Numerical Methods
Summary

September 28, 2020

Chapter 1

Computing with Matrices and Vectors

Elementary Operations: +,-,*,\, lowest level of real arithmetic available on computers usually implemented in hardware
Elementary Linear Algebra operations: The next level real arithmetic which is the computation on finite arrays of real numbers
Complex Algorithms: involves iterations and approximations

1.1 Fundamentals

$\mathbb{K}$ notation for a generic field of numbers i.e from $\mathbb{R}$ or $\mathbb{C}$

Vectors: one-dimensional array of real/complex numbers, the default for this lecture are column vectors:

$$\underbrace{\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{K}^n}_{\text{column vector}} \quad \underbrace{\begin{bmatrix} x_1 \cdots x_n \end{bmatrix} \in \mathbb{K}^{1,n}}_{\text{row vector}}$$

Unless stated otherwise, in mathematical formulas vector components are indexed from 1

Notations:

- column vectors: bold small roman letters e.g **x,y,z**
- row vectors: x^T, y^T, z^T
- Addressing vector components:

$$x = [x_1, ..., x_n]^T \rightarrow x_i, i = 1, ..., n$$
$$x \in \mathbb{K}^n \rightarrow (x)_i, i = 1, ..., n$$

- Selecting sub-vectors: $(x)_{k:l} = [x_k, ..., x_l]^T, 1 \leq k \leq l \leq n$
- j-th unit vector: $(e_j)_i = \sigma_{ij}$ where $sigma_{ij} := 1$ if $i = j$ 0 else (Kronecker symbol)

1.2 Software and Libraries

Eigen: A header-only C++ template library designed to enable easy, natural and efficient numerical linear algebra

Header-Only A library is called header-only if the full definitions of all macros functions and classes comprising the library are visible to the compiler in a header file form. Header-only libraries do not need to be separately compiled, packaged and installed in order to be used. All that is required is to point the compiler at the location of the headers, and then include the header files into the application source

Header File form: Many programming languages and other computer files have a directive, often called inlcude (or copy, import) that causes the contents of a second file to be inserted into the original file. These included files are called copybooks or header files.

Eigen Cheet Sheet:

```
// A simple quickref for Eigen. Add anything that's missing.
// Main author: Keir Mierle

#include <Eigen/Dense>

Matrix<double, 3, 3> A;                // Fixed rows and cols. Same as Matrix3d.
Matrix<double, 3, Dynamic> B;          // Fixed rows, dynamic cols.
Matrix<double, Dynamic, Dynamic> C;    // Full dynamic. Same as MatrixXd.
Matrix<double, 3, 3, RowMajor> E;      // Row major; default is column-major.
Matrix3f P, Q, R;                      // 3x3 float matrix.
Vector3f x, y, z;                      // 3x1 float matrix.
RowVector3f a, b, c;                   // 1x3 float matrix.
VectorXd v;                            // Dynamic column vector of doubles
double s;

// Basic usage
// Eigen          // Matlab           // comments
x.size()          // length(x)        // vector size
C.rows()          // size(C,1)        // number of rows
C.cols()          // size(C,2)        // number of columns
x(i)              // x(i+1)           // Matlab is 1-based
C(i,j)            // C(i+1,j+1)       //

A.resize(4, 4);   // Runtime error if assertions are on.
B.resize(4, 9);   // Runtime error if assertions are on.
A.resize(3, 3);   // Ok; size didn't change.
B.resize(3, 9);   // Ok; only dynamic cols changed.

A << 1, 2, 3,     // Initialize A. The elements can also be
     4, 5, 6,     // matrices, which are stacked along cols
     7, 8, 9;     // and then the rows are stacked.
B << A, A, A;     // B is three horizontally stacked A's.
A.fill(10);       // Fill A with all 10's.

// Eigen                                    // Matlab
MatrixXd::Identity(rows,cols)               // eye(rows,cols)
C.setIdentity(rows,cols)                    // C = eye(rows,cols)
MatrixXd::Zero(rows,cols)                   // zeros(rows,cols)
C.setZero(rows,cols)                        // C = zeros(rows,cols)
MatrixXd::Ones(rows,cols)                   // ones(rows,cols)
C.setOnes(rows,cols)                        // C = ones(rows,cols)
MatrixXd::Random(rows,cols)                 // rand(rows,cols)*2-1            // MatrixXd::Random returns uniform random numbers in (-1, 1).
C.setRandom(rows,cols)                      // C = rand(rows,cols)*2-1
VectorXd::LinSpaced(size,low,high)          // linspace(low,high,size)'
v.setLinSpaced(size,low,high)               // v = linspace(low,high,size)'
VectorXi::LinSpaced(((hi-low)/step)+1,      // low:step:hi
                    low,low+step*(size-1))  //


// Matrix slicing and blocks. All expressions listed here are read/write.
// Templated size versions are faster. Note that Matlab is 1-based (a size N
// vector is x(1)...x(N)).
// Eigen                           // Matlab
x.head(n)                          // x(1:n)
x.head<n>()                        // x(1:n)
x.tail(n)                          // x(end - n + 1: end)
x.tail<n>()                        // x(end - n + 1: end)
x.segment(i, n)                    // x(i+1 : i+n)
x.segment<n>(i)                    // x(i+1 : i+n)
P.block(i, j, rows, cols)          // P(i+1 : i+rows, j+1 : j+cols)
P.block<rows, cols>(i, j)          // P(i+1 : i+rows, j+1 : j+cols)
P.row(i)                           // P(i+1, :)
P.col(j)                           // P(:, j+1)
P.leftCols<cols>()                 // P(:, 1:cols)
P.leftCols(cols)                   // P(:, 1:cols)
P.middleCols<cols>(j)              // P(:, j+1:j+cols)
P.middleCols(j, cols)              // P(:, j+1:j+cols)
P.rightCols<cols>()                // P(:, end-cols+1:end)
P.rightCols(cols)                  // P(:, end-cols+1:end)
P.topRows<rows>()                  // P(1:rows, :)
P.topRows(rows)                    // P(1:rows, :)
P.middleRows<rows>(i)              // P(i+1:i+rows, :)
P.middleRows(i, rows)              // P(i+1:i+rows, :)
P.bottomRows<rows>()               // P(end-rows+1:end, :)
P.bottomRows(rows)                 // P(end-rows+1:end, :)
P.topLeftCorner(rows, cols)        // P(1:rows, 1:cols)
P.topRightCorner(rows, cols)       // P(1:rows, end-cols+1:end)
P.bottomLeftCorner(rows, cols)     // P(end-rows+1:end, 1:cols)
P.bottomRightCorner(rows, cols)    // P(end-rows+1:end, end-cols+1:end)
P.topLeftCorner<rows,cols>()       // P(1:rows, 1:cols)
P.topRightCorner<rows,cols>()      // P(1:rows, end-cols+1:end)
P.bottomLeftCorner<rows,cols>()    // P(end-rows+1:end, 1:cols)
P.bottomRightCorner<rows,cols>()   // P(end-rows+1:end, end-cols+1:end)
```

```
// Of particular note is Eigen's swap function which is highly optimized.
// Eigen                           // Matlab
R.row(i) = P.col(j);               // R(i, :) = P(:, j)
R.col(j1).swap(mat1.col(j2));      // R(:, [j1 j2]) = R(:, [j2, j1])

// Views, transpose, etc;
// Eigen                           // Matlab
R.adjoint()                        // R'
R.transpose()                      // R.' or conj(R')        // Read-write
R.diagonal()                       // diag(R)                // Read-write
x.asDiagonal()                     // diag(x)
R.transpose().colwise().reverse()  // rot90(R)               // Read-write
R.rowwise().reverse()              // fliplr(R)
R.colwise().reverse()              // flipud(R)
R.replicate(i,j)                   // repmat(P,i,j)


// All the same as Matlab, but matlab doesn't have *= style operators.
// Matrix-vector.  Matrix-matrix.   Matrix-scalar.
y  = M*x;          R  = P*Q;        R  = P*s;
a  = b*M;          R  = P - Q;      R  = s*P;
a *= M;            R  = P + Q;      R  = P/s;
                   R *= Q;          R  = s*P;
                   R += Q;          R *= s;
                   R -= Q;          R /= s;

// Vectorized operations on each element independently
// Eigen                       // Matlab
R = P.cwiseProduct(Q);         // R = P .* Q
R = P.array() * s.array();     // R = P .* s
R = P.cwiseQuotient(Q);        // R = P ./ Q
R = P.array() / Q.array();     // R = P ./ Q
R = P.array() + s.array();     // R = P + s
R = P.array() - s.array();     // R = P - s
R.array() += s;                // R = R + s
R.array() -= s;                // R = R - s
R.array() < Q.array();         // R < Q
R.array() <= Q.array();        // R <= Q
R.cwiseInverse();              // 1 ./ P
R.array().inverse();           // 1 ./ P
R.array().sin()                // sin(P)
R.array().cos()                // cos(P)
```

The Fundamental type of Eigen is the Matrix. There are two types of matrices:

- **fixed size:** size known at compile time
- **dynamic:** size known only at run time

Tensor product A column vector multiplied with a row vector

Dot Product Row vector multiplied with a column vector

1.3 1.2.3 Matrix Storage Formats

The entries of a (generic, dense i.e every entry matters) $A \in \mathbb{K}^{m,n}$ are stored in a contiguous linear array of size $m \cdot n$. An exception is structured/sparse matrices i.e matrices which have a certain structure or few non zero entries.

By Default Eigen stores matrices in Column major format as all the elements in this format are contiguous in memory. In Row major format the elements of the Matrix are scattered which results in cache misses. Hence Column major format is more efficient

Accessing Arrays in Eigen:

- A(i) $\rightarrow$ reference to the i-th element of the array
- A.data() $\rightarrow$ raw pointer

We declare a nxn matrix in Eigen like:

Eigen:: MatrixXd A = Eigen:: MatrixXd::Random(n,n)

Row/Col Access (j-th row):

A.row(j)
A.col(j)

1.3.1 Raw Pointers

A pointer is a type of variable. It stores the address of an object in memory and is used to access that object in memory and is used to access that object.
A **Raw Pointer** is a pointer whose lifetime is not controlled by an encapsulating object . A raw pointer can be assigned the address of another non-pointer variable, or it can be assigned a value of nullptr.

```
int* p = nullptr; // declare pointer and initialize it
                  // so that it doesn't store a random address
int i = 5;
p = &i; // assign pointer to address of object
int j = *p; // dereference p to retrieve the value at its address
```

1.4 1.4 Computational Effort

Traditional: number of elementary operations

Modern: The computational effort involved in a run of a numerical code is only loosely related to the overall execution time on modern computers. It is mainly determined by the memory access pattern and vectorization/pipelining.

1.4.1 1.4.1 Asymptotic complexity

Characterises the worst-case dependence of its computational effort on one or more problem size parameters when these tend to ∞

Notation: Landau-O notation

We define the computational effort as Cost(n) $= (O)(n^{\alpha}), \alpha > 0 \iff \exists C > 0, n_0 \in \mathbb{N} : cost(n) \leq Cn^{\alpha} \forall n > n_0$

Tacit Sharpness assumption: $cost(n) \neq \mathbb{O}(n^{\beta}), \forall \beta < \alpha$

Asymptotic complexity predicts the dependence of runtime on problem size for large problems, because for small problem sizes we can use the caches and hence we do not have the memory access bottleneck.

1.4.2 1.4.2 Computational Cost of basic numerical LA operations

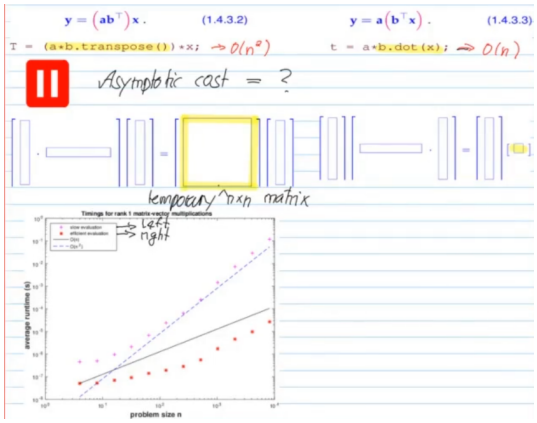
operation	description	#mul/div	#add/sub	asymp. complexity
dot product	$(\mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^n) \mapsto \mathbf{x}^H\mathbf{y}$	n	$n-1$	$O(n)$
tensor product	$(\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n) \mapsto \mathbf{x}\mathbf{y}^H$	mn	0	$O(mn)$
Matrix×vector	$(\mathbf{x} \in \mathbb{R}^n, \mathbf{A} \in \mathbb{R}^{m,n}) \mapsto \mathbf{A}\mathbf{x}$	mn	$(n-1)m$	$O(mn)$
matrix product[*]	$(\mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{B} \in \mathbb{R}^{n,k}) \mapsto \mathbf{A}\mathbf{B}$	mnk	$mk(n-1)$	$O(mnk)$

The matrix multiplication is implemented with a triple loop and hence not optimal.

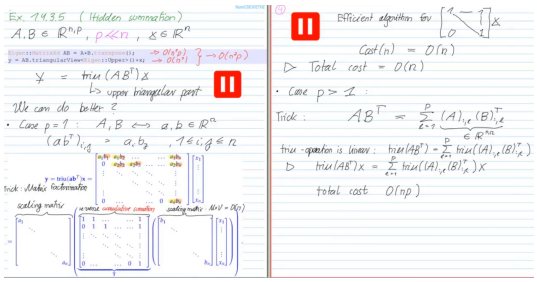
A doubly logarithmic plot is scaled logarithmically on the x and y axis. If Cost(n) = $\mathbb{N}(n^\alpha)$ then the data points aligned in a doubly logarithmic plot correspond to a straight line.

1.4.3 1.4.3 Tricks to improve complexity

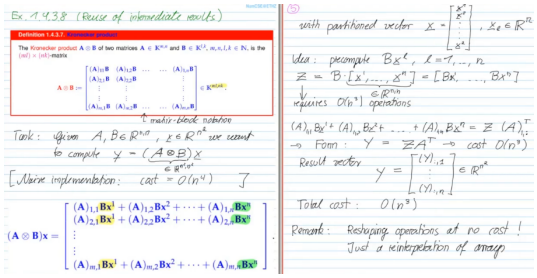
- **Exploit associativity**



- **Hidden Summation:**



- **Reuse of intermediate results**



1.5 1.5 Machine Arithmetic and Consequences

Computers can only calculate with finite numbers and hence cannot calculate with real numbers. Computers compute with machine numbers denoted: $\mathbb{M}$ The set M has two properties:

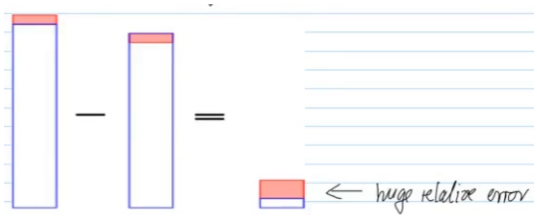
- **finite** I.e there is a lowest and a highest machine number any numbers larger or smaller than these bounds result in overflow/underflow
- **discrete in $\mathbb{R}$** hence there are gaps in the Machine numbers. When working in $\mathbb{R}$ our computer must round the numbers such that it can work with them.

Roundoff will effect any result of a numerical calculation. Hence a result which should be 0 might not be zero. Therefore we cannot test == 0 we must test for relative smallness

1.6 1.5.4 Cancellation

EPS- Errors: Max error you can have in a Machine set of numbers

EPS-sized errors are troublesome because of error amplification.



Cancellation: Extreme amplification of relative errors duirng the subtraction of numbers of equal size.

1.6.1 Avoiding Cancellation:

Rewrite expressions inequivalent form immune to cancellation. For x close to 1 cancellation is harmless.

1.7 Numerical Stability

Problem A function/mapping $F : X \to Y$ where X is the data space (input) and Y is the result space (output), in this course we usually consider normed Vector space. On $\mathbb{R}^n$ we have vector norms:

- $||.||_2$ Euclidean Norm
- $||.||_1$ One Norm (sum of the absolute value)
- $||.||_\infty$ Maximum Norm (max value)

The vector norms induce **Matrix Norms**. **D1.5.5.10 Matrix Norm:** Given vector norms $||.||_x$ and $||.||_y$ on $\mathbb{K}^n$ and $\mathbb{K}^m$, respectively, the associated matrix norm is defined by

$$M \in \mathbb{R}^{m,n} : ||M|| := \sup_{x \in \mathbb{R}^n \setminus \{0\}} \frac{||Mx||_y}{||x||_x}$$

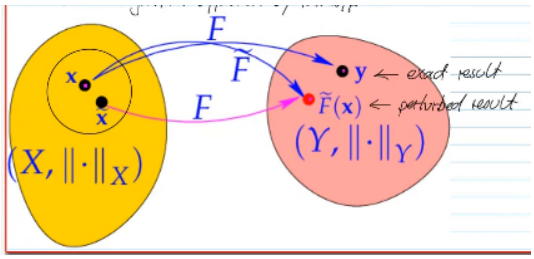
Norms are used to understand the size of pertubations.

Stable Algorithm: An algorithm $\tilde{F}$ for solving a problem $F : X \mapsto Y$ is **numerically stable** if for all $x \in X$ its result $\tilde{F}(x)$ (possibly affectedby roundoff) is the exact result for "slightly perturbed" data:

$$\exists C \approx 1 : \forall x \in X : \exists \tilde{x} \in X : ||x - \tilde{x}||_X \leq Cw(x)EPS||x||_X \wedge \tilde{F}(x) = F(\tilde{x})$$

Where:

- $\tilde{F}$: Algorithm affected by roundoff
- $\tilde{x}$. perturbed data
- Cw(x):= number of operations during the execution of the algorithm
- EPS:= machine precision



The impact of roundoff on a stable algorithm is of the same order of magnitude as the effect of the inevitable perturbations due to rounding the input data. When talking about perturbations we mean relative perturbations.

Sensitive Dependence: Given a slight perturbation the result varies largely:

$$||F(x) - F(\tilde{x})|| \text{ can be alrge for tiny } ||x - \tilde{x}||_x$$

Chapter 2

Chapter 2: Direct Methods for Linear Systems of Equations

2.1 Intro and Theory of Linear Systems of Equations (LSE's)

LSE:

$$A\underline{x} = \underline{b}$$

where:

- $A \in \mathbb{K}^{n,n}$ a square system/coefficient matrix
- $\underline{x} \in \mathbb{K}^n$
- $\underline{b} \in \mathbb{K}^n$

A being a square matrix is important, because this ensures that the number of equations is equal to the number of unknowns

2.2 Square LSE

2.2.1 Existence and uniqueness of solutions

1. If A is invertible (regular) :

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

   Do not use matrix inversion to solve LSE with numerical libraries
   A matrix $A \in \mathbb{K}^{n,n}$ is regular $\iff$:

   - $detA \neq 0$
   - columns of A are linearly independent
   - rows of A are linearly independent
   - $N(A) = \{z \in \mathbb{K}^n : A\underline{z} = \underline{0}\} = \{\underline{0}\}$

2.3 Theory: Linear Systems of Equations

2.3.1 Sensitivity/conditioning of LSE

Sensitivity: Quantities how small (relative) perturbation of data lead to changes of the output

Vector Norms & (induced) Matrix norms $||.||$

- $||M|| = \underset{\underline{x}\neq 0}{sup}\frac{||M\underline{X}}{||\underline{x}||} \Rightarrow ||M\underline{x}|| \leq ||M|| \cdot ||\underline{x}||$
- $||\underline{x} + \underline{y}|| \leq ||\underline{x}|| + ||\underline{y}||$
- $||\underline{y} - \underline{y}|| \geq ||\underline{x}|| - ||\underline{y}||$

Sensitivity strongly depends on the choice of norms

Lemma 2.2.2.5 Perturbation lemma

$$B \in \mathbb{R}^{n,n}, ||B|| < 1 \Rightarrow I + B \text{ regular } \wedge ||(I+B)^{-1}|| \leq \frac{1}{1-||B||}$$

T 2.2.2.4 Conditioning of LSE's If A is regular, $\| \Delta A \| < \| A^{-1} \|^{-1}$ and (2.2.2.3), then

- A + Δ A is regular/invertible
  - $\frac{\|x-\Delta x\|}{\|x\|} \leq \frac{\|A^{-1}\| \cdot \|A\|}{1-\|A^{-1}\|\|A\| \cdot \frac{\|\Delta A\|}{\|A\|}} \cdot \left(\frac{\|\Delta b\|}{\|b\|} + \frac{\|\Delta A\|}{\|A\|}\right)$

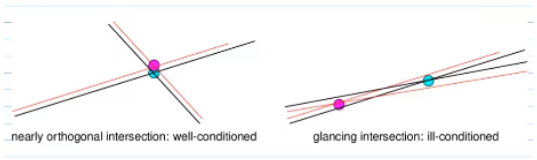
  where:

- $\frac{\|x-\Delta x\|}{\|x\|}$ is the relative error of the result
- $(\frac{\|\Delta b\|}{\|b\|} + \frac{\|\Delta A\|}{\|A\|})$ are the relative perturbations

D 2.2.2.7 Condition number of a matrix

$$Cond(A) := \| A^{-1} \| \cdot \| A \|$$

- **LSE: Well conditioned** $Cond(A) \approx 1$
- **LSE: Ill-conditioned** $Cond(A) >> 1$



cond(A)>> 1 $\iff$ columns/rows of A "almost linearly dependent"

2.4 2.3-2.5: Gaussian Elimination

Gauss Elim: Successive row manipulation of LSE $A\underline{x} = \underline{b}$ to convert it to triangular form. It consists of 2 steps:

1. **forward elimination:** $\mathcal{O}^3$
2. **back substitution:** $\mathcal{O}^2$

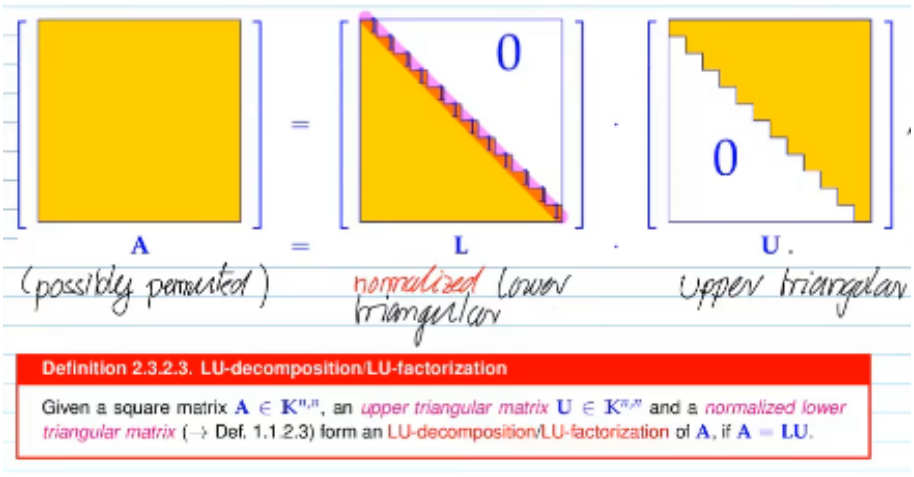
Block Gauss Elim:

Reminder: Matrix multiplication is not commutative!

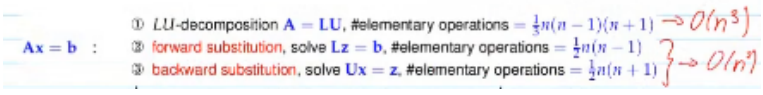
Given the LSE $A\underline{x} = \underline{b}$, where $A \in \mathbb{R}^{n,n}$ we split A into blocks where $A_{1,1} \in \mathbb{R}^{n,n}$ and regular. We can then apply Gauss like for scalars with the exception of commutivity

LU-Decomposition

equivalent to GE i.e cost(LU) = $\mathcal{O}(n^3)$



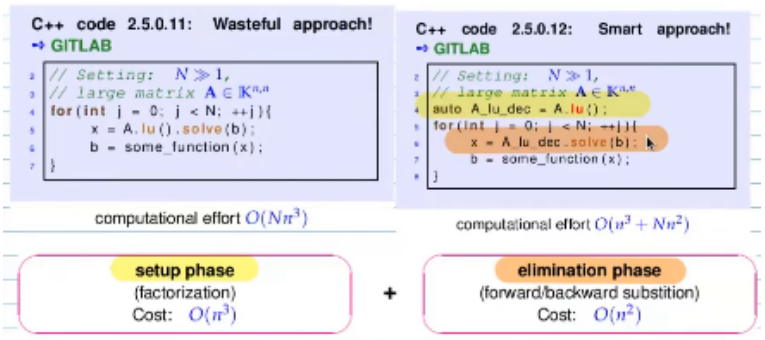
We can solve a LSE with LU-decomposition using the following steps:



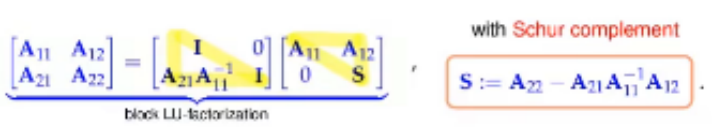
With eigen we can solve LSE with multiple right hand sides i.e we are solving the equation:

$$AX = B \text{ with } X, B \in \mathbb{R}^{n,l}$$

Eigen: X = A.lu().solve(B)
Where lu() calculates the LU-decomposition and solve(B) computes the forward and backward substitution for the columns of B

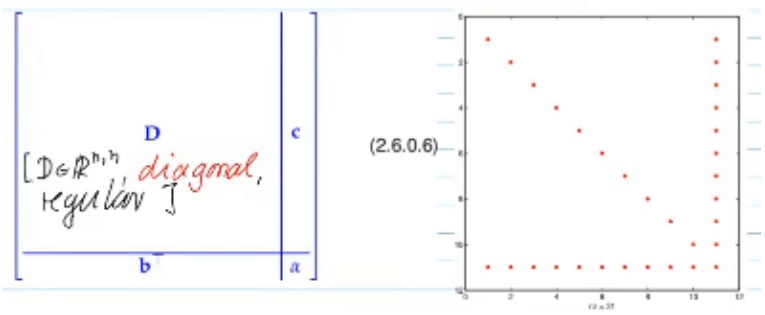


Block LU-Decomposition



2.5 Exploiting Structure when solving Linear Systems

LSE with arrow matrix: An arrow matrix is a matrix with the following form:



We can solve this LSE with either:

- LU-decomposition resulting in a runtime of $\mathcal{O}(n^3)$
- Block GE resulting in a runtime of $\mathcal{O}(n)$

The second method is faster but more vulnerable to round-off errors i.e it is more instable

LSE subject to low-rank modification: Assume Ax = b is easy to solve, because :

1. A has a special structure
2. LU-dec is available

Are goal is to solve $\tilde{A}\tilde{x} = b$ where $\tilde{A}$ arises from A by changing a single entry $(A)_{i^*j^*}$

$$\tilde{A} = A + uv^T, u, v \in \mathbb{R}^n$$

Using a modified Block elimination we get the equation:

$$\tilde{x} = A^{-1}b - \frac{A^{-1}u(v^H(A^{-1}b))}{1+v^H(A^{-1}u)}$$

The generalization of this formula is given by:

Lemma 2.6.0.21. Sherman-Morrison-Woodbury formula

For regular $\mathbf{A} \in \mathbb{K}^{n,n}$, and $\mathbf{U}, \mathbf{V} \in \mathbb{K}^{n,k}$, $n, k \in \mathbb{N}$, $k \leq n$, holds

$$(\mathbf{A} + \mathbf{U}\mathbf{V}^H)^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I} + \mathbf{V}^H\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^H\mathbf{A}^{-1},$$

if $\mathbf{I} + \mathbf{V}^H\mathbf{A}^{-1}\mathbf{U}$ is regular.

General rank k - matrix

2.6 Sparse Linear Systems

A sparse matrix is characterized by:

- "almost all" entries = 0
- for which it is worth while exploiting this fact i.e a 2x2 matrix is not sparse

Sparse LSE arise in models of large networks where each node is connected to only a few other nodes.

2.6.1 Sparse Matrix storage formats

A storage format should fulfill the following conditions:

- Memory nnz(A) (number of non zero elements in A)
- cost (A· vector) nnz(A)
- provide easily accessible information about the location of non zero entries

[COO/triplet format: Stores A as a list/sequence of tuples (i,j,value)

```
struct Triplet {
  size_t i; // row index
  size_t j; // column index
  scalar_t a; // additive contribution to matrix entry
};
using TripletMatrix = std::vector<Triplet>;
```

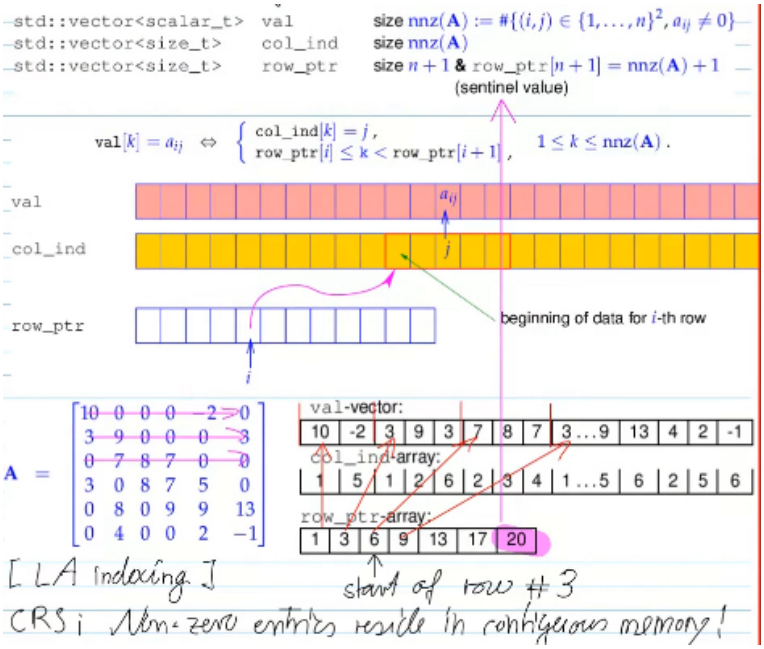
C++-code 2.7.1.2: Matrix×vector product $\mathbf{y} = \mathbf{A}\mathbf{x}+\mathbf{y}$ in triplet format

```
1 void multTripletMatvec(const TripletMatrix &A,
2                        const vector<scalar_t> &x,
3                        vector<scalar_t> &y)
4 for (size_t l=0; l<A.size(); l++) {
5   y[A[l].i] += A[l].a*x[A[l].j];
6 }
```

Allowed: repeated index pairs → values summed up

$$(A)_{i,j} = \sum_{k : A[k].i = i,\, A[k].j = j} A[k].a$$

Compresed row storage (CRS)



2.6.2 Sparse Matrices in Eigen

default format is CRS

```
#include <Eigen/Sparse>
Eigen::SparseMatrix<int, Eigen::ColMajor> Asp(rows,cols); // CCS
   format
Eigen::SparseMatrix<double, Eigen::RowMajor> Bsp(rows,cols); // CRS
   format
```

The challenge is to efficiently initialize the matrix.
Setting random entries is inefficient because it causes massive data movement. Setting an element which was 0 before means we have to insert it in val, update col_ind and$row - ptr$ arrays There are two possibilities to fix this:

1. Use intermediate COO/triplet format i.e we initialize with triplets and then convert to CRS format

```
std::vector <Eigen::Triplet <double > > triplets;
// ..  fill the std::vector triplets ..
Eigen::SparseMatrix<double, Eigen::RowMajor> spMat(rows, cols);
spMat.setFromTriplets(triplets.begin(), triplets.end());
```

O(#triplets) - complexity

2. Use reserve() & insert(), if nnz per row/column is known ahead of time

C++-code 2.7.2.1: Accessing entries of a sparse matrix: potentially inefficient!

```
unsigned int rows,cols,max_no_nnz_per_row;
.....
SparseMatrix<double, RowMajor> mat(rows,cols);
mat.reserve(RowVectorXi::Constant(cols,max_no_nnz_per_row)); --> Allocation of enough space
// do many (incremental) initializations
for ( ) {
  mat.insert(i,j) = value_ij;
  mat.coeffRef(i,j) += increment_ij;
}
mat.makeCompressed();
```

} O(1), if enough space reserve()'d

↳ squeeze out zeros

2.6.3 Direct Solution of Sparse LSE's

Assume: System matrix is in sparse matrix format i.e tells the location of the zero entries. Hence can be exploited by sparse elimination techniques.
Solving a sparse LSE with Eigen

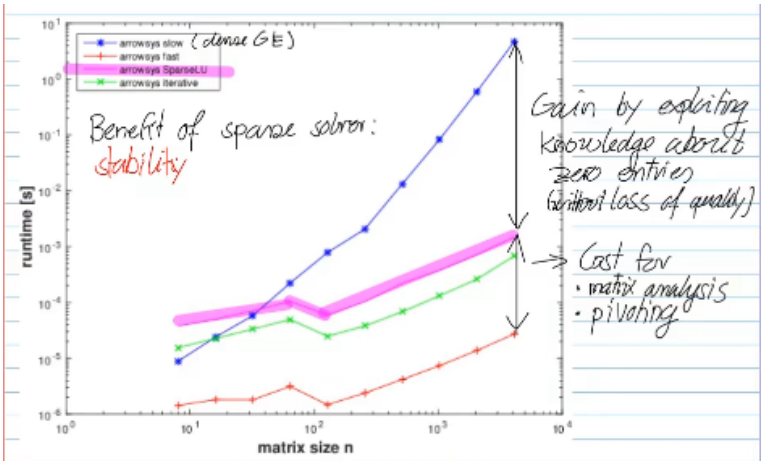
C++-code 2.7.3.1: Function for solving a sparse LSE with EIGEN ⇒ GITLAB

```
using SparseMatrix = Eigen::SparseMatrix<double>;
// Perform sparse elimination
void sparse_solve(const SparseMatrix &A, const VectorXd &b, VectorXd &x) {
  Eigen::SparseLU<SparseMatrix> solver(A);
  if (solver.info() != Eigen::Success) {
    throw "Matrix factorization failed";
  }
  x = solver.solve(b);
}
```

→ the expensive part
→ Check this
→ triangular solves : cheap

Sparse elimination can fail because:

- The matrix is not singular
- The matrix has an awkward structure



In practice when using sophisticated sparse matrix solver libraries the cost of solving Ax= b is $\mathcal{O}((nnz(A))^{\alpha})$ where $\alpha \approx 1.5 - 2.5$

Chapter 3

Direct Methosds for Linear Least Squares Problems

Overdetermined LSE: An LSE with more equations than unknowns i.e $m \geq n$

3.1 Overdetermined LiSE's

Linear parameter estimation in 1D We measure 2 quantities connected by a linear physical law

$$y = \alpha x + \beta \text{ for some unknown parameters } \alpha, \beta \in \mathbb{R}$$

We make n measured pairs i.e $(x_i, y_i) \in \mathbb{R}^2, i = 1, ..., n, n > 2$ in order to determine α, β
if the measurements were precise the following linear equations would be satisfied:

$$y_i = \alpha x_i + \beta \quad \forall i$$

this results in and LSE. However the measured values are affected by measurement errors e.g the y's are affected
$\Rightarrow$ The RHS might be perturbed
$\Rightarrow$ The RHS is no longer a solution for the matrix A $\Rightarrow$ No solution may exists

Linear regression: linear parameter estimation We are now given the following linear law:

$$y = \underline{a}^T \underline{x} + \beta \text{ with } x, a \in \mathbb{R}^n, \beta \in \mathbb{R}$$

Measurements:

$$(\underline{x}_i, y_i) \in \mathbb{R}^n \times \mathbb{R}, i = 1, ..., m$$

Thy system is Overdetermined if $m > n+1$. As in the above example the result might be perturbed and become unsolvable.

Principle in data science: You cannot afford not to use every piece of information available

Angles in triangulations

Triangulation: Covering an area in triangles to determine the distance of points.

We have the following pieces of information available:

- Each angle is supposed to be equal to its measured value
- The sum of interior angles is π for ecery triangle $\rightarrow$ +#triangles extra equations
- The sum of angles at an interior node is $2\pi \rightarrow$ +# interior nodes extra equations

Hence we have an OD LSE

3.2 Least Squares Solution Concepts

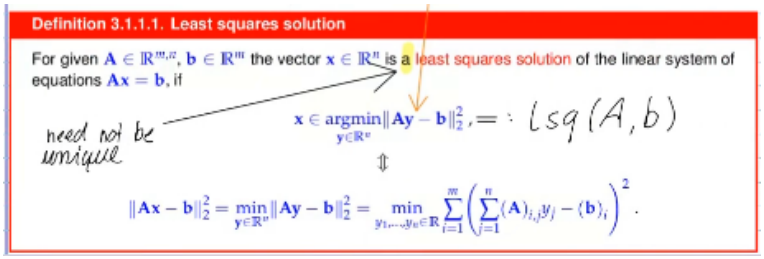
Setting: OD-LSE $A\underline{x} = \underline{b}$

3.2.1 Least Squares Solutions: Definition

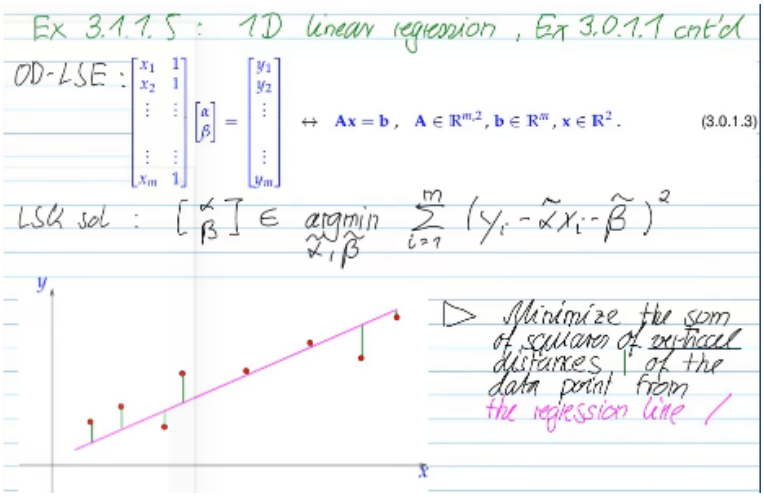
A LSQ solution of $A\underline{x} = \underline{b}$ is a vector $x \in \mathbb{R}^n$ that makes the **residual**:

$$r := \underline{b} - A\underline{x}$$

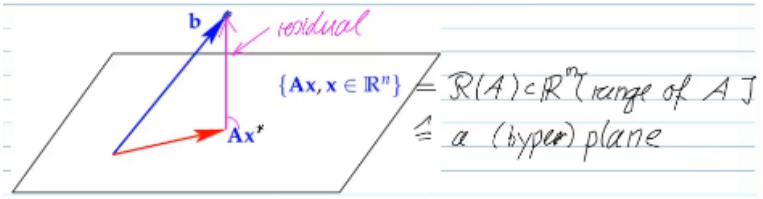
as small as possible with respect to the euclidean norm.



Generalization: $A \in \mathbb{R}^{n,n} regular \Rightarrow Lsq(A, \underline{b}) := \{A^{-1}, \underline{x}\}$



Geometric interpretation of least squares solution of $A\underline{x} = \underline{b}$



$A\underline{x}^*$ is closest to $\underline{b}$ in R(A), $x^* := LSQ$ sol
$\Rightarrow A\underline{x}^* = \underline{b}_0 =$ orthogonal projection of $\underline{b}$ onto R(A)

Geometric intuition: Since there will always be a point closest to b a least squares solution always exists:

$$\forall A \in \mathbb{R}^{m,n}, \underline{b} \in \mathbb{R}^m : lsq(A, b) \neq \emptyset$$

3.2.2 Normal Equations

The vector $x \in \mathbb{R}^n$ is a least squares solution of the linear system of equations:

$$Ax = b, A \in \mathbb{R}^{m,n}, b \in \mathbb{R}^m$$

if and only if it solves the **normal equations (NEQ)**

$$A^T Ax = A^T b$$

From this follows:

$$x \in lsq(A, \underline{b}) \iff A^T Ax = A^T b$$

Chapter 4

C++

4.1 Templates and Template Classes in C++

Templates: Let you define the behavior of the class without actually knowing what datatype will be handled by the operations of the class i,e a templated class does not depend on the datatype it deals with. The basic syntax for declaring a templated class:

```
template <class a_type> class a_class {...};
```

a_type is not a keyword, its an identifier that during the execution of the program will represent a single datatype

When defining a function as a member of a templated class, it is necessary to define it as a templated function

```
template<class a_type> void a_class<a_type>::a_function(){...}
```

specialization: An instantiated object of a templated class
Example:

```
class calc
{
  public:
    int multiply(int x, int y);
    int add(int x, int y);
 };
int calc::multiply(int x, int y)
{
  return x*y;
}
int calc::add(int x, int y)
{
  return x+y;
}
```

```
template <class A_Type> class calc
{
  public:
    A_Type multiply(A_Type x, A_Type y);
    A_Type add(A_Type x, A_Type y);
};
template <class A_Type> A_Type calc<A_Type>::multiply(A_Type x,A_Type y)
{
  return x*y;
}
template <class A_Type> A_Type calc<A_Type>::add(A_Type x, A_Type y)
{
  return x+y;
}
```

Keyword: typename In a template declaration "typename" can be used as an alternative to class to declare type template parameters

4.2 Scope Resolution Operator "::"

The scope Resolution Operator is used for one of the following:

- Access the global variable, if there is a local variable with the same name
- Define a function outside the class
- Access a class's static variable
- Multiple Inheritance
- Refer to a class inside another class

4.3 Namespaces

Namespaces provide a method for preventing name conflicts in large projects. Symbols declared inside a namespace block are placed in a named scope that prevents them from being mistaken for identically-named symbols in other scopes

using-declaration:

using $ns_$name::name

Makes the symbol name from the namespace *ns_name* accessible for unqualified lookup as if declared in the same class scope, block scope or namespace as where this using-declaration appears

unqualified lookup

4.4 Name Lookup

The procedure by which a name, when encountered in a program, is associated with the declaration that introduced

Unqualified name lookup An unqualified name, is a name that does not appear to the right of a scope resultion operator. Name lookup examines the scopes as described below, until it finds at least one declaration of any kind at which time the lookup stops and no further scopes are examined.