Basics of Computation
Basics of Computation

- Computers store information and allow us to operate on it.
  - That's basically it.
  - Computers have finite memory, so it is not possible to store the infinite range of numbers that exist in the real world, so approximations are made.

- Great floating point reference
  - *What Every Computer Scientist Should Know AboutFloating-Point Arithmetic* by D. Goldberg
Integers

- Basic unit of information in a computer is a **bit**: 0 or 1
- 8 bits = 1 byte
- Different types of information require more bits/bytes to store.
  - A **logical** (T or F) can be stored in a single bit.
    - C/C++: `bool` datatype
    - Fortran: `logical` datatype
  - **Integers**: Standard in many languages is 4-bytes. This allows for $2^{32}-1$ distinct values.
    - This can store: -2,147,483,648 to 2,147,483,647 (signed)
      - C/C++: `int` (usually) or `int32_t`
      - Fortran: `integer` or `integer*4`
    - Or it can store: 0 to 4,294,967,295 (unsigned)
      - C/C++: `uint` or `uint32_t`
      - Fortran (as of 95): `unsigned`
Integers (continued):

- Sometimes 2-bytes. This allows for $2^{16}-1$ distinct values.
  - This can store: -32,768 to 32,767 (signed)
    - C/C++: short (usually) or int16_t
    - Fortran: integer*2
  - Or it can store: 0 to 65,535 (unsigned)
    - C/C++: uint16_t
    - Fortran (as of 95): unsigned*2

- Note for IDL users: the standard integer in IDL is a 2-byte integer. If you do
  
  ```
  i = 2
  ``

  that's 2-bytes. To get a 4-byte integer, do:

  ```
  i = 2l
  ```
Overflow example...
program overflow

   integer i, iold

   iold = -1
   i = 0
   do while (i > iold)
      iold = i
      i = i+1
   enddo

   print *, i

end program overflow
Integers

- Python allows for the date size of the integer to scale with the size of the number: https://www.python.org/dev/peps/pep-0237/
  - Initially, it is the largest value supported in hardware on a machine (64-bits on a 64-bit machine—see `sys.maxint`)

```python
def fac(x):
    if x == 0:
        return 1
    else:
        return x*fac(x-1)
```

```python
a = fac(52)
print(a)
print(a.bit_length())
```

- This prints:

```
80658175170943878571660636856403766975289505440883277824000000000000000
226
```

Note: python 3.x does away with the distinction between int and long altogether.
Real Numbers

• **Floating point** format
  – Infinite real numbers on the number line need to be represented by a finite number of bits
  – **Approximations made**
    • Finite number of decimal places stored, maximum range of exponents.
  – Not every number can be represented.
    • In base-2 (what a computer uses), 0.1 does not have an exact representation (see S 2.1 of Goldberg)
Exact Representations

• \(0.1\) as represented by a computer:

```python
>>> b = 0.1
>>> print(type(b))
<class 'float'>
>>> print("{:30.20}".format(b))
  0.10000000000000000555

>>> import sys
>>> print(sys.float_info)
```
Floating Point

- In the floating point format, a number is represented by a **significand** (or mantissa), and an **exponent**, and a **sign**.
  - Base 2 is used (since we are using bits)
  - \[ 0.1 \sim (1 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} + 1 \cdot 2^{-5} + \ldots) \cdot 2^{-4} \]
    - significand

- All significand's will start with 1, so it is not stored

- **Single precision**:
  - Sign: 1 bit; exponent: 8 bits; significand: 24 bits (23 stored) = 32 bits
  - Range: \(2^{127-1}\) in exponent (because of sign) = \(2^{127}\) multiplier \(\sim 10^{38}\)
  - Decimal precision: \(~6\) significant digits

- **Double precision**:
  - Sign: 1 bit; exponent: 11 bits; significand: 53 bits (52 stored) = 64 bits
  - Range: \(2^{1023-1}\) in exponent = \(2^{1023}\) multiplier \(\sim 10^{308}\)
  - Decimal precision: \(~15\) significant digits
Floating Point

- Overflows and underflows can still occur when you go outside the representable range.
  - The floating-point standard will signal these (and compilers can catch them)

- Some special numbers:
  - $\text{NaN} = 0/0$ or $\sqrt{-1}$
  - $\text{Inf}$ is for overflows, like $1/0$
  - Both of these allow the program to continue, and both can be trapped (and dealt with)
  - $-0$ is a valid number, and $-0 = 0$ in comparison

- Floating point is governed by an IEEE standard
  - Ensures all machines do the same thing
  - Aggressive compiler optimizations can break the standard
Numerical Precision

- Finite number of bits used to represent #s means there is a finite precision, the **machine epsilon**, $\varepsilon$
  - One way to find this is to ask when $1 + \varepsilon = 1$
    
    ```python
    x = 1.0
    eps = 1.0
    while x + eps != x:
        eps = eps/2.0
    # machine precision is 2*eps, since that was the last
    # value for which 1 + eps was not 1
    print(2*eps)
    ```
    - This gives $2.22044604925e-16$
Integer Division

- How will your computer code evaluate:
  \[ \frac{1}{2} \]

- The answer depends on the language
  - Most programming languages will make the result of dividing 2 integers also be an integer
  - Python 3 is an exception

- Be careful with integer division—this is a common cause of bugs

- Let’s look at examples...
Numerical Precision

- This means that **most real numbers do not have an exact representation on a computer.**
  - Spacing between numbers varies with the size of numbers
  - Relative spacing is constant

\[
\text{relative roundoff error} = \frac{|\text{true number} - \text{computer number}|}{|\text{true number}|} \leq \epsilon
\]
Round-off Error

- **Round-off error** is the error arising from the fact that no every number is exactly representable in the finite precision floating point format.
  - Can accumulate in a program over many arithmetic operations.
Round-off Example 1

(Yakowitz & Szidarovszky)

- Imagine that we can only keep track of 4 significant digits
- Compute $\sqrt{x + 1} - \sqrt{x}$
  - Take $x = 1984$
  - Keeping only 4 digits each step of the way,
    $$\sqrt{x + 1} - \sqrt{x} = 44.55 - 44.54 = 0.01$$
  - We've lost a lot of precision
- Instead, consider:
  $$\sqrt{x + 1} - \sqrt{x} = (\sqrt{x + 1} - \sqrt{x}) \left( \frac{\sqrt{x + 1} + \sqrt{x}}{\sqrt{x + 1} + \sqrt{x}} \right) = \frac{1}{\sqrt{x + 1} + \sqrt{x}}$$
- Then
  $$\sqrt{1985} - \sqrt{1984} = \frac{1}{\sqrt{1985} + \sqrt{1984}} = \frac{1}{44.55 + 44.54} = 0.01122$$
Round-off Example 2
(Yakowitz & Szidarovszky)

- Consider computing \( e^{-24} \) via a truncated Taylor series
  \[
e^x \sim S(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}
  \]
- The error in this approximation is less than
  \[
  \frac{|x|^{n+1}}{(n + 1)!} \max\{1, e^x\}
  \]
- We can compute \( S(-24) \) by adding terms until the term is less than machine precision
  - We find \( S(-24) = 3.44305354288101977E-007 \)
  - But \( e^{-24} = 3.77513454427909773E-011 \)
  - The error is far bigger than the true answer!
- Let’s look at the code...
Look at the terms—we are relying on cancellation to get our result.
\begin{itemize}
  \item Instead recognize:
    \[ e^{-24} = \left(e^{-1}\right)^{24} \implies S(-24) = S(-1)^{24} \]
  \item \( S(-1) \) is well behaved, since each term is smaller in absolute magnitude than the previous.
    \begin{itemize}
      \item \( S(-1) = 0.36787944117144245 \)
      \item \( S(-1)^{24} = 3.77513454427912681E-011 \)
      \item \( \exp(-24) = 3.77513454427909773E-011 \)
    \end{itemize}
\end{itemize}
Round-off Example 3

- In spherically-symmetric codes (like a stellar evolution code), it is common to compute the volume of a spherical shell as

\[ V = \frac{4}{3} \pi (r_r^3 - r_l^3) \]

- This relies on the cancellation of two very big numbers.
- Rewriting reduces round-off errors:

\[ V = \frac{4}{3} \pi \Delta r (r_l^2 + r_l r_r + r_r^2) \]
Comparing Floating Point #s

- When comparing two floating point #s, we need to be careful.
- If you want to check whether 2 computed values are equal, instead of asking if
  \[ x = y \]
- it is safer to ask whether they agree to within some tolerance
  \[ |x - y| < \epsilon \]
Associative Property

- You learned in grade school that: \((a + b) + c = a + (b + c)\)
  - Not true with floating point
  - You can use parentheses to force order of operations
  - If you want to enforce a particular association, use parenthesis

\[
a = 1.0 \\
b = -1.0 \\
c = 2.0e-15
\]

\[
(a + b) + c \\
2e-15
\]

\[
a + (b + c) \\
1.9984014443252818e-15
\]
Associative Property

- Adding lots of numbers together can compound round-off error
  - One solution: sort and add starting with the smallest numbers
- Kahan summation (see reading list)
  - Algorithm for adding sequence of numbers while minimizing roundoff accumulation
  - Keeps a separate variable that accumulates small errors
  - Requires that the compiler obey parenthesis
Computer Languages

- You can write any algorithm in any programming language—they all provide the necessary logical constructs
  - However, some languages make things much easier than others
    - **C**
      - Excellent low-level machine access (operating systems are written in C)
      - Multidimensional arrays are “kludgy”
    - **Fortran** (Formula Translate)
      - One of the earliest complied languages
      - Large code base of legacy code
      - Modern Fortran offers many more conveniences than old Fortran
      - Great support for arrays
    - **Python**
      - Offers many high-level data-structures (lists, dictionaries, arrays)
      - Great for quick prototyping, analysis, experimentation
      - Increasingly popular in scientific computing
Computer Languages

- **IDL**
  - Proprietary
  - Great array language
  - Old: modern (like object-oriented programming) features break the “clean-ness” of the language
  - Losing popularity—no reason to start anything new with IDL
- **C++**
- **Others**
  - Julia, Ruby, Perl, shell scripts, ...
Roundoff vs. Truncation Error

- Roundoff error is just one of the errors we deal with.
- Translating continuous mathematical expressions into discrete forms introduces **truncation error**.
- Consider the Taylor series expansion for sine:

\[
\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots
\]

- If \(x\) is small, then we can neglect the higher-order terms (we truncate the series). This introduces an error.
Roundoff vs. Truncation Error
(Yakowitz & Szidarovszky)

- Truncation error will need to be understood for all of our methods
- Consider the definition of a derivative:

  \[ f'(x) = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h} \]

- We can't take the limit to 0. Our discrete version will be:

  \[ D_h(x) = \frac{f(x + h) - f(x)}{h} \]

- If we choose h small enough, this should be a good approximation.
Roundoff vs. Truncation Error
(Yakowitz & Szidarovszky)

Truncation error dominates
Roundoff error dominates
An error has occurred. To continue:

Press Enter to return to Windows, or

Press CTRL+ALT+DEL to restart your computer. If you do this, you will lose any unsaved information in all open applications.

Error: OE : 016F : BFF9B3D4

Press any key to continue
Bugs

One of the first real computer bugs (1947)
(Naval Surface Warfare Center, Dahlgren, VA.; Wikipedia)
Bugs

• Every code has bugs

• Good software engineering practices can help identify and prevent bugs
  – Testing allows you to
    • Find bugs
    • Prevent the bugs from reoccurring
    • We'll look at both unit testing and regression
      – Unit testing tests individual routines in isolation
      – Regression testing looks at the entire program and checks whether new changes result in a change in the answer
  – Version control tracks changes to our software and lets us manage development in a team