1 Introduction

The following are course notes made during the live lectures of EE 636, taken by Prof. Debasattam Pal (Spring 2021) **Logistics**

- Google Classroom - join codes shared on Moodle
- login to classroom regularly to check for assignments, announcements
- live lectures
- do not rely on the lectures being recorded

Reference books

- David Watkins - Matrix Computations (Main)
- Golub - Matrix Compuations

Grading Policy

- TA Proctored quizes - 20%
- Midsem - 20%
- Endsem - 40%
- Assignments - 10%
- TakeHomeExam/viva/project/coding - 10%

Take home exams will be significantly harder than assignments, which will be standard problems. Deadlines will be stricter for take-home-exams.

Learning more about HPC in specific will require you to look into topics that are beyond the scope of the course, but are in the reference book.

As such the main aim of this course is to learn what happens behind the scenes when we call library functions and learn how to write better code. **It is important that we understand when the result of a computation can be trusted or not.** We will see when a computer is prone to make errors. It has something to do with the condition number of the matrix.

Condition number is defined as

$$K_2(A) = \|A\|_2 \left\|A^{-1}\right\|_2.$$

We will spend a lot of time seeing how to solve $Ax = b$. We might not look at specific examples, but the content covered will apply more or less directly in some use cases e.g. image processing.

Starting off with a small simple assignment, to be submitted before the next lecture. **Assignment**

- Go through the syllabus
- Write briefly about a problem (from your area of expertise) that needs knowledge from any of the syllabus topics
- Feel free to come up with multiple examples. The more examples $\implies$ more credits.

- Submission due by next Monday (11th Jan)
- To be submitted on the classroom
- submission format `.pdf`

Professor asked us here what exactly are we looking for in this course. Some answers were given by students. The main purpose of this course is to understand what happens behind the scenes when we call linalg library functions, and consequently understand whether a particular computation is trustworthy or not.

1.1 Types of Problems in Matrix Computations

1.1.1 $\mathrm{A}\vec{x} = \vec{b}$

where we solve for x, where $A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m, x \in \mathbb{R}^n$. Also, note that $\mathbb{R}^n = \mathbb{R}^{n \times 1}$.

1.1.2 $\mathbf{argmin}_{\vec{x}} \left\| \mathrm{A}\vec{x} - \vec{b} \right\|$

Sometimes we cannot find solve for $\vec{x}$ exactly, in which case we would like to minimize some norm of the above kind.

1.1.3 $\mathrm{a}\vec{x} = \lambda\vec{x}$

Here we need to solve for both $\vec{x}$ and λ. This is a very important class of problems and also not straightforward. The characteristic equation to be solved is

$$|\mathrm{A} - \lambda \not{\mathbb{1}}| = 0 \tag{1}$$

And then we have the older problem of finding $\vec{x}$, the eigenvectors. The equation 1 is basically finding the roots of a polynomial, which is very nontrivial ($\because$ in general there is no closed form for degrees ≥ 5, as shown by Henry Abel.)

Something called the QR algorithm can find both eigvals and eigvecs in one shot.

1.1.4 $\mathrm{A}\vec{u} = œ\vec{v}$

A is known, the rest are unknown. And

$$\vec{u_1} \quad \vec{u_2} \ldots \vec{u_n} \tag{2}$$
$$\vec{v_1} \quad \vec{v_2} \ldots \vec{v_m} \tag{3}$$
$$\vec{u_i} \tag{4}$$

- Singular value decomposition
-

Questions that we are concerned with, given some problems

- How does a computer solve them?
- How trustworthy are the solutions?

- How efficient are the algorithms?

Something was said about the etymology of the word “algorithm". The prof then talks about the Caesar cypher.

The letter “E" is used most often in English. If you have a very large encrypted text, you could find the Caesar cypher key by finding the most frequent letter in the encrypted text. Pretty neat huh.

The enigma machine would randomly choose a key every day and the decoder machine could automatically figure out the key from the encrypted text.

Let's get down to computations and all. What does a computer do when you tell it to $\mathrm{A}\vec{x} = \vec{b}$.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13}\dots & a_{1n} \\ a_{21} & a_{22} & a_{23}\dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & a_{m3}\dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \tag{5}$$

Internally, it computes something like

$$\begin{pmatrix} \sum_{i=1}^{n} a_{1i}x_i \\ \vdots \\ \sum_{i=1}^{n} a_{mi}x_i \end{pmatrix} \tag{6}$$

Task: Compute the number of tasks required in matrix vector multiplication. Here, task means FLOP. So, find the number of FLOP. Prove that rowwise FLOP $= 2mn$ and columnwise FLOP $= 2mn$ and $\mathrm{A} \cdot B$ FLOP $= 2mnp$

Summary

The purpose of this course is to

- learn how a computer can solve the four types of problems mentioned a
- Effect of round-off errors
- When can a solution be trusted
- FLOPS to measure computation time
- Counted the FLOPS for basic matrix multiplication: $2mn$ for a matrix of size $m \times n$ and a vector of dim n.

Solvinig a linear system of linear equations

Triangular Matrix

$$\mathrm{A}\vec{x} = \vec{b} \tag{7}$$

where $\mathrm{A} \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^n$. The first step to learning how to solve such problems is learning about triangular matrix. Consider the following system with a lower

triangular matrix

$$\begin{bmatrix} l_{11} & 0 & 0 & \dots & 0 \\ l_{21} & l_{22} & 0 & \dots & 0 \\ l_{31} & l_{32} & l_{33} & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ l_{n1} & l_{n2} & l_{n3} & \dots & l_{nm} \end{bmatrix} \tag{8}$$

```
for k = 1,2,...n
        for i = 1,2,...,k-1
                b[k] <- b[k] - l[k][i]b[k]
        end
        if l[k][k] == 0 flag ``error"; exit
        else b[k] <- b[k]/l[k][k]
end
```

For the above algorithm (*forward substitution*), FLOPS $= n(n-1)+n = n^2$. Innermost loop has $2(k-1)$ FLOPS. The loop is called n times for $k = 1, 2, \dots n$. There is another division operation after a whole pass over the the inner loop. So, we have $n + \sum_{j=1}^{n} a_n z^n$ TODO.

In general, we can have a column-oriented algorithm for a row-oriented operation. The corresponding algo is called *backward substitution*. FLOPs will be the same as the row-oriented business.

The question of solvability

In real life we do not really get such nice looking lower-triangular or upper triangular matrices. Consider the following theorem

Theorem: Let A $\in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$ be given. Then the following are equivalent

1. A$\vec{x} = \vec{b}$ has a unique solution
2. A^{-1} exists
3. det$A \neq 0$
4. The rows of A are linearly independent
5. The columns are linearly independent
6. A$\vec{y} = 0 \iff y = 0$

How do we prove 1 $\implies$ 2? (Notation: $\vec{e_i}$ are the unit vectors). $\vec{b} = \sum_{j=1}^{n} a_j \vec{e_j}$ And $\vec{x} = \sum_{j=1}^{n} x_i \vec{e_i}$

$$\text{A}\vec{x} = \sum_{i=1}^{n} x_i \text{A}\vec{e_i}.$$

LOL IDK what I am doing.

Gaussian Elimination

You know what Guassian elimination is from MA 214. Summarizing naïve Gaussian elimination here. We use elementary transformation E to reduce A to an upper triangular form and solve then use backward substitution.

$$\begin{aligned}\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ -m_{21} & 1 & 0 & \dots & 0 \\ -m_{31} & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ -m_{n1} & 0 & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{n_1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix} \\ = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & \dots & a_{2n}^{(1)} \\ 0 & a_{32}^{(1)} & a_{33}^{(1)} & \dots a_{3n}^{(1)} & \\ \dots & \dots & \dots & \dots & \dots \\ 0 & a_{n2}^{(1)} & a_{n3}^{(1)} & \dots & a_{nn}^{(1)} \end{bmatrix}\end{aligned} \tag{9}$$

where

$$m_{j1} = \frac{a_{j1}}{a_{11}}, \quad a_{j1} \neq 0 \tag{10}$$

This completes one step of procuring an upper triangular matrix. The next step proceeds by pre-multiplying a similar elementary transformation matrix to equation (9)

$$\begin{aligned}\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & -m_{32} & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & -m_{n2} & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & \dots & a_{2n}^{(1)} \\ 0 & a_{32}^{(1)} & a_{33}^{(1)} & \dots & a_{3n}^{(1)} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & a_{n2}^{(1)} & a_{n3}^{(1)} & \dots & a_{nn}^{(1)} \end{bmatrix} \\ = \begin{bmatrix} a_{11} & a_{21} & a_{13} & \dots & a_{1n} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & \dots & a_{2n}^{(1)} \\ 0 & 0 & a_{33}^{(2)} & \dots & a_{3n}^{(2)} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & a_{n3}^{(2)} & \dots & a_{nn}^{(2)} \end{bmatrix},\end{aligned} \tag{11}$$

where

$$m_{j2} = \frac{a_{j2}^{(1)}}{a_{22}^{(1)}}. \tag{12}$$

You can see the upper triangular matrix forming. In general, we have

$$m_{ij} = \frac{a_{ij}^{(j-1)}}{a_{jj}^{(j-1)}}. \tag{13}$$

So, Gaussian elimination is basically the process

$$E_{n-1}E_{n-2}\dots E_2E_1A = \overline{A} \tag{14}$$

where $\overline{A}$ is an upper-triangular matrix. The product $E_{n-1}E_{n-2}\dots E_2E_1$ ends up being an upper triangular matrix.

When can we proceed with Gaussian elimination?

Of course, we would like to have $a_{jj}^{(j-1)}$ is non-zero. Else there will be singularieites as seen in (13).

Of course, checking those values after each step sounds very inefficient. An equivalent condition is - given A, we check if all of $A_1, A_2, \ldots, A_n$ are all invertible, where

$$A_k = A(1:k, 1:k)$$

is the $k \times k$ upper left submatrix of A. Formal proof ditch, we could look at some informal arguments. Consider $k = 1$. A_1 is invertible $\implies a_{11} \neq 0$. This could be our base case.

Now, the most important point here is that $\det A_k \forall k$ is invariant under elementary transformations. At step j, we already have the upper left $k-1 \times k-1$ matrix is upper triangular. It can be easily shown that the next elementary operation would create a $j \times j$ upper triangular matrix.

Also, the

Brief Recap...

Working with a real system

$$(A + \delta A)x = b + \delta b \tag{15}$$

The numerical solution has some error δx, with bounds given by

$$\frac{\|\delta x\|}{\|x\|} \leq \ldots \tag{16}$$

The larger the condition number, the closer it is to being a singular matrice and inversion is numerically unstable. Geometrically, solving $Ax = b$ is finding the intersection of planes. If the planes are nearly parallel to each other, then even small deviations in the planes will lead to large deviations in the intersection point.

Finite Precision Arithmetic

Even if we know any quantity with infinite precision, we can only store a representation of the quantity up to some finite precision.

The computers that use store finite precision binary representations, and computations of non-discrete systems are typically done using floating point numbers. A typical floating point representation is something that looks like this

$$x_1 \cdot x_2 x_3 x_4 x_5 \times 10^{y_1 y_2} \tag{17}$$

Where x_i are the digits in some base. Computers use binary, i.e. base 2. In general, there can be multiple representations that we can work with in computers. There was a need for standardization. So, most machines follow the IEEE standard.

IEEE 754

In (17), the x part is called the mantissa, and the y part is called the exponent. Typically there is also a sign bit. Per IEEE 754, single precision, typically implemented as a 32 bit word representation.

- 24 significand bits, including sign bit
- 8 exponent bits, with a bias of 127

In general, any floating point number (other than 0.0) can be represented as $1.x_1x_2x_3\ldots \times 10ey_1y_2\ldots$ In the IEEE representation, the 1. is implicit and only the digits after the decimal are stored. This results in a somewhat quirky represenation of 0.0.

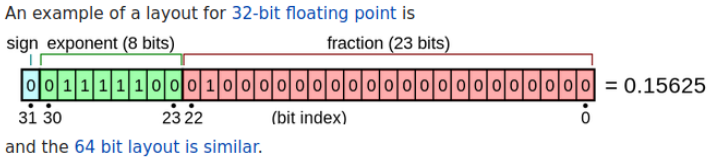

Figure 1: An example taken from Wikipedia

So, essentially your real line has been discretized.

Quantization Error

Given any $x \in \mathbb{R}$ the computer takes it as $\hat{x} := fl(x)$, i.e. converts it to a floating point representation in its word length.

$$\hat{x} = \text{fl}(x) = x(1+\epsilon) \tag{18}$$

ϵ is called the per-unit error.

What is the maximum possible error introduced when converting a real quantity from $\mathbb{R}$ to IEEE 754? It's the machine epsilon. Assume that the mantissa has s bits. The machine epsilon is given by

$$u \approx \frac{1}{2}10^{1-s} \tag{19}$$

Operations on Floating Point Numbers

If you have a binary operator that works with two numbers from $\mathbb{R}$, you want to be able to simulate it.

Let the 'real' operation be $O(x, y)$, where $x, y \in \mathbb{R}$. What the machine sees and does is $F(O(\hat{x}, \hat{y}))$, i.e. some errors will be introduced.

$$\begin{aligned} R &= F(O(x(1+\epsilon_1), y(1+\epsilon_2))) \\ &= O(x, y)(1+\epsilon) \end{aligned} \tag{20}$$

Let's start with looking at what happens to our basic arithmetic operations.

Multiplication

$$\begin{aligned} F(x_1x_2) &= (x_1(1+\epsilon_1)x_2(1+\epsilon_2))(1+\epsilon_3) \\ &= x_1x_2(1+\epsilon_1)(1+\epsilon_2)(1+\epsilon_3) \\ &= x_1x_2(1+\epsilon_1+\epsilon_2+\epsilon_3+\mathcal{O}(u^3)) \end{aligned} \tag{21}$$

So, for multiplication the significant error goes as $3u \approx \mathcal{O}(\sqcap)$

Division

$$F(\frac{x_1}{x_2}) \approx (\frac{x}{y})(1+\varepsilon+\mathcal{O}(u^2)) \tag{22}$$

Addition

This is troublesome

$$\begin{aligned} F(x_1+x_2) &= x_1(1+\epsilon_1)+x_2(1+\epsilon_2) \\ &= (x_1+x_2)(1+\frac{x_1\epsilon_1}{x_1+x_2}+\frac{x_2\epsilon_2}{x_1+x_2}) \end{aligned} \tag{23}$$

The relative error depends on the operands. So, the error will blow up if $x+y$ is small. This would happen if x and y have similar magnitudes, different signs.

E.g. if $x = 1.24456$ and $y = 1.23421$, so that $\hat{x} = 1.2346$ and $\hat{y} = 1.2342$. We have $\hat{x} - \hat{y} = 0.0004$, while $x - y = 0.00035$. This is like approximating 35 as 40. 14% error. Meanwhile, the truncation errors were some 0.001%. Absolutely mad.

This is called catastrophic cancellation. This error can reach as high as 30%. Imagine how much error is introduced in Gaussian elimination. Computer scientists were aware of the catastrophic cancellation errors.

The only way around this is to use different numerical algorithms, i.e. something that uses addition instead of subtraction, and figure out the upper bounds for errors in any numerical algorithm before we even start doing the computation.