Today:
- Computer arithmetic
- Fortran subroutines and functions

Monday:
- Fortran array storage
- Fortran modules
- Multi-file Fortran codes

Read: Class notes and references.
Floating point real numbers

**Base 10 scientific notation:**

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

**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, **Exponent:** -18

Binary floating point numbers:

**Example:** **Mantissa:** 0.101101, **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 (8 bytes)

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

We can hope to get at most 15 correct digits in computations.

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!
Absolute and relative error

Let $\hat{z} =$ exact answer to some problem,  
$z^* =$ computed answer using some algorithm.

Absolute error: $|z^* - \hat{z}|$

Relative error: $\frac{|z^* - \hat{z}|}{|\hat{z}|}$

If $|\hat{z}| \approx 1$ these are roughly the same.

But in general relative error is a better measure of how many correct digits in the answer:

Relative error $\approx 10^{-k} \implies \approx k$ correct digits.
Example:

Compute length of diagonal of 1 meter × 1 meter square.

True value: \( \hat{z} = \sqrt{2} = 1.4142135623730951 \ldots \) meters

We compute \( z^* = 1.413 \) meters

Absolute error: \( |z^* - \hat{z}| \approx 0.0012135 \approx 10^{-3} \) meters

Relative error: \( \frac{|z^* - \hat{z}|}{|\hat{z}|} \approx \frac{0.0012135}{1.414} \approx 0.000858 \approx 10^{-3} \)

Note: Relative error is dimensionless.

The absolute and relative errors are both \( \approx 10^{-3} \).

Roughly 3 correct digits in solution.
Exactly same problem but now measure in kilometers.

Compute length of diagonal of 0.001 km \( \times \) 0.001 km square.

True value: \( \hat{z} = \sqrt{2} \times 0.001 = 0.0014142135623730951 \ldots \) km

We compute \( z^* = 0.001413 \) km

**Absolute error:** \( |z^* - \hat{z}| \approx 0.0000012135 \approx 10^{-6} \) km

**Relative error:** \( \frac{|z^* - \hat{z}|}{|\hat{z}|} \approx \frac{0.0012135}{1.414} \approx 0.000858 \approx 10^{-3} \)

The absolute error is much smaller than before but there are still only 3 correct digits!
Exactly same problem but now measure in nanometers.

Compute length of diagonal of \(10^9\) nm \(\times\) \(10^9\) nm square.

True value: \(\hat{z} = \sqrt{2} \times 10^9 = 1414213562.3730951\ldots\) nm

We compute \(z^* = 1413000000\) nm

**Absolute error:** \(|z^* - \hat{z}| \approx 1213562.373 \approx 10^6\) nm

**Relative error:** 
\[
\frac{|z^* - \hat{z}|}{|\hat{z}|} \approx \frac{0.0012135}{1.414} \approx 0.000858 \approx 10^{-3}
\]

The absolute error is much larger than before but there are still 3 correct digits!
Machine epsilon (for 8 byte reals)

```python
>>> y = 1. + 3.e-16
>>> y
1.0000000000000002

>>> y - 1.
2.2204460492503131e-16
```

**Machine epsilon** is the distance between 1.0 and the next largest number that can be represented: $2^{-52} \approx 2.2204 \times 10^{-16}$

```python
>>> y = 1 + 1e-16
>>> y
1.0

>>> y == 1
True
```
We generally don’t need 16 digits in our solutions
But often need that many digits to get reliable results.

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

>>> y = pi * 1.e-10
>>> y
3.1415926535897934e-10

>>> z = 1. + y
>>> z
1.0000000003141594 # lost several digits!

>>> z - 1.
3.141593651889707e-10 # only 6 or 7 digits right!
```
Rounding errors can cause big errors!

Example: Solve $Ax = b$ using Matlab, for

$$A = \begin{bmatrix} 1 & 2 \\ 4 - 10^{-12} & 10 - 2 \times 10^{-12} \end{bmatrix}, \quad b = \begin{bmatrix} 5 \\ 10 - 2 \times 10^{-12} \end{bmatrix}. \quad \text{Solution: } \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$ 

>> format long e
>> A
A =
1.000000000000000e+000 2.000000000000000e+000
2.000000000000000e+000 3.999999999999000e+000

>> b
b =
5.000000000000000e+000
9.999999999999800e+000

>> x = A\b

x =
9.982238010657194e-001 rel. error 0.00178
2.000888099467140e+000 rel. error 0.00044

Note: This matrix is nearly singular (ill-conditioned).
Second column is almost a scalar multiple of the first.
Rounding errors can cause big errors!

Example: Solve $Ax = b$ using Matlab, for

$$A = \begin{bmatrix} 1 & 2 \\ 4 & 2 \times 10^{-12} \end{bmatrix}, \quad b = \begin{bmatrix} 10 - 2 \times 10^{-12} \end{bmatrix}.$$  

Solution: $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$.

```
>> format long e
>> A
A = 
    1.000000000000000e+000  2.000000000000000e+000
    2.000000000000000e+000  3.999999999999000e+000

>> b
b = 
    5.000000000000000e+000
    9.999999999999000e+000

>> x = A\b
x =
    9.982238010657194e-001  rel. error 0.00178
    2.000888099467140e+000  rel. error 0.00044
```

Note: This matrix is nearly singular (ill-conditioned).
Second column is almost a scalar multiple of the first.
Taylor series expansion for $\exp(x) = e^x$

The function $\exp(x)$ is provided by most computing languages. 

But how is this computed??

Computer hardware can only do addition, subtraction, multiplication, division.

Other functions must be approximated by some algorithm using only these.
Taylor series expansion for $\exp(x) = e^x$

The function $\exp(x)$ is provided by most computing languages.

But how is this computed??

Computer hardware can only do addition, subtraction, multiplication, division.

Other functions must be approximated by some algorithm using only these.

One useful tool is Taylor series expansions, e.g.

$$e^x = 1 + x + \frac{1}{2} x^2 + \frac{1}{3!} x^3 + \frac{1}{4!} x^4 + \cdots$$

Mathematically, this series converges for all $x$.

By taking enough terms, can make the error arbitrarily small.
Taylor series expansion for $\exp(x) = e^x$

The function $\exp(x)$ is provided by most computing languages.

But how is this computed??

Computer hardware can only do addition, subtraction, multiplication, division.

Other functions must be approximated by some algorithm using only these.

One useful tool is Taylor series expansions, e.g.

$$e^x = 1 + x + \frac{1}{2}x^2 + \frac{1}{3!}x^3 + \frac{1}{4!}x^4 + \cdots$$

Mathematically, this series converges for all $x$.

By taking enough terms, can make the error arbitrarily small.

But this is not true in finite precision computer arithmetic!
Example: Compute $e^x$ using Taylor series when $x < 0$.

```python
>>> import taylor; from numpy import exp
>>> exp(-20.)
2.0611536224385579e-09

>>> taylor.exp(-20., 20) # using N=20 terms
-21822593.77927747

>>> taylor.exp(-20., 100) # using 100 terms
5.6218844721304176e-09

>>> taylor.exp(-20., 1000) # using 1000 terms
5.6218844721304176e-09
```

Adding more terms does not help!!
Cancellation

Look more carefully at computation of $\exp(-7)$:

True: $\quad 0.0009118819655545 \quad 0.0009118819655545$

<table>
<thead>
<tr>
<th>$j$</th>
<th>$j$’th term</th>
<th>Partial sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$-7.00000000000000000000000$</td>
<td>$-6.000000000000000000000$</td>
</tr>
<tr>
<td>2</td>
<td>$24.500000000000000000000$</td>
<td>$18.500000000000000000000$</td>
</tr>
<tr>
<td>3</td>
<td>$-57.1666666666666666643$</td>
<td>$-38.6666666666666666643$</td>
</tr>
<tr>
<td>4</td>
<td>$100.041666666666666572$</td>
<td>$61.37499999999999929$</td>
</tr>
<tr>
<td>5</td>
<td>$-140.0583333333333371$</td>
<td>$-78.6833333333333371$</td>
</tr>
<tr>
<td>6</td>
<td>$163.401388888888888744$</td>
<td>$84.718055555555555372$</td>
</tr>
<tr>
<td>7</td>
<td>$-163.401388888888888744$</td>
<td>$-78.6833333333333371$</td>
</tr>
<tr>
<td>8</td>
<td>$142.97621527777777544$</td>
<td>$64.292881944444444173$</td>
</tr>
<tr>
<td>9</td>
<td>$-111.2037229938271423$</td>
<td>$-46.9108410493827250$</td>
</tr>
<tr>
<td>10</td>
<td>$77.8426060956790025$</td>
<td>$30.9317650462962774$</td>
</tr>
</tbody>
</table>

6th term is $\frac{7^6}{7!} = \frac{7 \cdot 7 \cdot 7 \cdot 7}{6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}$, decrease only for $j > 7$.

Note: Partial sums about $10^5$ times larger than answer, So will lose at least 5 digits. Expect at most 10 digits correct.
Look more carefully at computation of \( \exp(-7.) \):

<table>
<thead>
<tr>
<th>( j )</th>
<th>( j^{th} ) term</th>
<th>partial_sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(-7.0000000000000000)</td>
<td>(-6.0000000000000000)</td>
</tr>
<tr>
<td>6</td>
<td>(163.40138888888888744)</td>
<td>(84.7180555555555372)</td>
</tr>
</tbody>
</table>

Note: Only 10 correct digits.

Adding more terms won’t help!!
Better way to compute $\exp(-7.)$: $e^{-7} = 1/e^7$.

All terms in Taylor series for $\exp(7)$ are positive, no cancellation:

True: 1096.6331584284585006 1096.6331584284585006

<table>
<thead>
<tr>
<th>j</th>
<th>$j$'th term</th>
<th>partial_sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.0000000000000000</td>
<td>8.0000000000000000</td>
</tr>
<tr>
<td>2</td>
<td>24.5000000000000000</td>
<td>32.5000000000000000</td>
</tr>
<tr>
<td>3</td>
<td>57.1666666666666643</td>
<td>89.66666666666666572</td>
</tr>
<tr>
<td>4</td>
<td>100.0416666666666572</td>
<td>189.7083333333333144</td>
</tr>
<tr>
<td>5</td>
<td>140.0583333333333371</td>
<td>329.7666666666666515</td>
</tr>
<tr>
<td>6</td>
<td>163.4013888888888744</td>
<td>493.1680555555554974</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>41</td>
<td>0.00000000000000013</td>
<td>1096.6331584284575911</td>
</tr>
<tr>
<td>42</td>
<td>0.00000000000000002</td>
<td>1096.6331584284575911</td>
</tr>
</tbody>
</table>

Note: 15 correct digits

```python
>>> exp(-7.)
0.00091188196555451624
```

```python
>>> 1. / taylor.exp(7.,40)
0.000911881965554517
```
Some disasters

Usually rounding errors are negligible compared to other errors introduced by numerical methods (e.g. truncating the Taylor series, discretizing a differential equation, etc.)

But some notable disasters have been caused by rounding error, see

http://www.ima.umn.edu/~arnold/disasters/

These issues are explored more in courses on numerical analysis.

Numerical analysis mostly deals with much more interesting things, but some understanding of the limitations of computer arithmetic is essential.
For now, assume we have a single file `filename.f90` that contains the main program and also any functions or subroutines needed.

Next week will see how to split into separate files.

Will also discuss use of `modules`.
Fortran functions and subroutines

For now, assume we have a single file filename.f90 that contains the main program and also any functions or subroutines needed.

Next week will see how to split into separate files.

Will also discuss use of modules.

**Functions** take some input arguments and return a single value.

Usage: \[ y = f(x) \quad \text{or} \quad z = g(x,y) \]

Should be declared as **external** with the type of value returned:

\[ \text{real(kind=8), external :: f} \]
Fortran functions

```fortran
! $CLASSHG/codes/fortran/fcn1.f90

program fcn1
  implicit none
  real(kind=8) :: y, z
  real(kind=8), external :: f

  y = 2.
  z = f(y)
  print *, "z = ", z
end program fcn1

function f(x)
  implicit none
  real(kind=8), intent(in) :: x
  real(kind=8) :: f
  f = x**2
end function f
```

Prints out: \( z = 4.000000000000000 \)
Subroutines have arguments, each of which might be for input or output or both.

Usage: call sub1(x,y,z,a,b)

Can specify the intent of each argument, e.g.

real(kind=8), intent(in) :: x,y
real(kind=8), intent(out) :: z
real(kind=8), intent(inout) :: a,b

specifies that x, y are passed in and not modified,
z may not have a value coming in but will be set by sub1,
a, b are passed in and may be modified.

After this call, z, a, b may all have changed.
Fortran subroutines

```fortran
! $CLASSHG/codes/fortran/sub1.f90

program sub1
  implicit none
  real(kind=8) :: y,z

  y = 2.
  call fsub(y,z)
  print *, "z = ",z
end program sub1

subroutine fsub(x,f)
  implicit none
  real(kind=8), intent(in) :: x
  real(kind=8), intent(out) :: f
  f = x**2
end subroutine fsub
```
Fortran subroutines

A version that takes an array as input and squares each value:

```fortran
! $CLASSHG/codes/fortran/sub2.f90

program sub2
  implicit none
  real(kind=8), dimension(3) :: y,z
  integer n

  y = (/2., 3., 4./)
  n = size(y)
  call fsub(y,n,z)
  print *, "z = ", z
end program sub2

subroutine fsub(x,n,f)
  ! compute f(x) = x**2 for all elements of the array x
  ! of length n.
  implicit none
  integer, intent(in) :: n
  real(kind=8), dimension(n), intent(in) :: x
  real(kind=8), dimension(n), intent(out) :: f
  f = x**2
end subroutine fsub
```