Lecture 12 - Numpy Linear Algebra

* AMA / Quiz / Prayer

I. Numpy Linear Algebra

* We have spent a couple of days reviewing some important concepts in linear algebra and in programming. Fortunately, Python has made our life easier and included many functions for things we have had to code ourselves!

* These functions are found (for the most part) in the Numpy linear algebra module : numpy.linalg.

* Examples include:

  - Array math (elementwise addition, subtraction, multiplication, scalar multiplication)

    e.g. $\underline{v}+\underline{w}$, $\underline{v}-\underline{w}$, $c\underline{v}$, $\underline{v}*\underline{w}$

  - Sums and matrix-vector multiplication

    e.g. np.sum($\underline{v}*\underline{w}$), np.dot($\underline{v}$, $\underline{w}$)

    np.dot($\underline{\underline{A}}$, $\underline{v}$)

  - Transpose : $\underline{\underline{A}}^T \rightarrow$ A.T

  - Matrix and Vector Norms :

    $\|\underline{v}\| \rightarrow$ np.linalg.norm(v)

  - Linear solves (2 ways):

    $\underline{\underline{A}}\,\underline{x}=\underline{b} \longrightarrow$ x = np.linalg.solve(A, b)

    ⊛ This is the <u>better way</u>.

$\underline{x} = \underline{\underline{A}}^{-1} \cdot \underline{b}$

```
Ainv = np.linalg.inv(A)
x = np.dot(Ainv, b)
```

$\boxed{\text{Activity}}$

Practice with Numpy linear algebra functions.
(See Python file)

II. Computational cost.

* Now that we've talked a little bit about some methods for solving linear systems, we need to think a bit about how <u>good</u> they are.
Another way to say this is how much resources will it <u>cost</u> to solve a problem.

* As we discussed at the beginning of class, a computer has two kinds of resources: memory & the CPU

* We measure memory in bits:

e.g. array of 500 floats

1 float = 8 bytes = 64 bits

$$500 \text{ floats} \times 8 \frac{\text{bytes}}{\text{float}} = 4000 \text{ bytes}$$
$$= 4 \text{ kilobytes.}$$

8 bits = 1 byte

10^3 bytes = 1 KB 10^3 MB = 1 GB

10^3 KB = 1 MB 10^3 GB = 1 TB.

* The total memory consumed by your program is mostly the sum of all of the bits you have used in your variables.

* We measure the amount of CPU used in the <u>total running time</u> of the calculation. The total run time is determined by the number of "steps" or operations the CPU needs to execute.

* Usually <u>one line of executed code</u> is proportional to a fixed number of CPU execution units.

* <u>Example</u>: How long will this code take to execute?

		cost	# of times
1:	for i in range(n):	c_1	n
2:	print(i)	c_2	n

↑ How long it takes CPU to execute line #

```
>>> 0
    1
    2
    :
    n-1
```

Total time: $T(n) = c_1 n + c_2 n = (c_1 + c_2) n$

$\boxed{\text{Activity}}$ Calculate how long it takes the CPU to execute this code:

	Cost	# of times
for i in range(n):	c_1	n
for j in range(n):	c_2	n^2
A[i, j] = i*j	c_3	n^2

$$T(n) = c_1 n + c_2 n^2 + c_3 n^2 = c_1 + (c_2 + c_3) n^2$$

III. Asymptotic Behavior

* Cost (in CPU time or memory) is not really a big deal when 'n' is small. But, when 'n' is big we need to worry about it. So, what happens when $n \to \infty$?

* when $n \to \infty$, we only care about the largest power of n. For example, suppose:

$$T_1(n) = c_1 n^2 \qquad T_2(n) = c_2 n^2 + c_3 n$$

$$n = 10^5, \quad c_1 = 1.1, \quad c_2 = 1, \quad c_3 = 1$$

$$\left.\begin{aligned} T_1(n) &= 1.1 \times 10^{10} = 10^{10} + 10^9 \\ T_2(n) &= 10^{10} + 10^5 \end{aligned}\right\} T_1 >> T_2$$

lesson: only the largest power matters when n is big.

* Because we only care about the largest power, we use "asymptotic notation". we say:

$$T(n) = c_1 n^2 + c_2 n + \dots = \Theta(n^2)$$

"order n^2"

$$T(n) = c_1 n^3 + c_2 n^2 + \dots = \Theta(n^3)$$

"order n^3"

* If we can figure out what "order" our code is, it can help us predict how long it will take to run!

<u>Example</u>: Nested loop from activity. $T(n) = \Theta(n^2) = c n^2$

suppose for $n = 100 = 10^2$, $T(n) = 10$ sec.

How long will it take for $n = 10^4$?

$$c \cdot (10^2)^2 = 10 \text{ sec} \Rightarrow c = 10^{-3} \text{sec.}$$

$$T(10^4) = 10^{-3}\text{sec} \cdot (10^4)^2 = 10^5 \text{ sec.} \approx 1 \text{ day!}$$

100 x bigger takes 10,000 x longer!