CSC 336: Numerical Methods Week 2 - Floating Point

Summer 2020

Week 1 recap

- Readings + Lectures
  - Introduction
  - Scientific Computing
  - Approximation and Error
  - Conditioning and Stability
- Worksheets
  - Binary representations - motivating floating point
  - Approximating derivatives - truncation + rounding error

Week 2 plan

- Lecture + readings
- Worksheets will be posted by end of Monday
- Tuesday
    - Office hour 1-2pm (or by request)
    - **Homework #1 due by end of day**
- Wednesday 6-9pm
    - Q+A to start the first hour (will record)
    - Small group worksheet time / office hours (may record a group)
    - Possibly record a tutorial video in the last hour (about floating point arithmetic)
- Wednesday Homework #2 will be posted
- Thursday Office hour 1-2pm (or by request)
- Friday Assignment #1 will be posted (hopefully) and any extra materials from the week

Floating Point

- Last week we went through an exercise pointing out the limitations of using a fixed point number system to approximate the real numbers.
    - Large number of bits required to store numbers of different scales.
    - We touched on the idea that scientific notation is really the solution we need.
- Approximating the reals is necessary in order to perform any numerical computation
    - You *can* do exact symbolic computation, but that isn't the topic of this course (a good project for those interested though).

Additional Takeaway Points

- **Precision** refers to how many **significant digits** a number has - it is directly related to machine epsilon for normalized floating point numbers.

- A relative error of 10^{-p} corresponds to roughly p correct significant digits. **(from top of page 6 in Heath)**

- **Accuracy** refers to how many correct significant digits a number has. (Cancellation leads to a loss of precision, as we will see today.)

- If a number is denormalized, it reverts back to using a fixed point representation (with leading zeros) and so has less precision (also seen today).

Floating-point numbers

A simplified form for representing a floating-point number x in **base** b is the following:

$$x = (f)_b \times b^{(e)_b}$$

$f = \pm(.\, d_1 d_2 \cdots d_t)_b$ is called the **mantissa** (or **significand**).
$e = \pm(c_{s-1} c_{s-2} \cdots c_0)_b$ is an integer called **exponent** (or **characteristic**).
A computer in which numbers are represented as above is said to have t base-b digits **precision**.

The floating-point number is **normalised** if $d_1 \neq 0$, or else $d_1 = d_2 = \cdots = d_n = 0$.
Significant digits (of a nonzero floating-point number) are the digits following (including) the first nonzero digit of the mantissa. All the digits of the mantissa of a normalised floating-point number are significant.

Floating-point numbers -- limits

The absolute value of the mantissa is always ≥ 0 and < 1.

The exponent is (also) limited: $E_{\min} \leq e \leq E_{\max}$.
For a set of floating-point numbers in a simplified form, $-E_{\min} = E_{\max} = (aa \cdots a)_b$, where $a = b - 1$. (However, the IEEE Standard uses a slightly different range for the exponent.)

According to the above simplified form,
the largest floating-point number (often called **overflow level**, OFL) is
$N_{\max} = (.\, aa \cdots a)_b \times b^{(aa \cdots a)_b}$, where $a = b - 1$, while
the smallest in absolute value nonzero floating-point number (often called **underflow level**, UFL) is
$N_{\min} = (.\, 100 \cdots 0)_b \times b^{-(aa \cdots a)_b}$ if normalised, or
$N_{\min} = (.\, 00 \cdots 01)_b \times b^{-(aa \cdots a)_b}$ else.
From now on, we assume normalised numbers (mantissae), unless otherwise stated.

Set of floating-point numbers

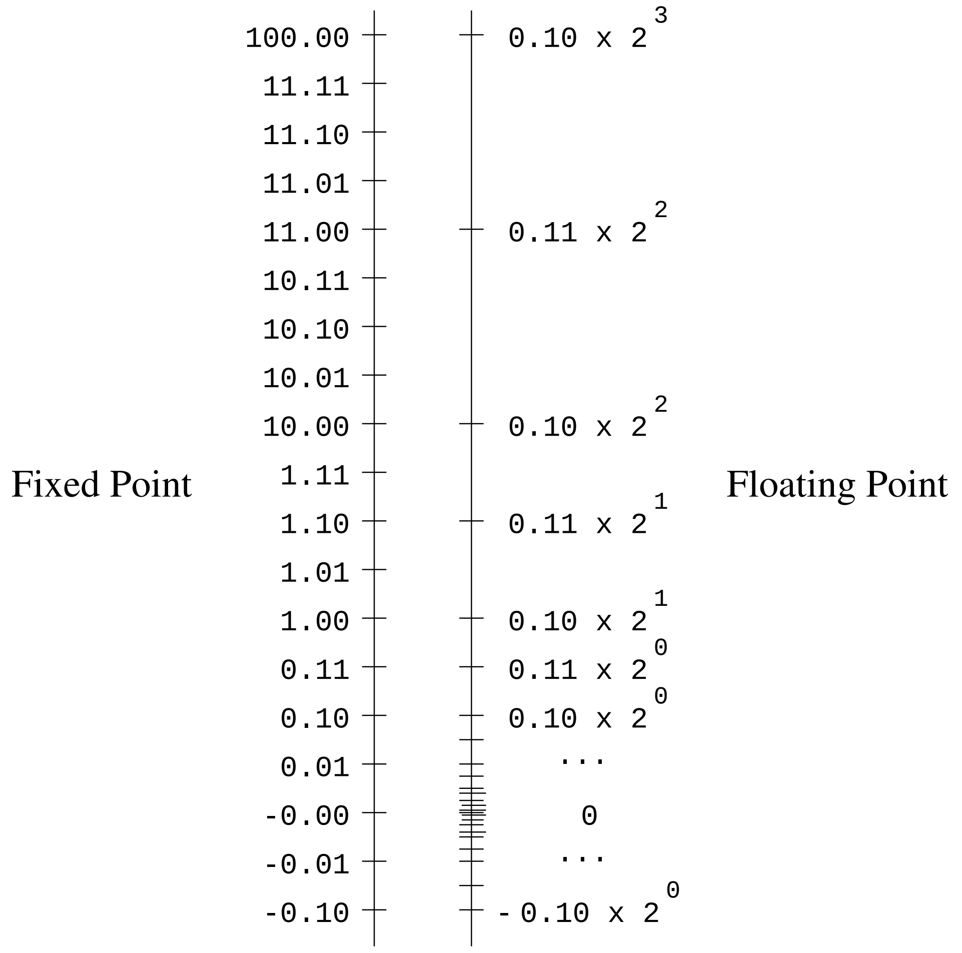
Notation: $\mathbb{R}_b(t, s)$ denotes the set of all base b floating-point numbers that can be represented by t b-digits mantissa (incl. sign), and s b-digits exponent (incl. sign).
Note: $\mathbb{R}_b(t, s) \subset [-OFL, -UFL] \cup \{0\} \cup [UFL, OFL]$.

Overflow occurs whenever a floating-point number greater than $N_{\max}$ or smaller than $-N_{\max}$ has to be stored in the computer.
Underflow occurs whenever a nonzero floating-point number in the range $(-N_{\min}, N_{\min})$ has to be stored in the computer. (Some exceptions may apply in IEEE Standard.)

Notes:

- $\mathbb{R}_b(t, s)$ is finite, while $\mathbb{R}$ is infinite.
- $\mathbb{R}$ is dense, while $\mathbb{R}_b(t, s)$ is not. There are many "holes" in the real axis, which are not uniform, and are not represented on the computer.
- The representable numbers are concentrated towards 0.

Comparing floating-point to fixed-point numbers



Fixed-point numbers look like integers multiplied by some (negative) power of the base. The number of digits to the left and to the right of the dot is fixed and there is no exponent.

Approximation of real numbers by floating-point numbers

A real number $x = \pm(x_I . x_F)_b \; = \pm(d_k d_{k-1} \cdots d_0 . d_{-1} \cdots)_b$ can be easily represented in a floating-point form (assuming no overflow or underflow occurs). First normalise the mantissa:

$$x = (d_k \cdots d_0 . d_{-1} \cdots)_b = (. D_1 D_2 \cdots)_b \times b^{k+1}$$

Then, there are two common ways to convert $x \in \mathbb{R}$ to a floating-point number $fl(x) \in \mathbb{R}_b(t, s)$.

(a) **Chopping**: chop after digit t of the mantissa.

(b) **Rounding (traditional)**: chop after digit t, then round D_t up or down, depending on whether $D_{t+1} \geq b/2$ or $D_{t+1} < b/2$ respectively, or in other words, add $b/2$ to D_{t+1}, then chop.

(c) **Rounding (proper or perfect)**: same as traditional, except when $D_{t+1} = b/2$ and $D_{t+2} = D_{t+3} = \cdots = 0$; then round D_t up or down, to the nearest even.

Approximation of real numbers by floating-point numbers

• Examples (in 2 decimal digits) (*):

x		chop	trad. round	proper round
2/3	.666...	.66	.67	.67
-3.05	$-.305 \times 10^1$	$-.30 \times 10^1$	$-.31 \times 10^1$	$-.30 \times 10^1$
-3.15	$-.315 \times 10^1$	$-.31 \times 10^1$	$-.32 \times 10^1$	$-.32 \times 10^1$
-3.155	$-.3155 \times 10^1$	$-.31 \times 10^1$	$-.32 \times 10^1$	$-.32 \times 10^1$
-3.055	$-.3055 \times 10^1$	$-.30 \times 10^1$	$-.31 \times 10^1$	$-.31 \times 10^1$

• Examples (in 2 binary digits) (*):

x		chop	trad. round	proper round
.101	$.101 \times 2^0$	$.10 \times 2^0$	$.11 \times 2^0$	$.10 \times 2^0$
.111	$.111 \times 2^0$	$.11 \times 2^0$	$.10 \times 2^1$	$.10 \times 2^1$

* We assume no overflow or underflow while rounding.

The IEEE Standard

In 1985, the Institute of Electrical and Electronics Engineers (IEEE) approved the so-called **IEEE Standard** for binary floating-point numbers representation and arithmetic operations. According to the IEEE Standard, the form for representing floating-point numbers is slightly more sophisticated than the above simplified form.

Without getting into details, IEEE Standard floating-point numbers are of the form

$$(-1)^q \times (d_0 . d_1 d_2 \cdots d_{t-1})_b \times 2^e$$

where $q = 0$ or 1, $(-1)^q$ is the sign of the number,
$d_i = 0$ or 1, for $i = 0, \cdots, t-1$, and
$E_{\min} \le e \le E_{\max}$ (with $E_{\min} = -E_{\max} + 1$), $e = (-1)^p \times (c_{s-2} \cdots c_1 c_0)_b$, where $p = 0$ or 1, $c_i = 0$ or 1, for $i = 0, \cdots, s-1$.

Essentially, only q, d_i, $i = 1, \cdots, t-1$, p and c_i, $i = 0, \cdots, s-2$, are stored.

There are numbers of **single precision** (binary32), **double precision** (binary64), and **quadruple precision** (binary128), with the characteristics as given in the table below.

The IEEE Standard

type of number	no. of bits	t	$E_{\min}$	$E_{\max}$	$\varepsilon_{\text{mach}}$
single precision, binary32	32	24 (=23+1)	-126	+127	1.2×10^{-7}
double precision, binary64	64	53 (=52+1)	-1022	+1023	2.2×10^{-16}
quadruple prec., binary128	128	113 (=112+1)	-16382	+16383	1.9×10^{-34}

The mantissa is normalised: $d_0 = 1$ (or else the number is 0); thus, d_0 need not be stored, and a t-bit mantissa needs $t - 1$ storage bits.
The IEEE Standard uses proper rounding.

The IEEE Standard also includes some "special" numbers, for example $+\infty$, $-\infty$, NaN (Not-a-Number), to handle indeterminate values that may arise in some computations.

The IEEE Standard is adopted today by many modern computer systems.

The respective characteristics in a decimal system are given (approximately) below:

type of number	t	$E_{\min}$	$E_{\max}$
single precision, binary32	7	-38	+38
double precision, binary64	16	-308	+308
quadruple prec., binary128	34	-4928	+4928

About $\varepsilon_{\text{mach}}$, later.

Round-off error (representation error)

The difference between x and $fl(x)$ is called the **round-off** error. It is roughly proportional to x, so we can write $fl(x) - x = x\delta$, or $fl(x) = x(1 + \delta)$, where δ is the **relative round-off error**. Note that δ may depend on x but there is a bound for it, independent of x, called **unit round-off** and often denoted by μ or u.

- $|\delta| \le b^{1-t}$ if normalised numbers and chopping are assumed,
- $|\delta| \le \frac{1}{2} b^{1-t}$ if normalised numbers and rounding are assumed.

An approximation $\hat{x}$ to x is said to be **correct in** r **significant** b**-digits**, if $\left|\frac{x - \hat{x}}{x}\right| \le \frac{1}{2} b^{1-r}$ $(= 5b^{-r}$ if $b = 10)$.

Absolute and relative errors

If x is a number and $\hat{x}$ an approximation to x, then
the **(absolute) error** in the approximation is $x - \hat{x}$, and
the **relative error** in the approximation is $\frac{x - \hat{x}}{x}$.

Often, we are interested only in the magnitude of the absolute and relative errors, in which case we take the **absolute value** of the respective errors.

Computer arithmetic

Let $x, y \in \mathbb{R}_b(t, s)$, i.e., assume the numbers x, y are exactly representable on the computer system, and $o \in \{+, -, \times, /\}$. Then, for the computer's floating-point operation $\bar{o}$ corresponding to o, the following holds in general:

$$x \ o \ y \neq fl(x \ \bar{o} \ y)$$

The operation $\bar{o}$ may vary from machine to machine, but, in general, it is constructed so that

$$x \ \bar{o} \ y = fl(x \ o \ y)$$

This is achieved with the use of one (or two) extra temporary digits (bits).

Thus, **the error of a computation is the round-off error of the correct result**.
In other words, computer operations return the correctly rounded result (i.e. the number closest to the correct answer in the representation being used).

Computer arithmetic

• Examples (in 2 decimal digits, proper rounding)

$$x = .2 \times 10^1,\ y = .51 \times 10^{-5},\ x + y = .2000051 \times 10^1$$
$$x \bar{+} y = fl(x + y) = .2 \times 10^1$$

$$x = .2 \times 10^1,\ y = .51 \times 10^{-2},\ x + y = .2051 \times 10^1$$
$$x \bar{+} y = fl(x + y) = .21 \times 10^1$$

$$x = .2 \times 10^1,\ y = .50 \times 10^{-2},\ x + y = .2050 \times 10^1$$
$$x \bar{+} y = fl(x + y) = .2 \times 10^1$$

$$x = .21 \times 10^1,\ y = .50 \times 10^{-2},\ x + y = .2150 \times 10^1$$
$$x \bar{+} y = fl(x + y) = .22 \times 10^1$$

Note 1: $\mathbb{R}_b(t, s)$ is not closed w.r.t addition (nor w.r.t other mathematical operations)!
Note 2: The phenomenon in which a non-zero number is added to another and the latter is left unchanged is often referred to as **saturation**.

Computation of functions

Similarly, the computer application $\bar{f}(x)$ of a function f to a number $x \in \mathbb{R}_b(t, s)$ is constructed so that

$$\bar{f}(x) = fl(f(x))$$

that is, the error of computing a function value is the round-off error of the correct result.

• Example (in 2 decimals)

$\sin(1) = 0.84147098\ldots, \quad \overline{\sin}(1) = fl(\sin(1)) = fl(0.84147098\ldots) = 0.84$

Machine epsilon

The smallest (non-normalised) floating-point number $\varepsilon_{\text{mach}}$ with the property $1 + \varepsilon_{\text{mach}} > 1$, is called the **machine epsilon** (in short, **mach-eps**).
$\varepsilon_{\text{mach}} = b^{1-t}$ if chopping,
$\varepsilon_{\text{mach}} = \frac{1}{2} b^{1-t}$ if traditional rounding and
$\frac{1}{2} b^{1-t} < \varepsilon_{\text{mach}} \le b^{1-t}$, ($\varepsilon_{\text{mach}} = \frac{1}{2} b^{1-t} + b^{-t}$) if proper rounding.
Thus, $-\varepsilon_{\text{mach}} \le \delta \le \varepsilon_{\text{mach}}$.

For IEEE Standard numbers, $\varepsilon_{\text{mach}} = 2^{1-t}$, i.e.
for single precision, $\varepsilon_{\text{mach}} = 2^{-23} \approx 1.19209 \times 10^{-7}$, and
for double precision, $\varepsilon_{\text{mach}} = 2^{-52} \approx 2.22045 \times 10^{-16}$.
for quadruple precision, $\varepsilon_{\text{mach}} = 2^{-112} \approx 1.9259 \times 10^{-34}$.

Note 1: There are alternative definitions of $\varepsilon_{\text{mach}}$ that result in slightly different values, still of the same order.

Note 2: We have $0 < \text{UFL} < \varepsilon_{\text{mach}} < \text{OFL}$.

Error propagation in simple arithmetic calculations

Recall that the relative round-off error δ in the floating-point representation $fl(x)$ of a number x (i.e. $\delta = \frac{fl(x) - x}{x}$) satisfies $-\varepsilon_{\text{mach}} \le \delta \le \varepsilon_{\text{mach}}$.

As soon as an error (round-off or by a floating-point operation) rises, it may then be amplified or reduced in subsequent operations.

Let $x, y \in \mathbb{R}$. Then $fl(x) = x(1 + \delta_1)$, $fl(y) = y(1 + \delta_2)$

(a) Multiplication

Assume we want to multiply x and y. Then the computer computes

$$fl(x) \bar{\times} fl(y) = fl(fl(x) \times fl(y)) = (x(1+\delta_1)y(1+\delta_2))(1+\delta_3) =$$
$$= xy(1+\delta_1+\delta_2+\delta_1\delta_2)(1+\delta_3) =$$
$$= xy(1+\delta_1+\delta_2+\delta_1\delta_2+\delta_3+\delta_1\delta_3+\delta_2\delta_3+\delta_1\delta_2\delta_3) \approx$$
$$\approx xy(1+\delta_1+\delta_2+\delta_3) = xy(1+\delta_\times)$$

where $$|\delta_\times| \le 3\varepsilon_{\text{mach}}$$

(b) Division Similar (do it!).

Error propagation in simple arithmetic calculations

(c) Addition

Assume we want to add x and y. Then the computer computes

$$fl(x) \bar{+} fl(y) = fl(fl(x) + fl(y)) = (x(1+\delta_1) + y(1+\delta_2))(1+\delta_3) =$$

$$= x(1+\delta_1)(1+\delta_3) + y(1+\delta_2)(1+\delta_3) \approx x(1+\delta_1+\delta_3) + y(1+\delta_2+\delta_3) =$$

$$= (x+y)(1 + \frac{x}{x+y}(\delta_1+\delta_3) + \frac{y}{x+y}(\delta_2+\delta_3)) = (x+y)(1+\delta_+)$$

where $$|\delta_+| \le |\frac{x}{x+y}| \cdot 2\varepsilon_{\text{mach}} + |\frac{y}{x+y}| \cdot 2\varepsilon_{\text{mach}} = \frac{|x|+|y|}{|x+y|} \cdot 2\varepsilon_{\text{mach}}$$

$$= 2\varepsilon_{\text{mach}} \quad \text{when} \quad xy > 0$$

$$= 2\varepsilon_{\text{mach}} \frac{|x-y|}{|x+y|} \quad \text{when} \quad xy < 0$$

Consider the case $x \approx -y$. The (relative) error blows up.

Example of catastrophic cancellation

• Example (in 8 decimal digits with chopping)

Consider the real numbers
$x = .123456790 \times 10^0$ and
$y = .123456789 \times 10^0$.
Then compute the difference

$$fl(fl(x) - fl(y)) = .00000001 \times 10^0 = .10000000 \times 10^{-7},$$

while

$$x - y = .000000001 \times 10^0 = .10000000 \times 10^{-8}.$$

In the above computed $fl(fl(x) - fl(y))$ no significant decimal digits are correct. Although the (relative) error in the floating-point representation of y is at the level of 10^{-8}, the (relative) error in $fl(fl(x) - fl(y))$ is too high (at the level of 10^0).

$$\frac{.1 \times 10^{-8} - .1 \times 10^{-7}}{.1 \times 10^{-8}} = \frac{-.9 \times 10^{-8}}{.1 \times 10^{-8}} = -9 \times 10^0$$

Important note

Adding nearly opposite (or subtracting nearly equal) numbers may result in having no correct digits at all. In some cases though, we are able to eliminate this **cancellation phenomenon** (often referred to as **catastrophic cancellation**).

In general, most arithmetic operations do not give rise to huge errors. Adding nearly opposite (or subtracting nearly equal) numbers is an exception.

Error propagation in computation: conditioning of problems

Consider some computation with input $x \in \mathbb{R}$ and output $f(x) \in \mathbb{R}$.
Let $fl(x) = x(1 + \delta_x)$, and assume $f(x)$ is twice differentiable in a neighbourhood of x.
Then, if $f(x) \neq 0$, using Taylor's series and ignoring terms of second or higher order ($O(\delta_x^2)$), we get

$$f(fl(x)) \approx f(x)(1 + \delta_f) \ \text{ with } \ \delta_f = \frac{xf'(x)}{f(x)} \delta_x.$$

If $|\delta_x| \le \varepsilon_{\text{mach}}$, we get

$$|\delta_f| \le \left| \frac{xf'(x)}{f(x)} \right| \varepsilon_{\text{mach}}.$$

The factor $\kappa_f = \left| \frac{xf'(x)}{f(x)} \right|$ is called **(relative) condition number** of $f(x)$ and is a measure of the relative sensitivity of the computation of $f(x)$ on relatively small changes in the input x, or in other words a measure of how the relative error in x propagates in $f(x)$.

A computation is called **well-conditioned** if relatively small changes in the input, produce relatively small changes in the output, otherwise it is called **ill-conditioned**.

Note: Once $f(fl(x))$ is computed, it is stored/represented as $fl(f(fl(x))) = fl(f(x)(1 + \delta_f)) = f(x)(1 + \delta_f)(1 + \delta) \approx f(x)(1 + \delta_f + \delta)$, where $|\delta| \le \varepsilon_{\text{mach}}$.

Error propagation in computation: conditioning of problems

- Example: Let $f(x) = \sqrt{x}$. Then, the condition number of $f(x)$ is $\left|\frac{xf'(x)}{f(x)}\right| = \frac{1}{2}$. For this function, the condition number happens to be small (and independent of x). Thus the computation of the square root of a number is a well-conditioned computation.

- Example: Let $f(x) = e^x$. Then, the condition number of $f(x)$ is $\left|\frac{xf'(x)}{f(x)}\right| = |x|$. For this function, the condition number depends (actually linearly) on x. For small $|x|$, the condition number is small, for large $|x|$, it is large. However, for large $|x|$, we will have overflow (note: $e^{700} \approx 10^{304}$), thus we can say that the computation of the exponential is well-conditioned for all acceptable values of x.

- Note 1: If, for some function, $\kappa_f > \varepsilon_{\text{mach}}{}^{-1}$, we risk having no correct digits at all in the computed result $\overline{f(x)}$.
- Note 2: The condition number of a function is a property inherent to the function itself, and not to the way the function is computed.

Error propagation in computation: stability of algorithms

Stability is a concept similar to conditioning, but it refers to a numerical algorithm, i.e. to the particular way a certain computation is carried out. A numerical algorithm is **stable** if small changes in the algorithm input parameters have a small effect on the algorithm output, otherwise it is called **unstable**.

• Example (in 3 decimal digits with rounding)

$$(a-b)^2 = a^2 - 2ab + b^2$$

Let $a = 15.6$, $b = 15.7$

$$(a-b)^2 = (-.1)^2 = .01 \to .1 \times 10^{-1}$$

$$a^2 - 2ab + b^2 = 243.36 - 2 \times 244.92 + 246.49$$

$$\to 243 - 490 + 246 = 489 - 490 = -1 = -.1 \times 10^1$$

Computing the rhs we don't even get the sign correct! In this case, computing the lhs is a more stable algorithm to compute the square of the difference of two numbers, than computing the rhs.

Moral: Mathematically equivalent expressions are not necessarily computationally equivalent.

Error propagation in computation: stability of algorithms

General considerations

Mathematically equivalent expressions are not necessarily computationally equivalent.

There is no general rule to pick the most stable algorithm for a certain computation. However, we make an effort

- to avoid adding nearly opposite (or subtracting nearly equal) numbers,
- to minimize the number of operations,
- when adding several numbers, to add them starting from the smallest and proceeding to the largest.
- to be alert when adding numbers of very different scales.

No rule, though, guarantees success. Each case may be special.

Also, keep in mind, that picking a stable algorithm for a certain computation does not improve the condition number of the computation.

Next

- Complete the readings for this week. Work through some textbook problems.
- Let me know if there is anything from this week that you would like more resources for.
- Please look at any assigned course work when it comes out and read through it to get a feel for how much time it will take to complete.
- Check Piazza if you have questions.