Numerical Method for visual computing and ML
Prof. Wenzel Jakob

Arthur Herbette

Thursday 18 September 2025

Note before we begin

This document is not a copy of all the materials of the course but more a summary that I will use for my revision.
Mistakes can also happens so if anyone is reading this, please be aware that if you think something is wrong, maybe it is. If you find a mistake you are more than welcome to send me a message or directly make a pull request on the repo .

What is the course about

First the course is given by Prof. Wenzel which leads the *Realistic Graphics Lab*. This lab research is on *physically-based rendering* and *inverse rendering*. The topics of this courses is used on a daily basis in their labs.
But numerical methods is not only use in physics simulation. It is also used in:

- Computer Graphics
- Computer Vision
- Machine learning
- Computational Photography
- Weather simulation
- Geometric information systems

Numerical Methods for rendering

If we want to have a visual effects it comes down to solve an integral:
for instance, to compute the color of a pixel it is computed as follow:

$$\text{pixel color} = \int \text{light} dx$$

If we want to compute the next frame of a simulation (a physics simulation) it comes down to solve an **enormous linear system**:

$$x = A^{-1}b$$

As we will see in the third homework, it can also be used for character animation. (we attach a virtual body to a skeleton usin optimization technique)

$$x = \underbrace{A^{+}}_{\text{Pseudoinverse}} \quad b$$

It is also used to computing derivatives of software for fun and profit (for maching learning), to do so we need to compute the partial derivative (or the hessian, jacobian, etc..) $\frac{\partial f(x)}{\partial x}$

A certain feeling of familiarity

So here all the techniques we saw above, we already know them, we know them from MATH-111 or MATH-101, MATH-106. The goal of this course is to find a way to implement those abstract concept.

1 IEEE 754 Floating Point Arithmetic

From $\mathbb{N}$ to $\mathbb{R}^n$

In computer science, everything is discrete, **everything**. This means that for instance the representation of π is impossible here.

$$\pi = 3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803$$

So we cannot store the π directly, we have to make **compromise**. The solution for that is: we don't store the correct value of π...

Number Systems

As we have seen in AICC I, AICC II, PPO, FDS, Computer Architecture, there is two main way to represents integer:

- Positive integer
- Signed integers (two's complement)

Here is the typical size of each fo them

`uint8`	0	255
`int8`	-128	127
`uint16`	0	65535
`int16`	-32768	32767
`uint32`	0	4294967295
`int32`	-2147483648	2147483647
`uint64`	0	18446744073709551615
`int64`	9223372036854775808	9223372036854775807

Fixed Point Arithmetic

The basic idea for this is just to allow negative exponent. Instead of having our representation begin at the power 0 we make it begin at a hardcode power for instance like this:

$$(b_3 b_2 b_1 b_0 b_{-1} b_{-2} b_{-3} b_{-4})$$

We are now able to represent the number 12.3125.

$$(11000101)_2 = 12.3125_{10}$$

There is two main drawbacks here:

Drawback 1 Big number requires a lot of storage

Drawback 2 Growth of exponent range during operations.

$$\left(\sum_{i=-k}^{k} a_k 2^k\right)\left(\sum_{i=-k}^{k} b_k 2^k\right) = a_{-k}b_{-k}2^{-2k} + \cdots + a_k b_k 2^{2k}$$

Rational numbers

Another issue that we have other than irrational number, is rational number. Let us take the number 0.1 and try to express is with fixed point arithmetic.
The solution for this is to use the definition of rational number:

$$\mathbb{Q} = \{\frac{a}{n} : a, b, \in \mathbb{Z}\}$$

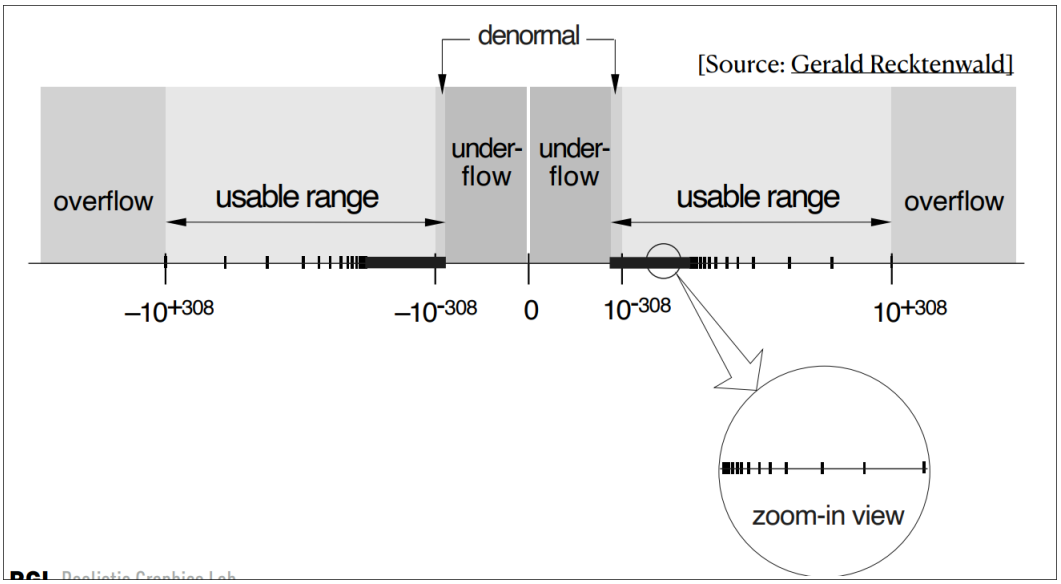
This means that in order to represent any rational number all we need is to store two integer. Operation on this representation gives us also the same representation. For intance if we take the addition:

$$\frac{a}{b} + \frac{c}{d} = \frac{(ad + cb)}{bd}$$

However this is not very good alternative. It is pretty costly in terms of computation and of space.

Another issues also with fixed point is the difference of scale. let us take for instance the size of a Proton and the one of a human. The scale is different. This makes it imposssible for us to have a good representation of both of them without using an **enormous** amount of memory.
This is where floating poit numbers comes in:

$$\mp 2^E \sum_{i=0}^{n} 2^{-i} A_i$$



Problem with this current version

- How to represent zero?
  - Easy, just don't store the leading one. Zero is signed in IEEE
- What if the number becomes too big during a calculation?
  - Return special number "infinity" (can be Positive/negative)
- What if the number becomes too small during a calculation?
  - Gradual underflow through denormalized numbers
- What if the user is trying to do a nonsensical calculation?
  - Return special number 'NaN' (Not a Number). NoN is infectious
- What to do when the result of the calculation cannot be represented exaclty (where should we round our error)
  - Use Banker's rounding (round to nearist tie to even) by default, other modes available.

IEEE 754: Nitty Gritty details, contd.

Here's the major innovations of this standard compared to previous ones

Accuracy guarantees

Elementary arithmetic operations (+, -, *, /, sqrt) are done in infinite precision and then rounded to a representable number. What this means is that the answer you get for each arithmetic operation must be the closest possible floating point number to the true mathematical answer.

The reason for this being a big innovation is that: before this *correct rounding*, different computers gave different results for the same floating point approximations which means that numerical code was unpredictable across machines.
IEEE-754 forces everyone to follow the same rule, so:

- Every platform produces the same value for (0.1 + 0.2) (hum hum javascript)etc.
- Maximum error is bounded and predictable

Denormalized

Enables gradual underflow (most controversial feature of IEEE 754)
Normally a floating-point number has the form

$$\pm 1.xxx \times 2^{exp}$$

But near zero, the exponent cannot go lower.
Before IEEE-754, number smaller than a certain size would immediately **underflow to zero**. To prevent that, IEEE-754 introduced **denormalized numbers** (subnormals)

$$\pm 0.xxx \times 2^{minExpo}$$

This create a **smooth ramp to zero** instead of a sudden drop.

> But why does it matter?
> Imagine if we didn't have denormals, very small results of calculations simply become 0 abruplty. This cause loss of information and instability in algorithms.
> When we add denormals, we get **gradual underflow**. You can represent number much closer to 0, though with reduced precision.

The reason for the *controversial* and slow of denormals is that processing denormals is much more complex for hardware:

- Many CPUs handle them using slow microcode
- Operations involving subnormals can be 100-1000x slower

Special case rules

	$-\infty$	-0	0	∞	NaN
$-\infty$	∞	NaN	NaN	$-\infty$	NaN
-0	NaN	0	-0	NaN	NaN
0	NaN	-0	0	NaN	NaN
∞	$-\infty$	NaN	NaN	∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN

```
def f(x):
        if x < 0 or x > 1:
                raise Exception("Invalid argument")
        return ..
```

```
def f(x):
        if not(x >= 0 and x <= 1):
                raise Exception("Invalid argument")
        return ..
```

Danger Zone

When dealing with IEEE-754 operations there is a couple of case which are dangerous:

- **Transcendental operations** (sin, cos, atan, exp, log) are much slower (50-300 clock cycles) and less accurate
- **Fused Multiply-Add** (FMA). Compute a*b+c directly (faster, with only oe rounding step). But now there are two ways to do the same thing.
- **Compiler optimizations** may violate expected IEEE-754 rounding behavior
- **Denormalized numbers** are often very slow and cause surprises. Can turn them off
- Carefule when converting floats to integers, and vice versa
- Arithmetic transformation that are 'pointlessc' on pencil and paper can make huge accuracy difference.

1.0.1 Errors

Catastrophic numerical errors

- Rounding error accumulation (1991: Patriot battery failed to interecpt missile in Dharan, Saudi Arabia)
- Ariane 5 Rocket failure

Model error

For instance if we wanted to compute the surface of the earth we would model the planet as a sphere which would give for the surface:

$$A = 4\pi r^2$$

Truncation error

What if we wanted to approximate the sum of $\frac{1}{3^i}$ if we take for intance:

$$\sum_{i=1}^{10} \frac{1}{3^i} \approx 0.499991532$$

$$\sum_{i=1}^{\infty} \frac{1}{3^i} \approx 0.5$$

We can also take the harmonic serie $\frac{1}{k}$ which converges in IEEE-754 arithmetic:

$$\sum_{k=1}^{\infty} \frac{1}{k} = 15.493683$$

Of all the mathematical operations, one particular case is **especially dangerous**

$$a - b \quad (a \approx b)$$

The also happens for $a + b$ when $a \approx -b$ This can lead to catastrophic cancellation even if the substraction is done without error (this can happend not only if the there is an rounding error in the operation but maybe from before).

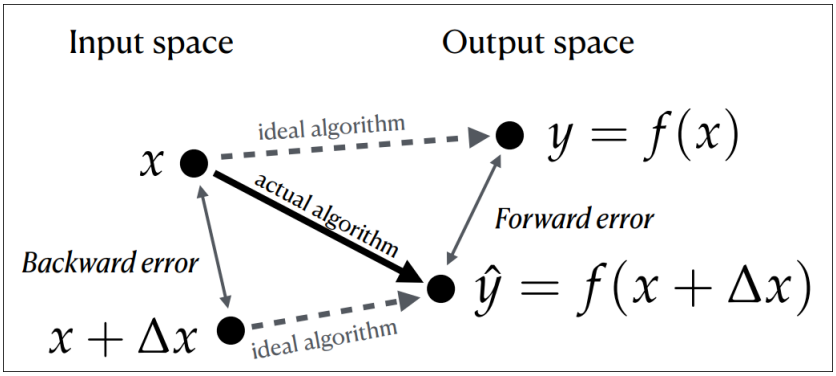
Example For instance $3.140 = 2^1 \cdot 1.10010010$ and $-3.149 = +2^1 1.10010010$ When you sum both you get:

$$= -2^{-7} 1.0_2$$

Quantiying error

- Absolute error: |true value − aprroximate value|
  - Example 2cm $\pm 0.1cm$
- Relative error: $\frac{\text{absolute error}}{|\text{true value}|}$
  - Example: 2cm $\pm 0.1\%$
  - Another common notation (true value) (1 $\pm$relative error)

Forward ans Backward Error



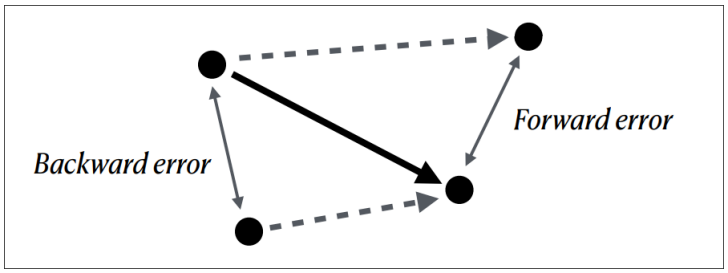
Example For instance let us take

$$f(x) := \sqrt{x}$$
$$y = f(x)$$
$$\hat{\boldsymbol{y}} = f_{IEE754}(x)$$

Here is you wanted to compute the the backward error you would do it like this:

$$error = x - (f_{IEE754}(x))^2$$

Conditioning of numerical problems



condition number: ratio $\dfrac{\text{forward error}}{\text{backward error}}$

2 Linear systems

Definition 1 *f is a linear function if and only if:*

- $f(x, y) = f(x) + f(y)$ *additivity*
- $f(\alpha x) = (x)$ *where* $\alpha \in \mathbb{R}$ *homogeneity*

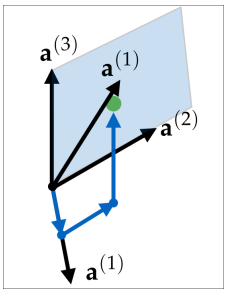
In this case $f(x) = Ax$ fully encodes everything that f does!

Geometric intrerpretation of matrix-vector product

$$(Ax)_i = \sum_{k=1}^{n} A_{ik} x_k \text{ where } A = \begin{pmatrix} | & | & | \\ a^{(1)} & a^{(2)} & a^{(3)} \\ | & | & | \end{pmatrix}$$

Example of linearly dependent columns span 2d Subspace:

$$Ax = a^{(1)}x_1 + a^{(2)}x_2 + a^{(3)}x_3$$

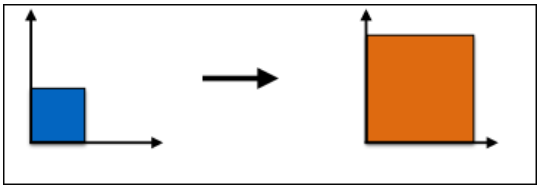


Basic matric transformation in 2D

Please see the video of 3blue1brown on essence of linear algebra to understand this well

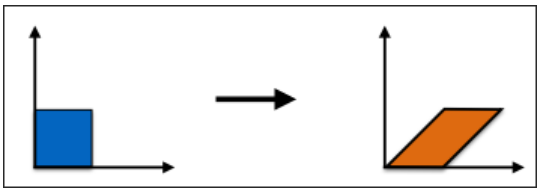
Scaling

$$\begin{pmatrix} v'_x \\ v'_y \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$



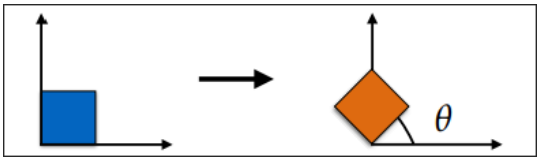
Shear

$$\begin{pmatrix} v'_x \\ v'_y \end{pmatrix} = \begin{pmatrix} 1 & c \\ 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$



Rotation

$$\begin{pmatrix} v'_x \\ v'_y \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$



Mirror

$$\begin{pmatrix} v'_x \\ v'_y \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

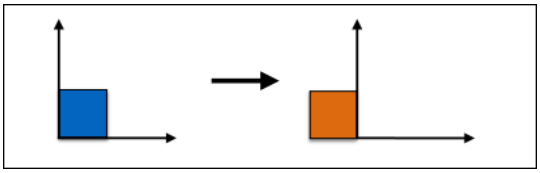


Figure 1: Geometric transformations using 2×2 matrices.

Why should we care about linear function?

- Solving linear problems is one of the (few) numerical problems that we know to solve **really well**
- When you can turn something into a linear system → good job, you are done.
- Nonlinear method are fragile

Almost everything boils down to a linear system

Definition 2 *Matrix multiplication:*

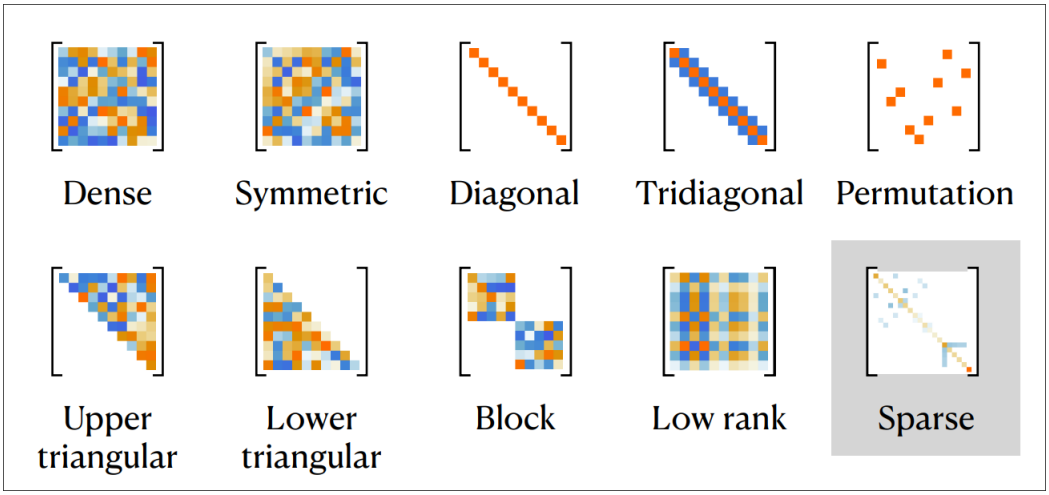
$$(AB)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

The question however is why is it defined this way?
For instance if we take $f(x) = Ax, g(x) = Bx$ then you get that

$$g(f(x))BAx$$

Matrix zoo

Here's a glimps into the great variety of matrix flavors that arise in practice:



Also important: positive definite and orthogonal matrices.

Sparse matrices

Definition 3 *(sparse matrix) A* ***sparse matrix*** *is a matrix for which almost all (> 99%) entries are zero*

But why are they useful?
Sparsity arises here because most things aren't connected for instance protein modeling, social networks, triangle meshes of 3D objects.
In fact they are so useful that almost every matrix algorithm has an analogous sparse version (or several!) By having an encoding of them, we can avoid storing the zeros and skip them during computation.

- Encodings: *Compressed Sparse Row* (CSR) or *Compressed Sparse Column* (CSC) format

Sparsity, continued

Sparsity also occurs when one discretizes continuous equations.
For instance let us take the Heat equation:

$$\frac{\partial^2 f(x)}{\partial x^2} = 0$$

The computer implementation of this equation is the following:

$$-f(x_{i-1}) + 2f(x_i) - f(x_{i+1}) = 0$$

Instead of having for instance a continuous line $\mathbb{R}$ we discretizes it by having a big sequence of x_i which represent each points.

The equation $\frac{\partial^2 f(x)}{\partial x^2} = 0$ intuitively says 'heat spreads out until it's evenly distributed.
As we know that computer can't handle continuous function directly; instead, they approximate them on a **grid** of **mesh** points. Suppose you pick poits $x_1, x_2, \ldots, x_n$ along the x-axis. You want to find the values $f(x_1), f(x_2), \ldots, f(x_n)$ that satisfy the equation **approximately**.
By using the definition of the second derivative we get the following approximation:

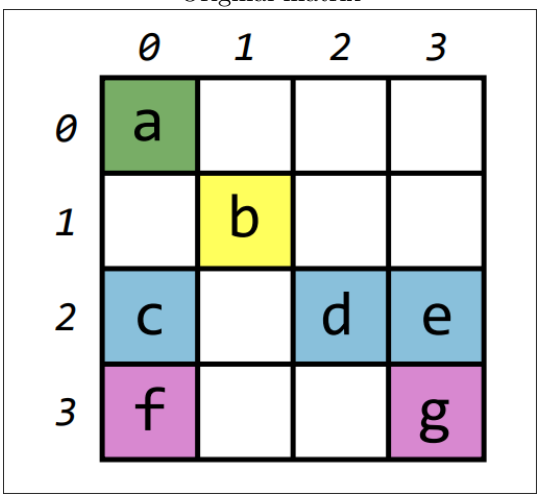
$$\frac{\partial^2 f(x)}{\partial x^2} \approx \frac{f(x_{i-1}) - 2f(x_i) + f(x_{i+1})}{h^2} = 0$$

where h is the distance between grid points.
Multiplying both sides by h^2, rearranging:

$$-f(x_{i-1}) + 2f(x_i) - f(x_{i+1}) = 0$$

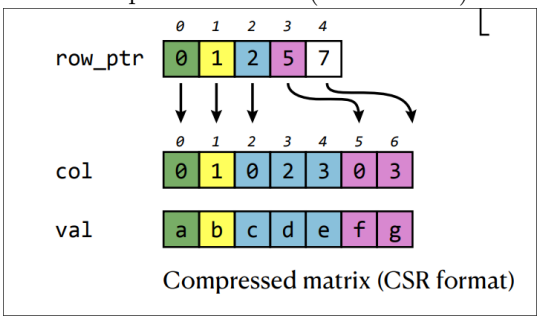
Sparse matrix-vector multiplication

Original matrix




```
y = np.zeros(n)
for i in range(n):
        for j in range(n):
                y[i] += A[i, j] * x[j]
```

Compressed matrix (CSR format)

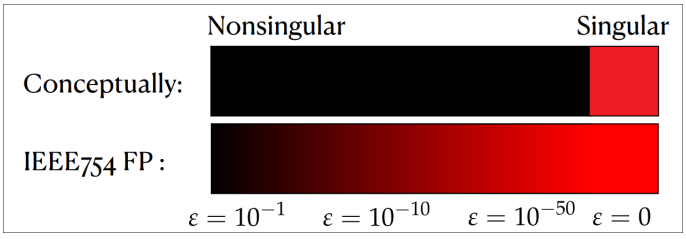



```
y = np.zeros(n)
for i in range(n):
        for k in range(A.rowptr[i], A.
            rowptr[i+ 1]):
                y[i] += A.val[k] * x[A.
                    col[k]]
```

Pure versus numerical mathematics

$$\begin{pmatrix} 1 & 0 \\ 1 & \varepsilon \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

In theory the matrix above is singular if and only if $\varepsilon = 0$
But in practrice it is more nuanced:



Solving linear system MATH-111 style

To solve linear system we can use Gaussian Elimination has seen previously with:

$$[A \mid b]$$

However when computing the elimination we have to be careful when choosing the pivot element → instabilities can occur even for nonzero elements.
The practical solution is just to pick the element with the largest absolute value in row.

A common situation

However it happends pretty often that we need to solved the same linear system over and over again.

$$\begin{aligned} Ax^{(1)} &= b^{(1)} \\ Ax^{(2)} &= b^{(2)} \\ Ax^{(3)} &= b^{(3)} \\ &\vdots \end{aligned}$$

This seems really inefficient. The question we should ask is: is it possible to compute something only one time and then use it for the next equation?

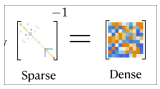
don't do this

The answer is yes, we could just compute A^{-1} from A which would gives us:

$$\begin{aligned} x^{(1)} &= A^{-1}b^{(1)} \\ x^{(2)} &= A^{-1}b^{(2)} \\ x^{(3)} &= A^{-1}b^{(3)} \\ &\vdots \end{aligned}$$

However this is a very not good idea but why?

- Computation of the inverse can be numerically unstable
- Computation of inverse + matrix multiplies require slightly more FLOPs than alternative discurssed next.
- Also, generally :

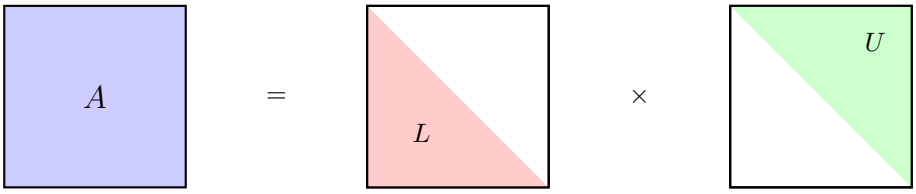


Which will make our program run out of memory

2.1 Lu Factorization

Here are the basic motivation of this method:

- Run the Gaussian elimination algorithm on A alone (no right-hand side b)
- Keep track of what transformations ir perform (e.g., adding scaled rows)
- Those transformations can be written down as matrice L and U
- We can then 'apply' L and U to any right hand side b very efficiently



We won't be looking to musch in the details of how the LU decomposition is implemented. Furthermore as computer scientist (apart for those who will work on them) we won't implement this methods and we should always use the framework that has been already created.

Using a LU decomposition

Once the decomposition is done, how do we actually use it? We have the relation

$$A = LU$$

Therefore we found that:

$$x = U^{-1}\big(L^{-1}b\big)$$

You could think like: 'but we have now to inverse of matrix to compute, this is worse!'. It would be if tose matric where some randoms ones but here we know how they behave. Those are triangular matrices. As you can try on your own those linear system are very easy to solve $\implies$ Solving a triangular system takes $O\big(n^2\big)$!

Version with pivoting

It exists also a version with column pivoting, also known as partial pivoting:

$$A = PLU$$

The way of doing it is the same but the solution is then:

$$x = U^{-1}\big(L^{-1}\big(P^{-1}b\big)\big)$$

Implementation in SciPy

```
import scipy.linalg as la
lu, piv = la.lu_factor(A)
x = la.lu_solve((lu, piv), b)
```

Symmetric positive definite matrices: Cholesky decomposition

First the definition of the symmetric matrix:

Definition 4 *A matrix A is* ***symmetric*** *if*

$$A = A^T$$

Definition 5 *A symmetric matrix is* ***positive definite*** *if*

$$x^T Ax > 0 \quad \forall \text{ nonzero } x$$

In practice what this means is:

- No negative eigenvalues
- It's 'nicely behaved'
- It appears all the time in physics, optimization, machine learning, etc.

Now let's see what happen with the LU decomposition but when we are with a symmetric matrix:

$$A = LU$$

Which implies directly:

$$A = LU = (LU)^T = U^T L^T$$

So we have two relations $A = LU$ and $A = U^T L^T$ since factorization of a matrix are unique for triangular factors, that menas the left hand and right hand factos must match. So we must have:

$$L = U^T \text{ and } U = L^T$$

Therefore:

$$A = LL^T$$

Where L is lower triangular with **positive** diagonal entries.
This is great! We only need to store one matrix instead of two, we saved half of the storage. For instance given:

$$\begin{pmatrix} A_{11} & A_{21} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} \\ 0 & L_{22} \end{pmatrix}$$

We have that:

$$\begin{aligned} L_{11} &= \sqrt{A_{11}} \\ L_{21} &= \frac{A_{21}}{L_{11}} \\ L_{22} &= \sqrt{A_{22} - L_{21}^2} \end{aligned}$$

2.2 Cost of matrix operations

Now we will explore the cost of matrix operations. First let us see the how to compute it in python.
If we take the matric vector multiplication we have the following code:

```
b = np.zeros(n)
for j in range(n):
        for i in range(n):
                b[i] += A[i, j] * x[j]
```

However this is not really efficient the best way of doing it is by using the numpy library like this:

```
b = A @ x
```

For the first one we have 30'282 ms, for the second one with have **11ms**!
For the matrix-matrix multiplication:

```
C = np.zeros((n, n))
for i in range(n):
        for j in range(n):
                for k in range(n):
                        C[i, j] += A[i, k] * B[k, j]
```

The 'correct way' of doing it is:

```
C = A @ B
```

- python $\rightarrow$ 341'495ms
- numpy $\rightarrow$ 19ms
- Matrix-Vector multiplication $\rightarrow O(n^2)$
- Matrix-Matrix multiplication $\rightarrow O(n^3)$

Is it really $O(n^3)$

- Volker Strassen 1969 $\rightarrow O(n^{2.8074})$
- Virginia Williams et al 2023 $\rightarrow O(n^{2.3751552})$

Now the question is what if we have triangular matrix, this way facorization becomes way faster, we only have to 'go up'.

- Triangular system solving aka. 'substitution'
  - Facorization $\rightarrow O(n^3)$
  - Substitution $\rightarrow O(n^2)$

2.3 Regression

Imagine that:

- We have a model of the **expected behaviour**
- The model contains unknown parameter values
- We have data/ measurements that may be supported by the model
- **Goal**: find parameters that best support the data
- Can we express this as a solution to a linear system?
  - In that case, this is called **linear regression**

For instance, let us take an ideal ballistic motion

$$y = ax^2 + bx + c$$

All we know is the point where the ball has been, we have a lot of $(x, , y)$ data points. Our unknown is a, b, c.

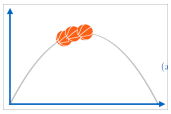
The question we need to answer is, how many data points (i.e. measurements) do we need to solve this system. For us as we have three unkowns we need to solve a matrix like this:

$$\begin{pmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{pmatrix} \cdot \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

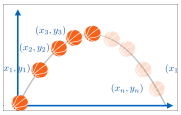
However this can be generalize: **Polynomial regression**

The question we tried to answer before is: Find a polynomial that perfectly interpolates a given set of points so that:

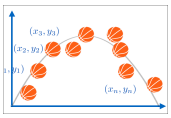
$$y(t_i) = p_i$$



What if the observation are weak



What if there are > 3 observations? Use subset? Which ones? use all somehow?



What if there is noicse? What if the model violates the assumptions?