaN)

! $CLASSHG/codes/fortran/nan.f90
program nan
    implicit none
    real (kind=8) :: x, y, z

    x = 0.d0
    y = 1.d0 / x
    print *, "y = ", y  prints  y = +Infinity

    z = 0.d0 / x
    print *, "z = ", z  prints  z = NaN

    z = 0.d0 * y
    print *, "z = ", z  prints  z = NaN
end program nan
Trapping floating point exceptions

Often we want the program to crash instead of continuing with Infinity or NaNs.

Can compile with fpe-trap flag set to the list of exceptions to trap: overflow, underflow, or divide by zero:

```
$ gfortran -ffpe-trap=zero,overflow,underflow \nan.f90

$ ./a.out
Floating point exception
```

**Note:** Not at all informative about where it crashed. (Need to use a debugger to figure out where.)