Numerical Methods for Computer Science
HS2023 Essentials

Simon Sure
info@simonsure.com / simon.sure@inf.ethz.ch

January 17, 2024

This is an unofficial script containing information from the Numerical Methods for Computer Science course as taught in HS of 2023. Although I am fairly confident that this contains everything mentioned in the lecture videos, ... - there is no guarantee for correctness of completeness!

If you have remarks or questions, or find any mistakes, don't hesitate to send me a mail to info@simonsure.com.

Also, feel free to share this script via https://simonsure.com/ethz which will always contain a link to the newest version. Use the date of this document to identify the newest version.

Contents

1 Nice to Know

We often consider graphs to analyze runtimes etc. In many cases we consider lin-log graphs, which have a linear scale on the x-axis and a logarithmic scale on the y-axis. This exhibits exponential growth as linear graphs. Also, we have log-log graphs at times, which ave a logarithmic scale on both axis. Such settings are used to demonstrate polynomials, which are shown as lines. Monomials with higher coefficients correspond to higher exponents.

2 Computing with Matrices and Vectors

This heavily bases on Linear Algebra. We will deal with vectors $\mathbf{x} \in \mathbb{K}$ where $\mathbb{K}$ is either $\mathbb{R}$ or $\mathbb{C}$. We will mostly consider real numbers here. Similarly, we work with matrices $\mathbf{A} \in \mathbb{K}^{n,m}$ where n is the number of rows and m is the number of columns. We write

$$\mathbf{A} = \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & \vdots & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix}$$

and denote individual entries with $(\mathbf{A})_{i,j} = a_{ij}, 1 \leq i \leq n, 1 \leq j \leq m$.

We alwo consider submatrices. Those are defined by boundary indices:

$$(\mathbf{A})_{k:l,r:s} := [a_{ij}]_{\substack{i=k,\dots,l \\ j=r,\dots,s}}, 1 \leq k \leq l \leq n, 1 \leq r \leq s \leq m$$

A special case of submatrices are row and column vectors of some matrix $\mathbf{A}$. The i-th row is $(\mathbf{A})_{i,:} := [a_{i,1}, \dots, a_{i,m}], 1 \leq i \leq n$. The j-th columns is $(\mathbf{A})_{:,j} := [a_{1,j}, ..., a_{n,j}]^\top, 1 \leq j \leq m$.

> Notice that the indexing is different between math notation, where we start with 1, and programming/C++/code notation, where we start enumeration with 0.

There are some special matrices, which are of utmost importance:

- identity matrix

$$I := I_n := \begin{bmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{bmatrix} \in \mathbb{K}^{n,n}$$

- zero matrix

$$O := O_{n,m} := \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix} \in \mathbb{K}^{n,m}$$

- diagonal matrix

$$\text{diag}(d_1, ..., d_n) := \begin{bmatrix} d_1 & & 0 \\ & \ddots & \\ 0 & & d_n \end{bmatrix} \in \mathbb{K}^{n,n}, d_j \in \mathbb{K}, j = 1, ..., n$$

2.1 Software and Libraries

2.1.1 Eigen - a C++ library

Using performant numerical libraries is important for fast and efficient code. We use Eigen, which is a header-only C++ library. Header-only means that the library is compiled each time we compile a program. This is achieved through the library being entirely in its header files. Thus, the compiler can perform more optimizations.

Generally, there are many different ways to implement things in Eigen. And different (shorthand) notations. We won't make all this explicit. Consider the following code examples to understand how to work with types, templates, ...

Types & Creating Matrices and Vectors The fundamental data type in Eigen is `Matrix`. We can define three types of matrices:

- fixed size: size known at compile time
- dynamic size: size only known at runtime
- sparse/special matrices: only considered later (implicitly)

Additionally, we have to specify which scalar type we want the matrix entries to be. This will mostly be real numbers but can also be complex numbers. Below we will assume real numbers. If one uses different/custom scalars, one must cast appropriately.

Listing 1: Declaring Types in Eigen

```
using dynMat_t = Eigen::Matrix<Scalar, Eigen::Dynamic, Eigen::
    Dynamic>;
using dynColVec_t = Eigen::Matrix<Scalar, Eigen::Dynamic, 1>;
using dynRowVec_t = Eigen::Matrix<Scalar, 1, Eigen::Dynamic>;
using index_t = typename dynMat_t::Index;
using entry_t = typename dynMat_t::Scalar;
```

Listing 2: Declaring Matrices & Vectors

```
Eigen::Matrix<Scalar, Eigen::Dynamic, 1> colvec(dim);
Eigen::Matrix<Scalar, 1, Eigen::Dynamic> rowvec(dim);
Eigen::Matrix<Scalar, Eigen::Dynamic, Eigen::Dynamic> testmat(
    dim);
Eigen::Matrix<Scalar, Eigen::Dynamic, Eigen::Dynamic> testmat2(
    dim, dim-2);

Eigen::MatrixXd testmat(dim);
Eigen::VectorXd colvec(dim);
Eigen::RowVectorXd rowvec(dim);
Eigen::Matrix3d test3mat;
Eigen::Vector4d col4vec;
```

Here, `X` corresponds to dynamic size while `d` indicates that we use doubles as the scalar type. Those are just shorthand notations for what we have done manually above. Furthermore, for fixed sized matrices, we can only use this notation for sizes up to including 5. Afterwards, we use dynamic sizes with this shorthand-notation or must manually write the entire type.

Listing 3: Initializing Matrices & Vectors

```
+ zero
+ constant
+ diagonal
+ random
+ identity
```

Listing 4: Accessing Matrices & Vectors

```
colvec[i] = 1; // access vector with square brackets
vecprod(1,1) = 2; // access matrix with round brackets

```

```
M.block(1,1,p,q) += Eigen::MatrixBase<MatType>::Constant(p,q
    ,1.0);
MatirxXd B = M.block(1,1,p,q);

M.topRows(p);
M.bottomRows(p);
M.leftCols(p);
M.rightCols(p);
M.template triangularView<Upper>(); // .template is used for
    additional safety but mostly unnecessary
```

In Linear Algebra more generally, we can consider submatrices. This is implemented through blocks in Eigen. `.block(i,j,p,q)` corresponds to $(\mathbf{M})_{i+1:i+p,j+1:j+p}$. The $+1$ comes from the different indexing schemes.

Notice that through blocking we access/edit the original matrix and don't create a copy!

Listing 5: Computing with Matrices & Vectors

```
dynMat_t vecprod = colvec*rowvec;
```

The computations we can perform are very intuitive. Multiplying matrices corresponds to normal matrix multiplication. In the library, this is achieved through operator overloading. When we want to perform element-wise operations such as element-wise multiplication, we need to work with the matrices as arrays instead.

Listing 6: Matrix & Vector Methods

```
const int nrows = M.rows();
const int ncols = M.cols();
M.transpose(); // reinterpreting matrix for use as transpose for
    lazy evaluation (not modification)
M.transposed(); // creating a new transposed matrix at call
    point (no modification)
```

2.1.2 Dense Matrix Storage Formats

An interesting question is to ask how matrices are stored physically in memory. Conceptually, memory is a linear array while matrices are 2D. To store dense matrices are two basic approaches. Dense is not a precisely defined term. It means that basically all matrix entries contain information. On the other hand, we have sparse matrices. Those are matrices, which have a special structure so that only a few entries contain relevant information. Diagonal matrices and identity matrices are example. For those, we only want to store the relevant entries and the special form. But for dense matrices we must consider all entries.

Generic dense matrices are stored in row- or column-major order. Row-major means that the elements of rows are contiguous in memory, while column-major means that column entries are contiguous in memory.

- row-major: C, C++, Bitmaps, Python
- column-major: Fortran, Matlab, Eigen (default)

With `A(i)`, where A is a matrix in Eigen, one gets the i-th element when considering the linear row-major memory orientation. And with `A.data(i)` one can get a raw pointer to the corresponding element.

The default majority in Eigen ist column-major. While the majority doesn't matter for writing code, it provides optimization opportunities for certain operations. The performance benefits mainly stem from cache and memory miss latencies. Which majority is faster depends on the application. Thus, Eigen also offers the option to store matrices in column-major form.

Eigen is column-major by default. Codes optimizes for column majority usually run faster.

2.2 Computational Effort/Computational Complexity

Computational complexity is introduced as asymptotic complexity in other courses. But for numerical programs, constants determined by memory access patterns etc. have a big impact. Thus, we also consider asmyptotic behavior as a good indicator. But for actual performance assessments, we must time the real-world performance of our programs.

When wanting to time executing to measure performance, always execute multiple iterations of our programs, measure the runtime, and take the average result.

```
1  void timing(void)
2  {
3    const int K = 3; // Number of repetitions
4    // store time before all iterations
5    for(int k=0;k<K;k++) {
6      auto tic =  high_resolution_clock::now(); // time before
           this iteration
7      // OPERATION TO BE TIMED
8      auto toc = high_resolution_clock::now(); // time after this
           iteration
9      double t = (double)duration_cast<microseconds>(toc-tic).
           count()/1E6; // duration of this iteration
10     // store time o fthis iteration
11   }
12   // time after all iterations
13   // store time of all iterations
14   // compute average among all iterations
15   // print timing results
16 }
```

2.2.1 Asymptotic Complexity

Definition (Asymptotic) Complexity: The asymptotic complexity of an algorithm characterizes the worst-case dependence of its computational effort on one or more problem size parameter(s) when these tend to ∞.

The problem size parameter usually is the matrix/vector size. We have been familiarized with $\mathcal{O}, \Theta, o$, and Ω notation. Here, we mostly consider $\mathcal{O}$-notation (Landau notation). $cost(n) = \mathcal{O}(f(n))$ (or technically correct $cost(n) \in \mathcal{O}(f(n))$) means

$$\exists C > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, cost(n) \leq C \cdot f(n)$$

In most cases we consider polynomial runtimes $\mathcal{O}(n^\alpha), \alpha > 0$. For precise statements, we want our runtimes to be sharp and make sharpness assumptions regarding given runtimes. That

means if $cost(n) = \mathcal{O}(n^\alpha)$ we have $cost(n) \neq \mathcal{O}(n^\beta), \beta < \alpha$. Sharpness not only holds for polynomial runtimes. For an even more precise statements we may also consider Θ-notation, which specify lower and up bounds.

As with asymptotic complexity, this only makes statements for large input sizes. For small matrices, the constant factors will dominate the runtime.

2.2.2 Computational Cost of Basic Numerical Linear Algebra Operations

operation	description	#mult/#div	#add/#sub	asympt. complexity
dot product	$\mathbf{x}, \mathbf{y} \in \mathbb{R} \mapsto \mathbf{x}^H\mathbf{y}$	n	$n-1$	$\mathcal{O}(n)$
tensor product	$\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n \mapsto \mathbf{x}\mathbf{y}^H$	nm	0	$\mathcal{O}(nm)$
matrix-vector nult.	$\mathbf{x} \in \mathbb{R}^n, \mathbf{A} \in \mathbb{R}^{m,n} \mapsto \mathbf{A}\mathbf{x}$	mn	$(n-1)m$	$\mathcal{O}(mn)$
matrix product	$\mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{B} \in \mathbb{R}^{n,k} \mapsto \mathbf{A}\mathbf{B}$	mnk	$mk(n-1)$	$\mathcal{O}(mnk)$

Figure 1

Of those three elementary operations, only matrix multiplication isn't optimal. With Strassen's algorithm or similar algorithms, we can get close to $\mathcal{O}(n^2)$ for square matrices but reduce the exponent to some fraction in any case. But due to very high constants, we implement matrix multiplication using a triple loop in basically all cases.

2.2.3 Some Tricks to Improve Complexity

This considers some very simple tricks/approaches to reduce the asymptotic complexity of evaluating some expression. Here, assume that expressions such as $\mathbf{A}\mathbf{x}^H\mathbf{y}$ are given. We want to see how to optimize evaluation compared to just evaluating always the leftmost possible exprsesion until we have a result.

Exploit Associativity While matrix and vector operations generally aren't commutative, they are associative. And changing the order of evaluation can noticeably improve complexity.

Consider the column vectors $\mathbf{a}, \mathbf{b}, \mathbf{x} \in \mathbb{R}^n$. Computing $(\mathbf{a}\mathbf{b}^\top)\mathbf{x}$ and $\mathbf{a}(\mathbf{b}^\top\mathbf{x})$ yield the same result. But the former has complexity $\mathcal{O}(n^2)$ while the latter has complexity $\mathcal{O}(n)$.

Change the order of evaluation of algebraic expressions to improve asymptotic runtime.

Hidden Summation This concept is more difficult to express generally. But it many cases, computations with matrices and vectors can be interpreted in a meaningful way. For instance, multiplying a vector $\mathbf{a} \in \mathbb{R}^n$ with a matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, where $(\mathbf{A})_{i,j} = \begin{cases} 0, & i > j \\ 1, & \text{else} \end{cases}$, i.e., an upper triangular matrix, corresponds to the result $\mathbf{y} = \mathbf{A}\mathbf{a}$ having $(\mathbf{y})_i = \sum_{j=1}^{i}(\mathbf{a})_j$. Then, the elements of $\mathbf{y}$ can be computed with DP by just continuously adding elements of $\mathbf{y}$ and storing the intermediate results as the elements of $\mathbf{y}$. Here we go from the trivial multiplication in $\mathcal{O}(mn)$ to $\mathcal{O}(n)$ with DP.

Let's consider a more complex example. Given $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,p}, p \ll n, \mathbf{x} \in \mathbb{R}^n$, we want to compute $\mathbf{y} = \text{upperTriangular}(\mathbf{A}\mathbf{B}^\top)\mathbf{x}$. Trivially, this is $\mathcal{O}(n^2p)$ if we do the matrix multiplication first, the n set the lower triangle to zero and multiply with $\mathbf{x}$. If we consider the special case

$p = 1$ we would have $\mathcal{O}(n^2)$. But in that case we can perform a matrix factorization of $\mathbf{ab}^\top$ so that we can benefit from cumulative summation.

$$y = triu(ab^\top)x = \begin{bmatrix} a_1b_1 & a_1b_2 & \dots & & \dots & a_1b_n \\ 0 & a_2b_2 & a_2b_3 & \dots & \dots & a_2b_n \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & \vdots \\ 0 & \dots & & \dots & 0 & a_nb_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \\ \vdots \\ x_n \end{bmatrix}$$

$$= \begin{bmatrix} a_1 & & \\ & \ddots & \\ & & a_n \end{bmatrix} \left(\begin{bmatrix} 1 & 1 & \dots & & \dots & 1 \\ 0 & 1 & 1 & \dots & \dots & 1 \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & \vdots \\ 0 & \dots & & \dots & 0 & 1 \end{bmatrix} \left(\begin{bmatrix} b_1 & & & \\ & \ddots & & \\ & & \ddots & \\ & & & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \\ \vdots \\ x_n \end{bmatrix} \right) \right)$$

Here we then have two scalings and one reverse summation. All can be done in $\mathcal{O}(n)$. So we have $\mathcal{O}(n)$ instead of $\mathcal{O}(n^2)$ by utilizing hidden summations.

This then can also be generalized to $p > 1$ by recognizing that $\mathbf{AB}^\top = \sum_{l=1}^{p} (\mathbf{A})_{:,l}(\mathbf{B})_{:,l}^\top$ and that the triangular operation is linear. So:

$$triu(\mathbf{AB}^\top)\mathbf{x} = \sum_{l=1}^{p} triu((\mathbf{A})_{:,l}(\mathbf{B})_{:,l}^\top)x$$

The inner computation can be done in $\mathcal{O}(n)$ as above. With the p summands we then get $\mathcal{O}(np)$ instead of $\mathcal{O}(n^2p)$ originally.

If an arithmetic expression can be rewritten to contain a matrix with all ones in the upper triangle, one can often use this to reduce a matrix multiplication to $\mathcal{O}(n)$.
Also, matrix-vector multiplications with diagonal matrices are just scalings which can be done in $\mathcal{O}(n)$.

Reuse of Intermediate Results .

Definition Kronecker Product: The Kronecker product $A \otimes B$ of two matrices $A \in \mathbb{K}^{m,n}$ and $B \in \mathbb{K}^{l,k}, m, n, l, k \in \mathbb{N}$ is the $(ml) \times (nk)$-matrix

$$A \otimes B = \begin{bmatrix} (A)_{1,1}B & (A)_{1,2}B & \dots & \dots & (A)_{1,n}B \\ (A)_{2,1}B & (A)_{2,2}B & & \vdots & \\ \vdots & \vdots & & & \vdots \\ (A)_{m,1}B & (A)_{m,2}B & \dots & \dots & (A)_{m,n}B \end{bmatrix} \in \mathbb{K}^{ml,nk}$$

With $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,n}$ and $\mathbf{x} \in \mathbb{R}^{n^2}$ we can compute $\mathbf{y} = (\mathbf{A} \otimes \mathbf{B})\mathbf{x}$. Trivially this can be done

in $\mathcal{O}(n^4)$. But the product reveals that some parts re-reused:

$$(\mathbf{A}\otimes\mathbf{B})\mathbf{x} = \begin{bmatrix} (\mathbf{A})_{1,1}\mathbf{B}\mathbf{x}^1 + (\mathbf{A})_{1,2}\mathbf{B}\mathbf{x}^2 + \cdots + (\mathbf{A})_{1,n}\mathbf{B}\mathbf{x}^n \\ (\mathbf{A})_{2,1}\mathbf{B}\mathbf{x}^1 + (\mathbf{A})_{2,2}\mathbf{B}\mathbf{x}^2 + \cdots + (\mathbf{A})_{2,n}\mathbf{B}\mathbf{x}^n \\ \vdots \\ \vdots \\ (\mathbf{A})_{m,1}\mathbf{B}\mathbf{x}^1 + (\mathbf{A})_{m,2}\mathbf{B}\mathbf{x}^2 + \cdots + (\mathbf{A})_{m,n}\mathbf{B}\mathbf{x}^n \end{bmatrix}, \mathbf{x} = \begin{bmatrix} \mathbf{x}^1 \\ \mathbf{x}^2 \\ \vdots \\ \mathbf{x}^n \end{bmatrix}, \mathbf{x}^i \in \mathbb{R}^n$$

The idea is to precompute $\mathbf{B}\mathbf{x}^l, l = 1, ..., n$, because those factors are roused n times each: $\mathbf{Z} = \mathbf{B}\begin{bmatrix}\mathbf{x}^1, ..., \mathbf{x}^n\end{bmatrix} = [\mathbf{B}\mathbf{x}^1, ..., \mathbf{B}\mathbf{x}^n]$. The precomputation can be done in $\mathcal{O}(n^3)$.

For one line of $(\mathbf{A}\otimes\mathbf{B})\mathbf{x}$ we compute $(\mathbf{A})_{1,1}\mathbf{B}\mathbf{x}^1 + (\mathbf{A})_{1,2}\mathbf{B}\mathbf{x}^2 + ... + (\mathbf{A})_{1,n}\mathbf{B}\mathbf{x}^n = \mathbf{Z}(\mathbf{A})_{1,:}^\top$ in $\mathcal{O}(n^2)$. If we compute all lines we get $\mathbf{Y} = \mathbf{Z}\mathbf{A}^\top$ which can then be done in $\mathcal{O}(n^3)$. In the end, we must reshape to get the result:

$$\mathbf{y} = \begin{bmatrix} (\mathbf{Y})_{:,1} \\ \vdots \\ (\mathbf{Y})_{:,n} \end{bmatrix} \in \mathbb{R}^{n^2}$$

This is also straightforward to implement in C++:

```
template <class Matrix, class Vector>
Vector kronmultv(const Matrix &A, const Matrix &B, const Vector
    &x){
  unsigned int m = A.rows(); unsigned int n = A.cols();
  unsigned int l = B.rows(); unsigned int k = B.cols();
  // 1st matrix mult. computes the products Bx^j
  // 2nd matrix mult. combines them linearly with the
     coefficients of A
  Matrix t = B * Matrix::Map(x.data(),k,n) * A.transpose();
  return Matrix::Map(t.data(), m*l, 1);
}
```

2.3 Machine Arithmetic and Consequences

From Linear Algebra we are already familiar with the Gram-Schmidt algorithm to find an orthogonal basis for some vector space given some non-orthogonal basis. The algorithm outputs an orthogonal basis. Meaning, they are pairwise orthogonal and have norm 1. With $\mathbf{Q} = \begin{bmatrix}\mathbf{q}_1 & \dots & \mathbf{q}_n\end{bmatrix}$ we then have $\mathbf{Q}^\top\mathbf{Q} = \mathbf{I}$.

Listing 7: Gram-Schmidt Algorithm

```
template <class Matrix> Matrix gramschmidt(const Matrix &A) {
  Matrix Q = A;
  // First vector just gets normalized
  Q.col(0).normalize();
  for (unsigned int j = 1; j < A.cols(); ++j) {
    // Replace inner loop over each previous vector in Q with
       fast
    // matrix-vector multiplication
    Q.col(j) -= Q.leftCols(j) * (Q.leftCols(j).adjoint() * A.col
       (j));
```

```
    // Normalize vector, if possible.
    // Otherwise colums of A must have been linearly dependent
    if (Q.col(j).norm() <= 10e-9 * A.col(j).norm()) {
      std::cerr << "Gram-Schmidt failed: A has lin. dep columns.
          " << std::endl;
      break;
    } else {
      Q.col(j).normalize();
    }
  }
  return Q;
}
```

But if we run this implementation on $\mathbf{A} = \left[\frac{1}{i+j-1}\right]_{i,j=1}^{n} \in \mathbb{R}^{n,n}$ (called Hilbert Matrix), we get $\mathbf{Q}$ with $\mathbf{Q}^\top\mathbf{Q} \neq \mathbf{I}$! Thus, something must have went wrong.

The reason is that computers can't compute with real numbers directly. Instead, they work with an abstraction of machine numbers. Computing with those only approximates real numbers and, thus, we incur errors in most computations. There are certain situations in which those errors become severe. We will consider those (and how to avoid them) below.

2.3.1 Machine Numbers

Computers store real numbers as floating point numbers. This representation (other than real numbers) is finite. We compute with machine numbers $\mathbb{M}$ instead of real numbers $\mathbb{R}$. Hence, the result of numerical Linear Algebra may differ noticeably from theoretical Linear Algebra.

There are three major error sources with machine numbers:

- Overflow: If some (intermediate) result after rounding is larger than the largest encodable number, we get infinity.
- Underflow: If some (intermediate) result after rounding is smaller than the smallest non-zero number, we get zero.
- Roundoff: There are gaps in between machine numbers as computers work with discrete numbers. This leads to roundoff errors if a result is between to machine numbers.

2.3.2 Roundoff Errors

We already mentioned that some real numbers can't be expressed precisely as machine numbers. More formally, elementary arithmetic operations such as $+, -, *, /, ...$ are not closed with respect to machine numbers. Accordingly, we can't use those operations with machine numbers. Instead we consider modified operations, which round the results to machine numbers: $\tilde{op} := \text{rounding} \circ op$.

Definition Rounding Up: Correct rounding ("rounding up") is given by the function

$$rd : \begin{cases} \mathbb{R} \to & \mathbb{M} \\ x \mapsto & \max \operatorname{argmin}_{\tilde{x} \in \mathbb{M}} |x - \tilde{x}| \end{cases}$$

This is what we consider here. But modern x86 machines round to the closest even number if we are exactly in between two machine numbers.

The error we incur can be precisely defined:

$$\text{relative error of } rd \hat{=} \frac{|\tilde{op}(x,y) - op(x,y)|}{|op(x,y)|} = \frac{|(rd - ld)op(x,y)|}{|op(x,y)|}$$

The precision of machine numbers may vary between machines and different floating point types (floats, doubles, ...). But the machine precision is generally given by

$$\max_{x \in |\mathbb{M}|} \frac{|rd(x) - x|}{|x|} =: ESP \approx 2.2 \cdot 10^{-16} \text{ for double} \Rightarrow rd(x) = x(1+\epsilon), |\epsilon| \leq ESP$$

Notice that there is a difference between normalized and denormalized values (compare SPCA for distinguishing those two types). For normalized values, we can get the precision simply by considering the smallest value so that when added to 1 results in a number $\neq 1$. This is considered the machine epsilon/precision, although we have worse precision for denormalized values.

In C++, we can access the epsilon with `std::numeric_limits<double>::epsilon();`.

Listing 8: Gram Schmidt roundoff - simple vs. QR

```
void gsroundoff(MatrixXd& A){
  // Gram-Schmidt orthogonalization of columns of A
  MatrixXd Q = gramschmidt(A);
  // Test orthonormality of columns of Q, which should be an
  // orthogonal matrix according to theory
  cout << setprecision(4) << fixed << "I = " << endl << Q.
     transpose()*Q << endl;
  // Eigen's stable internal Gram-Schmidt orthogonalization by
  // QR-decomposition (discussed later)
  HouseholderQR<MatrixXd> qr(A.rows(),A.cols());
  qr.compute(A); MatrixXd Q1 = qr.householderQ();
  // Test orthonormality
  cout << "I1 = " << endl << Q1.transpose()*Q1 << endl;
  // Check orthonormality and span property
  MatrixXd R1 = qr.matrixQR().triangularView<Upper>();
  cout << scientific << "A-Q1*R1 = " << endl << A-Q1*R1 << endl;
}
```

This shows that roundoff errors are significant. But by using special implementations and optimizing programs for roundoff, one can mitigate their impact. Here, through the QR-factorization, we avoid roundoff errors.

2.3.3 Cancellation

Roundoff by itself is just a phenomenon. But cancellation is how this phenomenon impacts computations in a bad way. Cancellation describes the phenomenon how even tiny relative errors can amplify through multiple computations to huge errors. Thus, even small errors can be destructive.

Cancellation occurs when subtracting (adding with different sign) two numbers. During this process the existing errors in the operands remain, while the actual number becomes almost zero. Then, the relative error usually becomes very big, even if we don't introduce any new error or even reduce the absolute error.

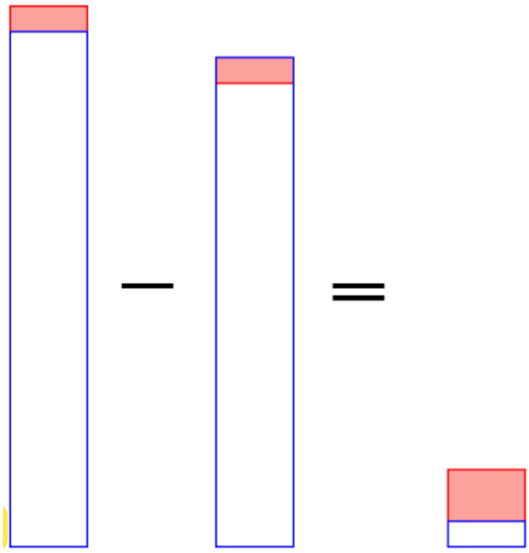


Figure 2: Cancellation - small relative error becomes large relative error

So cancellation occurs when we have subtraction (or addition with negative summand). If we afterwards multiply with a large value or divide with a small value, we blow up the value so that the error becomes a large absolute error.

Example Root of 2nd Degree Polynomial: We use the discriminant formula to compute the roots of a square polynomial:

```
Vector2d zerosquadpol(double alpha, double beta) {
  Vector2d z;
  double D = std::pow(alpha, 2) - 4 * beta;
  if (D < 0)
    throw "no real zeros";
  else {
    // The famous discriminant formula
    double wD = std::sqrt(D);
    z << (-alpha - wD) / 2, (-alpha + wD) / 2;
  }
  return z;
}
```

With $p(x) = x^2 + (y + \frac{1}{y})x + 1 = (x - y)(x - \frac{1}{y})$ we have the two roots y and $\frac{1}{y}$. But when using this function, we get small relative errors with y but very high relative errors for $\frac{1}{y}$.
With roundoff, we easily understand the small relative errors for y. The large relative errors with $\frac{1}{y}$ come from cancellation, because the result is very close to zero after subtraction in the term `-alpha + wD`.

Now we can also understand why the Gram Schmidt algorithm is vulnerable to cancellation. While subtracting the projections on existing orthogonal basisvectors, we incur cancellation as the resulting vector may be small with a large relative error. But normalization then blows up the large relative/small absolute error to also be a large absolute error.

Example Cancellation in Difference Quotient: Consider $f : \mathbb{R} \to \mathbb{R}$ being smooth and $f'(x) \approx \frac{f(x+h)-f(x)}{h}$ for $h \ll 1$. $f(x+h) - f(x)$ will be almost zero with a high relative error then. And if h is almost, then be blow up the high relative error but absolutely small, to a large absolute error.

```
void diffq() {
  double h = 0.1, x = 0.0;
  for (int i = 1; i <= 16; ++i) {
    double df = (exp(x + h) - exp(x)) / h;
    cout << setprecision(14) << fixed;
    cout << setw(5) << -i << setw(20) << abs(df - 1) << endl
       ;
    h /= 10;
  }
}
```

$\log_{10}(h)$	relative error
-1	0.05170918075648
-2	0.00501670841679
-3	0.00050016670838
-4	0.00005000166714
-5	0.00000500000696
-6	0.00000049996218
-7	0.00000004943368
-8	0.00000000607747
-9	0.00000000827403
-10	0.00000000827403
-11	0.00000000827403
-12	0.000000890058234
-13	0.000079927783736
-14	0.000079927783736
-15	0.11022302462516
-16	1.00000000000000

Figure 3

This figure shows that the relative error first becomes small. But for very small h, the relative error becomes large again due to cancellation.
It can be mathematically shown that the best choice of h is $\approx \sqrt{EPS}$.

Numerical codes vulnerable to cancellation are mostly useless.

To test for cancellation, we usually need to check if some value is 'almost' zero (the result and divisor to avoid blowing up the error).
Do NOT do so by testing `==0`. Instead, test for relative smallness!

2.3.4 Avoiding Cancellation

Now it is obvious that we want to avoid cancellation. We can't give one general method to avoid cancellation. But there are multiple techniques of which we will highlight a few.

Changing the Computation Here, we consider the example of findings the roots of a square polynomial of earlier. Vieta's theorem tells us that $x_1 \cdot x_2 = \beta$. Thus, it is sufficiently to accurately compute only one of x_1 and x_2. Then, we get the other through $\frac{\beta}{x_i}$.

For the first root, we can use the discriminant formula. But only use it to compute the large solution, which we can compute accurately as seen in the above example.

```
Eigen::VectorXd zerosquadpolstab(double alpha, double beta) {
  Eigen::Vector2d z(2);
  double D = std::pow(alpha, 2) - 4 * beta; // discriminant
  if (D < 0)
    throw "no real zeros";
  else {
    double wD = std::sqrt(D);
    // Use discriminant formula only for zero far away from 0
    // in order to void cancellation. For the other zero
    // use Vieta's formula.
    if (alpha >= 0) {
      double t = 0.5 * (-alpha - wD);
      z << t, beta / t;
    } else {
      double t = 0.5 * (-alpha + wD);
      z << beta / t, t;
    }
  }
  return z;
}
```

Rewriting Expressions With many expressions, we can perform some trivial arithmetic transformations to get a form not vulnerable to cancellation. We consider two kind of common transformations:

- $\int_0^x \sin t dt = 1 - \cos x$. For $x \approx 0$ we incur cancellation. But with trigonometric identities we get $= 2\sin^2(\frac{x}{2})$, which we can compute with higher accuracy also for small x.
- $y = \sqrt{1+x^2} - \sqrt{1-x^2}$ has cancellation for small x. But with a neat trick we can use the identity $a^2 - b^2 = (a+b)(a-b)$:

$$\frac{(\sqrt{1+x^2} - \sqrt{1-x^2})(\sqrt{1+x^2} + \sqrt{1-x^2})}{\sqrt{1+x^2} + \sqrt{1-x^2}} = \frac{2x^2}{\sqrt{1+x^2} + \sqrt{1-x^2}}$$

  This does not exhibit cancellation.

Trading Cancellation for Approximation Expressions exhibit cancellation only for certain values. For those, we can use the Taylor approximation to get an expression without cancellation. Of course, the Taylor polynomial only is an approximation. However, this approximation may be better compared to error through cancellation.

For some smooth function $f : \mathbb{R} \to \mathbb{R}$ we know the Taylor polynomial holds for some $\zeta \in]x_0, x_0 + h[$

$$f(x_0 + h) = \sum_{k=0}^{m} \frac{f^{(k)}(x_0)}{k!} h^k + \frac{f^{(m+1)}(\zeta)}{(m+1)!} h^{m+1}$$

We then simply disregard the additional term $\frac{f^{(m+1)}(\zeta)}{(m+1)!} h^{m+1}$ to get an approximation. So that we get a good approximation we must choose a sufficiently large m.

Example $\int e^{at} dt$:
$I(a) = \int_0^1 e^{at} dt = \frac{e^a - 1}{a}$ exhibits cancellation for $a \approx 0$.
Thus, we approximate $e^a - 1$ with its Taylor polynomial but leave the factor $\frac{1}{a}$ as it is. Then we get

$$I(a) = \frac{\exp(a) - 1}{a} = \sum_{k=0}^{m} \frac{1}{(k+1)!} a^k + \frac{1}{(m+1)!} \exp(\zeta) a^{m+1}, \text{ for some } \zeta \in [0, a]$$

We then just neglect $R_m(a)$ to get an approximation. Here, we can also give an upper bound for the relative error of this approximation:

$$\begin{aligned} \text{relative error} &= \frac{|I(a) - \tilde{I}_m(a)|}{|I(a)|} = \frac{\frac{e^a - 1}{a} - \sum_{k=0}^{m} \frac{1}{(k+1)!} a^k}{\frac{e^a - 1}{a}} \\ &\leq \frac{1}{(m+1)!} \exp(\zeta) a^{m+1} \leq \frac{1}{(m+1)!} \exp(a) a^m \end{aligned}$$

We can compute the relative error depending on m for some given a. If we are interested in $a = 10^{-3}$ we get:

m	1	2	3	4	5
relative error	$1.0010 \cdot 10^{-3}$	$5.0050 \cdot 10^{-7}$	$1.6683 \cdot 10^{-10}$	$4.1708 \cdot 10^{-14}$	$8.3417 \cdot 10^{-18}$

Figure 4

Based on when the relative error is sufficiently small, we can start using the Taylor polynomial approximation. The approximate number of correct digits can be identified with $\log_{10}$(relative errot).

3 Direct Methods for Linear Systems of Equations

3.1 Theory

Linear Systems of Equations were introduced in Linear Algebra. We have seen that those can be expressed as

$$\mathbf{Ax} = \mathbf{b}$$

with $\mathbf{A} \in \mathbb{R}^{n,n}$ and $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$. $\mathbf{A}$ and $\mathbf{b}$ are given while $\mathbf{x}$ contains the unknowns to be found. We may also consider $\mathbf{A}$s which are not square, but that is a less common case. Especially if want that a solution exists.

The i-th equation for $i = 1, ..., n$ of the LSE corresponds to $(\mathbf{A})_{i,:}\mathbf{x} = (\mathbf{b})_i$.

Existence and Uniqueness of Solutions .

Definition Invertible Matrix: $\mathbf{A} \in \mathbb{K}^{n,n}$ is invertible/regular $:\Leftrightarrow \exists \mathbf{B} \in \mathbb{K}^{n,n} : \mathbf{AB} = \mathbf{BA} = \mathbf{I}$ Then, $\mathbf{B}$ is called the inverse of $\mathbf{A}$. We write $\mathbf{B} = \mathbf{A}^{-1}$.

There are some equivalent criteria to test if some matrix $\mathbf{A}$ is invertible:

- det $\mathbf{A} \neq 0$
- columns or rows of $\mathbf{A}$ linearly independent
- $\mathcal{N}(\mathbf{A}) = \{\mathbf{z} \in \mathbb{K}^n | \mathbf{Az} = 0\} = \{\mathbf{0}\}$ (trivial null space)

If A is invertible, then there exists a unique solution to a SLE $\mathbf{Ax} = \mathbf{b}$ with arbitrary $\mathbf{b}$ as $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \Leftrightarrow \mathbf{x}{=}\mathbf{Bb}$.

DO NOT use matrix inversion to solve LSEs! Instead use the methods detailed later.

3.2 Gaussian Elimination (GE)

3.2.1 Basic Algorithm

Everything in this section should be known from Linear Algebra. We do successive row transformations to turn $\mathbf{A}$ into an upper triangular matrix. This is called forward elimination. Afterwards, we perform back substitution. The details of this should be known and won't be elaborated.

- forward elimination: $\mathcal{O}(n^3)$
- back substitution: $\mathcal{O}(n^2)$

Instead of considering $\mathbf{Ax} = \mathbf{b}$, we can also consider multiple right hand sides at once with $\mathbf{AX} = \mathbf{B}$. This has runtime $\mathcal{O}(n^3) \cdot \mathcal{O}(n) = \mathcal{O}(n^4)$.

Block Gaussian Elimination Instead of considering all elements of $\mathbf{A}$ individually, we can also consider blocks of $\mathbf{A}$ and blocked Gaussian elimination.

$$\left[\begin{array}{cc|c} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{b}_1 \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{b}_2 \end{array}\right] \xrightarrow{\text{forward elimination}} \left[\begin{array}{cc|c} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{b}_1 \\ 0 & \mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12} & \mathbf{b}_2 - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{b}_1 \end{array}\right]$$

$$\xrightarrow{\text{back substitution}} \left[\begin{array}{cc|c} \mathbf{I} & 0 & \mathbf{A}_{11}^{-1}(\mathbf{b}_1 - \mathbf{A}_{12}\mathbf{S}^{-1}\mathbf{b}_s) \\ 0 & \mathbf{I} & \mathbf{S}^{-1}\mathbf{b}_s \end{array}\right],$$

$$\mathbf{S} = \mathbf{A}_{22} - A_{21}\mathbf{A}_{11}^{\top}\mathbf{A}_{12} \text{ is called the Schur complement}$$

$$\mathbf{b_S} = \mathbf{b}_2 - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{b}_1$$

Considering blocked GE comes in handy for certain optimizations down the road.

3.2.2 LU-Decomposition

This is equivalent to Gaussian Elimination but written as a matrix factorization. This makes it simpler to solve the same system for multiple right-hand sides - without runtime penalties.

Definition LU-Decomposition/-Factorization: Given a square matrix $\mathbf{A} \in \mathbb{K}^{n,n}$, an upper triangular matrix $\mathbf{U} \in \mathbb{K}^{n,n}$ and a normlized lower triangular matrix $\mathbf{L} \in \mathbb{K}^{n,n}$ form an LU-decomposition/LU-factorization of $\mathbf{A}$, if $\mathbf{A} = \mathbf{LU}$.

Normalized L means here that the diagonal entries of L are all 1. This is necessary so that the decomposition is unique for given $\mathbf{A}$. The value of $\mathbf{L}$ and $\mathbf{U}$ can be acquired during GE without additional cost. Computing the LU-Decomposition takes $\mathcal{O}(n^3)$. This follows directly from obtaining the coefficients from GE.

Solving LSE via LU We use the LU-Decomposition to solve an LSE.

$$\mathbf{Ax} = \mathbf{B} \Leftrightarrow \mathbf{L}(\mathbf{Ux}) = \mathbf{b}$$
$$\text{solve } \mathbf{Lz} = \mathbf{b} \text{ for } \mathbf{z}$$
$$\text{solve } \mathbf{Mx} = \mathbf{z} \text{ for } \mathbf{x}$$

Thus, solving with LU can be split into three stages:

1. Compute LU-Decomposition: $\mathbf{A} = \mathbf{LU}$.
   $\frac{1}{3}n(n-1)(n+1)$ elementary operations
2. Forward Substitution: solve $\mathbf{Lz} = \mathbf{b}$.
   $\frac{1}{2}n(n+1)$ elementary operations
3. Backward Substitution: solve $\mathbf{Ux} = \mathbf{z}$.
   $\frac{1}{2}n(n+1)$ elementary operations

The setup phase (step 1) completes in $\mathcal{O}(n^3)$. The elimination phase (step 1 & step 2) complete in $\mathcal{O}(n^2)$. Thus, for one solve, the runtime is the same as GE directly. But we can also solve for multiple right-hand sides at an additional cost of $\mathcal{O}(n^2)$ each.

If we consider multiple right-hand sides with $\mathbf{AX} = \mathbf{B}$, we get runtime $\mathcal{O}(n^3) + \mathcal{O}(n^2) \cdot n = \mathcal{O}(n^3)$. This is better than GE directly!

Block LU-Decomposition With the Schur complement $\mathbf{S} = \mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}$ we have:

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{A}_{21}\mathbf{A}_{11}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{0} & \mathbf{S} \end{bmatrix}$$

to get the LU-Decomposition.

3.2.3 Gaussian Elimination & LU-Decomposition in Eigen

We consider the system $\mathbf{AX} = \mathbf{B} \in \mathbb{R}^{n,l}$ with $\mathbf{A} \in \mathbb{R}^{n,n}$ and $\mathbf{X} \in \mathbb{R}^{n,l}$.

In Eigen, we can compute the LU decomposition, i.e., perform phase 1 (setup phase/factorization) on $\mathbf{A}$ with `A.lu();`. Then, we can use the resulting object to perform phrase 2 (elimination) on a vector or matrix with `.solve(B);`.

`.lu();` is $\mathcal{O}(n^3)$. And `.solve(B);` is $\mathcal{O}(n^2 \cdot l)$.

> Generally, use good implementations from Eigen or other numerical libraries. Creating a performant implementation oneself while considering caching, architecture optimizations, etc. is very difficult!

To benefit from LU, make sure to reuse the LU object whenever possible.

Listing 9: How to use LU in Eigen - bad code - $\mathcal{O}(Nn^3)$

```
for (int j = 0; j< N; ++j) {
  x = A.lu().solve(b);
  b = some_function(x);
}
```

Listing 10: How to use LU in Eigen - good code - $\mathcal{O}(n^3 + Nn^2)$

```
auto A_lu_dec = A.lu();
for(int j = 0; j< N; ++j) {
  x = A_lu_dev.solve(b);
  b = some_function(x);
}
```

3.2.4 Special Cases

Here we consider two cases in which we can use a special approach to improve runtime. Specifically, we utilize the special structure of matrices to be able to use block LU form above.

> **Definition** Sherman-Morrison-Woodbury Formula: For regular $A \in \mathbb{K}^{n,n}$, and $U, V \in \mathbb{K}^{n,k}, n, k \in \mathbb{N}, k \leq n$, holds:
>
> $$(A + UV^H)^{-1} = A^{-1} - A^{-1}U(I + V^H A^{-1}U)^{-1}V^H A^{-1}$$
>
> If $I + V^H A^{-1}U$ is regular. UV^H is a general rank k matrix.

LSE with Arrow Matrix Here we want to compute the LU-Decomposition of an arrow matrix. An arrow matrix is a matrix $\mathbf{A} \in \mathbb{R}^{n,m}$ with $(\mathbf{A})_{ij} = 0 \Leftrightarrow (i \neq j) \wedge (i \neq n) \wedge (j \neq m)$. The

resulting matrix then looks like an arrow that points to the bottom right.

$$\begin{bmatrix} ? & 0 & 0 & \cdots & 0 & ? \\ 0 & ? & 0 & \cdots & 0 & ? \\ 0 & 0 & ? & \cdots & 0 & ? \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & ? & ? \\ ? & ? & ? & \cdots & ? & ? \end{bmatrix} = \left[\begin{array}{c|c} \mathbf{D} & \mathbf{c} \\ \hline \mathbf{b}^\top & \alpha \end{array}\right]$$

where $\mathbf{D}$ is a diagonal matrix.

When $\mathbf{D}$ is given by its diagonal and the other vectors and constants are given in obvious form, constructing the matrix and solving for some right-hand side $\mathbf{y}$ as introduced above leads to a runtime of $\mathcal{O}(n^3)$. But this matrix has a special form so that we can optimize the runtime. We use block GE.

Block GE works well here, because it is easy to invert $\mathbf{D}$. We can do so in $\mathcal{O}(n)$. We just apply block GE as introduced above and get:

$$\mathbf{A}\mathbf{x} = \left[\begin{array}{c|c} \mathbf{D} & \mathbf{c} \\ \hline \mathbf{b}^\top & \alpha \end{array}\right] \left[\begin{array}{c} \mathbf{x}_1 \\ \hline \zeta \end{array}\right] = \mathbf{y} := \left[\begin{array}{c} \mathbf{y}_1 \\ \hline \eta \end{array}\right]$$
$$\zeta = \frac{\eta - \mathbf{b}^\top \mathbf{D}^{-1} \mathbf{y}_1}{\alpha - \mathbf{b}^\top \mathbf{D}^{-1} \mathbf{c}}$$
$$\mathbf{x}_1 = \mathbf{D}^{-1}(\mathbf{y}_1 - \zeta \mathbf{c})$$

Remember to check that the denominator is $\neq 0$. But besides that we can perform those evaluations in $\mathcal{O}(n)$ - a significant runtime improvement.

This comes at the cost of decreased numerical stability. This is more vulnerable to roundoff/-cancellation. But it is only relevant in edge cases.

Listing 11: Arrow Matrix Solve

```
VectorXd arrowsys_fast(const VectorXd &d, const VectorXd &c,
    const VectorXd &b, const double alpha, const VectorXd &y) {
  int n = d.size();
  VectorXd z = c.array() / d.array();
  VectorXd w = y.head(n).array() / d.array();
  const double den = alpha - b.dot(z);
  // this is a safe test for approximate equality to 0.0
  if(std::abs(den) < std::numeric_limits<double>::epsilon() * (b
      .norm() + std::abs(alpha))) {
    throw std::runtime_error("Near singular system");
  }
  const double xi = (y(n) - b.dot(w)) / den;
  return (VectorXd(n+1) << w - xi * z, xi).finished();
}
```

Rank-1 Modifications Assume that solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ is easy to solve (having LU-Decomposition, special structure of $\mathbf{A}$, ...). Then we will see that solving $\tilde{\mathbf{A}}\tilde{\mathbf{x}} = \mathbf{b}$ is comparatively easy if $\tilde{\mathbf{A}}$ is a low-rank modification of $\mathbf{A}$. A low-rank modification of $\mathbf{A}$ is $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{M}'$, where M' has a low rank. Here, we consider rank-1 modifications specifically, i.e., M' has rank 1. Any rank 1 matrix can be represented as $uv^\top$ with $u, v \in \mathbb{R}^n$. One common such low-rank modification is to only

alter one entry of the matrix $\mathbf{A}$ to get $\tilde{\mathbf{A}}$. This can be done, because such a rank-1 modification matrix can be easily constructed as $z \cdot e_i \cdot e_j^\top$.

We use block GE to solve this more efficiently:

$$\begin{bmatrix} \mathbf{A} & \mathbf{u} \\ \mathbf{v}^\top & -1 \end{bmatrix} \cdot \begin{bmatrix} \tilde{\mathbf{x}} \\ \zeta \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix}$$

From row 2 we get $\zeta = \mathbf{v}^\top \tilde{\mathbf{x}}$. With that from the first row follows $(\mathbf{A} + \mathbf{u}\mathbf{v}^\top)\tilde{\mathbf{x}} = \mathbf{b}$. This shows that solving for $\tilde{\mathbf{x}}$ would solve our rank-1 modified LSE.

When solving row 1 directly by itself, we get $\tilde{\mathbf{x}} = \mathbf{A}^{-1}(\mathbf{b} - \zeta\mathbf{u})$. Inserting that into the equation for row 2 yields $\zeta = \mathbf{v}^\top \mathbf{A}^{-1}(\mathbf{b} - \zeta\mathbf{u})$. Solving for ζ leads to $\zeta = \frac{\mathbf{v}^\top \mathbf{A}^{-1}\mathbf{b}}{1+\mathbf{v}^\top \mathbf{A}^{-1}\mathbf{u}}$. Alternatively, this is directly corresponds to the formulas from Block GE.

Then, we insert the solution for ζ into the equation for $\tilde{\mathbf{x}}$ or use the Block GE formulas to get:

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

This can then be solved by only solving for systems with $\mathbf{A}$, i.e., to compute $\mathbf{A}^{-1}b$ and $\mathbf{A}^{-1}u$.

Applying this method involves checking that the denominator is $\neq 0$ (in a numerically stable way!). That corresponds to near singular systems.

```
template <class LUDec>
Eigen::VectorXd smw(const LUDec &lu, const Eigen:VectorXd &u,
    const Eigen::VectorXd &v, const Eigen::VectorXd &b) {
  const Eigen::VectorXd z = lu.solve(b); // z = A^-1 b, O(n^2)
  const Eigen::VectorXd w = lu.solve(u); // w = A^-1 u, O(n^2)
  double alpha = 1.0 + v.dot(w);
  double beta = v.dot(z);
  if (std::abs(alpha) < std::numeric_limits<double>::epsilon() *
      std::abs(beta))
    throw std::runtime_error("A nearly singular");
  else
    return (z - w * beta / alpha);
}
```

If we can easily solve systems with $\mathbf{A}$, i.e., in $\mathcal{O}(n^2)$ through LU, this has runtime only $\mathcal{O}(n^2)$. But we pay a price with worse numeric stability.

Low Rank Modification We can generalize this approach to higher rank modifications.

Theorem Sherman-Morrison-Woodbury Formula: For regular $A \in \mathbb{K}^{n,n}$, and $U, V \in \mathbb{K}^{n,k}, n, k \in \mathbb{N}, k \leq n$, holds:

$$(A + UV^H)^{-1} = A^{-1} - A^{-1}U(I + V^H A^{-1}U)^{-1}V^H A^{-1}$$

If $I + V^H A^{-1}U$ is regular. UV^H is a general rank k matrix.

3.3 Sparse Linear Systems

With general LSE solve methods we consider all matrices. This means we consider dense matrices in which all entries are relevant/contain information. But in many cases we know, for instance,

that certain elements will be zero. In certain such cases we can exploit this special structure to do certain computations faster.

> **Example**:
> Simulating circuits requires modeling them as matrices. The number of non-zero entries is proportional to the nodes, which the matrix size is quadratic in the number of nodes. Thus, most entries will be zero for large matrices.
> Large real-world graphs/networks often also exhibit a lot of zeros.

3.3.1 Sparse Matrix Storage Formats

First and foremost, if most entries are known to be zero, it is wasteful to store all those zeros. Thus, we have special storage formats for sparse matrices to avoid such wasteful use of memory. We have multiple goals:

- Memory use should be proportional to the number of non-zero entries instead of the number of total entries.
- The cost of a matrix vector multiplication should be linear in the number of non-zero entries instead of $\mathcal{O}(nm)$.

A common side benefit is that we can easily obtain information about the location of non-zero entries.

COO/triplet format We store $\mathbf{A}$ as a list of tuples $(i, j, value)$. We allow i, j pairs to be repeated so that multiple values sum up to the final value for some position in the matrix: $(A)_{i,j} = \sum_{k, A[k].i=i, A[k].j=j} A[k].a$. Adding matrices then becomes very easy. It is just a matter of concatenation. Multiplication is not much more difficult - compare the below C++ implementation.

```
struct Triplet {
  size_t i;
  size_t j;
  scalar_t a;
}
using TripletMatrix = std::vector<Triplet>;

void multTriplMatvec(const TripletMatrix &A, const vector vector
    <scalar_t> &x, vector<scalar_t> &y) {
  for (size_t i=0; i<A.size(); i++) {
    y[A[i].i] += A[i].a*x[A[i].j];
  }
}
```

COO is still quite wasteful, because we store indices for every element and may even store indices repeatedly.

Compressed Row Storage (CRS) CRS is a bit more sophisticated (but also more complex). Notice that we also distinguish between row and column majority here. As Eigen is column major, we will consider the column major version. Then, we would name this paragraph Compressed Column Store to be more accurate.

We store three vectors here :

- `std::vector<scalar_t> val` of size #non-zero-entries of $\mathbf{A}$.

  It contains the non-zero values of the matrix in column-major order. This vector stores the actual data while the other two are used for specifying their position in the matrix.

- `std::vector<size_t> row_ind` of size #non-zero-entries of $\mathbf{A}$.

  For each index k, this stores the row of the entry in the 'val' vector.

- `std::vector<size_t> col_ptr` of size $n + 1$.

  `row_ptr[i]` stores the position/index of the first non-zero entry of row i of the matrix in the vector val. Additionally, we have the sentinel value `row_ptr[n+1]=non-zero-etnries of` $\mathbf{A}$`+1` for computational reasons.

$$val[k] = a_{ij} \Leftrightarrow \begin{cases} row\_ind[k] = i, \\ col\_ptr[j] \leq k \leq col\_ptr[j+1], \end{cases} \qquad 1 \leq k \leq nnz(A)$$

If some columns only contain zeroes, their `col_ptr` entries just point to the first element of the next column with non-zero entries. Then, at least two `col_ptr` entries point to that element.

3.3.2 Sparse Matrices in Eigen

Eigen relies mostly on the Compressed Column Storage (CCS) format. We use `Eigen::SparseMatrix` for CCS:

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

The difficult part is initialization. Just setting entries in random order and with an unknown number of entries leads to bad performance, because the data structure needs to be resizes multiple times involving massive data movement.

Listing 12: SparseMatrix Adding One-By-One without Reservation

```
unsigned int rows, cols;
SparseMatrix<double, rowMajor> mat(rows,cols);
// do many (incremental) initializations
for( /* ... */) {
  // insert or modify, may involve relocation/resizing
  mat.insert(i,j) = value_ij;
  mat.coeffRef(i,j) += increment_ij;
}
mat.makeCompressed(); // reduce data structure size overhead
```

Of all option, the just given trivial version is probably the worse. The two following approaches have better runtime performance - specifically $\mathcal{O}(n)$. But the latter has even better constants than the former.

Intermediate COO/triplet format Alternatively, we can first create a vector of triplets and then initialize the sparse matrix based on those triplets. This has runtime $\mathcal{O}(n)$ with n being the number of triplets.

Listing 13: SparseMatrix Initialization from Triplets

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

SparseMatrix Adding One-By-One with Reservation Here, we reserve space for a certain number of entries per row when creating the sparse matrix data structure. Then, we can insert and modify elements with constant cost.

Listing 14: SparseMatrix Adding One-By-One with Reservation

```
unsigned int rows, cols, max_no_nnz_per_column;
SparseMatrix<double, rowMajor> mat(rows,cols);
mat.reserve(RowVectorXi::Constant(cols,max_no_nnz_per_row)); //
    allocate enough space here
for( /* ... */) {
  // O(1) each if enough space reserved
  mat.insert(i,j) = value_ij;
  mat.coeffRef(i,j) += increment_ij;
}
mat.makeCompressed(); // reduce data structure size overhead
```

3.3.3 Direct Solution of Sparse Linear Systems of Equations

When working with sparse matrices, always use the appropriate data structures. Otherwise, library functions don't know that we work with sparse matrices or would struggle to identify zero elements - as being zero is measured by being almost zero to have numerically stable programs.

In this section we won't dive into the details of the algorithms but only see how to apply them. This is as solving sparse linear systems is an active field of research and the algorithms are very complex. But one should be able to use the algorithms and know about their performance (tradeoffs).

The Eigen library provides a `SparseLU` solver. This can be used just as the regular LU-Decomposition in the two steps of factorization and elimination. But we must check whether the solve was successful. It may be that we have a singular matrix or an obscure matrix structure prevents the solver in Eigen to function properly. We can't specify a runtime, because the performance heavily depends on the matrix structure and the algorithm implemented by the library.

Listing 15: Sparse LU

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

Solving Sparse Systems - Performance Tradeoffs We have seen the arrow matrix above. Using that sparse matrix, this demonstrates the performance difference of various approaches:

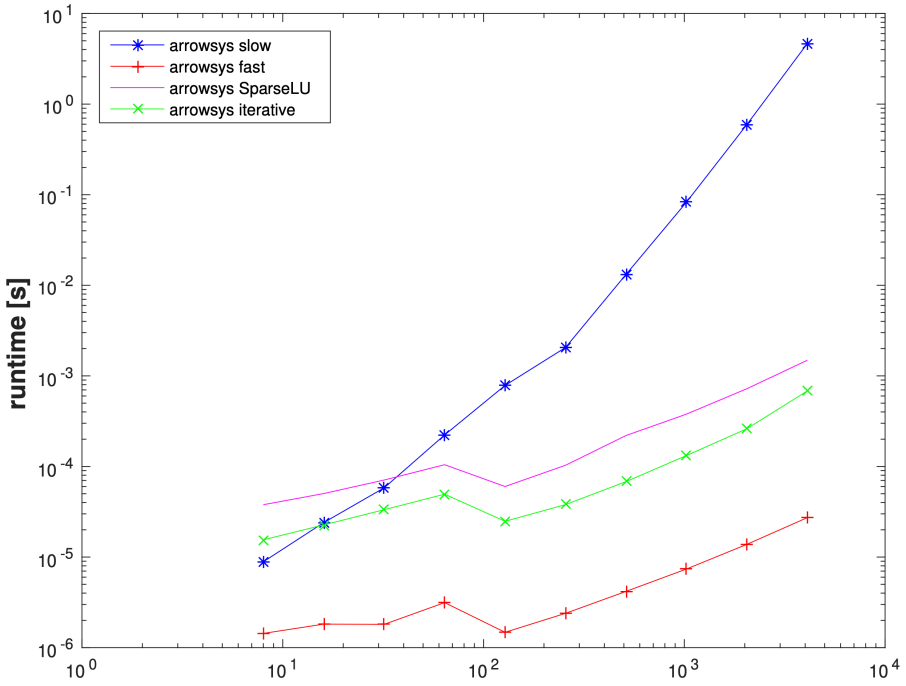


Figure 5

We see that going from the slow to the SparseLU approach leads to significant performance improvements. This exploits the basic knowledge about some sparse structure. Then, through further optimizations based on knowledge about the specific matrix structure, we can improve the performance even further.

In many cases, we choose the SparseLU approach over specialized versions, because of the usually higher numerical stability in library implementations.

While the runtimes of sparse solvers depends on the specific matrix formats, they usually perform better than dense solvers (regarding runtime). The cost of solving a sparse LSE $\mathbf{Ax} = \mathbf{b}$ is $\mathcal{O}(n^\alpha)$ in most cases, where n is the number of non-zero matrix entires and $\alpha \in [1.5, 2.5]$ approximately.

Different algorithms applied to sparse symmetric positive definite matrices reveal an average runtime of $\mathcal{O}(n^{1.5})$. Matrices, which have a less common/clear structure show worse performance when applying sparse solvers.

4 Direct Methods for Linear Least Squares Problems

Linear Least Squares Problems have already been introduced in Linear Algebra. They are of great relevance in many applications such as machine learning, imaging science, data science, etc. We will discuss the theory and numerical approaches to solving them here.

Linear Least Squares Problems arise from the desire to solve overdetermined LSEs. Meaning, LSEs which have more linearly independent equations than unknowns. The system matrices of overdetermined LSEs are tall, i.e., $\mathbf{A} \in \mathbb{R}^{m,n}$ with $m > n$. We must generalize the concept of solving LSEs to also consider such cases.

Examples of Overdetermined Linear Systems of Equations - Motivation .

Example Linear Parameter Estimation in 1D: We propose a physical law $y = \alpha x + \beta$ but don't know α nor β. Thus, we perform measurements $(x_i, y_i) \in \mathbb{R}^2, i = 1, ..., n$. Our intuition tells us: We want to perform as many measurements as possible ($n \gg 2$) and get the average of our measurements to reduce the error. Solving the Linear Least Squares Problem corresponds to finding the solution for α and β that best fits all our measurements. In this case, the system would look like this:

$$\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_m & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

The principle we see from this initial example is: One cannot afford not to use all available information.

Example Linear Regression - Linear Parameter Estimation:
This is a generalization of the before described parameter estimation to multiple dimensions. Our proposed law now is $y = \mathbf{a}^\top \mathbf{x} + \beta$. We want to identify $\mathbf{a}$ and β based on our observations $(x_i, y_i) \in \mathbb{R}^n \times \mathbb{R}, i = 1, ..., n$. If $m > n + 1$ and linearly independent measurements, we have an overdetermined LSE with equations $y_i = \mathbf{a}^\top \mathbf{x}_i + \beta$, which leads to this system:

$$\begin{bmatrix} \mathbf{x}_1^\top & 1 \\ \vdots & \vdots \\ \mathbf{x}_m^\top & 1 \end{bmatrix} \begin{bmatrix} \mathbf{a} \\ \beta \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}$$

Example triangles: Consider measuring the three angles of a triangle. With entirely accurate measurements, this system has a solution:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} \tilde{\alpha} \\ \tilde{\beta} \\ \tilde{\gamma} \\ \pi \end{bmatrix}$$

But through measurement errors we likely won't be able to find a precise solution. But then, we can reduce the variance of our measurements by applying a Least Squares Method to find a solution minimizing the error across all four penalized properties.

4.1 Least Squares Solutions Concepts

Definition Least Squares Solution: For given $\mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{b} \in \mathbb{R}^m$ the vector $\mathbf{x} \in \mathbb{R}^n$ is a least squares solution of the linear system of equations $Ax = b$, if

$$x \in \text{argmin}_{y \in \mathbb{R}^n} \|Ay - b\|_2^2 =: lsq(A, b)$$

$$\Leftrightarrow \|Ax - b\|_2^2 = \min_{y \in \mathbb{R}^n} \|Ay - b\|_2^2 = \min_{y_1,\dots,y_n \in \mathbb{R}} \sum_{i=1}^{m} \left(\sum_{j=1}^{n} (A)_{i,j} y_j - (b)_i \right)^2$$

Nice to know: We consider the square of the norm simply to not have the root, which is a hassle to work with. We don't hesitate to do so, because taking the square of all values doesn't change the minima.

This is a true generalization of solving invertible LSEs. We know that those have exactly one solution. Thus, that solution is the only one which makes the squared norm 0. We get $lsq(\mathbf{A}, \mathbf{b}) = \{\mathbf{A}^{-1}\mathbf{b}\}$ with $\mathbf{A} \in \mathbb{R}^{m,m}$ being invertible.

Geometric Interpretation Consider $\mathcal{R}(\mathbf{A}) \subset \mathbb{R}^m = \{\mathbf{Ax}, \mathbf{x} \in \mathbb{R}^n\}$, which is a hyperplane and the range of the linear projection $\mathbf{A}$. The vector $\mathbf{b} \in \mathbb{R}^m$ may not lie on that hyperplane but somewhere else in $\mathbb{R}^m$. So, there is no solution to the overdetermined system.

But we can draw the residual vector. It is the orthogonal projection $\mathbf{x}^*$ of $\mathbf{b}$ onto the hyperplan $\mathcal{R}(A)$, which corresponds to the smallest deviation from $\mathbf{b}$ of solutions $\mathbf{x}$. Although visualization usually show the projection in 3D onto a plane, don't forget that the dimensional difference can be higher so that multiple best projections exist. The solution is not guaranteed to be unique!

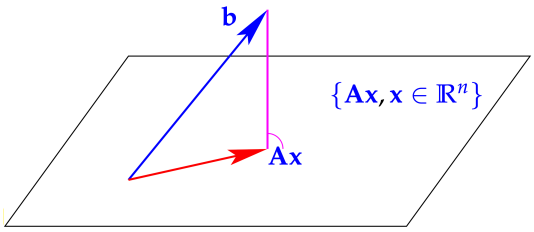


Figure 6

From this interpretation we can see that such a projection must always exist. Hence, the minimum also exists and we can define least squares solutions which always exist via the minimum and not the infimum.

4.1.1 Normal Equations

Here we learn how to recover least squares solutions by using a square system matrix, whose solutions are the least square solutions.

Remember that the residual is orthogonal to $\mathcal{R}(\mathbf{A})$: $\mathbf{r} := \mathbf{b} - \mathbf{Ax}^* \perp \mathcal{R}(\mathbf{A})$. By applying the mathematical definition of orthogonality, we get $(\mathbf{Ay})^\top(\mathbf{b} - \mathbf{Ax}^*) = 0, \forall \mathbf{y} \in \mathbb{R}^n$. Because this does hold for all $\mathbf{y}$, we can also write:

$$\mathbf{A}^\top(\mathbf{b} - \mathbf{Ax}^*) = 0$$

Theorem Least Squares Solutions from Normal Equations: The vector $\mathbf{x} \in \mathbb{R}^n$ is a least squares solution of the linear system of equations $\mathbf{Ax} = \mathbf{b}, \mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{b} \in \mathbb{R}^m$, if and only if it solves the normal equations (NEQ)

$$\mathbf{A}^\top \mathbf{Ax} = \mathbf{A}^\top \mathbf{b}$$

The system matrix $\mathbf{A}^\top \mathbf{A}$ is symmetric.

The normal equation system collapses to a small $n \times n$ system, which exhibits a unique solution if invertible or otherwise all possible solutions.

"Formal" Derivation using Analysis We define $J : \mathbb{R}^n \to \mathbb{R}$ with $J(\mathbf{y}) := \|\mathbf{b} - \mathbf{Ay}\|_2^2$ and get

$$\begin{aligned} J(\mathbf{y}) &= \mathbf{y}^\top \mathbf{A}^\top \mathbf{Ay} - 2\mathbf{b}^\top \mathbf{Ay} + \mathbf{b}^\top \mathbf{b} \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} (\mathbf{A}^\top \mathbf{A})_{ij} \mathbf{y}_i \mathbf{y}_j - 2\sum_{i=1}^{n}\sum_{j=1}^{n} \mathbf{b}_i (\mathbf{A})_{ij} \mathbf{y}_j + \sum_{i=1}^{m} \mathbf{b}_i^2 \end{aligned}$$

This is a multi-variate polynomial in $\mathbf{y}$, i.e., J is smooth. So if we now consider some $\mathbf{x}^* \in \mathbb{R}^n$ with $\mathbf{x}^* \in \operatorname{argmin}_{\mathbf{x} \in \mathbb{R}^n} J(\mathbf{x})$, we see that $\mathbf{x}^*$ is a minima of J. Hence, the gradient must be zero: $\nabla J(\mathbf{x}^*) = 0$. Through partial differentiation:

$$\nabla J(\mathbf{x}^*) = \left[\frac{\partial J}{\partial \mathbf{y}_i}(\mathbf{y})\right]_{i=1}^{n} = 2\mathbf{A}^\top \mathbf{Ay} - 2\mathbf{A}^\top \mathbf{b}$$

Notice: This only shows that the Normal Equations are a necessary condition. That the condition is sufficient must be shown separately.

Uniqueness of LSQ Solutions .

Theorem Kernel and Range of $\mathbf{A}^\top \mathbf{A}$: For $\mathbf{A} \in \mathbb{R}^{m,n}, m \geq n$ holds

$$\begin{aligned} \mathcal{N}(\mathbf{A}^\top \mathbf{A}) &= \mathcal{N}(\mathbf{A}) \\ \mathcal{R}(\mathbf{A}^\top \mathbf{A}) &= \mathcal{R}(\mathbf{A}^\top) \end{aligned}$$

Proof. First equality:

- $\mathbf{z} \in \mathcal{N}(\mathbf{A}) \Rightarrow \mathbf{z} \in \mathcal{N}(\mathbf{A}^\top \mathbf{A})$
- $\mathbf{z} \in \mathcal{N}(\mathbf{A}^\top \mathbf{A}) \Rightarrow \mathbf{A}^\top \mathbf{Az} = 0 \Rightarrow \mathbf{z}^\top \mathbf{A}^\top \mathbf{Az} = 0 \Leftrightarrow \|\mathbf{Az}\|^2 = 0 \Leftrightarrow \mathbf{Az} = 0 \Leftrightarrow \mathbf{z} \in \mathcal{N}(\mathbf{A})$

The proof of the second equality is analogously. □

Corollary Uniqueness of Least Squares Solutions: If $m \geq n$ and $\mathcal{N}(\mathbf{A}) = \{0\}$, then the linear system of equations $\mathbf{Ax} = \mathbf{b}, \mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{b} \in \mathbb{R}^m$ has a unique least squares solution $\mathbf{x} = (\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top \mathbf{b}$.

If $\mathcal{N}(\mathbf{A}) \neq \{0\}$, then, we have infinitely many solutions, because there are linearly dependent equations so that the system is actually underdetermined.

The corollary easily follows from the fact that if $\mathcal{N}(\mathbf{A}) = \{0\}$ we also have $\mathcal{N}(\mathbf{A}^\top\mathbf{A}) = \{0\}$ from the earlier theorem. Then, $\mathbf{A}^\top\mathbf{A}$ is regular and its LSE always has a unique solution.

The condition for the uniqueness of the Least Squares Solution is $\mathcal{N}(\mathbf{A}) = \{0\}$, which is equivalent to $\mathbf{A}$ having full rank, i.e., $rank(\mathbf{A}) = n$.

4.1.2 Moore-Penrose Pseudoinverse

If the full-rank condition is not satisfied ($\mathcal{N}(\mathbf{A}) \neq \{0\}$), we cannot expect a unique solution to a least squares problem. We now specify the minimal norm condition to always get one unique solution of those many solutions.

Definition Generalized Solution of a Linear System of Equations: The generalized solution $\mathbf{x}^\dagger \in \mathbb{R}^n$ of a linear system of equations $\mathbf{Ax} = \mathbf{b}, \mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{b} \in \mathbb{R}^m$ is defined as

$$\mathbf{x}^\dagger := \text{argmin}\{\|\mathbf{x}\|_2 \mid \mathbf{x} \in lsq(\mathbf{A}, \mathbf{b})\}$$

$lsq(\mathbf{A}, \mathbf{b}) = \{\mathbf{x} \in \mathbb{R}^n | \mathbf{A}^\top\mathbf{Ax} = \mathbf{A}^\top\mathbf{b}\}$ is an affine subspace of $\mathbb{R}^n$, which can be expressed as $lsq(\mathbf{A}, \mathbf{b}) = \mathbf{x}_0 + \mathcal{N}(\mathbf{A}^\top\mathbf{A}) = \mathbf{x}_0 + \mathcal{N}(\mathbf{A})$ with $\mathbf{x}_0 \in lsq(\mathbf{A}, \mathbf{b})$.

So, the affine subspace of $lsq(\mathbf{A}, \mathbf{b})$ is parallel to $\mathcal{N}(\mathbf{A})$. To find the element with minimal norm form $lsq(\mathbf{A}, \mathbf{b})$, we need to find the element, whose vector is orthogonal to $\mathcal{N}(\mathbf{A})$. This is as we know that the shortest distance between a point and a space is always there, where the direct line is orthogonal on the space.

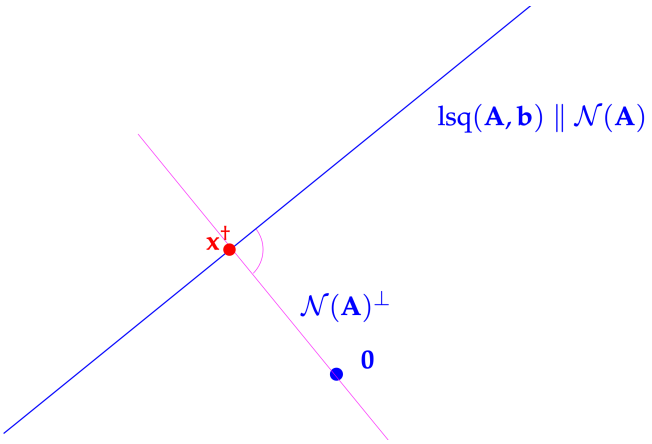


Figure 7

So, for $\mathbf{x}^\dagger$ must hold that $\mathbf{x}^\dagger \perp \mathcal{N}(\mathbf{A}) \Leftrightarrow \mathbf{x}^\dagger \in \mathbb{N}(\mathbf{A})^\perp$.

If $\{\mathbf{v}_1, ..., \mathbf{v}_k\}$ is a basis of $\mathcal{N}(\mathbf{A})^\perp$ (k :)dim$\mathcal{N}(\mathbf{A})$) we see that $\mathbf{x}^\dagger = \mathbf{Vy}$ for some $\mathbf{y} \in \mathbb{R}^k$ where $\mathbf{V} = [\mathbf{v}_1, ..., \mathbf{v}_k] \in \mathbb{R}^{n,k}$.

From this we can extend the normal equations:

$$\mathbf{A}^\top\mathbf{AVy} = \mathbf{A}^\top\mathbf{b}$$
$$\Rightarrow \mathbf{V}^\top\mathbf{A}^\top\mathbf{AVy} = \mathbf{V}^\top\mathbf{A}^\top\mathbf{b}$$

We show that $\mathbf{V}^\top\mathbf{A}^\top\mathbf{AV}$ is always regular. observing that $\mathbf{Vx}$ is not null if $\mathbf{x}$ is not null because $\mathbf{V}$ is regular. But $\mathbf{V}$ spans the complement of the null space of $\mathbf{A}$, so if $\mathbf{x} \neq 0, \mathbf{Vx}$ will

orthogonal to every null vector of $\mathbf{A}$. Hence, $\mathbf{AVx} \neq 0$. Thus, $\mathbf{AVx} = 0$ if and only if $\mathbf{x} = 0$. This implies that $\mathcal{N}(\mathbf{AV}) = \{0\}$ and $\mathbf{AV}$ is regular. This then quickly leads to $\mathbf{V}^\top\mathbf{A}^\top$ being regular too.

When having found a solution for $\mathbf{y}$, $\mathbf{x}^\dagger$ then is given by $\mathbf{Vy}$. Thus, a unique solution for $\mathbf{y}$ exists.

Theorem Moore-Penrose Pseudoinverse - Formula for Generalized Solution: Given $\mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{b} \in \mathbb{R}^m$, the generalized solution $\mathbf{x}^\dagger$ of the linear system of equations $\mathbf{Ax} = \mathbf{b}$ is given by

$$\mathbf{x}^\dagger = \mathbf{V}(\mathbf{V}^\top\mathbf{A}^\top\mathbf{AV})^{-1}(\mathbf{V}^\top\mathbf{A}^\top\mathbf{b})$$

where $\mathbf{V}$ is any matrix whose columns form a basis of $\mathcal{N}(\mathbf{A})^\perp$.
$\mathbf{V}(\mathbf{V}^\top\mathbf{A}^\top\mathbf{AV})^{-1}\mathbf{V}^\top\mathbf{A}^\top$ is called the Moore-Penrose Pseudoinverse of $\mathbf{A}$.

4.2 Normal Equation Method

After having understood the theoretical foundation of Least Squares Problems, we now consider algorithms for solving least squares problems numerically. Specifically, this section will discuss the normal equations $\mathbf{A}^\top\mathbf{Ax} = \mathbf{A}^\top\mathbf{b}$. Here, we also assume that $\mathbf{A}$ satisfies the full-rank condition (FRC). Thus, also $\mathbf{A}^\top\mathbf{A}$ is regular.

Definition Symmetric Positive Definite (s.p.d.) Matrices: $\mathbf{M} \in \mathbb{K}^{n,n}, n \in \mathbb{N}$ is symmetric (Hermitian) positive definite (s.p.d) if (a) it is symmetric/hermitian:

$$\mathbf{M} = \mathbf{M}^H$$

and (b) positive definite

$$\forall \mathbf{x} \in \mathbb{K}^n : \mathbf{x}^H\mathbf{Mx} > 0 \Leftrightarrow \mathbf{x} \neq 0$$

If $\mathbf{x}^H\mathbf{Mx} \geq 0$ for all $\mathbf{x} \in \mathbb{K}^n$, then $\mathbf{M}$ is positive semi definite.

$\mathbf{A}^\top\mathbf{A}$ is also symmetric positive definite. This is of relevance, because such matrices can be treated in a special ways which allows for further optimizations. Specifically, computing the LU-Decomposition of a s.p.d. matrix can be done in Eigen with the function `.ltt()`, which is slightly more efficient that `.lu()`.

Proof. That $\mathbf{C}$ is symmetric is clear from $A^\top A = (A^\top A)^\top$.

And positive definiteness is also easy:

$$\mathbf{x}^\top\mathbf{Cx} = \mathbf{x}^\top\mathbf{A}^\top\mathbf{Ax} = (\mathbf{Ax})^\top\mathbf{Ax} = \|\mathbf{Ax}\|_2^2 > 0 \Leftrightarrow \mathbf{x} \neq 0$$

This last equivalence holds only because we assume the full-rank condition. □

The basic algorithm to solve least squares problems based on the normal equations proceeds in three steps:

1. Compute the regular matrix $\mathbf{C} := \mathbf{A}^\top\mathbf{A} \in \mathbb{R}^{n,n}$.
2. Compute the right hand side vector $\mathbf{c} := \mathbf{A}^\top\mathbf{b}$.
3. Solve the s.p.d. LSE $\mathbf{Cx} = \mathbf{c}$.

Listing 16: Solving Least Squares Problems with Normal Equations

```
VectorXd normeqsolve(const MatrixXd &A, const VectorXd &b) {
  if (b.size() != A.rows())
    throw runtime_error("Dimension mismatch");
  VectorXd x = (A.transpose()*A).llt().solve(A.transpose()*b);
  return x;
}
```

- `A.transpose()*A` is $\mathcal{O}(mn^2)$
- `A.transpose()*b` is $\mathcal{O}(mn)$
- `.llt()/.lu()` is $\mathcal{O}(n^3)$

Thus, the total runtime is $\mathcal{O}(mn^2 + n^3)$. For some fixed n, this is linear in m.

Roundoff Errors Clearly this approach is impacted by roundoff in certain cases. We illustrate this with

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ \delta & 0 \\ 0 & \delta \end{bmatrix} \Rightarrow \mathbf{A}^\top \mathbf{A} = \begin{bmatrix} 1+\delta^2 & 1 \\ 1 & 1+\delta^2 \end{bmatrix}$$

The following shows that $1+\delta^2$ with $\delta \approx \sqrt{EPS}$ becomes 1 in machine arithmetics, because the relative error from 1 is smaller than EPS.

$$1 = (1+\delta^2)(1+\epsilon) \Rightarrow \epsilon = \frac{\delta^2}{1+\delta^2} < EPS$$

Thus, for $\mathbf{A}^\top \mathbf{A}$ we actually compute $\mathbf{A}^\top \mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$ on computers. This is a fatal error, because now $\mathbf{A}^\top \mathbf{A}$ is not regular and we can't determine the least squares solution.

Extended Normal Equations If $\mathbf{A}$ is sparse, then $\mathbf{A}^\top \mathbf{A}$ generally is not sparse. This becomes a problem for large m and n, because the computational expense then is increased noticeably. But there is a trick to avoid computing $\mathbf{A}^\top \mathbf{A}$: Extended normal equations.

Instead of considering the normal equations in their common form $\mathbf{A}^\top \mathbf{A}\mathbf{x} = \mathbf{A}^\top \mathbf{b}$, we take a different but equivalent form: $\mathbf{A}^\top(\mathbf{b} - \mathbf{A}\mathbf{x}) = 0$. This corresponds to two linear equations:

- $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$
- $\mathbf{A}^\top \mathbf{r} = 0$

We may express those in a LSE:

$$\begin{bmatrix} \mathbf{I} & \mathbf{A} \\ \mathbf{A}^\top & 0 \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix}$$

The benefit is that if $\mathbf{A}$ is sparse, so is the new system matrix. However, the cost is the size: $\mathbb{R}^{m+n,m+n}$. But if this is sparse it may still be noticeably faster than the unextended approach.

4.3 Orthogonal Transformation Methods (QR-Decomposition)

In this section we consider another approach to solving least squares problems. Specifically, we will use the QR-Decomposition (introduced below) to perform the solve. But the general setting is as before: We consider an overdetermined LSE $\mathbf{Ax} = \mathbf{b}$ and assume the FRC for $\mathbf{A}$ so that the least squares solution is unique.

Orthogonal Unitary Matrices Knowing about orthogonal and unitary matrix is the foundation for the introduction of the QR-Decomposition below.

Definition Unary and Orthogonal Matrices:

- $\mathbf{Q} \in \mathbb{K}^{n,n}, n \in \mathbb{N}$, is unary if $\mathbf{Q}^{-1} = \mathbf{Q}^H$
- $\mathbf{Q} \in \mathbb{R}^{n,n}, n \in \mathbb{N}$, is orthogonal if $\mathbf{Q}^{-1} = \mathbf{Q}^\top$

Equivalent definition: Orthogonality/Unitarity of a matrix is equivalent to norm preservation.

$$\|\mathbf{Qx}\|_2 = \|\mathbf{x}\|_2, \forall \mathbf{x} \in \mathbb{R}^n$$

The original definition of $\mathbf{T}$ being orthogonal (unitary not mentioned every time from now on) corresponds to $\mathbf{x}^\top \mathbf{T}^\top \mathbf{Tx} = \mathbf{x}^\top \mathbf{y}$. The following shows that this is equivalent to $\|\mathbf{Ty}\|_2 = \|\mathbf{y}\|_2, \forall \mathbf{y} \in \mathbb{R}^n$. For this, we only use the polarization identity: $\mathbf{x}^\top \mathbf{y} = \frac{1}{2}(\|\mathbf{x}+\mathbf{y}\|_2^2 - \|\mathbf{x}\|_2^2 - \|\mathbf{y}\|_2^2)$.

$$(\mathbf{Tx})^\top (\mathbf{Tx}) = \frac{1}{2}(\|\mathbf{T}(\mathbf{x}+\mathbf{y})\|_2^2 - \|\mathbf{Tx}\|_2^2 - \|\mathbf{Ty}\|_2^2) = \frac{1}{2}(\|\mathbf{x}+\mathbf{y}\|_2^2 - \|\mathbf{x}\|_2^2 - \|\mathbf{y}\|_2^2) = \mathbf{x}^\top \mathbf{y}$$

Thus, the equality holds if and only if $\mathbf{x}^\top \mathbf{T}^\top \mathbf{Tx} = \mathbf{x}^\top \mathbf{y}$.

Equivalent definition: The columns of an orthogonal/unitary matrix form an orthonormal basis of $\mathbb{R}^n$.

4.3.1 Transformation Idea

Remember Gaussian Elimination. The idea behind GE is to transform the LSE so that one gets a LSE which is easier to solve but equivalent regarding the solution(s). We want to do the same with least squares problems, i.e., transform $\mathbf{Ax} = \mathbf{b}$ to $\tilde{\mathbf{A}}\mathbf{x} = \tilde{\mathbf{b}}$ so that (a) the least squares solution is easy to compute and (b) the solution remains unchanged, i.e., $lsq(\mathbf{A}, \mathbf{b}) = lsq(\tilde{\mathbf{A}}, \tilde{\mathbf{b}})$.

First, we see that the desired form is also a triangular form. We want to transform $\mathbf{A}$ into upper triangular form.

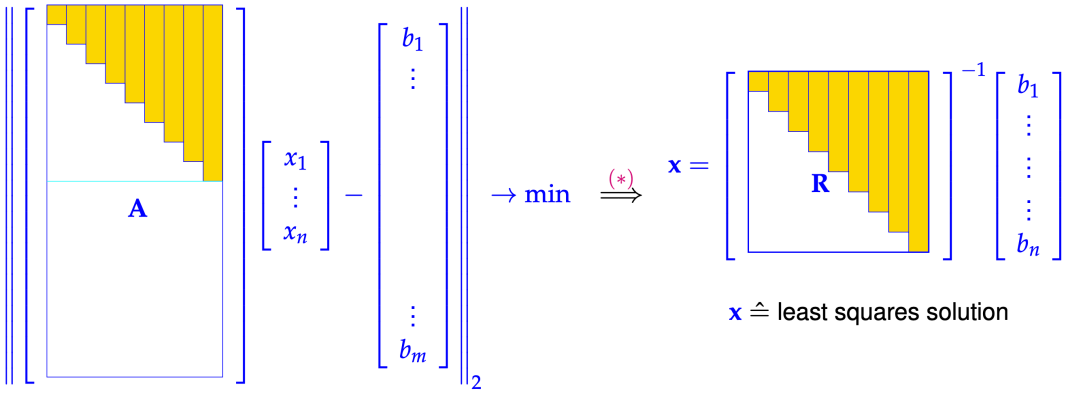


Figure 8

Potential solutions for $\mathbf{x}$ will be measured based on the norm of the difference on the left side of the figure ($\|\mathbf{Ax} - \mathbf{b}\|_2$). But independent of the choice of $\mathbf{x}$, the lower $m - n$ components will remain unchanged. Thus, we must optimize $\mathbf{x}$ so that the upper n components are minimal.

If we call the upper triangular rectangular submatrix of $\mathbf{A}$ matrix $\mathbf{R}$, we have to solve the LSE $\mathbf{Rx} = \mathbf{b}_{1:n}$. Then, the upper n components are zero (the optimum) and the lower $m - n$ can't be changed so that we have found an x which minimizes the norm of the given difference.

After having understood why upper triangular form makes solving the least squares problem easy (it is reduced to a simple elimination in $\mathcal{O}(n^2)$) we are interested to find out how to transform a given $\mathbf{A}$ and $\mathbf{b}$ into such easy to solve form.

While deriving the normal equations, we have seen that those minimize the residual vector - specifically the norm of the residual vector. Thus, any transformations which leave the norms unchanged, also maintain that the solution $\mathbf{x}$ has the smallest residual vector. This means that our transformations must leave the norm $\|\cdot\|_2$ unchanged.

Now we remember orthogonal and unitary transformations from above. Those are characterized by preserving the norm. Thus, we want to perform transformations using orthogonal matrices to transform $\mathbf{A}$ into upper triangular form. In the next sections we see how this can be done.

If we have a transformation matrix $\mathbf{T} \in \mathbb{R}^{m,m}$ satisfying $\|\mathbf{Ty}\|_2 = \|\mathbf{y}\|_2$ for all $\mathbf{y} \in \mathbb{R}^m$, then:

$$lsq(\mathbf{A}, \mathbf{b}) = \text{argmin}_{\mathbf{y}\in\mathbb{R}^n} \|\mathbf{Ay} - \mathbf{b}\|_2 = \text{argmin}_{\mathbf{y}\in\mathbb{R}^n} \|\tilde{\mathbf{A}}\mathbf{y} - \tilde{\mathbf{b}}\|_2 = lsq(\tilde{\mathbf{A}}, \tilde{\mathbf{b}})$$

4.3.2 QR-Decomposition

When having done GE, we collected the elements of $\mathbf{L}$ throughout the process and $\mathbf{U}$ as the result of transformations. So, GE provided the LU-Decomposition as a side effect. Similarly, we may obtain the QR-Decomposition when performing the correct orthogonal transformations. We will now consider how that can be done.

The basic algorithm from Linear Algebra which provides the QR-Decomposition is the Gram-Schmidt algorithm, which orthonormalizes linearly independent vectors $\{\mathbf{a}^1, \ldots, \mathbf{a}^2\} \subset \mathbb{R}^m, m \geq n$.

Listing 17: Gram Schmidt Algorithm Pseudocode

```
1 q^1 := a^1 / ||a^1||_2
```

```
for j = 2, ..., n do
  {
    q^j := a^j
    for l = 1, 2, ..., j − 1 do
      { q^j = q^j − ⟨a^j, q^l⟩q^l; }
    if (q^j = 0) then STOP
    else {q^j = q^j / ||q^j||_2;}
  }
```

Theorem Span Property of Gram Schmidt Vectors: If $\{\mathbf{a}^1, ..., \mathbf{a}^n\} \subset \mathbb{R}^m$ is linearly independent, then the Gram Schmidt Algorithm computes orthonormal vectors $\mathbf{q}^1, ..., \mathbf{q}^n \in \mathbb{R}^m$ satisfying

$$\text{span}\{\mathbf{q}^1, ..., \mathbf{q}^l\} = \text{span}\{\mathbf{a}^1, ..., \mathbf{a}^l\}$$

for all $l \in \{1, ..., n\}$.

Orthonormal means: $(\mathbf{q}^i)^\top \mathbf{q}^j = \delta_{ij}$. So the vectors are pairwise orthogonal and have norm 1. We can collect the n computed vectors $\mathbf{q}^1, \ldots, \mathbf{q}^n$ and the n vectors $\mathbf{a}^1, \ldots, \mathbf{a}^n$ from which we construct two matrices:

$$\mathbf{Q} := \begin{bmatrix} \mathbf{q}^1 & \ldots & \mathbf{q}^n \end{bmatrix} \in \mathbb{R}^{m,n}$$
$$\mathbf{A} := \begin{bmatrix} \mathbf{a}^1 & \ldots & \mathbf{a}^n \end{bmatrix} \in \mathbb{R}^{m,n}$$

As $\mathbf{Q}$ has orthonormal columns, Q is unitary/orthogonal as introduced before. Hence, we notice that $\mathbf{Q}^\top \mathbf{Q} = \mathbf{I}$.

Generally, we know that any vector $\mathbf{q}^i$ can be written as a linear combination of the column vectors of $\mathbf{A}$, because $\mathbf{Q}$ and $\mathbf{A}$ span the same space. From the Gram Schmidt algorithm follows that to express $\mathbf{q}^l$ we even only need the first l column vectors of $\mathbf{A}$.

$$\mathbf{q}^l = \sum_{i=1}^{l} \mathbf{t}_{il} \mathbf{a}^i \text{ with } l = 1, ..., n$$

- $\mathbf{q}^1 = t_{11}\mathbf{a}^1$
- $\mathbf{q}^2 = t_{12}\mathbf{a}^1 + t_{22}\mathbf{a}^2$
- $\mathbf{q}^3 = t_{13}\mathbf{a}^1 + t_{23}\mathbf{a}^2 + t_{22}\mathbf{a}^3$

  $\vdots$

- $\mathbf{q}^n = t_{1n}\mathbf{a}^1 + t_{2n}\mathbf{a}^2 + \cdots + t_{nn}\mathbf{a}^n$

We may define $\mathbf{T} \in \mathbb{R}^{n,n}$ as $(\mathbf{T})_{ij} := \begin{cases} t_{ij}, & i \leq j \\ 0, & \text{else} \end{cases}$. Then it holds that $\mathbf{Q} = \mathbf{AT}$ and we see that $\mathbf{T}$ is an upper triangular matrix.

Because we assume the full rank condition to hold for $\mathbf{A}$ in this whole section as stated in the beginning, we know that $\text{rank}(\mathbf{A}) = n$. Because $\mathbf{A}$ and $\mathbf{Q}$ span the same space, we have $\text{rank}(\mathbf{Q}) = n$ too. Thus, the transformation $\mathbf{T}$ cannot reduce the number of dimensions and must be regular. Less formally, it can also be seen by intuition that from the linear independence of the vectors and the Gram Schmidt algorithm follows that $t_{ii} \neq 0$ for all valid i.

Lemma Group of (Regular) Diagonal/Triangular Matrices: If $\mathbf{A}, \mathbf{B}$ are diagonal/upper triangular/lower triangular, then: $\mathbf{AB}$ and $\mathbf{A}^{-1}$ (assuming $\mathbf{A}$ is regular) are also diagonal/upper triangular/lower triangular.

With $\mathbf{T}$ being invertible and upper triangular, we see that $\mathbf{A} = \mathbf{QT}^{-1}$ and $\mathbf{T}^{-1}$ is upper triangular. Thus, we have a decomposition of $\mathbf{A}$ as an orthogonal and upper triangular matrix. If we define $\mathbf{R} := \mathbf{T}^{-1}$ we get the economical QR-Decomposition:

$$\mathbf{A} = \mathbf{QR}, \quad \mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{Q} \in \mathbb{R}^{m,n}, \mathbf{R} \in \mathbb{R}^{n,n}$$

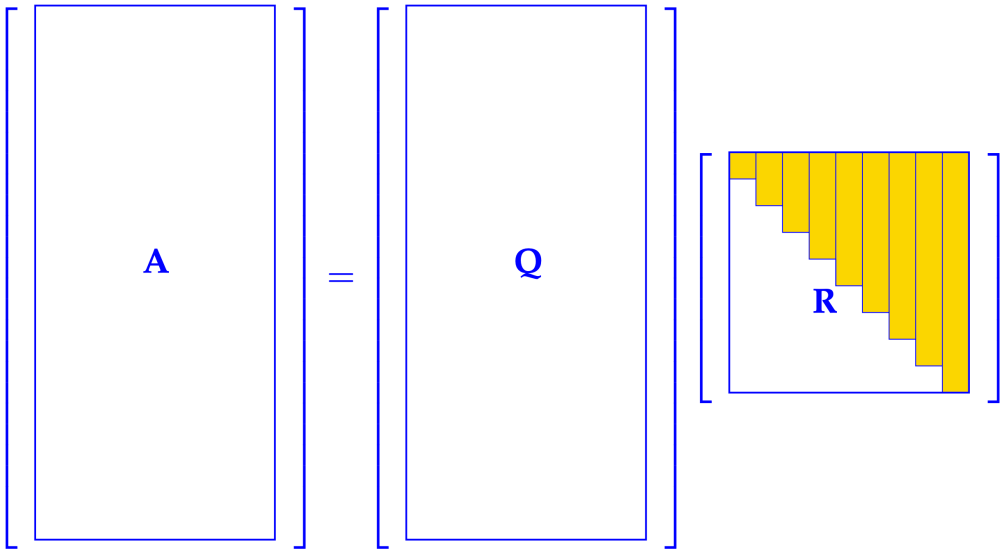


Figure 9

We can extend the $\mathbf{Q}$ matrix with $m - n$ additional orthogonal vectors. If we also extend $\mathbf{R}$ to the bottom with zeros so that it has height m, the equality with $\mathbf{A}$ still holds. Then we have the full QR-Decomposition:

$$\mathbf{A} = \tilde{\mathbf{Q}}\tilde{\mathbf{R}}, \quad \mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{Q} \in \mathbb{R}^{m,m}, \mathbf{R} \in \mathbb{R}^{m,n}$$

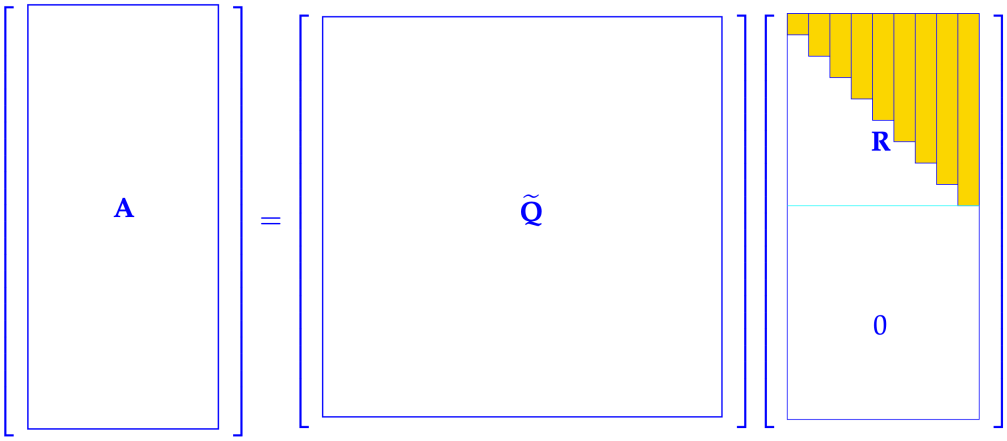


Figure 10

Theorem QR-Decomposition: For any matrix $\mathbf{A} \in \mathbb{K}^{n,k}$ with $\text{rank}(\mathbf{A}) = k$ there exists

- a unique matrix $\mathbf{Q}_0 \in \mathbb{R}^{n,k}$ that satisfies $\mathbf{Q}_0^H \mathbf{Q}_0 = \mathbf{I}_k$, and a unique upper triangular matrix $\mathbf{R}_0 \in \mathbb{K}^{k,k}$ with $(\mathbf{R})_{i,i} > 0, i \in \{1, ..., k\}$ such that

$$\mathbf{A} = \mathbf{Q}_0 \cdot \mathbf{R}_0$$

- a unitary Matrix $\mathbf{Q} \in \mathbb{K}^{n,n}$ and a unique upper triangular matrix $\mathbf{R} \in \mathbb{K}^{n,k}$ with $(\mathbf{R})_{i,i} > 0, i \in \{1, ..., n\}$, such that

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R}$$

If $\mathbb{K} = \mathbb{R}$ all matrices will be real and $\mathbf{Q}$ then is orthogonal. Otherwise, $\mathbf{Q}$ is unitary.

Corollary Uniqueness of QR-Decomposition:
As implicitly stated in the before theorem: The economical QR-Decomposition of $\mathbf{A} \in \mathbb{K}^{m,n}, m \geq n$ with $\text{rank}(\mathbf{A}) = n$ is unique, if we demand $(\mathbf{R}_0)_{ii} > 0$ for all $i = 1, ..., n$.

Proof. If $\mathbf{A}$ has full rank n, then $\mathbf{R}$ must be regular. Remember that the upper triangular matrices form a group under multiplication as stated in above. We will assume that two decompositions exist and see that the two decompositions must then be equal.

$$\mathbf{Q}_1\mathbf{R}_1 = \mathbf{Q}_2\mathbf{R}_2 \Rightarrow \mathbf{Q}_1 = \mathbf{Q}_2\mathbf{R} \text{ with } \mathbf{R} := \mathbf{R}_2\mathbf{R}_1^{-1}$$
$$\mathbf{I} = \mathbf{Q}_1^H\mathbf{Q}_1 = \mathbf{R}^H\mathbf{Q}_2^H\mathbf{Q}_2\mathbf{R} = \mathbf{R}^H\mathbf{R}$$

From some Lemma about the Cholesky Decomposition then follows that $\mathbf{R}$ must be unique. Specifically, $\mathbf{R} = \mathbf{I}$ so that then $\mathbf{R}_2 = \mathbf{R}_1$. □

4.3.3 Computation of QR-Decomposition

In the previous section we used the Gram Schmidt algorithm to derive the QR-Decomposition. But that algorithm is numerically very unstable and introduces very large errors. Thus, we consider some numerically stable approaches here.

The basic idea is to do iterative steps on $\mathbf{A}$ through orthogonal transformations (OTs) which preserve the norm to turn it into an upper triangular matrix. With each OT, we get closer to the desired upper triangular form. In the end, we can combine all OTs to one single orthogonal transformation (justified with following corllary) and get $\mathbf{Q}^*\mathbf{A} = \mathbf{R}$ from which we can form the QR-Decomposition $\mathbf{A} = (\mathbf{Q}^*)^{-1}\mathbf{R}$.

Corollary Composition of Orthogonal Transformations: The product of two orthogonal/unitary matrices of the same size is again orthogonal/unitary.

We will discuss two approaches to get such orthogonal transformations to reach an upper triangular form: Householder reflections and Givens rotations. This uses the realization that orthogonal matrices (as being invertible) correspond to full-rank linear transformations. Furthermore, their specific structure implies that they correspond to rotations, reflections, and arbitrary combinations of those. With Householder reflections, we perform many consecutive reflections and with Givens rotations we perform many consecutive rotations.

Let's consider the 2D case to illustrate what needs to be done. Given some $\mathbf{A} = \begin{bmatrix} a_1 & b_1 \\ a_2 & b_2 \end{bmatrix}$, we want to apply some orthogonal transformation $\mathbf{Q}$ so that we get $\mathbf{Q}\begin{bmatrix} \mathbf{a} & \mathbf{b} \end{bmatrix} = \begin{bmatrix} * & * \\ 0 & * \end{bmatrix}$. If we consider Householder reflections, we need to find a reflection so that $\mathbf{a}$ is reflected onto the x-axis. And if we consider Givens rotations, we need to find a rotation so that $\mathbf{a}$ is rotated onto the x axis.

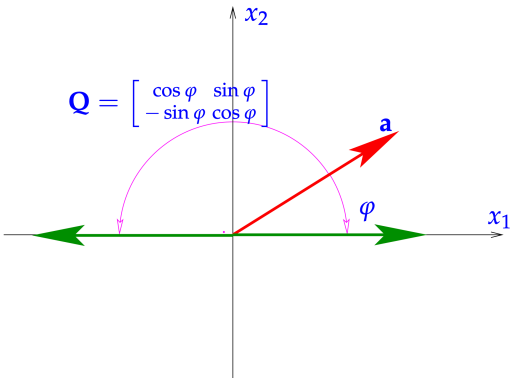


Figure 11: Rotation

If we want $\mathbf{R}$ to be in its unique form with $(\mathbf{R})_{ii} > 0$ for all valid i, we must always map on the x-axis in the direction of the unit vector $\mathbf{e}_1$.

Householder Refelctions This generalizes the approach by using reflections from 2D to arbitrarily many dimensions. If we have some vector $\mathbf{a}$ and find a reflection so that $\mathbf{a}$ is reflected to become $\mathbf{b}$, where $\|\mathbf{a}\|_2 = \|\mathbf{b}\|_2$ and $\mathbf{b}$ is parallel to $\mathbf{e}_1$, we define $\mathbf{v}$ to be orthogonal to the reflection plane. $v := a - b \perp a + b$.

$$\mathbf{b} = \mathbf{a} - \mathbf{v} = \mathbf{a} - \mathbf{v}\frac{\mathbf{v}^\top\mathbf{v}}{\mathbf{v}^\top\mathbf{v}} \overset{(*)}{=} \mathbf{a} - 2\mathbf{v}\frac{\mathbf{v}^\top\mathbf{a}}{\mathbf{v}^\top\mathbf{v}} = (\mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^\top}{\mathbf{v}^\top\mathbf{v}})\mathbf{a} = \mathbf{H}(\mathbf{v})\mathbf{a}$$

Equality $(*)$ is justified through

$$\mathbf{v}^\top\mathbf{v} = (\mathbf{a}-\mathbf{b})^\top(\mathbf{a}-\mathbf{b}) \overset{(\square)}{=} (\mathbf{a}-\mathbf{b})^\top(\mathbf{a}-\mathbf{b}+\mathbf{a}+\mathbf{b}) = 2(\mathbf{a}-\mathbf{b})^\top\mathbf{a} = 2\mathbf{v}^\top\mathbf{a}$$

where we can add $+\mathbf{a} + \mathbf{b}$ in step $(\square)$, because of the orthogonality $(\mathbf{a} - \mathbf{b}) \perp (\mathbf{a} + \mathbf{b}) \Leftrightarrow (\mathbf{a} - \mathbf{b})^\top(\mathbf{a} + \mathbf{b}) = 0$.

To find the vector $\mathbf{v}$ in practice we choose $\mathbf{v} = \mathbf{a} + \text{sign}(\mathbf{a}_1)\|\mathbf{a}\|\mathbf{e}_1$. The factor $\text{sign}(\mathbf{a}_1)$ is used to avoid small vectors $\mathbf{v}$ if $\mathbf{a}$ is almost on the x-axis and has negative $\mathbf{a}_1$ sign, i.e., to avoid cancellation.

$$\mathbf{H}(\mathbf{v})\mathbf{a} = \pm\|\mathbf{a}\|_2\mathbf{e}_1 : \mathbf{H}(\mathbf{v}) \begin{bmatrix} * \\ * \\ \vdots \\ * \end{bmatrix} = \begin{bmatrix} \pm\|\mathbf{a}\|_2 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Notice that this may map on the x-axis with negative or positive sign. But to get the unique form of $\mathbf{R}$ we must map with positive sign. This would require additional work to be numerically stable.

$\mathbf{H}(\mathbf{v})$ is called a Householder matrix. It is orthogonal and symmetric from its construction as a reflection. The symmetry of the matrix follows from the property reflection specifically.

We apply those Householder matrices as orthogonal transformations. For consecutive steps, we choose smaller householder matrices to eliminate continuous columns. Therefore, we fill the diagonal not covered by $\mathbf{H}(\mathbf{v})$ with ones and everything else with zeroes.

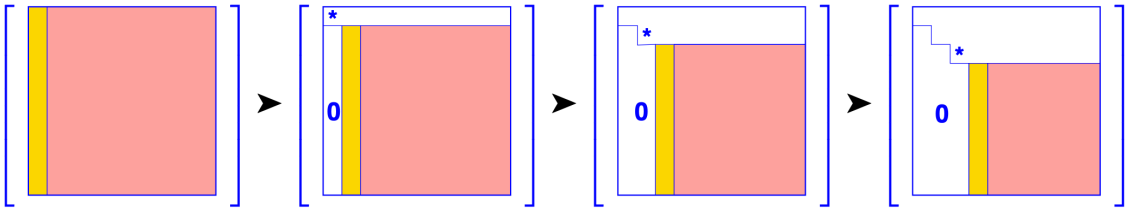


Figure 12

The figure applies some $\mathbf{H}(\mathbf{v})$ in the first step. Then some $\begin{bmatrix} 1 & 0 \\ 0 & \mathbf{H}(\mathbf{v}) \end{bmatrix}$ in the second step, some $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \mathbf{H}(\mathbf{v}) \end{bmatrix}$ in the third step, etc. In total we get $\mathbf{Q}_{n-2}\mathbf{Q}_{n-2}\dots\mathbf{Q}_1\mathbf{A} = \mathbf{R}$ with $\mathbf{A} = \mathbf{QR}$ where $\mathbf{Q} = \mathbf{Q}_1^H\mathbf{Q}_2^H\dots\mathbf{Q}_{n-1}^H$ by using that the inverse of orthogonal matrices is its hermitian.

We will mostly consider overdetermined systems as those are the ones for which we are interested in the least squares solutions. But Householder transformations can be applied to any matrix. Thus, this is suitable to determine the QR-Decomposition of arbitrary matrices.

When computing the QR-Decomposition in practice, we usually don't compute and store $\mathbf{Q}$. Instead, only the normalized $\mathbf{v}$ vectors are computed and stored, because it is much cheaper computationally, and in many cases it is sufficient to know the vectors $\mathbf{v}$ to compute even with $\mathbf{Q}$. When having stored some $\mathbf{v}$ we can just compute $\mathbf{H}(\mathbf{v}) = \mathbf{I} - 2\mathbf{v}\mathbf{v}^\top \in \mathbb{R}^{m,m}, \mathbf{v} \in \mathbb{R}^m, \|\mathbf{v}\|_2 = 1$.

All together, implementations usually compute $\mathbf{R}$ but only encode $\mathbf{Q}$ by its normalized reflection vectors $\mathbf{v}$.

Grivens Rotations Householder matrices implement the idea of using reflections. With Givens rotations we generalize the concept of rotating onto the x-axis to arbitrary dimensions. This is very similar to 2D rotations, because with one Givens rotation we single out two axis and perform a 2D rotation in its plane. With Householder matrices, we have made all elements below one in $\mathbf{A}$ zero. With Givens rotations we may only attack two entries of which we want to zero one. Thus, it may take us $n - 1$ givens rotations to zero all lower elements.

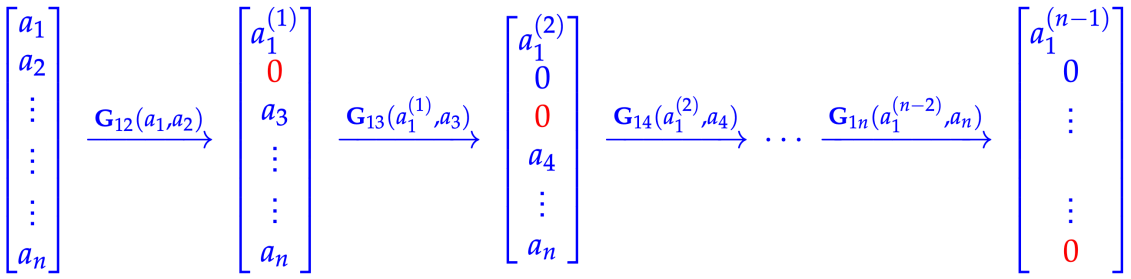


Figure 13

One such Givens rotation is written as $\mathbf{G}_{ij}(a_i, a_j)$, where we want to modify a_i to make a_j zero and a_i/a_j are the corrent elements in the matrix. For $\mathbf{G}_{ik}(a_1, a_k)$, ghe matrix specifically is

$$\begin{bmatrix} \gamma & \dots & \sigma & \dots & 0 \\ \vdots & & \vdots & & \vdots \\ -\sigma & \dots & \gamma & \dots & 0 \\ \vdots & & \vdots & & \vdots \\ 0 & \dots & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} a_1^{(1)} \\ \vdots \\ 0 \\ \vdots \\ a_n \end{bmatrix}$$

$$\gamma = \frac{a_1}{\sqrt{|a_1|^2 + |a_k|^2}} \quad \text{and} \quad \sigma = \frac{a_k}{\sqrt{|a_1|^2 + |a_k|^2}}$$

When we just look at the 2D plane spanned by $\mathbf{e}_1$ and $\mathbf{e}_k$, γ corresponds to $\cos(\mathbf{e}_1 \angle \mathbf{a}) = \cos(\varphi)$ and σ corresponds to $\cos(\mathbf{e}_k \angle \mathbf{a}) = \sin(\mathbf{e}_1 \angle \mathbf{a}) = \sin(\varphi)$. Hence, this is just as the rotation matrix we know from Linear Algebra and the 2D introduction from above.

Because Givens rotation matrices are rotations, they are orthonormal matrices.

To store Givens rotations it is sufficient to store the two indices on which we work and the angle between the vector $\mathbf{a}$ and $\mathbf{e}_i$, which we named φ above. But to make this process numerically stable, we choose a more complex encoding for some $\mathbf{G} = \begin{bmatrix} \gamma & \sigma \\ -\sigma & \gamma \end{bmatrix}$:

$$\text{Encoding: } \rho := \begin{cases} sign(\sigma), & \gamma = 0 \\ \sqrt{2} sign(\gamma)\sigma, & |\sigma| < |\gamma| \\ \frac{1}{2}\sqrt{2} sign(\sigma)/\gamma, & |\sigma| \geq |\gamma| \end{cases}$$

$$\text{Decoding: } \begin{cases} \rho = \pm 1 & \Rightarrow \gamma = 0, \sigma = \pm 1 \\ |\rho| < 1 & \Rightarrow \sigma = \frac{1}{2}\sqrt{2}\rho, \gamma = \sqrt{1-\sigma^2} \\ |\rho| > 1 & \Rightarrow \gamma = \frac{1}{2}\sqrt{2}/\rho, \sigma = \sqrt{1-\gamma^2} \end{cases}$$

This encoding forgets the sign of the matrix $\mathbf{G}$. So the signs of the corresponding rows of the matrix $\mathbf{R}$ must be changed accordingly.

Computational Cost of Householder QR Householder-based QR of some $\mathbf{A} \in \mathbb{R}^{m,n}, m \geq n$ is done by applying various $\mathbf{H}(\mathbf{v})$ for $n-1$ times total. Applying one householder matrix $\mathbf{H}(\mathbf{v})$ to some $\mathbf{X} \in \mathbb{R}^{m,k}$ corresponds to computing $\mathbf{H}(\mathbf{v})\mathbf{X} = \mathbf{X} - 2\mathbf{v}(\mathbf{v}^\top \mathbf{X})$, which can be done in $\mathcal{O}(mk)$. So, applying one householder matrix is proportional to the number of entries in $\mathbf{X}$. While applying householder matrices consecutively, the householder matrices shrink.

$$\sum_{k=1}^{n-1} 2(m-k+1)(n-k) = \mathcal{O}(mn^2)$$

QR-Decompositoin of Banded Matrices Givens rotations are often not chosen in general cases, because we must apply quadratically more transformation matrices. Although the asymptotic cost is also $\mathcal{O}(mn^2)$ as with Householder transformations. However, in certain cases (such as with banded matrices) Givens rotations work very well. This is usually the case (as also with banded matrices), if only a few elements must be zeroed.

With banded matrices, specifically with an upper and lower bandwidth of 1, we only need to apply one Given's rotation per column. Thus, a total of $n - 1 = \mathcal{O}(n)$ transformation matrices suffices.

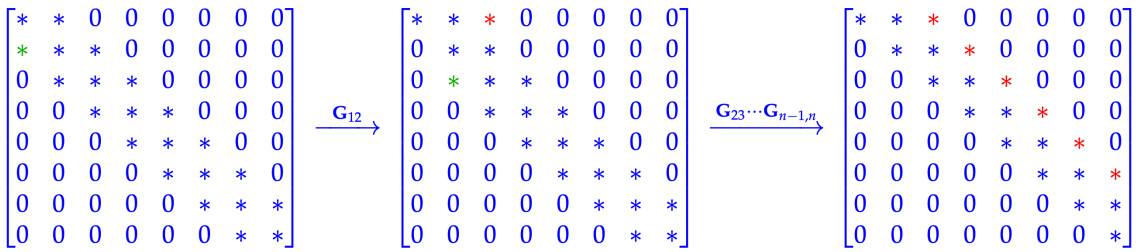


Figure 14

From the structure of the Givens rotation matrix and the structure of the banded matrix follows that we only turn one zero into a non-zero element with each Givens rotation (at most). So the total number of non-zero entries remains constant (or decreases). By visual inspection we see that the resuling matrix after all transformations is still tridiagonal, i.e., the bandwidth remains unchanged. Such tridiagonal matrices are just a special case of general banded matrices for which the bandwidth cannot increase in the resulting $\mathbf{R}$ reached through Givens rotations.

In this case, one Givens rotation only affects two rows of the matrix and, hence, it suggests to have runtime $\mathcal{O}(n)$. But in fact the runtime is $\mathcal{O}(1)$, because we already know most entries to be zero (≤ 3 non-zero entries in every row of the matrix) and mustn't even consider them. So, we have a total cost of $\mathcal{O}(n)$ for computing the QR decomposition.

4.3.4 QR-Decomposition in Eigen

We compute the QR-Decomposition of some matrix $\mathbf{A}$ after importing `#include <Eigen/QR>` with `Eigen::HousholderQR<MatrixXd> qr(A).` We can access $\mathbf{Q}$ through `qr.householderQ()` and $\mathbf{R}$ through `qr.matrixQR().template triangularView<Eigen::Upper>(). .matrixQR()` returns a special matrix which contains $\mathbf{R}$ in the upper triangle and the $\mathbf{v}$ vectors from the Householder matrices below the diagonal.

Listing 18: Full & Economical QR-Decomposition in Eigen

```
#include <Eigen/QR>

std::pair<MatrixXd,MatirxXd> qr_decomp_full(const MatrixXd& A) {
  Eigen::HouseholderQr<MatrixXd> qr(A);
  MatrixXd Q = qr.householderQ();
  MatrixXd R = qr.matrixQR().template triangularView<Eigen::
     Upper>();
  return std::pair<MatrixXd,MatrixXd>(Q,R);
}

std::pair<MatrixXd,MatrixXd> qr_decomp_eco(const MatrixXd& A) {
```

```
11   using index_t = MatrixXd::Index;
12   const index_t m = A.rows(), n = A.cols();
13   Eigen::HouseholderQR<MatrixXd> qr(A);
14   MatrixXd Q = (qr.householderQ()*MatrixXd::Identity(m,n));
15   MatrixXd R = qr.matrixQR().block(0,0,n,n).template
        triangularView<Eigen::Upper>();
16   return std::pair<MatrixXd,MatrixXd>(Q,R);
17 }
```

The function `.householderQ()` returns an expression of the unitary matrix $\mathbf{Q}$ as a sequence of Householder transformations. The benefit of this only returning an expression and not the actual matrix is: All the expensive multiplications are only done if really necessary. But through optimizations, only a few multiplications may actually be performed. For instance, when multiplying with a vector.

Computing the QR-Decomposition runs in $\mathcal{O}(mn^2)$ as explained in the previous section.

4.3.5 QR-Decomposition based Solver for Linear Least Squares Problems

Remember the goal of least squares problems: Finding $\mathbf{x} \in \mathbb{R}^n$ so that $\|\mathbf{Ax} - \mathbf{b}\|_2$ is minimal where $\mathbf{A} \in \mathbb{R}^{m,n}$ and $\mathbf{b} \in \mathbb{R}^m$. Now we assume that the full QR-Decomposition of $\mathbf{A}$ is given, or rather computed with one oft the above outlined methods, and want to understand how to utilize this Decomposition to solve least squares problems.

$$\|\mathbf{Ax} - \mathbf{b}\|_2 = \|\mathbf{QRx} - \mathbf{b}\|_2 = \|\mathbf{Q}(\mathbf{Rx} - \mathbf{Q}^\top \mathbf{b})\|_2 = \|\mathbf{Rx} - \tilde{\mathbf{b}}\|_2, \quad \tilde{\mathbf{b}} := \mathbf{Q}^\top \mathbf{b}$$

If we assume full rank for $\mathbf{A}$, also $\mathbf{R}$ has full rank. Then, the minimization problem looks like:

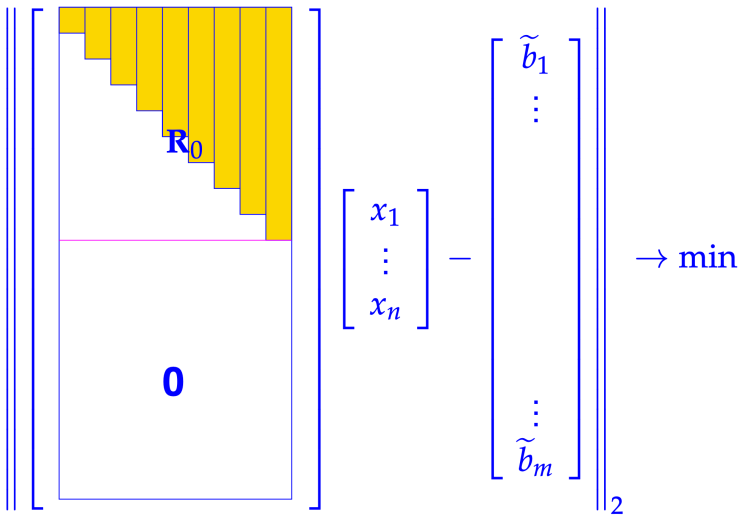


Figure 15

When discussing the use of the QR-Decomposition before, we have seen that now we must only consider the upper n components to identify $\mathbf{x}$ which minimizes this expression. Specifically, we must solve the system $\mathbf{R}_0\mathbf{x} = \tilde{\mathbf{b}}_{1:n}$.

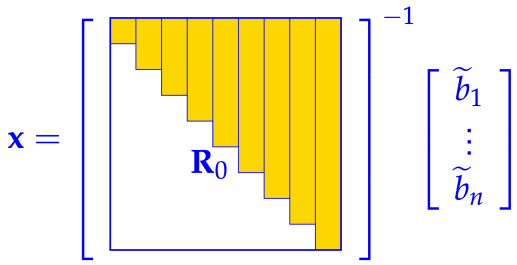


Figure 16

The residual then is $\mathbf{Q}(\mathbf{R}\mathbf{x} - \tilde{\mathbf{b}}) = \mathbf{Q}\left(\begin{bmatrix}\tilde{b}_1\\ \vdots \\ \tilde{b}_n \\ 0 \\ \vdots \\ 0\end{bmatrix} - \tilde{\mathbf{b}}\right) = \mathbf{Q}\begin{bmatrix}0\\ \vdots \\ 0 \\ \tilde{b}_{n+1} \\ \vdots \\ \tilde{b}_m\end{bmatrix}$.

QR-based Linear Least Squares Solving in Eigen After having computed the QR-Decomposition with `.householderQR()`, one can use `.solve(x)` on the resulting object, where `x` is a suitable vector or matrix.

Listing 19: Least Squares Solve in Eigen with QR

```
VectorXd lsqsolve_eigen(const MatrixXd& A, const VectorXd& b,
    VectorXd& x) {
  x = A.householderQR().solve(b);
  return x;
}
```

This is a two-stage approach with a total runtime of $\mathcal{O}(mn^2)$.

1. Compute $\mathbf{R}$ and Householder reflection vectors $\mathbf{v}$ in $\mathcal{O}(mn^2)$.
2. Computing $\mathbf{Q}^\top\mathbf{b}$ can be done in $\mathcal{O}(mn)$ by utilizing the reflection vectors and avoiding to actually compute the Householder matrices.
3. Solving $\mathbf{R}_0\mathbf{x} = \tilde{\mathbf{b}}_{1:n}$ in $\mathcal{O}(n^2)$.

The QR-Decomposition can be reused to solve multiple least squares problems. We only need to repeat stages 2 and 3 then. An additional solve, thus, works in $\mathcal{O}(mn + n^2)$.

4.4 Singular Value Decomposition (SVD)

4.4.1 Definition & Theory

Theorem (Full) Singular Value Decomposition: For any $\mathbf{A} \in \mathbb{K}^{m,n}$ there are unitary/orthogonal matrices $\mathbf{U} \in \mathbb{K}^{m,m}, \mathbf{V} \in \mathbb{K}^{n,n}$ and a (generalized) diagonal matrix $\mathbf{\Sigma} = diag(\sigma_1, ..., \sigma_p) \in \mathbb{R}^{m,n}, p := \min\{m, n\}, \sigma_1 \geq \sigma_2 \geq ... \geq \sigma_p \geq 0$ such that

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H$$

$\sigma_1, ..., \sigma_p$ are the singular values of $\mathbf{A}$.

To my knowledge, the proof of this theorem wasn't considered and, thus, isn't relevant. (?)

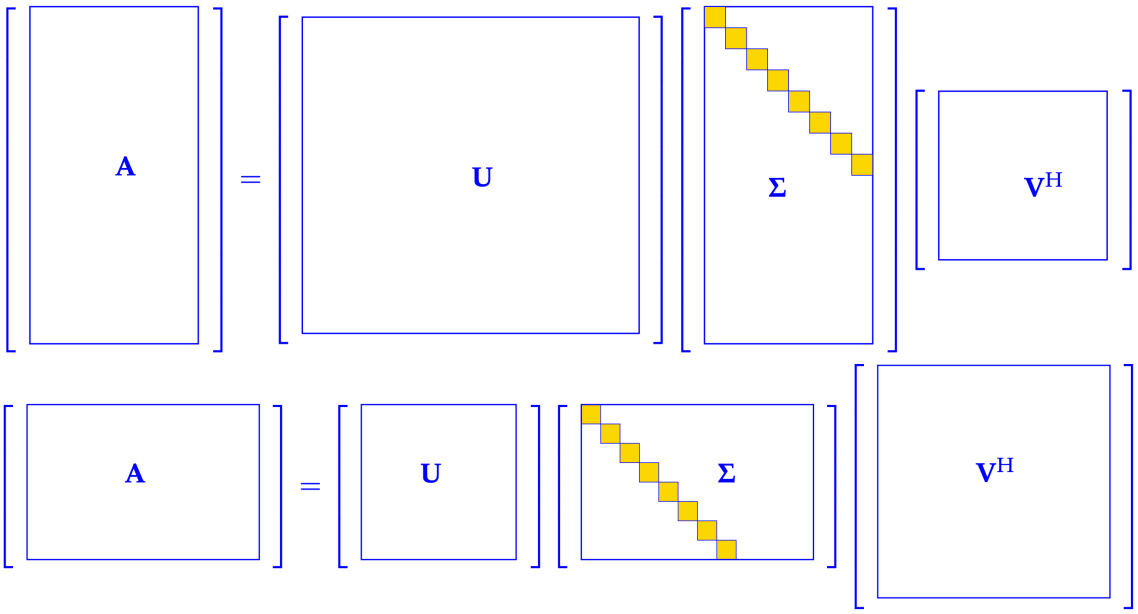


Figure 17: full SVD of fat and tall matrices

But we may also consider the thin SVD with $\mathbf{\Sigma} \in \mathbb{K}^{p,p}, \mathbf{U} \in \mathbb{K}^{m,p}, \mathbf{V}^H \in \mathbb{K}^{p,n}$ by removing the zero rows of Σ and remove the correspondingly irrelevant

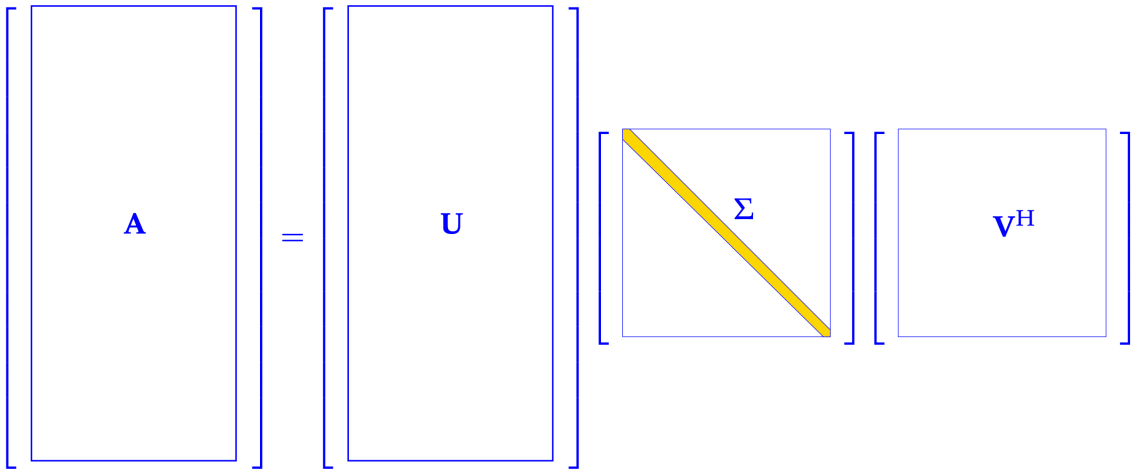


Figure 18: economical SVD

Definition Singular Value Decomposition: The decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H$ from the preceding theorem is called Singular Value Decomposition (SVD) of $\mathbf{A}$. The diagonal entries σ_i of $\mathbf{\Sigma}$ are the singular values of $\mathbf{A}$. The columns of $\mathbf{U}/\mathbf{V}$ are the left/right singular vectors of $\mathbf{A}$.

The decomposition has an intuitive geometric meaning. $\mathbf{V}$ and $\mathbf{U}$ are orthogonal transformations. $\mathbf{V}^H$ transforms the space orthogonally (combination or rotations and reflections) so that the deformations of the transformation $\mathbf{A}$ to the space happen along the axis. We use $\mathbf{\Sigma}$ to scale along

those axis and then rotate/reflect the space again. So $\mathbf{V}$ and $\mathbf{U}$ perform basis changes so that we can apply simple scalings though $\mathbf{\Sigma}$.

Lemma: The squares σ_i^2 of the non-zero singular values of $\mathbf{A}$ are the non-zero eigenvalues of $\mathbf{A}^H\mathbf{A}, \mathbf{A}\mathbf{A}^H$ with associated eigenvectors $(\mathbf{V})_{:,1}, ..., (\mathbf{V})_{:,p}$ or rather $(\mathbf{U})_{:,1}, ..., (\mathbf{U})_{:,p}$ respectively.

Proof. We consider $\mathbf{A}^H\mathbf{A}$ but the case for $\mathbf{A}\mathbf{A}^H$ is analogous.

$$\mathbf{A}^H\mathbf{A} = (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^H)^H(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^H) = \mathbf{V}\mathbf{\Sigma}^H\mathbf{U}^H\mathbf{U}\mathbf{\Sigma}\mathbf{V}^H = \mathbf{V}\mathbf{\Sigma}^H\mathbf{\Sigma}\mathbf{V}^H$$

$\mathbf{A}^H\mathbf{A}$ and $\mathbf{\Sigma}^H\mathbf{\Sigma}$ are similar matrices with the same eigenvalues. And $\mathbf{V}$ contains the eigenvectors of $\mathbf{A}^H\mathbf{A}$. □

SVD Additive Rank-1 Decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H$ may also be written as $\sum_{j=1}^{p} \sigma_j u_j v_j^H$. The tensor product clearly has rank one, so we add p rank-1 matrices. If $\sigma_{r+1} = \cdots = \sigma_p = 0$:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H = \sum_{j=1}^{r} \sigma_j u_j v_j^H$$

This different way of writing the SVD is helpful in many situations! Remember it.

For instance, notice that if $\sigma_r \neq 0$:

$$\mathbf{A}\mathbf{v}_l = \sum_{j=1}^{r} \sigma_j u_j v_j^H v_l = \begin{cases} \sigma_l u_l, & l \leq r \\ 0, & l > r \end{cases}$$

Then, it follows that

- $\mathcal{R}(\mathbf{A}) = span\{\mathbf{u}_1, \ldots, \mathbf{u}_r\}$ and $rank(\mathbf{A}) = r$
- $\mathcal{N}(\mathbf{A}) = span\{\mathbf{v}_{r+1}, \ldots, \mathbf{v}_n\}$

Lemma: If for some $1 \leq r \leq p := \min\{m, n\}$, the singular values of $A \in \mathbb{K}^{m,n}$ satisfy $\sigma_1 \geq ... \geq \sigma_r > \sigma_{r+1} = ... = \sigma_p = 0$, then

- $rank(\mathbf{A}) = r$
- $\mathcal{N}(\mathbf{A}) = span\{(\mathbf{V})_{:,r+1}, ..., (\mathbf{V})_{:,n}\}$
- $\mathcal{R}(\mathbf{A}) = span\{(\mathbf{U})_{:,1}, ..., (\mathbf{U})_{:,r}\}$

The given vectors spanning $\mathcal{N}$ and $\mathcal{R}$ are even an orthonormal basis each.

4.4.2 SVD in Eigen

To work with the SVD in Eigen, one shall include `#include <Eigen/SVD>`. Then with `Eigen::JacobiSVD<MatrixXd> svd(A)` can be used to compute the SVD of $\mathbf{A}$. By default, the matrices $\mathbf{U}$ and $\mathbf{V}$ are not explicitly computed. But one can request so and specify whether one wants full/economical form by passing options as demonstrated in the following listing.

Listing 20: SVD in Eigen

```
#include <Eigen/SVD>

std::tuple<MatrixXd, MatrixXd, MatrixXd> svd_full(const MatrixXd
    & A) {
  Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeFullU | Eigen
     ::ComputeFullV);
  MatrixXd U = svd.matrixU();
  MatrixXd V = svd.matrixV();
  VectorXd sv = svd.singularValues();
  MatrixXd Sigma = MatrixXd::Zero(A.rows(), A.cols());
  const unsigned p = sv.size();
  Sigma.block(0,0,p,p) = sv.asDiagonal();
  return std::tuple<MatrixXd, MatrixXd, MatrixXd>(U, Sigma, V);
}

std::tuple<MatrixXd, MatrixXd, MatrixXd> svd_eco(const MatrixXd&
     A) {
  Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeThinU | Eigen
     ::ComputeThinV);
  MatrixXd U = svd.matrixU();
  MatrixXd V = svd.matrixV();
  VectorXd sv = svd.singularValues();
  MatrixXd Sigma = sv.asDiagonal();
  return std::tuple<MatrixXd, MatrixXd, MatrixXd>(U, Sigma, V);
}
```

- Despite not being shown, one can also specify `Eigen::ComputeFullU | Eigen::ComputeThinV`, for instance.
- One shall not call `.matrixU()` nor `.matrixV()` if one did not request for them to be computed as mentioned before.
- The provided singular values are sorted in non-decreasing order.

Although we don't discuss the workings of the algorithms used to compute the SVD we shall memorize its runtime:

$$\mathcal{O}(\min\{m,n\}^2 \cdot \max\{m,n\})$$

SVD based rank computation Using the SVD one can quite easily compute the rank of some matrix. From an above Lemma we know that $rank(\mathbf{A}) = r$ with r being the number of non-zero singular values. When determining r we shall NOT determine r by testing the singular values to be zero using $== 0$. Instead, we shall test for relative smallness.

```
MatrixXd::Index rank_by_svd(const MatrixXd &A, double tol =ESP){
  if (A.norm()==0)
    return MatrixXd::Index(0);
  Eigen::JacobiSVD<MatrixXd> svd(A);
  const VectorXd sv = svd.singularValues();
  MatrixXd::Index n = sv.size();
```

```
  MatrixXd::Index r = 0;
  while ((r<n) && (sv(r) >= sv(0)*tol))
    r++;
  return r;
}
```

Anyway, Eigen provides its own utility function for this which should generally be used. If `.setThreshold(tol)` was not called, Eigen uses the machine precision epsilon for that datatype.

```
MatrixXd::Index rank_eigen(const MatrixXd &A, double tol = ESP)
    {
  return A.jacobiSvd().setThreshold(tol).rank();
}
```

Orthonormal Bases of Range and Nullspace of A using SVD .

```
MatrixXd nullspace(const MatrixXd &A, double tol = EPS) {
  using index_t = MatrixXd::Index;
  Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeFullV);
  index_t r = svd.setThreshold(tol).rank();
  MatrixXd Z = svd.matrixV().rightCols(A.cols()-r);
  return Z;
}

MatrixXd rangespace(const MatrixXd &A, double tol = EPS) {
  using index_t = MatrixXd::Index;
  Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeFullU);
  index_t r = svd.setThreshold(tol).rank();
  return svd.matrixU().leftCols(r);
}
```

4.4.3 SVD based Solver for Linear Least Squares Problems

Now we discuss how to solve least squares problems using SVD. This extends our repertoire of methods to solve SVD from QR and normal equations. Remember that for arbitrary $\mathbf{A}$ we can compute the unique solution of $\mathbf{Ax} = \mathbf{b}$ according to least squares and the minimal norm condition. For that, we learned about the Moore-Penrose Pseudoinverse, which we now denote $\mathbf{A}^+$: $\mathbf{x}^\dagger = \mathbf{A}^+\mathbf{b}$.

We derived the Moore-Penrose Pseudoinverse by considering the normal equations and the additional criteria $\mathbf{x}^\dagger \in \mathcal{N}(\mathbf{A})^\perp$ so that we get the unique solution with minimum norm through the requirement of orthogonality of the vector to the set of all possible solutions.

Now we notice that $\mathcal{N}(\mathbf{A})^\perp = span\{\mathbf{v}_1, ..., \mathbf{v}_r\} = \mathcal{R}(\mathbf{V}_1) = \{\mathbf{V}_1\mathbf{y} | \mathbf{y} \in \mathbb{R}^r\}$. Here we have split $\mathbf{V} = \begin{bmatrix}\mathbf{V}_1 & \mathbf{V}_2\end{bmatrix}$ with $\mathbf{V}_1$ being the upper r columns of $\mathbf{V}^\top$ and $\mathbf{V}_2$ the lower $n - r$ rows.

So, $\mathbf{x}^\dagger = \mathbf{V}_1\mathbf{y}$ for some $\mathbf{y} \in \mathbb{R}^r$. We can combine this with the normal equation:

$$\begin{aligned}
&\mathbf{A}^\top\mathbf{A}\mathbf{x} = \mathbf{A}^\top\mathbf{b} \\
\Rightarrow\ &\mathbf{A}^\top\mathbf{A}\mathbf{V}_1\mathbf{y}^+ = \mathbf{A}^\top\mathbf{b} \\
\Rightarrow\ &\mathbf{V}_1^\top\mathbf{A}^\top\mathbf{A}\mathbf{V}_1\mathbf{y}^+ = \mathbf{V}_1^\top\mathbf{A}^\top\mathbf{b} \\
\Rightarrow\ &\mathbf{V}_1^\top\mathbf{V}\boldsymbol{\Sigma}\mathbf{U}^\top\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top\mathbf{V}_1\mathbf{y}^+ = \mathbf{V}_1^\top\mathbf{V}\boldsymbol{\Sigma}^\top\mathbf{U}^\top\mathbf{b} \\
\Rightarrow\ &\mathbf{V}_1^\top[\mathbf{V}_1,\mathbf{V}_2]\begin{bmatrix}\boldsymbol{\Sigma}_r & 0\\ 0 & 0\end{bmatrix}\begin{bmatrix}\boldsymbol{\Sigma}_r & 0\\ 0 & 0\end{bmatrix}\begin{bmatrix}\mathbf{V}_1^\top\\ \mathbf{V}_2^\top\end{bmatrix}\mathbf{V}_1\mathbf{y}^+ = \mathbf{V}_1^\top\begin{bmatrix}\mathbf{V}_1 & \mathbf{V}_2\end{bmatrix}\begin{bmatrix}\boldsymbol{\Sigma}_r & 0\\ 0 & 0\end{bmatrix}\begin{bmatrix}\mathbf{U}_1^\top\\ \mathbf{U}_2^\top\end{bmatrix}\mathbf{b} \\
&\Rightarrow \boldsymbol{\Sigma}_r^2\mathbf{y}^+ = \boldsymbol{\Sigma}_r\mathbf{U}_1^\top\mathbf{b} \\
&\Rightarrow \boldsymbol{\Sigma}_r\mathbf{y}^+ = \mathbf{U}_1^\top\mathbf{b} \\
&\Rightarrow \mathbf{y}^+ = \boldsymbol{\Sigma}_r^{-1}\mathbf{U}_1^\top\mathbf{b}
\end{aligned}$$

We can extend this formula to compute the solution to least squares problem: $\mathbf{V}_{:,1:r}\boldsymbol{\Sigma}_r^{-1}\mathbf{U}_1^\top\mathbf{b}$.

Listing 21: Least Squares Solve using SVD in Eigen

```
#include <Eigen/SVD>
VectorXd lsqsvd(const MatrixXd &A, const VetorXd &b) {
  Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeThinU | Eigen
     ::ComputeThinV);
  VectorXd sv = svd.singularValues();
  unsigned int r = svd.rank();
  MatrixXd U = svd.matrixU(), V=svd.matrixV();
  return V.leftCols(r) * (sv.head(r).cwiseInverse().asDiagonal()
      * (U.leftCols(r).adjoint() * b));
}

VectorXd lsqsvd_eigen(const MatrixXd &A, const VectorXd &b) {
  Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeThinU | Eigen
     ::ComputeThinV);
  return svd.solve(b);
}
```

Theorem Pseudoinverse and SVD: If $\mathbf{A} \in \mathbb{K}^{m,n}$ has the SVD $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H$, then its Moore-Penrose pseudoinverse is given by $\mathbf{A}^\dagger = \mathbf{V}_1\boldsymbol{\Sigma}_r^{-1}\mathbf{U}_1^H$.

4.4.4 SVD based Optimizations and Approximations

Norm-Constrained Extrema of Quadratic Forms Given some $\mathbf{A} \in \mathbb{K}^{m,n}, m \geq n$, we consider how to find $\mathbf{x} \in \mathbb{K}^n$ with $\|\mathbf{x}\|_2 = 1$ such that $\|\mathbf{A}\mathbf{x}\|_2^2$ is maximal. Through this maximum, we can define the Euclidean matrix norm.

The Euclidean matrix norm of $\mathbf{A}$ is defined as:

$$\|\mathbf{A}\|_2 := \max_{\|\mathbf{x}\|} \|\mathbf{A}\mathbf{x}\| = \sigma_1$$

With the SVD $\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top$ we can solve this problem as it corresponds to maximizing $\|\mathbf{A}\mathbf{x}\|_2 = \|\boldsymbol{\Sigma}\mathbf{V}^\top\mathbf{x}\|_2$. If we consider $\mathbf{y} = \mathbf{V}^\top\mathbf{x}$ with $\|\mathbf{y}\| = 1$ we intend to maximize $\|\mathbf{A}\mathbf{x}\|_2 =$

$\sum_{l=1}^{n} \sigma_l^2 y_l^2$. As $\mathbf{y}$ must have norm 1, we can maximize this by putting all weight on the largest singular value. So, $\mathbf{y} = \mathbf{e}_1$ and $\mathbf{x} = \mathbf{V}\mathbf{e}_1 = (\mathbf{V})_{:,1}$. The maximal value is $\|\mathbf{A}\mathbf{x}\|_2 = \sigma_1$.

Fitting Hyperplanes We can express a hyperplane by using a vector $\mathbf{n}$ orthogonal to the plane and one point of the plane expressed through $\mathbf{p}$.

$$\mathbf{n}^\top(\mathbf{x}-\mathbf{p}) = 0 \Rightarrow \mathbf{n}^\top\mathbf{x} - \mathbf{n}^\top\mathbf{p} = 0 \Rightarrow \mathbf{n}^\top\mathbf{x} - \mathbf{c}' = 0 \Rightarrow \mathbf{n}^\top + \mathbf{c} = 0$$
$$\Longrightarrow \mathcal{H} := \{\mathbf{x} \in \mathbb{R}^d | c + \mathbf{n}^\top\mathbf{x} = 0\}, \|\mathbf{n}\|_2 = 1, c \in \mathbb{R}$$

Given point coordinate vectors $\mathbf{y}_1, ..., \mathbf{y}_m \in \mathbb{R}^d, m > d$, we want to find $\mathcal{H} \leftrightarrow \{c \in \mathbb{R}, \mathbf{n} \in \mathbb{R}^d, \|\mathbf{n}\|_2 = 1\}$ such that $\sum_{j=1}^{m} dist(\mathcal{H}, \mathbf{y}_j)^2 = \sum_{j=1}^{m} |c + \mathbf{n}^\top\mathbf{y}_j|^2 \to \min$. I.e.:

$$\left\| \begin{bmatrix} 1 & y_{1,1} & \dots & y_{1,d} \\ 1 & y_{2,1} & \dots & y_{2,d} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & y_{m,1} & \dots & y_{m,d} \end{bmatrix} \begin{bmatrix} c \\ n_1 \\ \vdots \\ n_d \end{bmatrix} \right\|_2 \to \min, \qquad \text{with } \|\mathbf{n}\|_2 = 1$$

We rewrite this using the QR-Decomposition of $\mathbf{A}$.

$$\|\mathbf{A}\mathbf{x}\|_2 \to \min$$
$$\Longleftrightarrow$$
$$\|\mathbf{R}\mathbf{x}\|_2 = \left\| \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1,d+1} \\ 0 & r_{22} & \cdots & r_{2,d+1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_{d+1,d+1} \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} \begin{bmatrix} c \\ n_1 \\ \vdots \\ n_d \end{bmatrix} \right\|_2 \to \min$$

We can choose c freely. Thus, we can optimize $\mathbf{n}$ to minimize the cost through $\tilde{\mathbf{A}} = \begin{bmatrix} r_{22} & \cdots & r_{2,d+1} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & r_{d+1,d+1} \\ 0 & \dots & 0 \\ \vdots & \vdots & \vdots \\ 0 & \dots & 0 \end{bmatrix}$.

We can do this minimization under the constraint $\|\mathbf{n}\| = 1$ just as in the previous paragraph. But instead of choosing the maximum singular value, we now look at the minimum singular value and the corresponding vector $(\mathbf{V})_{:,r}$ from $\mathbf{V}$. Once having found this optimal solution, we choose $c := -\frac{1}{r_{11}} \sum_{l=1}^{d} n_l r_{1,l}$.

We can generalize to let an arbitrary number of parts of the vector $\mathbf{x}$ be without constraint.

Listing 22: Fitting under Constraint in Eigen

```
void clsq(const MatrixXd& A, const unsigned dim, double& c,
    VectorXd& n) {
  unsigned p = A.cols(), m = A.rows();
  if (p < dim + 1) {
    cerr << "not enough unknowns\n";
    return;
```

```
  }
  if (m < dim) {
    cerr << "not enough equations\n";
  }

  m = std::min(m, p);
  MatrixXd R = A.householderQr().matrixQR()
    .template triangularView<Eigen::Upper>();
  MatrixXd V = R.block(p-dim, p-dim, m+dim-p, dim)
    .jacobiSvd(Eigen::ComputeFullV).matrixV();
  n = V.col(dim - 1);

  MatrixXd R_topleft = R.topLeftCorner(p-dim, p-dim);
  c = -(R_topleft.template triangularView<Eigen::Upper>()
    .solve(R.block(0,p-dim,p-dim,dim)) * n)(0);
}
```

Best Low-Rank Approximation In this section we want to get approximations of matrices through matrices with a lower rank. A low-rank matrix is a matrix $\mathbf{A} \in \mathbb{R}^{m,n}$ if $r := rank(\mathbf{A}) \ll \min\{m, n\}$. This enables compression, because we can store the lower-rank matrices using less memory by utilizing its SVD in the form $\mathbf{A} = \sum_{l=1}^{r} \sigma_l (\mathbf{U})_{:,l} (\mathbf{V})_{:,l}^{\top}$. This only takes $\mathcal{O}(r(m + n)) \ll \mathcal{O}((m \cdot n))$ memory.

Before we introduce how to get the best low-rank approximation, we introduce the Frobenius norm.

Definition Frobenius Norm: The Frobenius norm of $A \in \mathbb{K}^{m,n}$ is defined as

$$\|\mathbf{A}\|_F^2 := \sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2$$

Theorem Best Low-Rank Approximation: Let $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H$ be the SVD of $\mathbf{A} \in \mathbb{K}^{m,n}$. For $1 \leq k \leq rank(\mathbf{A})$ set

$$\mathbf{A}_k := \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^H = \sum_{l=1}^{k} \sigma_l (\mathbf{U})_{:,l} (\mathbf{V})_{:,l}^H$$

- $U_k := [(U)_{:,1}, ..., (U)_{:,k}] \in \mathbb{K}^{m,k}$
- $V_k := [(V)_{:,1}, ..., (V)_{:,k}] \in \mathbb{K}^{n,k}$
- $\Sigma_k := diag(\sigma_1, ..., \sigma_k) \in \mathbb{K}^{k,k}$

Then for $\|\cdot\| = \|\cdot\|_F$ and $\|\cdot\| = \|\cdot\|_2$ holds true

$$\|A - A_k\| \leq \|A - F\|, \forall F \in \mathcal{R}_k(m, n)$$

that is, A_k is the rank-K best approximation of A in the matrix norms $\|\cdot\|_F$ and $\|\cdot\|_2$.

$$\mathcal{R}_k(m, n) = \{\mathbf{M} \in \mathbb{K}^{m,n} | rank(\mathbf{M}) \leq k\}$$

As with any approximation, this introduces an error. The error depends on the chosen norm:

- Euclidean norm: $\|\mathbf{A} - \mathbf{A}_k\|_2 = \| \begin{bmatrix} 0 & 0 \\ 0 & \mathbf{\Sigma}_{k+1-n} \\ 0 & 0 \end{bmatrix} \|_2 = \sigma_{k+1}$
- Frobenius norm: $\|\mathbf{A} - \mathbf{A}_k\|_F^2 = \| \begin{bmatrix} 0 & 0 \\ 0 & \mathbf{\Sigma}_{k+1-n} \\ 0 & 0 \end{bmatrix} \|_F^2 = \sigma_{k+1}^2 + ... + \sigma_n^2 = \sum_{j=k+1}^{\min\{m,n\}} \sigma_j^2$

This reveals that the decay of the singular values predicts the quality of low-rank approximations. If we want the relative error to be below some tolerance, we can use this formula tho choose the approximation rank k:

$$\frac{\|\mathbf{A} - \mathbf{A}_k\|_2}{\|\mathbf{A}\|_2} = \frac{\sigma_{k+1}}{\sigma_1} \leq \text{target tolerance}$$

Principal Component Data Analysis (PCD) PCD could be said to be a method of data science. Most generally, it consists of two things: Trend detection in a dataset and data classification according to those trends for individual data vectors.

Instead of providing a purely theoretical and formal introduction to the topic, we will just go hands-on with an example and illustrate everything important along the way. This better equips the reader with the necessary intuition for this topic.

Trend Analysis We have n stocks of which we know the price at m times. We fit this data in a matrix $\mathbf{A} \in \mathbb{R}^{m,n}$ where each column corresponds to one stock. We now want to find the dominant trends for stock price behavior, i.e., $\mathbb{R}^m$ vectors which showcase some most typical stock trends.

In Linear Algebra formality, this corresponds to us wanting to find orthonormal vectors $\mathbf{u}_1, \ldots, \mathbf{u}_p \in \mathbb{R}^m$ such that for all $k \in [n]$: $(\mathbf{A})_{:,k}$ from our dataset is in $span\{\mathbf{u}_1, \ldots, \mathbf{u}_p\}$ + error. The goal is to choose $\mathbf{u}_1, \ldots, \mathbf{u}_p$ so that the accumulated error is minimal. Then, the p vectors $\mathbf{u}_i$ correspond to the trends in the dataset.

The approach is very similar to low-rank approximations: We want to keep the components of the SVD of $\mathbf{A}$ with high singular values.

$$A = \underbrace{\sum_{l=1}^{p} \sigma_l u_l v_l^\top}_{\text{reconstruction from trends}} + \underbrace{\sum_{l=p+1}^{\min\{n,m\}} \sigma_l u_l v_l^\top}_{\text{error/deviation from actual value}}$$

The p trends are meaningful if the singular values decay quickly so that with $p \ll m, n$ we already get good information for the overall system.

Assuming p is already known, the p dominant trends can then be found in the p leftmost columns of $\mathbf{U}$ as those being the vectors, which span the space corresponding to the p largest singular values. The factor σ_l then corresponds to the importance of this trend. And $(\mathbf{v}_l)_k$ denotes the strength of the l-th trend in the data vector a_k.

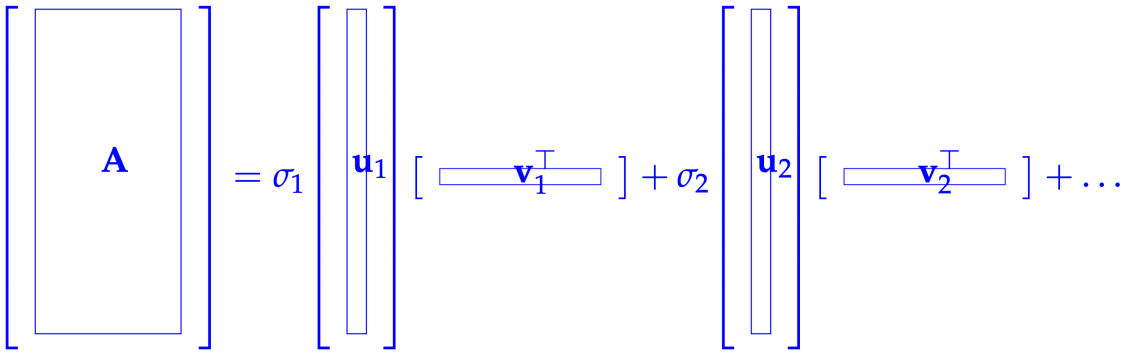


Figure 19: Help to Understand the Strength of a Trend in a Data Vector

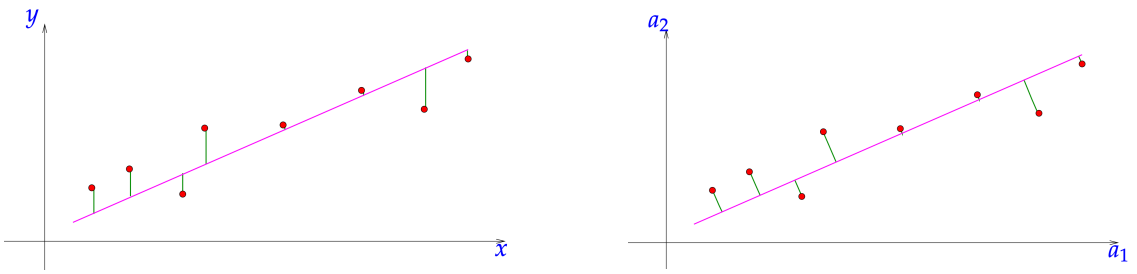
One may consider a lin-log plot of the singular values depending on their number in the ordering. From this graph, one can often quickly see which few singular values with highest number are most important.

Data Classification As just mentioned, the entries of $\mathbf{V}$ can be understood to identify the strength of some trend in some data vector $\mathbf{a}_k$. So we can simply plot all data vectors as a point in a (multi-dimensional) graph, where the 'strength' values for the main identified trends are the values for the different axis. Then, we can observe how different groups/clusters emerge.

If we had a new data vector which is was not part of the optimization and we want to identify the strengths of our trends for this new vector, we must find its best approximation in in the basis $\{\mathbf{u}_1, \ldots, \mathbf{u}_k\}$. The coefficients then correspond to $\sigma_i \cdot s_i$, where s_i can be said to correspond to the strength of the i-th trend.

Proper Orthogonal Decomposition (POD) Given some dataset, this is about approximating this set with a subspace of the dataspace of some arbitrary dimension. With lower dimensions, our approximation will clearly be less accurate but also more abstract.

We can compare this approach to least squares on a high level. With least squares we always try to find parameters for a linear model so that the model has the lowest deviation possible from the true data. The deviation is measured by $\|\mathbf{Ax} - \mathbf{b}\|_2$. POD wants to find an approximation but differs in some ways: (1) Instead of finding parameters for an arbitrary linear model, POD always consider a subspace. (2) This subspace cannot be affine. It must always go through zero. (3) The measure for deviation from the actual data is different. With POD, we consider the distance across all dimensions from our data vectors to the selected subspace.



Least Squares Deviation (vertical distances)

POD Deviation (minimal distances)

Figure 20: Comparison of Deviations of POD and Least Squares with Linear Regression

More formally: Given data points $\mathbf{a}_1, \ldots, \mathbf{a}_n \in \mathbb{R}^m$ with $m, n \in \mathbb{N}$, for some $k \leq \min\{m, n\}$

we want to find a subspace $U_k \subset \mathbb{R}^m$ such that

$$U_k = \text{argmin}_{W \subset \mathbb{R}^m, \dim W = k} \sum_{j=1}^{n} \inf_{\mathbf{w} \in W} \|\mathbf{a}_j - \mathbf{w}\|_2^2.$$

We want to find U_k in the form of a set of orthonormal basis vectors for U_k. Let $\mathbf{W} = \begin{bmatrix} \mathbf{w}_1 & \dots & \mathbf{w}_k \end{bmatrix} \in \mathbb{R}^{m,k}, k \leq m$ denote this basis.

If $\mathbf{x}$ is an arbitrary vector in $\mathbb{R}^m$ we measure the quality of the approximation $\mathbf{W}$ for $\mathbf{x}$ through the distance of $\mathbf{x}$ from the subspace U_k / space spanned by $\mathbf{W}$.

$$dist(\mathbf{x}, \mathbf{W}) = \min_{\mathbf{y} \in \mathbb{R}^k} \|\mathbf{x} - \mathbf{W}\mathbf{y}\|_2 = \|\mathbf{x} - \mathbf{W}\mathbf{W}^\top \mathbf{x}\|_2$$

The minimization through $\mathbf{y} = \mathbf{W}^\top \mathbf{x}$ holds, because $\mathbf{W}\mathbf{W}^\top$ is the (orthogonal) projection of $\mathbf{x}$ onto the subspace spanned by $\mathbf{W}$ (not justified in the lecture).

If we apply this quality measure for all our data points, we can express the criteria for the POD solution in another way:

$$\sum_{j=1}^{n} \inf_{\mathbf{w} \in W} \|\mathbf{a}_j - \mathbf{w}\|_2^2 = \sum_{j=1}^{n} \|\mathbf{a}_j - \mathbf{W}\mathbf{W}^\top \mathbf{a}_j\|_2^2 = \|\mathbf{A} - \mathbf{W}\mathbf{W}^\top \mathbf{A}\|_F^2$$
$$\implies \text{argmin}_{\mathbf{W} \in \mathbb{R}^{m,k}, \mathbf{W}^\top \mathbf{W} = \mathbf{I}} \|\mathbf{A} - \mathbf{W}\mathbf{W}^\top \mathbf{A}\|_F^2$$

Notice that $\mathbf{W}\mathbf{W}^\top$ has at most rank k, and then also $\mathbf{W}\mathbf{W}^\top \mathbf{A}$ has at most rank k. Remember that we have discussed before how to get the best rank k approximation of some matrix. Let's use that optimal approximation for $\mathbf{W}\mathbf{W}^\top \mathbf{A}$, which then can only have lower distance.

$$\|\mathbf{A} - \mathbf{W}\mathbf{W}^\top \mathbf{A}\|_F \geq \|\mathbf{A} - \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^H\|_F, \quad \forall \mathbf{W} \in \mathbb{R}^{m,k}, \mathbf{W}^\top \mathbf{W} = \mathbf{I}$$

With the SVD of $\mathbf{A}$ we see that there is an assignment to $\mathbf{W}$ which is exactly equal to this lower bound. This assignment then must be the optimally minimizing solution. The assignment ist $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ (its SVD) and $\mathbf{W} = \mathbf{U}_k$. Then:

$$\mathbf{W}\mathbf{W}^\top \mathbf{A} = \mathbf{U}_k \mathbf{U}_k^\top \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top = \mathbf{U} \left[\mathbf{I}_k \; 0\right] \mathbf{\Sigma}\mathbf{V}^\top = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^\top$$

Theorem Solution of POD Problems: The subspace U_k spanned by the first k left singular vectors of $\mathbf{A} = \begin{bmatrix} \mathbf{a}_1 & \dots & \mathbf{a}_n \end{bmatrix} \in \mathbb{R}^{m,n}$ solves the POD problem

$$U_k = \mathcal{R}((\mathbf{U})_{:,1:k})$$

with $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H$ being the SVD of $\mathbf{A}$.

We can compute the POD error like this:

$$\sum_{j=1}^{n} \inf_{\mathbf{w} \in U_k} \|\mathbf{a}_j - \mathbf{w}\|_2^2 = \sum_{l=k+1}^{p} \sigma_l^2$$

That statement follows from the error of the low-rank approximation, which is the same of our POD solution.

The POD problem is basically solved as a low rank approximation. Above we have shown that those two problems can generally be understood equivalently.

4.5 Comparison of Normal Equations, QR, SVD

Normal Equations are only of practical use for very large sparse matrices, because the other approaches struggle to utilize that sparsity/the orthogonal transformations destroy the sparsity. But is the matrix is not sparse or not very large, we prefer the other approaches for their better numeric stability.

5 Filtering Algorithms

A signal is obtained from measuring some time-dependent value $X(t)$ at n sampling points $t_0, t_1, \ldots, t_{n-1}$, which we write as a vector $\in \mathbb{R}^n$. The process of acquiring such a vector is called sampling. Usually, we sample at equidistant points and denote the time-difference between t_i and t_{i+1} with Δt. Those are time-discrete (discrete sampling points), finite (finite sampling points) and analog (values of X are $\in \mathbb{R}$). If we omit finite, we leave the scope of Linear Algebra as we get infinite sequences in the real numbers: $(X_k)_{k\in\mathbb{Z}} \in \mathbb{R}^{\mathbb{Z}}$.

In this section, we will consider how to work with signals, analyze them, decompose them, ... We start with a basic theoretical introduction and then discuss the Fourier Transform in length, specifically the Discrete Fourier Transform (DFT).

5.1 Filters and Convolutions

A filter/channel is a function F : input signals $\to$ output signals. More formally, $F : l^\infty(\mathbb{Z}) \to l^\infty(\mathbb{Z})$ where $l^m(\mathbb{K}) := \{(x_j)_{j\in\mathbb{K}} | \sup |x_j| < m\}$. So that we can use Linear Algebra to work with signals and describe filters, we introduce four properties which we will assume in the following.

5.1.1 LT-FIR

Finiteness means that finite input signals are mapped to finite output signals.

Definition Finite Channel/Filter: A channel/filter $F : l^\infty(\mathbb{Z}) \to l^\infty(\mathbb{Z})$ is called finite, if every input signal of finite duration produces an output signal of finite duration,

$$(\exists M \in \mathbb{N} : |j| > M \Rightarrow x_j = 0)$$
$$\Longrightarrow$$
$$(\exists N \in \mathbb{N} : |k| > N \Rightarrow (F((x_j)_{j\in\mathbb{Z}}))_k = 0)$$

Time invariance means that is should not matter when a signal arrives. With S_m being the shift operator (/channel/filter) as $S_m((x_j)_{j\in\mathbb{Z}}) = (x_{j-m})_{j\in\mathbb{Z}}$:

Definition Time-invariant channel/filter:
A filter $F : l^\infty(\mathbb{Z}) \to l^\infty(\mathbb{Z})$ is called time-invariant (TI), if shifting the input in time leads to the same output shifted in time by the same amount.

$$\forall (x_j)_{j\in\mathbb{Z}} \in l^\infty(\mathbb{Z}), \forall m \in \mathbb{Z} : F(S_m((x_j)_{j\in\mathbb{Z}})) = S_m(F((x_j)_{j\in\mathbb{Z}}))$$

Time-invariant filters/channels commute with the time shift operator.

Linearity means that the channel/filter is a linear map.

Definition Linear Channel/Filter: A filter $F : l^\infty(\mathbb{Z}) \to l^\infty(\mathbb{Z})$ is called linear, if F is a linear mapping:

$$F(\alpha(x_j)_{j\in\mathbb{Z}} + \beta(y_i)_{j\in\mathbb{Z}}) = \alpha F((x_j)_{j\in\mathbb{Z}}) + \beta F((y_j)_{j\in\mathbb{Z}})$$

for all sequences $(x_j)_{j\in\mathbb{Z}}, (y_j)_{j\in\mathbb{Z}}$ and real numbers $\alpha, \beta \in \mathbb{R}$.

Causality means that the output should only arrive after the input.

Definition Causal Channel/Filter: A filter $F : l^\infty(\mathbb{Z}) \to l^\infty(\mathbb{Z})$ is called causal (or physical, or nonanticipative), if the output does not start before the input

$$\forall M \in \mathbb{N} : (x_j)_{j\in\mathbb{Z}} \in l^\infty(\mathbb{Z}), x_j = 0, \forall j \leq M \Rightarrow F((x_j)_{j\in\mathbb{Z}})_k = 0, \forall k \leq M$$

We may use the acronym LT-FIR for filters that are linear, time-invariant, finite, and causal. From now on, we only consider LT-FIR filters with finite, time-discrete, and analog signals - unless explicitly stated otherwise.

Impulse Response A linear mapping is uniquely defined by actions on the basis vector. For $l^\infty(\mathbb{Z})$, a basis is given by $e_k := (\delta_{jk})_{j\in\mathbb{Z}}$, which we call an impulse. Form time-invariance follows that we can shift any basis signal e_k to the 'origin' e_0, evaluate there, and then shift back. Thus, from $S_{-k}(e_k) = e_0$ follows that knowing $F(e_0)$ fully defines a LT-FIR.

Definition Impulse Response: The impulse response (IR) of a channel/filter is the output for a single unit impulse at $t = 0$ as input, that is, the input signal is

$$x_j = d_{j,0} := \begin{cases} 1, & \text{if } j = 0 \\ 0, & \text{else} \end{cases} \text{ (Kronecker symbol)}$$

Now we will reason more formally, why a LT-FIR signal is fully defined by its impulse response. From $F \circ S_m = S_m \circ F, \forall m \in \mathbb{Z}$ follows with time-invariance that

$$F((\delta_{j,k})_{j\in\mathbb{Z}}) = (h_{j-k})_{j\in\mathbb{Z}} = \begin{pmatrix} \dots & 0 & h_0 & h_1 & \dots & h_{n-1} & 0 & \dots \end{pmatrix}, \; h_0 \text{ being at } t_k = k \cdot \Delta t$$

With linearity follows:

$$F((x_j)) = \sum_k x_k F(S_k(\delta_{j,0})_{j\in\mathbb{Z}}) = \sum_k x_k S_k(F((\delta_{j,0})_{j\in\mathbb{Z}}))$$

5.1.2 LT-FIR Linear Mappings

We have just seen that any signal is a superposition of shifted impulse responses. For some input signal of length m, a filter with an n impulse response produces an output of length $m + n - 1$. Generally, "output duration is = input duration + duration of impulse response − 1".

$$y_k = \sum_{j=0}^{m-1} h_{k-j} x_j, k = 0, ..., m + n - 2$$

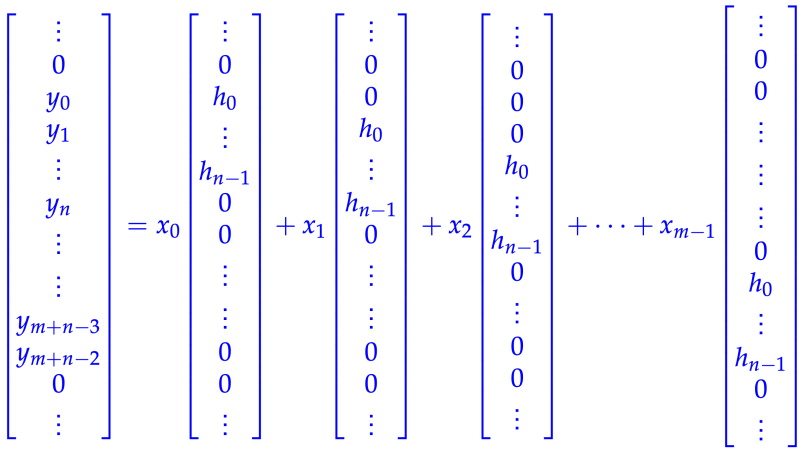


Figure 21

We can extend this to infinite signals (if they are zero after some time), because the additionally considered h values are trivially 0:

$$y_k = \sum_{j=0}^{m-1} h_{k-j} x_j = \sum_{j \in \mathbb{Z}} h_{k-j} x_j$$

Definition Convolution of Sequences: Given two sequences $(h_k)_{k\in\mathbb{Z}}$, $(x_k)_{k\in\mathbb{Z}}$, at least one of which is finite or decays sufficiently fast, their convolution is another sequence $(y_k)_{k\in\mathbb{Z}}$, defined as

$$y_k = \sum_{j \in \mathbb{Z}} h_{k-j} x_j, k \in \mathbb{Z}$$

We write $(y_j) = (h_x) * (x_k)$.

We require finiteness of one or sufficiently fast decay so that this sum exists. For practical applications we want finiteness, so that we can represent y_k in memory.

Theorem Convolution of Sequences Commutes: If well-defined, the convolution of sequences commutes

$$(x_k) * (h_k) = (h_k) * (x_k)$$

Convolutions can be used to just arbitrarily combine two signals. Or we can also explicitly use it to apply a filter to a signal by considering the cnovolution with its impulse response.

5.1.3 Discrete Convolutions

Just as a reminder: We restrict ourselves to finite sequences/signals with finite filters/channels. This enables us to use methods form Linear Algebra. As such, (discrete) convolution, i.e., convolution of finite signals, can be understood as a matrix vector product.

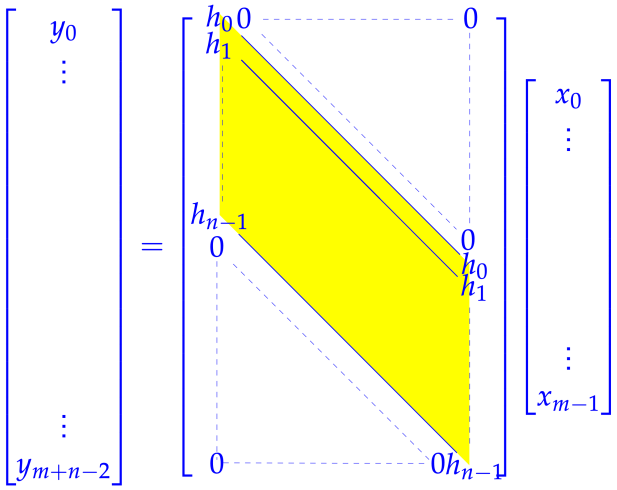


Figure 22: Discrete Convolution

The first column is the zero-extended/-padded impulse response. The columns in the matrix arise from cyclic permutation.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = x_0 \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \\ 0 \\ 0 \\ 0 \end{bmatrix} + x_1 \begin{bmatrix} 0 \\ h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \\ 0 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \\ 0 \end{bmatrix} + x_3 \begin{bmatrix} 0 \\ 0 \\ 0 \\ h_0 \\ h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} \qquad \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} h_0 & 0 & 0 & 0 \\ h_1 & h_0 & 0 & 0 \\ h_2 & h_1 & h_0 & 0 \\ h_3 & h_2 & h_1 & h_0 \\ h_4 & h_3 & h_2 & h_1 \\ 0 & h_4 & h_3 & h_2 \\ 0 & 0 & h_4 & h_3 \\ 0 & 0 & 0 & h_4 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_4 \end{bmatrix}$$

Discrete Convolution (Formula/Definition) Discrete Convolution (Linear Algebra)

Figure 23: Discrete Convolution Example

Definition Discrete Convolution: Given $\mathbf{x} = [x_0, ..., x_{m-1}]^\top \in \mathbb{K}^m$ and $\mathbf{h} = [h_0, ..., h_{n-1}]^\top \in \mathbb{K}^n$, their discrete convolution is the vector $\mathbf{y} \in \mathbb{K}^{m+n-1}$ with components

$$y_k = \sum_{j=0}^{m-1} h_{k-j} x_j, k = 0, ..., 2n-2$$

where we have adopted the convention $h_j := 0$ for $j < 0$ or $j \geq n$.

The notation is as for sequences: $y = h * x = x * h$ with y, h, x being vectors.

Example: (Discrete) Convolutions are not only used for signal processing/applying filters. The multiplication of two polynomials $p(x)$ and $q(x)$ of degree $\leq n-1$ can also be achieved by computing the discrete convolution of their coefficient vectors.

5.1.4 Periodic Convolution

So far we looked at finite, time-discrete, and analog signals. We can omit finiteness for a certain case while still being able to use Linear Algebra. This case are periodic signals.

Definition Periodic Time-Discrete Signal: An n-periodic signal, $n \in \mathbb{N}$, is a sequence $(x_j)_{j\in\mathbb{Z}} \in l^\infty(\mathbb{Z})$ satisfying

$$x_{j+n} = x_j, \forall j \in \mathbb{Z}$$

Such period signals can still be represented by vectors $\mathbf{x} = \begin{bmatrix} x_0 \\ \vdots \\ x_{n-1} \end{bmatrix} \in \mathbb{R}^n$.

If we apply a LT-FIR filter to a periodic signal, the output signal is periodic itself. If we apply a filter with impulse response $(h_k)_{k\in\mathbb{Z}}$ to an n-periodic signal/sequence (x_k) we can compute $(y_k) := F((x_k))$ like this:

$$y_k = \sum_{j\in\mathbb{Z}} h_{k-j}x_j = \sum_{j\in\mathbb{Z}} h_j x_{k-j}$$

$$\overset{j=v+ln}{=} \sum_{v=0}^{n-1}\sum_{l\in\mathbb{Z}} h_{v+ln}x_{k-v-ln} = \sum_{v=0}^{n-1}\left(\sum_{l\in\mathbb{Z}} h_{v+ln}\right)x_{k-v}$$

$$= \sum_{v=0}^{n-1} p_v x_{k-v} \quad \text{with } (p_v) \text{ being a periodic sequence from the impulse response } (h_k)$$

So, the output signal of applying a LT-FIR filter to an n-periodic signal is again an n-periodic signal. We see that we can express this as a discrete periodic convolution of the n-periodic input signal with the n-periodic sequence (p_v) created from the impulse response (h_k). Such discrete periodic convolutions are different from the discrete (non-periodic) convolutions in the size of the output vector.

Definition Discrete Periodic Convolution: The discrete periodic convolution of two n-periodic sequences $(p_k)_{k\in\mathbb{Z}}$, $(x_k)_{k\in\mathbb{Z}}$ yields the n-periodic sequence $(y_k)_{k\in\mathbb{Z}}$.

$$(y_k) := (p_k) *_n (x_k)$$

$$y_k := \sum_{j=0}^{n-1} p_{k-j}x_j = \sum_{j=0}^{n-1} x_{k-j}p_j, k \in \mathbb{Z}$$

Discrete periodic convolutions can be interpreted as applying an LT-FIR filter to a periodic signal. And remembering the LT-FIR filters are linear, leads us to realizing that the periodic convolution is a linear map. Just as the discrete (non-periodic) convolution could be expressed as a matrix multiplication, we can also write this linear map $\mathbb{R}^2 \to \mathbb{R}, p \times x \mapsto y, y := p *_n x$ (with $y, p, x \in \mathbb{R}^n$) as a matrix multiplication.

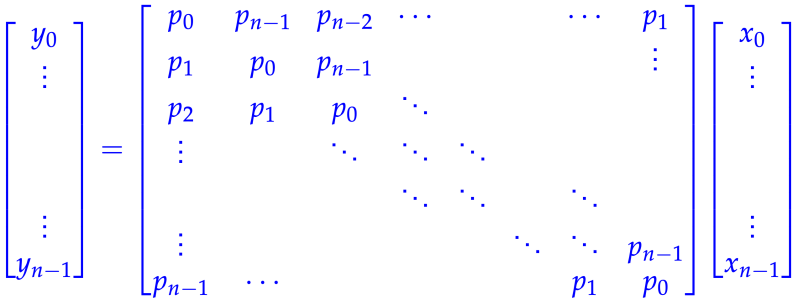


Figure 24: Discrete Periodic Convolution as Matrix-Vector Multiplication

The matrix used for such a discrete periodic convolution has a special form. We call such matrices circulant matrices.

Definition Circulant Matrix: A matrix $\mathbf{C} = [c_{ij}]_{i,j=1}^{n} \in \mathbb{K}^{n,n}$ is circulant

$$:\iff \exists (p_k)_{k\in\mathbb{Z}}\ n\text{-periodic sequence: } c_{ij} = p_{j-i}, 1 \leq i,j \leq n$$

Notice that this definition uses different indices than the above figure.

A circulant matrix is fully defined by its associated n-periodic signal/sequence (p_k).

Convolution by Periodic Convolution So far we have defined discrete (non-periodic) convolutions and discrete periodic convolutions differently. And by how we defined them they are actually different. Specifically, the matrix for discrete (non-periodic) convolutions did not have to be square while it is square with discrete periodic convolutions. However, we can express discrete (non-periodic) convolutions as discrete periodic convolutions.

Consider $a = \begin{bmatrix} a_0 \\ \vdots \\ a_{n-1} \end{bmatrix}, b = \begin{bmatrix} b_0 \\ \vdots \\ b_{n-1} \end{bmatrix}, n \in \mathbb{N}$ and let $z = a * b, z_k = \sum_{j=0}^{n-1} a_{j-k} b_j, k = 0, ..., 2n-2$. The idea to express this discrete (non-periodic) convolution as a periodic one, is to suppress the interactions between different periods by padding the sequences with zeros. Instead of a and b, let's consider $x_k := \begin{cases} a_k, & 0 \leq k < n \\ 0, & n \leq k < 2n-1 \end{cases}$ and $y_k := \begin{cases} b_k, & 0 \leq k < n \\ 0, & n \leq k < 2n-1 \end{cases}$. Those are called the associated $2n-1$ periodic sequences. Then we have $z = a * b = x *_{2n-1} y$.

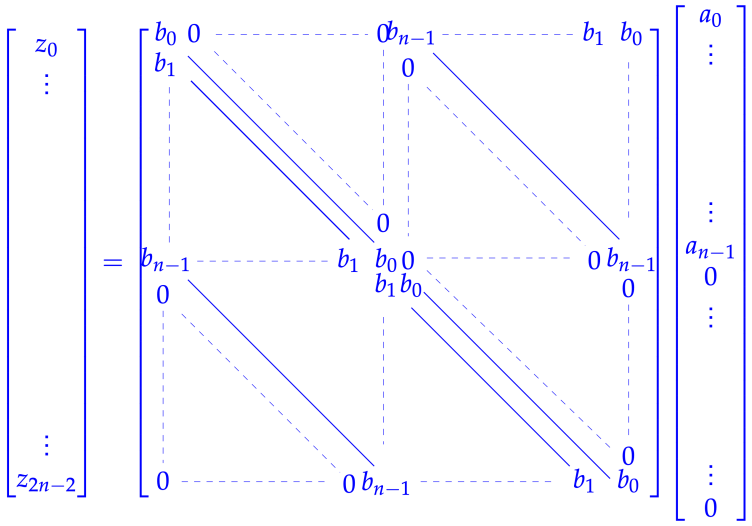


Figure 25: Discrete (Non-Periodic) Convolution by Discrete Periodic Convolution

5.2 Discrete Fourier Transform (DFT)

The Fourier Transform is one of the most important achievements of mathematics and has a huge impact across a very wide range of applications. For instance, without the Fourier transform, modern cellular communication would not be possible as is. The basic idea of the Fourier Transform is, given a signal, to decompose it into pure $\sin$ and $\cos$ waves, which sum up to the actual signal when scaled and added. This can be understood as wanting to find the "ingredients" of some signal. This is very important to transmit information using signals and of utmost importance in signal processing.

Here, we consider the Discrete Fourier Transform (DFT), which deviates from the theoretical Fourier Transform in that its signals are sampled in a time-discrete fashion. We will base our theory of DFS on circulant matrices.

5.2.1 Diagonalizing Circulant Matrices

This section will look at DFT and various of its properties.

Experiment: Eigenvectors of Circulant Matrices This provides some motivation to how we arrive at DFT, what all of it has to do with $\sin$ and $\cos$, the connection to circulant matrices, ... Lets consider a numerical experiment to get a better feeling for the situation. Let $\mathbf{C} =$ `circul(VectorXd::Random(8));` be some arbitrary 8×8 circulant matrix. If we compute the eigenvectors of such $\mathbf{C}$ we make an interesting observation. For all $n \times n$ circulant matrices, the eigenvectors are identical!

If we consider circulant matrices in $\mathbb{R}^{n,n}$, the eigenvectors are usually in the complex numbers and their real and imaginary components remind us of sampled $\cos$ / $\sin$ signals with different frequencies.

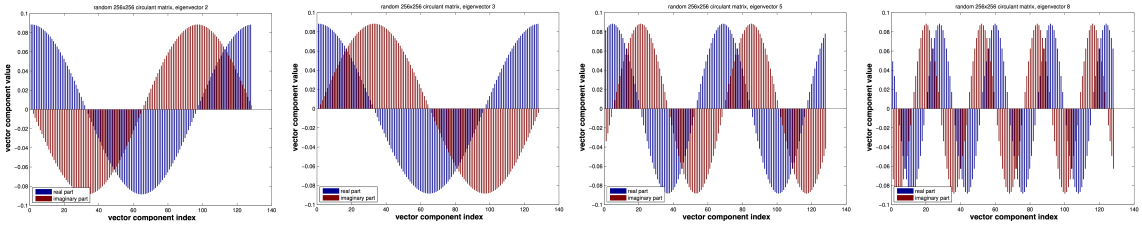


Figure 26: Eigenvectors of Circulant Matrix circul$([1, 2, \ldots, 128]^\top)$

Eigenvectors of Circulant Matrices Now we will continue with a formal treatment. Recall that trigonometric functions are closely linked to complex calculus through

$$e^{it} = \cos t + i \cdot \sin t$$

We will proceed with suggesting a formal definition of eigenvectors for circulant matrices and will afterwards verify that those are indeed the eigenvectors for all circulant matrices. As an answer to the question how to find those eigenvectors: Intuition, ingenuitity, luck, try-and-error, ...

The definition relies on the n-th roots of unity $\sqrt[n]{1}$. But instead of considering the trivial solution 1, we look at $\sqrt[n]{1} = w_n = e^{-\frac{2\pi i}{n}}$. Generally, the n-th roots of unity have many properties:

- $\overline{w}_n = w_n^{-1}$
- $w_n^n = 1$
- $w_n^{\frac{n}{2}} = -1$
- $w_n^k = w_n^{k+n}$
- $\sum_{k=0}^{n-1} w_n^{kj} = \sum_{k=0}^{n-1} (w_n^j)^k = \begin{cases} n, & j = 0 \mod n \\ 0, & j \neq 0 \mod n \end{cases}$

We define

$$\mathbf{v_k} := \left[\mathbf{w_n^{-kj}}\right]_{\mathbf{j=0}}^{\mathbf{n-1}} \in \mathbb{C}^n, \quad k = 0, ..., n-1$$

We will show that $v_k, k = 0, ..., n-1$, are the eigenvectors of any circulant $\mathbf{C} \in \mathbb{C}^{n,n}$.

$$\begin{aligned}
(\mathbf{C}\mathbf{v}_k)_j &= \sum_{l=0}^{n-1} (\mathbf{C})_{j,l} (\mathbf{v}_k)_l = \sum_{l=0}^{n-1} u_{j-l} w_n^{-kl} \\
&\overset{l_{old}=j-l_{new}}{=} \sum_{l=0}^{n-1} u_l w_n^{(l-j)k} = w_n^{-jk} \sum_{l=0}^{n-1} u_l w_n^{lk} \\
&= (\mathbf{v}_k)_j \cdot \lambda_k \quad \text{with } \lambda_k = \sum_{l=0}^{n-1} u_l w_n^{lk}
\end{aligned}$$

This holds for $j = 0, ..., n-1$. Thus, $\mathbf{C}\mathbf{v}_k = \lambda_k \mathbf{v}_k$. This means that with $\mathbf{v}_k$ we have the eigenvectors of $\mathbf{C}$. As there are n different $\mathbf{v}_k$ and at most n eigenvectors, those are all eigenvectors of $\mathbf{C}$. Furthermore, all eigenvectors $\mathbf{v}_k$ are pairwise orthogonal:

$$\mathbf{v}_k^H \mathbf{v}_m = \sum_{j=0}^{n-1} w_n^{kj} w_n^{-mj} = \sum_{n=0}^{n-1} w_n^{(k-m)j} = 0, \quad \text{if } k \neq m$$

However, they for $k = m$ we DON'T have $\mathbf{v}_k^H \mathbf{v}_m = 1$, but $= n$. Thus, this does NOT form an orthonormal basis but only an orthogonal basis. We can and will still use the orthogonal basis to diagonalize $\mathbf{C}$ below.

$$\{\mathbf{v}_0, \dots, \mathbf{v}_{n-1}\} = \left\{ \begin{bmatrix} \omega_n^0 \\ \vdots \\ \omega_n^0 \end{bmatrix}, \begin{bmatrix} \omega_n^0 \\ \omega_n^{-1} \\ \vdots \\ \omega_n^{1-n} \end{bmatrix}, \cdots, \begin{bmatrix} \omega_n^0 \\ \omega_n^{2-n} \\ \omega_n^{2(2-n)} \\ \vdots \\ \omega_n^{-(n-1)(n-2)} \end{bmatrix}, \begin{bmatrix} \omega_n^0 \\ \omega_n^{1-n} \\ \omega_n^{2(1-n)} \\ \vdots \\ \omega_n^{-(n-1)^2} \end{bmatrix} \right\}$$

Figure 27: Orthogonal Eigenvector Basis for Circulant Matrices

If we write the orthogonal vectors $\mathbf{v}_k$ in a matrix, we call that matrix the Fourier matrix.

$$F_n = \begin{bmatrix} \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \cdots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \cdots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_n^0 & \omega_n^{(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix} = \left[\omega_n^{lj}\right]_{l,j=0}^{n-1} \in \mathbb{C}^{n,n}$$

But we can scale the vectors with $\frac{1}{\sqrt{n}}$ so that they have norm 1. Then, we have an orthonormal basis.

Lemma Properties of the Fourier Matrix: The scaled Fourier-matrix $\frac{1}{\sqrt{n}}\mathbf{F}_n$ is unitary:

$$\mathbf{F}_n^{-1} = \frac{1}{n}\mathbf{F}_n^H = \frac{1}{n}\overline{\mathbf{F}}_n$$
$$\Leftrightarrow \frac{1}{\sqrt{n}}\mathbf{F}_n^{-1} = \frac{1}{\sqrt{n}}\mathbf{F}_n^H$$

Lemma Diagonalization of Circulant Matrices: For any circulant matrix $\mathbf{C} \in \mathbb{K}^{n,n}$, $c_{ij} = u_{i-j}$, and $(u_k)_{k\in\mathbb{Z}}$ being an n-periodic sequence, holds true:

$\mathbf{C}\overline{\mathbf{F}}_n = \overline{\mathbf{F}}_n diag(d_1, ..., d_n)$ i.e., eigenvectors are just scaled by their eigenvalue $(d_i = \lambda_i)$

$$\left[d_0, ..., d_{n-1}\right]^\top = \mathbf{F}_n \left[u_0, ..., u_{n-1}\right]^\top$$

Proof. Above we derived that $\mathbf{v}_k$ are the eigenvectors with eigenvalues λ_k. For the eigenvalues we had the expression $\lambda_k = \sum_{l=0}^{n-1} u_l w_n^{lk}$. If we denote $d_i = \lambda_i$, the first statement follows directly from that derivation.

If we perform this derivation again (we skip some steps, see earlier derivation for details) with slightly different notation, we quickly see that the diagonal entries/eigenvalues can be computed as given by the second expression.

$$\mathbf{C}\mathbf{v}_k = \mathbf{C}(\overline{\mathbf{F}}_n)_{:,k} = (\overline{\mathbf{F}}_n)_{:,k} \sum_{l=0}^{n-1} u_l w_n^{lk} = (\overline{\mathbf{F}}_n)_{:,k} \sum_{l=0}^{n-1} u_l (\mathbf{F}_n)_{k,l} = (\overline{\mathbf{F}}_n)_{:,k} (\mathbf{F}_n \mathbf{u})_k$$

Generalizing from one $\mathbf{v}_k$, i.e., one column of $\overline{\mathbf{F}}_n$ leads to the statements of the Lemma. □

With $\overline{\mathbf{F}}_n^{-1} = n\mathbf{F}_n^{-1}$ from the Lemma on "Properties of the Fourier Matrix" follows:

$$\Longrightarrow \mathbf{C} = \mathbf{F}_n^{-1}\mathbf{D}\mathbf{F}_n$$

This is the diagonalization of any circulant matrix $\mathbf{C}$.

Definition Discrete Fourier transform (DFT): The linear map

$$\begin{aligned} \mathrm{DFT}_n &: \mathbb{C}^n \to \mathbb{C}^n \\ \mathrm{DFT}_n(\mathbf{y}) &:= \mathbf{F}_n\mathbf{y}, \quad \mathbf{y} \in \mathbb{C}^n \end{aligned}$$

is called Discrete Fourier Transform (DFT).
For example: $\left[c_0, ..., c_{n-1}\right] := \mathrm{DFT}_n(y)$

$$c_k = \sum_{j=0}^{n-1} y_j w_n^{jk}, \quad k = 0, ..., n-1$$

5.2.2 DFT in Eigen

The implementation of DFT in Eigen isn't optimal. Anyway, implementing DFT in a performant and stable way is tricky. Generally, we should use library functions.

To use DFT in Eigen, we must `#include <unsupported/Eigen/FFT>`.

Listing 23: DFT in Eigen

```
int main() {
  using Comp = complex<double>;
  const VectorXcd::Index n = 5;
  VectorXcd y(n), c(n), x(n);
  y << Comp(1,0), Comp(2,1), Comp(3,2), Comp(4,3), Comp(5,4);
  FFT<double> fft; // DFT transform object
  c = fft.fwd(y);  // DFT of y
  x = fft.inv(c);  // inverse DFT of c

  cout << "y = " << y.transpose() << endl
       << "c = " << c.transpose() << endl
       << "x = " << x.transpose() << endl;
  return 0;
}
```

We will have $x = y$ in the end.

5.2.3 Discrete Convolution via DFT

All operations with circulant matrices can be implemented by means of Fourier matrices/DFT. We will now see how this can be done and discuss the benefits of doing so.

Recall discrete periodic convolution:

$$(y_k) = (p_k) *_n (x_k) \Leftrightarrow y_k = \sum_{j=0}^{n-1} p_{k-j} x_j \text{ with } k = 0, ..., n-1$$

In Eigen, we can implement this as given by the formula in $\mathcal{O}(n^2)$:

Listing 24: Discrete Periodic Convolution in Eigen

```
Eigen::VectorXcd pconv(const Eigen::VectorXcd &u, const Eigen::
    VectorXcd &x) {
  const int n = x.size();
  Eigen::VectorXcd z = VectorXcd::Zero(n);
  for (int k = 0; k<n; ++k) {
    for (int j = 0, l = k; j<= k; ++j, --l)
      z[k] += u[l] * x[j];
    for (int j = k+1, l = n-1; j<n; ++j, --l)
      z[k] += u[l] * x[j];
  }
  return z;
}
```

Now we will so how to compute this convolution with DFT. For this we assume that our implementation of DFT is "ultra fast". Later we will discuss the Fast Fourier Transform and see that this assumption is (somewhat) realistic.

Already in the beginning we have expressed discrete periodic convolutions with circulant matrices. We will do so now again and additionally write the circulant matrix by its DFS.

$$z_k = \sum_{j=0}^{n-1} u_{k-j} x_j \longleftrightarrow \mathbf{z} = \mathbf{C}\mathbf{x}, \quad \mathbf{C} = \left[u_{i-j}\right]_{i,j=0}^{n-1}$$

$$\mathbf{z} = \mathbf{F}_n^{-1} diag(\mathbf{F}_n \mathbf{u}) \mathbf{F}_n \mathbf{x}$$

This reduces the computation to a few multiplications with the Fourier transform and its inverse (being how we construct $\mathbf{F}_n^{-1}$). If this is very fast (by assumption), computing the convolution suddenly also is very fast.

Theorem Convolution Theorem: The discrete periodic convolution $*_n$ between n-dimensional vectors $\mathbf{u}$ and $\mathbf{x}$ is equal to the inverse DFT of the component-wise product between the DFTs of $\mathbf{u}$ and $\mathbf{x}$.

$$(u) *_n (x) := \left[\sum_{j=0}^{n-1} u_{k-j} x_j\right]_{k=0}^{n-1} = F_n^{-1} \left[(F_n u)_j (F_n x)_j\right]_{j=1}^{n}$$

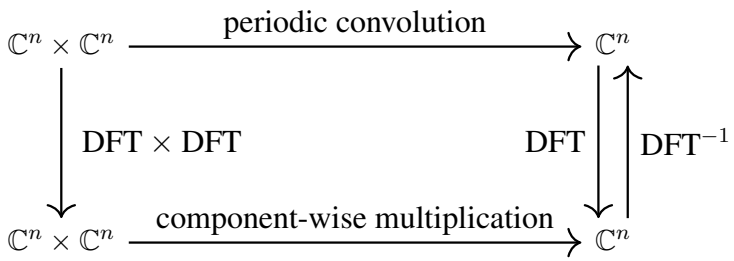


Figure 28: Commuting Diagram for Periodic Convolution with DFT

Implementing convolutions using DFT is easy.

Listing 25: Discrete Periodic Convolutions using DFT in Eigen

```
Eigen::VectorXcd pconvftt(const Eigen::VectorXcd &u, const Eigen
    ::VectorXcd &x) {
  Eigen::FFT<double> fft;
  return fft.inv(((ftt.fwd(u)).cwiseProduct(ftt.fwd(x))).eval())
     ;
}
```

With this technique we can also compute discrete (non-periodic) convolutions by means of zero-extension/-padding.

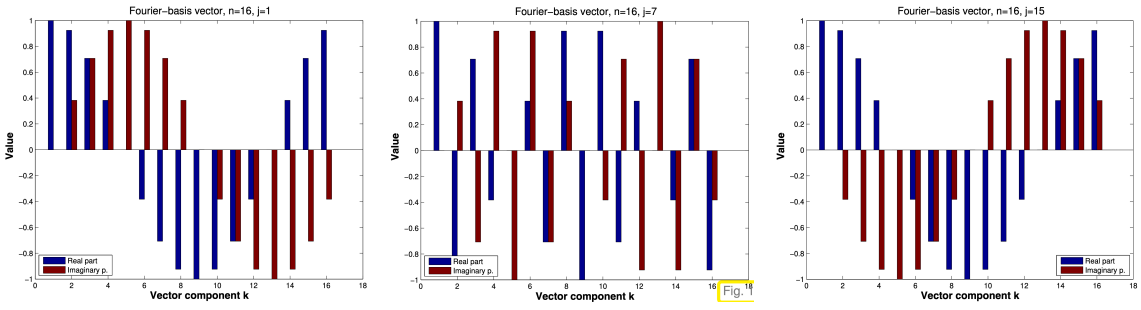
Listing 26: Discrete (Non-Periodic) Convolution using DFT in Eigen

```
Eigen::VectorXcd fastconv(const Eigen::VectorXcd &h, const Eigen
    ::VectorXcd &x) {
  assert(x.size() == h.size());
  const Eigen::Index n = h.size();
  return pconvfft(
    (Eigen::VectorXcd(2 * n - 1) << h, Eigen::VectorXcd::Zero(n
       -1)).finished(),
    (Eigen::VectorXcd(2 * n - 1) << x, Eigen::VectorXcd::Zero(n
       -1)).finished(),
  );
}
```

5.2.4 Frequency Filtering via DFT

Going back to the beginning, circulant matrices correspond to some LT-FIR filter. What we have derived with DFT is: Such circulant matrices correspond to multiplication with $\mathbf{F}_n$, some scaling (with eigenvalues), and some multiplication with $\mathbf{F}_n^{-1} = \frac{1}{n}\overline{\mathbf{F}}_n$. This corresponds to a basis change, scaling in this different basis, and reversal of the basis change. What we will see here is that this intermediate basis in which we perform the scaling has significance (and, thus, is important in many applications).

If we visualize some columns of a Fourier matrix (the intermediate basis), we see that the real and imaginary components of those columns correspond to oscillations with different frequencies.



slow oscillation/low frequency fast oscillation/high frequency slow oscillation/low frequency

Figure 29: Imaginary & Real Components of Columns of $\mathbf{F}_{16}$

We may consider the definition of the Fourier matrix and express its entries by means of $\cos$ and $\sin$. Then we see that the k-th column corresponds to the sampled trigonometric function of frequency k.

$$(\mathbf{F}_n)_{:,k} = [w_n^{kj}]_{j=0}^{n-1} = [e^{\frac{-2\pi i}{n}kj}]_j = \left[\cos\left(\frac{2\pi kj}{n}\right)\right]_j - i\left[\sin\left(\frac{2\pi kj}{n}\right)\right]_j$$

As stated before, applying a filter/circulant matrix corresponds to:

1. computing DFT: $c_k = \sum_{j=0}^{n-1} y_j w_n^{kj}$
2. performing scaling on c_k
3. computing inverse DFT: $y_j = \frac{1}{n}\sum_{k=0}^{n-1} c_k w_n^{-kj}$

The following reveals that the result y_j of the inverse DFT we compute as the last step corresponds to the superposition of oscillations of different frequencies. The values of $\mathbf{c}$, i.e., $|c_k|$, correspond to the strength of different the frequency/mode k in the output signal $\mathbf{y}$. For simplicity in the notation we consider $\mathbf{y} \in \mathbb{R}^n, n = 2m+1$.

$$\begin{aligned}
ny_j &= c_0 + \sum_{k=1}^{m} c_k w_n^{-kj} + \sum_{k=m+1}^{2m} c_k w_n^{-kj} \\
&= c_0 + \sum_{k=1}^{m} c_k w_n^{-kj} + c_{n-k} w_n^{(k-n)j} \\
&= c_0 + \sum_{k=1}^{m} c_k w_n^{-kj} + c_{n-k} w_n^{kj} \quad \text{property of } w_n \text{ from the beginning} \\
&= c_0 + \sum_{k=1}^{m} c_k \left(\cos(2\pi kj/n) + i\sin(2\pi kj/n)\right) + c_{n-k}\left(\cos(2\pi kj/n) + i\sin(-2\pi kj/n)\right) \\
&= c_0 + \sum_{k=1}^{m} c_k \left(\cos(2\pi kj/n) + i\sin(2\pi kj/n)\right) + \overline{c}_k\left(\cos(2\pi kj/n) - i\sin(2\pi kj/n)\right)
\end{aligned}$$

$$\begin{aligned}
&= c_0 + \sum_{k=1}^{m} (Re(c_k) + iIm(c_k))\left(\cos(2\pi kl/n) + i\sin(2\pi kl/n)\right) \\
&\quad + (Re(c_k) - iIm(c_k))\left(\cos(2\pi kl/n) - i\sin(2\pi kl/n)\right)
\end{aligned}$$

$$\begin{aligned}
&= c_0 + \sum_{k=1}^{m} 2Re(c_k)\cos(2\pi kl/n) - 2Im(c_k)\sin(2\pi kl/n) \\
&= c_0 + 2\sum_{k=1}^{m} Re(c_k)\cos(2\pi kj/n) - Im(c_k)\sin(2\pi kj/n)
\end{aligned}$$

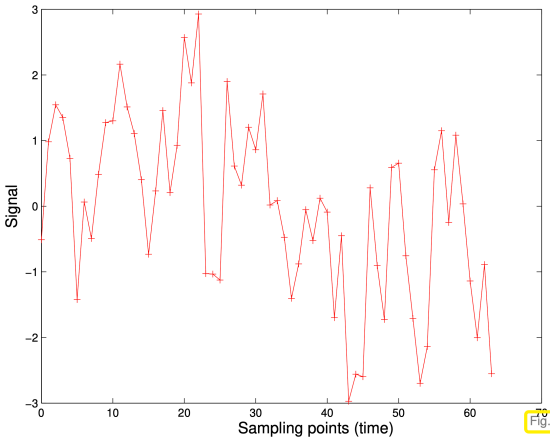
Now that we have understood the connection between DFT, signals, and frequencies, we will continue by discussing how to apply this interpretation of DFT.

Frequency Identification with DFT For illustratory purposes, we will consider an artificially generated signal with two dominant modes/frequencies:

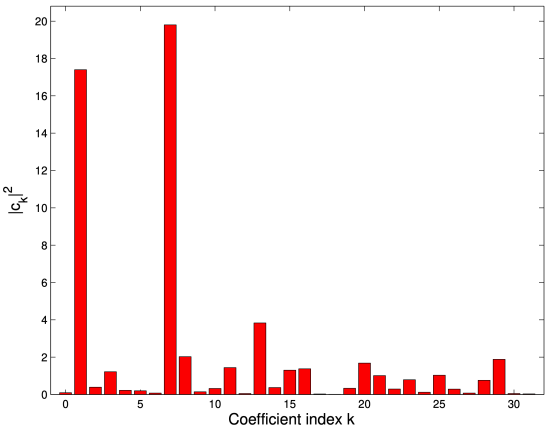
Listing 27: Two Mode Signal

```
VectorXd signalgen() {
  const int N = 64;
  const ArrayXd t = ArrayXd::LinSpaced(N,0,N);
  const VectorXd x = ((2*M_PI/N*t).sin() + (14*M_PI/N*t).sin()).
     matrix();
  return x + VectorXd::Random(N);
}
```

Below we see the raw signal. From it, we can't clearly identify the two dominant frequencies. However, through DFT we get vectors whose content describes the strength of different frequencies. There we clearly see the frequencies corresponding to indices 2 and 7 dominate.



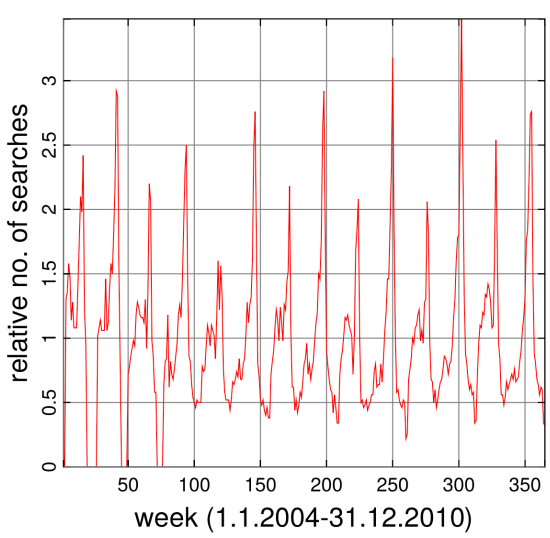
raw randomized two mode signal



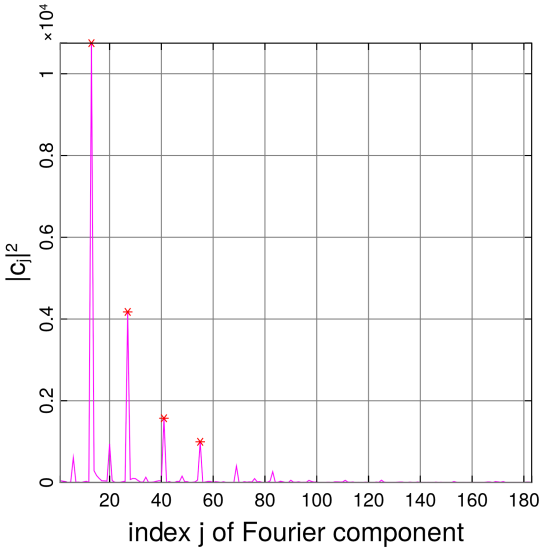
frequency decomposition through DFT

Figure 30: Frequency Identification with DFT

Detecting Periodicity in Data The fact that DFT decomposes a signal into its frequencies allows us to apply DFT to data and see whether there are some dominant frequencies in the dataset. In this example we consider the count of google searches for the term "Vorlesungsverzeichnis" (lecture catalog). The analysis reveals that most requests come at frequencies of 1 term (14 weeks) and 1 semester (1/2 year).



Google Searches "Vorlesungsverzeichnis"



Fourier Spectrum

Figure 31: Frequency of Google Searches for "Vorlesugnsverzeichnis"

'low' and 'high' frequencies The columns of $\mathbf{F}_n$ correspond to the frequencies which we can identify with DFT. And it is sufficient to consider the first half of the frequencies, because the last half columns simply correspond to the complex conjugate of the first half without introducing new frequencies.

$$(\mathbf{F}_n)_{:,n-k} = [w_n^{j(n-k)}]_{j=0}^{n-1} = [w_n^{-jk}]_{j=0}^{n-1} = (\overline{\mathbf{F}}_n)_{:,k}, \quad \text{with } w_n^n = 1$$

We may get a better intuition for this, when considering the frequencies, which are defined by the different n-th roots of unity, on the unit circle.

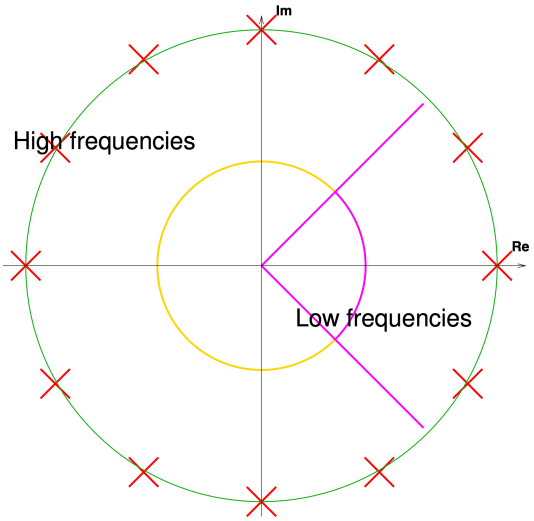


Figure 32: Low and High Frequencies of Fourier Matrices on Unit Circle

Filtering Frequencies Now we have all the necessary background to understand how frequency filtering by using DFT works. We proceed in those steps:

1. start: signal $\mathbf{y}$ (signal in time-domain)
2. apply DFT to $\mathbf{y}$ to get vector $\mathbf{c}$ (frequency-domain)
3. do some operations/filtering on the frequencies to get vectors $\tilde{\mathbf{c}}$
4. apply inverse DFT to get a time-domain signal again

The operations and filtering in step 3 mostly corresponds to very basic numerical operations. If those operation are linear operations, we call it linear filtering.

- low-pass filter: This removes high frequencies but lets low frequencies persist. It sets $c_j = 0$ for high frequencies ($|j - \frac{n}{2}| < k$ for some threshold k)
- high-pass filter: This removes low frequencies but lets high frequencies persist. It sets $c_j = 0$ for low frequencies ($|j - \frac{n}{2}| \geq k$ for some threshold k)

Listing 28: Low- and High-Pass Filter in Eigen

```
void freqfilter(const VectorXd &y, int k, VectorXd &low,
    VectorXd &high) {
  const VectorXd::Index n = y.size();
  if (n%2 != 0)
    throw std::runtiem_error("Even vector length required!");
  const VectorXd::Index m = y.size() / 2;

  Eigen::FFT<double> fft; //DFT helper object
  VectorXcd c = fft.fwd(y);

  VectorXcd clow = c;
  // set high frequency coefficients to zero
  for (int i = -k; j<= k; ++j)
    clow(m+j) = 0;
  // (complementary) vector of high frequency coeffieicnts
  VectorXcd chigh = c - clow;

  // recover filtered time-domain signals
  low = fft.inv(clow).real();
  high = fft.inv(chigh).real();
}
```

Sound Filtering & Compression with DFT When given an audio recording as a frequency, we can perform DFT. The result is the frequency decomposition. In many recordings (such as of the human voice), only a small subset of all frequencies contain important information. Instead of storing the audio file, it is often cheaper to only store the weight for some select frequencies.

With such an approach we might also consider that the human ear can only hear certain frequencies. Frequencies not in that range can be neglected for some good compression.

5.2.5 Two-Dimensional DFT

The idea of DFT can be extended from one to multiple dimensions. For instance, we can consider images (grayscale so that 2D) to be a two-dimensional signal. The applications from the previous subsection (compression, filtering, ...) then can be transferred to such multi-dimensional signals. The core to this transfer is to understand the Fourier transform for mulutiple dimensions.

Matrix Fourier Models We will now discuss how to get from 1D DFT to 2D DFT. In 1D, DFT corresponds to a change of basis so that we are in the frequency-domain instead of the time-domain. For 2D, we take the frequency vectors from 1D and 'blow them up' as tensor products. The resulting rank-1 matrices can be understood to each correspond to one combination of frequency for each dimension.

$$\{(\mathbf{F}_m)_{:,j}(\mathbf{F}_n)_{:,l}^\top\}_{\substack{j=0,\dots,n-1\\ l=0,\dots,n-1}} \subset \mathbb{C}^{m,n}$$

The 2D DFT of $\mathbf{Y} \in \mathbb{C}^{n,n}$ is

$$\mathrm{DFT}_{m,n}(\mathbf{Y}) = \sum_{j=0}^{m-1}\sum_{l=0}^{n-1}(\mathbf{Y})_{j,l}(\mathbf{F}_m)_{:,j}(\mathbf{F}_n)_{:,l}^\top =: \mathbf{C}$$

$$\Rightarrow (\mathbf{C})_{k_1,k_2} = \sum_{j_1=0}^{m-1}\sum_{j_2=0}^{n-1} y_{j_1,j_2} w_m^{j_1k_1} w_n^{j_2k_2} = \sum_{j_1=0}^{m-1} w_m^{j_1k_1}\left(\sum_{j_2=0}^{n-1} w_n^{j_2k_2} y_{j_1,j_2}\right),$$

$$0 \le k_1 < m, 0 \le k_2 < n$$

The part in the brackets corresponds to 1D DFT_n for one row of $\mathbf{Y}$. Then, the outer part computes the 1D DFT_m for the computed DFT/for one column of $\mathbf{Y}$/for one row of $\mathrm{DFT}_n(\mathbf{Y}^\top)$.

$$(\mathbf{C})_{k_1,k_2} = \sum_{j_1=0}^{m-1}(\mathbf{F}_n(\mathbf{Y})_{j_1,:})_{k_2} w_m^{j_1k_1} \Longrightarrow \mathbf{C} = \mathbf{F}_m(\mathbf{F}_n\mathbf{Y}^\top)^\top = \mathbf{F}_m\mathbf{Y}\mathbf{F}_n$$

Knowing the inverse of $\mathbf{F}_i$ also lets use quite easily compute the inverse for 2D DFT:

$$\mathbf{C} = \mathbf{F}_m\mathbf{Y}\mathbf{F}_n \Rightarrow \mathbf{Y} = \mathbf{F}_m^{-1}\mathbf{C}\mathbf{F}_n^{-1} = \frac{1}{mn}\overline{\mathbf{F}}_m\mathbf{C}\overline{\mathbf{F}}_n$$

Listing 29: 2D DFT in Eigen

```
template <typename Scalar>
void fft2(Eigen::MatrixXcd &C,const Eigen::MatrixBaseyScalar> &Y
    ) {
  using idx_t = Eigen::MatrixXcd::Index;
  const idx_t m = Y.rows(), n = Y.cols();
  C.resize(m,n);
  Eigen::MatrixXcd tmp(m,n);

  Eigen::FFT<double> fft;
  for (idx_t k = 0; k<m; k++) {
    Eigen::VectorXcd tv(Y.row(k));
    tmp.row(k) = fft.fwd(tv).transpose();
  }

  for (idx_t k = 0; k<n; k++) {
    Eigen::VectorXcd tv(tmp.col(k));
    C.col(k) = fft.fwd(tv);
  }
}

template <typename Scalar>
```

```
void ifft2(Eigen::MatrixXcd &C, const Eigen::MatrixBase<Scalar>
    &Y) {
  using idx_t = Eigen::MatrixXcd::Index;
  const idx_t m = Y.rows(), n = Y.cols();
  fft2(C, Y.conjugate());
  C = C.conjugate()/(m*n);
}
```

Periodic Convolution of Matrices This paragraph justifies why $\{(\mathbf{F}_m)_{:,j}(\mathbf{F}_n)_{:,l}^\top\}_{\substack{j=0,\dots,n-1\\ l=0,\dots,n-1}} \subset \mathbb{C}^{m,n}$ is the set of eigen"matrices" for all 2D convolutions/for the equivalent of the circulant matrices for 2D DFT.

We assume $\mathbf{Y} = (\mathbf{F}_m)_{:,r}(\mathbf{F}_n)_{s,:} \in \mathbb{C}^{m,n} \longleftrightarrow (\mathbf{Y})_{i,j} = \omega_m^{ri}\omega_n^{sj}$:

$$\begin{aligned}\mathbf{X} \times_{m,n} \mathbf{Y} := (B(\mathbf{X},\mathbf{Y}))_{k,\ell} &= \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\mathbf{X})_{i,j}(\mathbf{Y})_{(k-i) \mod m,(\ell-j) \mod n} \\ &= \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\mathbf{X})_{i,j}\omega_m^{r(k-i)}\omega_n^{s(\ell-j)} \\ &= \left(\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\mathbf{X})_{i,j}\overline{\omega_m^{ir}\omega_n^{js}}\right)\cdot \omega_m^{rk}\omega_n^{s\ell}\end{aligned}$$

$$\implies B(\mathbf{X}, \underbrace{(\mathbf{F}_m)_{:,r}(\mathbf{F}_n)_{s,:}}_{\text{eigenmatrix/eigenvector}}) = \underbrace{\left(\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\mathbf{X})_{i,j}\overline{\omega_m^{ri}\omega_n^{sj}}\right)}_{\text{eigenvalue}}(\mathbf{F}_m)_{:,r}(\mathbf{F}_n)_{s,:}$$

$$\implies B(\mathbf{X}, \underbrace{\overline{(\mathbf{F}_m)_{:,r}(\mathbf{F}_n)_{s,:}}}_{\text{eigenmatrix/eigenvector}}) = \underbrace{\left(\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\mathbf{X})_{i,j}\omega_m^{ri}\omega_n^{sj}\right)}_{\text{eigenvalue}}\overline{(\mathbf{F}_m)_{:,r}(\mathbf{F}_n)_{s,:}}$$

As $\overline{(\mathbf{F}_m)_{:,j}(\mathbf{F}_n)_{:,l}^\top}$ indeed corresponds to the eigenmatrices, we can diagonalize 2D convolution using those basis matrices. Also, this reveals the eigenvalues, which are entries of $\mathrm{DFT}_{m,n}(\mathbf{X})$.

Theorem 2D Convolution Theorem: For any $\mathbf{X}, \mathbf{Y} \in \mathbb{C}^{m,n}$, we have

$$\mathbf{X} *_{m,n} \mathbf{Y} = \mathrm{DFT}_{m,n}^{-1}(\mathrm{DFT}_{m,n}(\mathbf{X}) \odot \mathrm{DFT}_{m,n}(\mathbf{Y}))$$

where $\odot$ stands for the entrywise multiplication of matrices of equal size.

The theorem can be justified analogous to the 1D case. But due to the complexity of dealting with the additional dimensions, we don't do so here.

Listing 30: 2D Convolution in Eigen

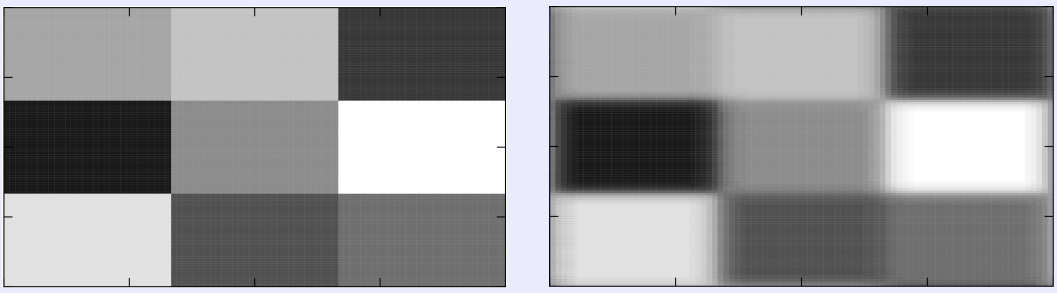
```
template <typename Scalar1, typename Scalar2, class EigenMatrix>
void pmconv(
  const Eigen::MatrixBase<Scalar1> &X,
  const Eigen::MatrixBase<Scalar2> &Y,
  EigenMatrix &
```

```
) {
  using Comp = std::complex<double>;
  using idx_t = typename EigenMatrix::Index;
  using val_t = typename EigenMatrix::Scalar;
  const idx_t n = X.cols(), m = X.rows();
  if ((m != Y.rows()) || (n != Y.cols()))
    throw std::runtime_error("pmconv: size mismatch");
  Z.resize(m, n);
  Eigen::MatrixXcd Xh(m, n), Yh(m, n);
  // Step \ding{202}: 2D DFT of \Blue{$\VY$}
  fft2(Yh, (Y.template cast<Comp>()));
  // Step \ding{203}: 2D DFT of \Blue{$\VX$}
  fft2(Xh, (X.template cast<Comp>()));
  // Steps \ding{204}, \ding{205}: inverse DFT of component-wise
      product
  ifft2(Z, Xh.cwiseProduct(Yh));
}
```

Deblurring by DFT Before we can deblur, we need to understand blurring. Let's consider an image $\mathbf{P} \in \mathbb{R}^{m,n}$ (being a pixel matrix). The blurring operator $\mathcal{B}$ is defined through a point spread function (PSF) $\mathbf{S}_{k,q}$, which identified through a small $2L \times 2L$ matrix indexed through $[-L, L] \times [-L, L]$ ($L \ll n, m$). Applying the blurring operator to some image ($\mathcal{B}(\mathbf{P})$) corresponds to computing each pixel by moving the PSF across the image and using its entries as weights for the new pixel.

$$(\mathbf{C} := \mathcal{B}(\mathbf{P}))_{l,j} := \sum_{k,q=-L}^{L} \mathbf{S}_{k,q}(\mathbf{P})_{(l+k) \mod m,(j+q) \mod n}$$

Example: Let $\mathbf{S}_{k,q} = \frac{1}{1+k^2+q^2}$ and $L = 5$.



Original Image Blurred Image

Figure 33: Blurring Example

Through a lengthy but actually rather easy transformation chain, one can see that blurring

corresponds to a 2D discrete periodic convolution.

$$
\begin{aligned}
c_{l,j} &= \sum_{k=-L}^{L} \sum_{q=-L}^{L} \mathbf{S}_{k,q}(\mathbf{P})_{(l+k) \mod m_i,(j+q) \mod n} \\
&= \sum_{k=-L}^{L} \sum_{q=-L}^{L} \mathbf{S}_{-k,-q}(\mathbf{P})_{(l-k) \mod m_i,(j-q) \mod n} \\
&= \sum_{k=0}^{L} \sum_{q=0}^{L} \mathbf{S}_{-k,-q}(\mathbf{P})_{(l-k) \mod m_i,(j-q) \mod n} \\
&+ \sum_{k=0}^{L} \sum_{q=-L}^{-1} \mathbf{S}_{-k,-q}(\mathbf{P})_{(l-k) \mod m_i,(j-q+n) \mod n} \\
&+ \sum_{k=-L}^{-1} \sum_{q=0}^{L} \mathbf{S}_{-k,-q}(\mathbf{P})_{(l-k+m) \mod m_i,(j-q) \mod n} \\
&+ \sum_{k=-L}^{-1} \sum_{q=-L}^{-1} \mathbf{S}_{-k,-q}(\mathbf{P})_{(l-k+m) \mod m_i,(j-q+n) \mod n} \\
&= \sum_{k=0}^{L} \sum_{q=0}^{L} \mathbf{S}_{-k,-q}(\mathbf{P})_{(l-k) \mod m_i,(j-q) \mod n} \\
&+ \sum_{k=0}^{L} \sum_{q=n-L}^{n-1} \mathbf{S}_{-k,-q+n}(\mathbf{P})_{(l-k) \mod m_i,(j-q) \mod n} \\
&+ \sum_{k=m-L}^{m-1} \sum_{q=0}^{L} \mathbf{S}_{-k+m,-q}(\mathbf{P})_{(l-k) \mod m_i,(j-q) \mod n} \\
&+ \sum_{k=m-L}^{-1} \sum_{q=n-L}^{-1} \mathbf{S}_{-k+m,-q+n}(\mathbf{P})_{(l-k) \mod m_i,(j-q) \mod n}
\end{aligned}
$$

Notice that the form of this is quite similar to:

$$
\mathbf{X} *_{m,n} \mathbf{Y} := B(\mathbf{X}, \mathbf{Y}) = \left[\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (\mathbf{X})_{i,j} (\mathbf{Y})_{(k-i) \mod m,(l-j) \mod n} \right]_{\substack{k=0,\ldots,m-1 \\ l=0,\ldots,n-1}}
$$

with $\mathbf{X}, \mathbf{Y} \in \mathbb{C}^{m,n}$

$\mathbf{P}$ perfectly corresponds to $\mathbf{Y}$. And we can modify $\mathbf{S}$ to correspond to $\mathbf{X}$ (remember $L \ll n, m$):

$$
(\mathbf{S})_{k,q} = \begin{cases} \mathbf{S}_{-k,-q}, & 0 \leq k, q \leq L \\ \mathbf{S}_{-k+m,-q}, & m - L \leq k < m, 0 \leq q \leq L \\ \mathbf{S}_{-k,q+n}, & 0 \leq k \leq L, n - L \leq q < n \\ \mathbf{S}_{-k+m,-q+m}, & m - L \leq k < m, n - L \leq q < n \\ 0, & \text{else} \end{cases}
$$

This corresponds to putting the PSF weights into squares in the corners of $\mathbf{S}$.

If we are given a blurred image $\mathbf{C}$ and the blurring weights $\mathbf{S}$, we can reconstruct $\mathbf{P}$. From the 2D convolution theorem follows:

$$\mathbf{P} = \mathrm{DFT}^{-1}_{m,n}(\mathrm{DFT}_{m,n}(\mathbf{C}) \div_{entrywise\ division} \mathrm{DFT}_{m,n}(\mathbf{S}))$$

We can only perform this deblurring if $\mathrm{DFT}_{m,n}(\mathbf{S})$ is element-wise $\neq 0$. If that doesn't hold, we can intuitively understand the blurring to be such that deblurring isn't possible.

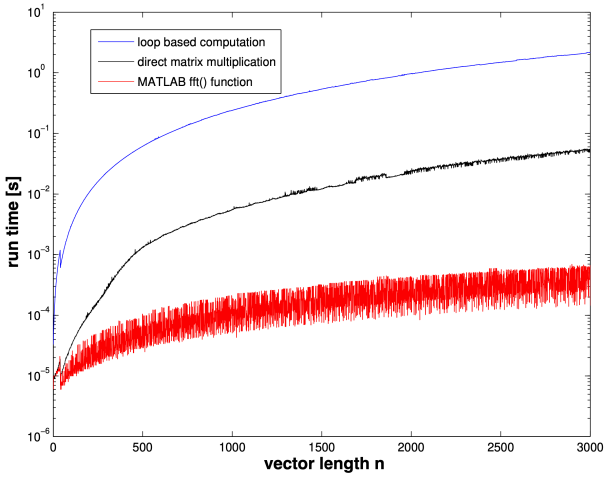
Listing 31: DFT Based Deblurring in Eigen

```
MatrixXd deblur(const MatrixXd &C, const MatrixXd &S, const
    double tol = 1e-3) {
  const long m = C.rows(), n = C.cols(), M = S.rows(), N = S.
     cols();
  const L = (M-1) / 2;
  if (M != N)
    throw std::runtime_error("Error: S not quadratic!");
  MatrixXd Spad = MatrixXd::Zero(m,n);
  // filling S matrix
  Spad.block(0, 0, L+1, L+1) = S.block(L, L, L+1, L+1);
  Spad.block(m-L, n-L, L, L) = S.block(0, 0, L, L);
  Spad.block(0, n-L, L+1, L) = S.block(L, 0, L+1, L);
  Spad.block(m-L, 0, L, L+1) = S.block(0, L, L, L+1);
  // inverse of blurring operator
  MatrixXcd SF = fft2(Spad.cast<complex>());
  // test for invertibility
  if (SF.cwiseAbs().minCoeff() < tol * SF.cwiseAbs().maxCoeff())
    std::cerr << "Error: Deblurring impossible!\n";
  // DFT based deblurring
  return ifft2(fft2(C.cast<complex>()).cwiseQuotient(SF)).real()
     ;
}
```

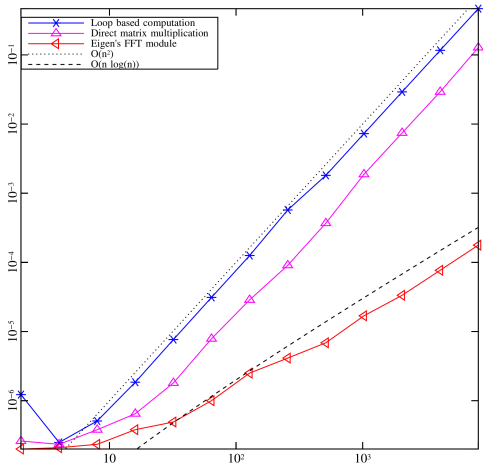
5.3 Fast Fourier Transform (FFT)

In previous sections we have derived methods using DFT. Initially, we assumed that we have some "ultra-fast" DFT implementation. Otherwise, all those methods would not yield performance improvements. In this section we take a look at the Fast Fourier Transform, which provides this fast DFT implementation.

At its core, DFT corresponds to matrix multiplication: $c = \mathbf{F}_n\mathbf{y}$ which can be computed in $\mathcal{O}(n^2)$ for $n \to \infty$. But when looking at different implementations of DFT (manual loops, Eigen matrix multiplication, native FFT function), we see that the performance differs substantially.



Timing in MATLAB



Timing in Eigen

Figure 34: Performance Comparison of different DFT Implementaitons

The reason for the superior performance of the native functions is that they use FFT.

The Fast Fourier Transform (FFT) algorithm was discovered by C.F. Gauss in 1805 and rediscovered by Cooley & Tuckey in 1965. It is considered one of the most influential/important algorithms of the century.

5.3.1 FFT Derivation & Complexity

The basic idea of FFT is divide-and-conquer. We will split the computation of DFT into two smaller DFT computations. Repeating the process of splitting the computation into smaller DFT computations yields FFT. We will now do the formal derivation of one such 'split' with $n = 2m, m \in \mathbb{N}$ for simplicity.

We do the 'split' by separating even from odd indices:

$$c_k = \sum_{j=0}^{n-1} y_j e^{-\frac{2\pi i}{n}jk} = \sum_{j=0}^{m-1} y_{2j} e^{\frac{-2\pi i}{n}2jk} + \sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{n}(2j+1)k}$$

$$= \underbrace{\sum_{j=0}^{m-1} y_{2j} \underbrace{e^{-\frac{2\pi i}{\mathbf{m}}jk}}_{=\omega_m^{jk}}}_{=:\tilde{c}_k^{even}} + e^{-\frac{2\pi i}{n}k} \cdot \underbrace{\sum_{j=0}^{m-1} y_{2j+1} \underbrace{e^{-\frac{2\pi i}{\mathbf{m}}jk}}_{=\omega_m^{jk}}}_{=:\tilde{c}_k^{odd}}$$

$$\Longrightarrow \begin{cases} \tilde{c}_k^{even} = \mathrm{DFT}_m[y_{2j}]_{j=0}^{m-1} \\ \tilde{c}_k^{odd} = \mathrm{DFT}_m[y_{2j+1}]_{j=0}^{m-1} \end{cases}, \quad \text{and } c_k = \tilde{c}_k^{even} + e^{-\frac{2\pi i}{n}k}\tilde{c}_k^{odd}$$

Observe the m-periodicity of $\tilde{c}_k^{odd}$ and $\tilde{c}_k^{even}$.

If $n = 2^k$, we can implement this by just using this recursion:

Listing 32: FFT for $n = 2^k$

```
1 VectorXcd fftrec(const VectorXcd& y) {
2   using idx_t = VectorXcd::Index;
```

```
  using comp = std::complex<double>;
  // nothing to do for DFT of length 1
  const idx_t n = y.size();
  if (n==1) return y;
  if (n%2 != 0)
    throw std::runtime_error("size(y) must be even!");
  const Eigen::Map<const Eigen::Matrix<comp, Eigen::Dynamic,
     Eigen::Dynamic, Eigen::RowMajor>> Y(y.data, n/2, 2);
  const VectorXcd c1 = fftrec(Y.col(0)), c2 = fftrec(Y.col(1));
  const comp omega = std::exp(-2*M_PI/n*comp(0,1));
  comp s(1.0, 0.0);
  VectorXcd c(n);
  // scaling of DFT of odd components plus periodic copying
  for (long k = 0; k<n; ++k) {
    c(k) = c1(k%(n/2)) + c2(k%(n/2))*s;
    s *= omega;
  }
  return c;
}
```

This has the runtime $T(n) = \mathcal{O}(n) + 2 \cdot T(\frac{n}{2})$. That leads to $\mathcal{O}(n \log n)$, which is noticeably better than $\mathcal{O}(n^2)$ of the trivial implementation.

Special Case. Of course, this can be generalized for n not even and $n \neq 2^k, k \in \mathbb{N}$. One special generalization is for $n = pq$ where n is not prime.

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{jk} \overset{[j=:lp+m]}{=} \sum_{m=0}^{p-1} \sum_{l=0}^{q-1} y_{lp+m} e^{-\frac{2\pi i}{pq}(lp+m)k} = \sum_{m=0}^{p-q} \omega_n^{mk} \sum_{l=0}^{q-1} y_{lp+m} \omega_q^{l(k \mod q)}$$

We compute c_k not directly but in two steps:
1.: $\sum_{l=0}^{q-1} y_{lp+m} w_q^{l(k \mod q)}$ corresponds to some DFT_q. We compute p DFTs of length q.

$$z_m := \text{DFT}_q[y_{lp+m}]_{l=0}^{q-1}$$
$$(z_m)_k = z_{m,k} := \sum_{l=0}^{q-1} y_{lp+m} w_q^{lk}, \quad 0 \leq m < p, 0 \leq k < q$$

2.: With $k = rq + s$ $(0 \leq r < p, 0 \leq s < q)$ we can compute c_k:

$$c_{rq+s} = \sum_{m=0}^{p-1} e^{-\frac{2\pi i}{pq}(rq+s)m} z_{m,s} = \sum_{m=0}^{p-1} (w_n^{ms} z_{m,s}) w_p^{mr}, \quad 0 \leq r < p, 0 \leq s < q-1$$

This is strongly related to 2D DFT. There, we split DFT to separate DFTs for each dimension. Here, we also split the DFT so that we can consider two DFTs of smaller dimension.

Runtime The execution time for fft depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors. This explains the high variability of execution time/performance seen in the introductory runtime observations in MATLAB. It usually pays off to use DFTs on vectors of length of powers of two even if it requires to artificially length 'data' vectors noticeably.

To actually achieve great performance, this approach has to be implemented in a smart way (considering caches, computer architecture, compiler optimizations, ...). Thus, don't lightly implement DFT/FFT by yourself. Generally, use high-quality numerical libraries.

6 Data Interpolation and Data Fitting in 1D

Our goal here is to perform ***one-dimensional interpolation***:
We are given data points $(t_i, y_i), i = 0, ..., n, n \in \mathbb{N}, t_i \in I \subset \mathbb{R}, y_i \in \mathbb{R}$. t_i are called nodes and y_i are called values.
Our objective is the reconstruction of a function $f : I \to \mathbb{R}$:

- satisfying the $n + 1$ interpolation conditions (IC) $f(t_i) = y_i, i = 0, ..., n$
- and belonging to a set V of eligible functions.

The function f is called the interpolant of the given data set $\{(t_i, y_i)\}_{i=0}^{n}$.

Through V one can add additional constraints on the model for the data set. Such constraints may include: Smoothness, differentiability, periodicity, behavior towards limits, ...

1D interpolation is important for numerical applications. Given a few precise measurement pairs (voltage & current, pressure & density, ...), the task of finding an interpolant means to model the relationship functionally. A functional model/relation $y = f(t)$ is needed for many numerical analysis. Of course, math doesn't restrict us to one dimension. But we restrict ourselves to 1D here for simplicity.

Functions in Code When implementing 1D interpolation, we usually do so with functors (at least in C++).

Listing 33: 1D Interpolation Functors

```
class Interpolant {
  private:
    // various internal data describing f
    // can be the coefficients of a basis representation
  public:
    // constructor: computation of coefficients of
       representation
    Interpolant(const vector<double>& t, const vector<double>& y
       );
    // evaluation operator for interpolant f
    double operand() (double t) const;
}
```

Various methods may allow for special functionality such as adding individual data points with an additional method. But this is the basic framework we will target when implementing different 1D interpolation methods considered below.

6.1 Abstract Interpolation

Here, we will consider how to compute an internal representation of the function given a set of data points. We consider the special case of eligible functions being from a vector space:

$$V = span\{\{t \to b_0(t)\}, ..., \{t \to b_m(t)\}\} = span\{b_0, ..., b_m\}$$

We call those functions $b_0, ..., b_m$ the basis functions and generally assume that they are linearly independent. (If they are not, just remove linearly dependent ones.) $\dim V = m + 1$. The

basis property guarantees that every $f \in V$ has a unique representation in the basis functions: $\exists \alpha_j \in \mathbb{R}$ so that $f(t) = \sum_{j=0}^{n} \alpha_j b_j(t)$. The α-values are called basis expansion coefficients.

Finding those coefficients α can be done by solving a linear system of equations. For instance if we only have finitely many b_i, the system may be overdetermined so that we can only approximate.

$$\mathbf{Ac} := \begin{bmatrix} b_0(t_0) & \dots & b_m(t_0) \\ \vdots & \vdots & \vdots \\ b_0(t_n) & \dots & b_m(t_n) \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \vdots \\ \alpha_m \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix} =: \mathbf{y}$$

Again, a unique solution only exists if $m = n$ and $\mathbf{A}$ is regular. Then: $f(t) = \sum_{j=0}^{n} (\mathbf{A}^{-1}\mathbf{y})_j b_j(t)$.

We can understand interpolation for some basis through a **linear interpolation operator**: $\begin{cases} \mathbb{R}^{n+1} \to V \\ y \mapsto f \end{cases}$. Linear interpolation corresponds to a linear map.

Definition Linear Interpolation Operator: An interpolation operator $I : \mathbb{R}^{n+1} \mapsto C^0([t_0, t_m])$ for the given nodes $t_0 < t_1 < ... < t_n$ is called linear, if

$$I(\alpha y + \beta z) = \alpha I(y) + \beta I(z), \forall y, z \in \mathbb{R}^{n+1}, \alpha, \beta \in \mathbb{R}$$

C^0 are the set of continuous function. This contains the V we considered before.

A basis for which $A = I \Leftrightarrow b_i(t_j) = \delta_{ij}$ holds is called a **cardinal basis**. Then: $f = \sum_{j=0}^{n} y_j b_j$.

Example Piecewise Linear Interpolation:
This is a simple example of a cardinal basis. For V let:

$$V = \left\{ f \in C^0([t_0, t_m]) | \begin{cases} f(t) = \alpha_i + \beta_i t, & \alpha_i, \beta_i \in \mathbb{R} \quad , t \in [t_i, t_{i+1}] \\ f(t) = 0, & \quad , \text{otherwise} \end{cases} \right\}$$

A simple cardinal basis for which this holds are the "tent" functions:

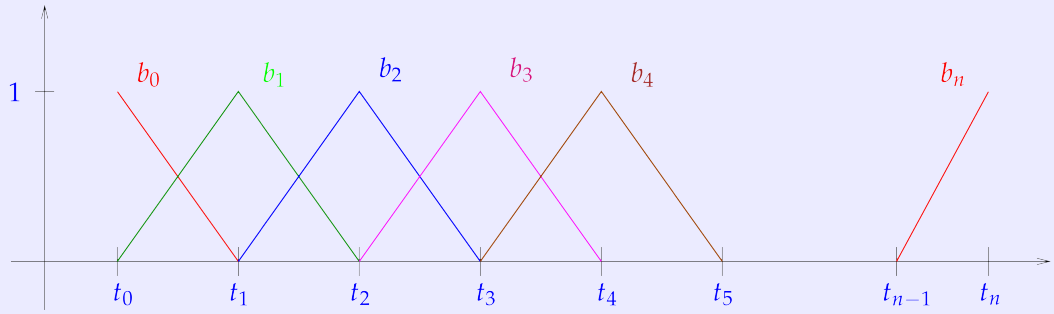


Figure 35: Cardinal Basis - Tent Functions

6.2 Global Polynomial Interpolation

In this section we look at how interpolation in 1D works if we want to model our data set with polynomials. There are many reasons why we want to work with polynomials:

- Polynomials form a vector space. Thus, we can easily compute with them.

- Multiplying two polynomials has a very predictable effect on the form of the resulting polynomial. Specifically, the new degree is the sum of the two input degrees. (Similarly for other operations.)
- Polynomials are easy to derive and integrate.
- Polynomials appear in Taylor polynomials with suggests that we can use them well for approximations.
- Polynomials are easy to evaluate.

Polynomial interpolation is done using the **Horner scheme**. Let $p \in \mathcal{P}_n$ be in monomial representation. Thus, it is represented by a vector of its monomial coefficients. We can compute $p(t)$ with cost $\mathcal{O}(n)$ using the Horner scheme. This algorithm becomes apparent when writing the polynomial in special form:

$$p(t) = t(\dots t(t(\alpha_n t + \alpha_{n-1}) + \alpha_{n-2}) + \dots + \alpha - 1) + \alpha_0$$

Listing 34: Horner Scheme in C++

```
// IN: p = vector of monomial coefficients starting with leading
     coefficient, length = degree + 1
// IN: t = vector of evaluation points  t_i
// OUT: vector of values of polynomial evaluated at t_i
Eigen::VectorXd horner(const Eigen::VectorXd &p, const Eigen::
    VectorXd &t) {
  const VectorXd::Index n = t.size();
  Eigen::VectorXd y{p[0] * VectorXd::Ones(n)};
  for (unsigned i = 1; i<p.size(); ++i)
    y = t.cwiseProduct(y) + p[i] * VectorXd::Ones(n);
  return y;
}
```

6.2.1 Uni-Variate Polynomials

Polynomials in 1D are functions $\mathbb{R} \to \mathbb{R}$ and written as linear combinations on monomials. For polynomials up to including degree k we have:

$$\mathcal{P}_k := \{t \mapsto \alpha_k t^k + \alpha_{k-1} t^{k-1} + \dots + \alpha_1 t + \alpha_0, \alpha_j \in \mathbb{R}\}$$

It is already known that polynomials form a vector space. Here, we will consider interpolation with V as the set of uni-variate polynomials (i.e., polynomials with one variable). The monomial basis of $\mathcal{P}_k$ is the set of monomials $\{t \mapsto t^l\}_{l=0}^k$. We immediately see that $\dim \mathcal{P}_k = k + 1$.

6.2.2 Lagrange Polynomial Interpolation Problem

Definition Lagrange Polynomial Interpolation Problem: Given the set of interpolation nodes $\{t_0, \dots, t_n\} \subset \mathbb{R}, n \in \mathbb{N}$, and the values $y_0, \dots, y_n \in \mathbb{R}$, compute $p \in \mathcal{P}_n$ such that it satisfies the interpolation conditions (IC)

$$p(t_j) = y_j \text{ for } j = 0, \dots, n$$

Existence and uniqueness follows from considering a cardinal basis $\{L_i\}_{i=0}^n$ of $\mathcal{P}_n$: $L_i(t_j) = \delta_{ij}$ for $i, j \in \{0, ..., n\}$. Such a cardinal basis is given by the Lagrange polynomial:

$$L_i(t) = \frac{(t-t_0)\dots(t-t_{i-1})(t-t_{i+1})\dots(t-t_n)}{(t_i-t_0)\dots(t_i-t_{i-1})(t_i-t_{i+1})\dots(t_i-t_n)}$$

As this is a cardinal basis (which can be easily seen and of which the proof is omitted here), we get $p(t) = \sum_{i=0}^n y_i L_i(t)$. The interpolation operator $\begin{cases} \mathcal{P}_n \to \mathbb{R}^{n+1} \\ p \mapsto [p(t_j)]_{j=0}^n \end{cases}$ is clearly a linear mapping. As we can generate any vector in $\mathbb{R}^{n+1}$ by appropriately choosing $p \in \mathcal{P}_n$, this mapping is surjective. But as $\dim \mathcal{P}_n = \dim \mathbb{R}^{n+1}$ the mapping then must also be bijective. This implies existence and uniqueness of a solution.

Theorem Existence & Uniqueness of Lagrange Interpolation Polynomial: The general Lagrange polynomial interpolation problem admits a unique solution $p \in \mathcal{P}_n$ for any (t_i, y_i).

6.2.3 Polynomial Interpolation Algorithms

Polynomial interpolation is relevant in different use cases with different requirements. We may be given some initial data set but then want to evaluate the interpolating polynomial in multiple points. Or we may only want to interpolate in the same point. Or we may want to continue adding data points to the data set while evaluating only at one point/in various points. Or ...

According to this different requirements, there are different algorithms for polynomial interpolation which optimize the runtime for specific use cases.

Multiple Evaluations

- fixed set of nodes $\{t_0, \dots, t_n\}$ for $i = 0, ..., n$
- many different data values y_i for $i = 0, ..., n$
- many arguments x_k for $k = 1, ..., N$ with $N \gg 1$

We should efficiently compute all $p(x_k)$ for $p \in \mathcal{P}_n$ interpolating in (t_i, y_i) for $i = 0, ..., n$.

Listing 35: Multiple Evaluations Interpolation C++ Interface

```
class PolyInterp {
  private:
    Eigen::VectorXd t;
  public:
    PolyInterp(const Eigen::VectorXd &_t);
    // evaluation
    Eigen::VectorXd eval(const Eigen::VectorXd &y, const Eigen::
       VectorXd &x) const;
}
```

A straightforward implementation takes the Lagrange polynomials and computes $p(x_k) = \sum_{i=0}^n y_i L_i(x_k)$ with the Lagrange polynomial $L_i(t) = \prod_{\substack{j=0 \\ j \neq i}}^n \frac{t-t_j}{t_i-t_j}$. Evaluating a single Lagrange polynomial has cost n, which we must do n times to compute $p(x_k)$. As there are N x_k values, we get the runtime $\mathcal{O}(n^2 N)$.

But this does a lot of unnecessary computations. We will precompute some of this and reuse it to improve asymptotic runtime. First, we will precompute:

$$\lambda_i = \frac{1}{(t_i - t_0)\dots(t_i - t_{i-1})(t_i - t_{i+1})\dots(t_i - t_n)}, \quad i = 0, ..., n$$

According to the lectures, this can be done in $\mathcal{O}(n^2)$. But it should also be possible in $\mathcal{O}(n)$.

$$p(t) = \sum_{i=0}^{n} L_i(t) y_i = \sum_{i=0}^{n} \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{t - t_j}{t_i - t_j} y_j = \sum_{i=0}^{n} \prod_{\substack{j=0 \\ j\neq i}}^{n} \lambda_i (t - t_j) y_j$$
$$= \left[\prod_{j=0}^{n} (t - t_j)\right] \sum_{i=0}^{n} \frac{\lambda_i}{t - t_i} y_i = \frac{\sum_{i=0}^{n} \frac{\lambda_i}{t-t_i} y_i}{\sum_{i=0}^{n} \frac{\lambda_i}{t-t_i}}$$

where we use $1 = \prod_{j=0}^{n}(t - t_j) \sum_{i=0}^{n} \frac{\lambda_i}{t-t_i} \Rightarrow \prod_{j=0}^{n}(t - t_j) = \frac{1}{\sum_{i=0}^{n} \frac{\lambda_i}{t-t_i}}$. This is called the **Barycentric interpolation formula**.

- Construction/initialization requires computing λ_i in $\mathcal{O}(n^2)$.
- Evaluation requires evaluating the Barycentric interpolation formula which can be done in $\mathcal{O}(Nn)$.

Listing 36: Multiple Evaluations Interpolation

```
template<typename NODESCALAR>
template<typename RESVEC, typename DATAVEQ>
RESVEC BarycPolyInterp<NODESCALAR>::eval(const DATAVEC &y, const
    nodeVec_t &x) const {
  const idx_t N = x.size();
  RESVEC p(N);
  for(int i = 0; i<N; ++i) {
    nodeVec_t z = (x[i] * nodeVec_t::Ones(n) - t);
    // check if we want to evaluate close to a node
    const double tref{z.cwiseAbs().maxCoeff()};
    idx_t k;
    if (z.cwiseAbs().minCoeff(&k) < tref * std::abs(std::
       numeric_limits<NODESCALAR>::epsilon())) {
      p[i] = y[k];
    } else {
      const ndoeVec_t mu = lambda.cwiseQuotient(z);
      p[i] = (mu.cwiseProduct(y)).sum() / mu.sum();
    }
  }
  return p;
}
```

We check $z \approx 0$ elementwise numerically, to avoid division by zero for stability/to avoid cancellation. If $z \approx 0$ for some index k, then we just return y_k.

Single Evaluation

- fixed set of nodes $\{t_0, \ldots, t_n\}$ with one set of data values y_i for $i = 0, \ldots, n$

  We will extend this algorithm this in the end so that the set of data points is extendable.

- one argument x

We should efficiently compute all $p(x_k)$ for $p \in \mathcal{P}_n$ interpolating (t_i, y_i) for $i = 0, \ldots, n$.

Listing 37: Single Evaluation Interpolation C++ Interfact

```
class PolyInterp {
  private:
  public:
    double eval(const Eigen::VectorXd &t, const Eigen::VectorXd
       &y, double x);
}
```

The approach for this is called partial polynomial interpolation. Let $p_{k,l} \in \mathcal{P}_{l-k}$ be defined as $p_{k,l}(t_j) = y_j, j = k, ..., l$ to interpolate a certain 'subset' of the given data points. Then, we compute p recurisvely.

- base case: $p_{k,k}(x) \equiv y_k$ (constant polynomial)
- $p_{k,l}(x) = \frac{(x-t_k)p_{k+1,l}(x)-(x-t_l)p_{k,l-1}(x)}{t_l-t_k} = p_{k+1,l}(x) + \frac{x-t_l}{t_l-t_k}(p_{k+1,l}(x) - p_{k,l-1}(x))$ for $0 \leq k \leq l \leq n$

Althrough we require $0 \leq k \leq l \leq n$ there seems no reason why the points should be ordered by their x-coordinate, i.e., by nothing it seems necessary that $t_0 \leq t_1 \leq t_2 \leq \cdots \leq t_n$ must hold. The correctness of partial polynomial interpolation, i.e., that the resulting p satisfies the interpolation conditions, can be shown by induction.

The evaluation according to this recursion can be done with the Aitken-Neville scheme/algorithm for some point x.

Listing 38: Aitken-Neville Scheme Evaluation in C++

```
double ANipoleval(const VectorXd& t, VectorXd y,const double x){
  for (int i = 0; i<y.size(); ++i) {
    for (int k = i-1; k >= 0; --k) {
      y[k] = y[k + 1] + (y[k + 1]-y[k])*(x-t[i])/(t[i]-t[k]);
    }
  }
  return y[0];
}
```

$n =$	0	1	2	3
t_0	$y_0 =: p_{0,0}(x)$	$\nearrow p_{0,1}(x)$	$\nearrow p_{0,2}(x)$	$\nearrow p_{0,3}(x)$
t_1	$y_1 =: p_{1,1}(x)$	$\nearrow p_{1,2}(x)$	$\nearrow p_{1,3}(x)$	
t_2	$y_2 =: p_{2,2}(x)$	$\nearrow p_{2,3}(x)$		
t_3	$y_3 =: p_{3,3}(x)$			

Figure 36: Aitken-Neville Scheme

The computational cost is $\mathcal{O}(n^2)$ with memory $\mathcal{O}(n)$. The polynomial interpolant is never directly computed. For a single evaluation this is not asymptotically faster compared to the approach using the Barycentric interpolation formula. However, we can very simply extend this so that we can add more data points. We will simply consider an additional y_i and run one more iteration of the Aitken Neville scheme. This can be done in $\mathcal{O}(n)$. Furthermore, evaluation then is trivially $\mathcal{O}(1)$ at any point. Memory utilization stays at $\mathcal{O}(n)$, which is very efficient.

Listing 39: Partial Polynomial Interpolation and Atiken-Neville based Interpolation

```
class PolyEval {
  private:
  public:
    // constructor taking the evaluation points as argument
    PolyEval(double x);
    // add another data point and update internal information
    void addPoint(t,y);
    // value of current interpolating polynomial at x
    double eval(void) const;
}
```

Extrapolation to Zero This is a somewhat special case of interpolation. The goal is to evaluate some function as some specific point. Without loss of generality we can assume that we want to compute $\phi(0)$ for some smooth $\phi : [-1, 1] \to \mathbb{R}$. We have a function in procedural form (i.e., a C++ function `double psi(double h)`). But this function is not reliable for $|h| \ll 1$. Thus, we want to interpolate the function ϕ so that it is most accurate at $x = 0$. To do so, we can use an arbitrary data set which we gain from evaluating ϕ for stable points.

> **Example**: Let $f \in C^2$ and $\psi(h) := \frac{f(x+h)-f(x-h)}{2h}$ the difference quotient that approximates the derivative at $x = 0$. We can't evaluate this for $h = 0$ and due to cancellation evaluations close to zero will be unreliable.

Here we think that polynomial approximation is a good approach after looking at the Taylor approximation. For small h, the remainder will be zero and the polynomial approximation will be close to perfect.

$$\phi(h) = \sum_{l=0}^{L} \frac{f^{(k)}(a)}{k!} h^l + \mathcal{O}(h^{L+1}) = \sum_{l=0}^{L} c_l h^l + \mathcal{O}(h^{L+1})$$

One question remains: How to get this approximation polynomial. The idea to approximate this polynomial so that we get a close to perfect approximation for $h = 0$ is:

1. Pick $h_0, \ldots, h_n$ for which ϕ can be evaluated safely.
2. Evaluate ϕ for all those h_i.
3. Then we conclude $\phi(0) \approx p(0)$ with the **extra**polating polynomial $p \in \mathcal{P}_n$.

   We say extrapolation here, because the evaluation point is not in the interval covered by the nodes.

We compute $p(0)$ using the Aitken-Neville scheme. As discussed, it is update friendly for a single evaluation so that we can continue to add h_i until we are satisfied with our approximation

(if the procedural ϕ allows further evaluations). Alternatively and as implemented below, we can define an upper bound for the data set size and just terminate before the entire set is considered if the error between iterations vanishes below some absolute or relative tolerance.

Listing 40: Extrapolation to Zero with Aitken-Neville

```
template <class Function>
double diffex(Function& f, const double x, const double h0,
    const double rtol, const double atol) {
  const unsigned nit = 10;
  VectorXd h(nit); h[0] = h0;
  VectorXd y(nit);
  y[0] = (f(x+h0)-f(x-h0))/(2*h0);

  for (unsigned i = 1; i<nit; ++i) {
    h[i] = h[i-1]/2;
    y[i] = (f(x+h[i])--f(x-h[i]))/(2.0 * h[i])
    for (int k = i-1; k>=0; --k)
      y[k] = y[k+1] - (y[k+1]-y[k]) * h[i]/(h[i]-h[k]);
    cosnt double errest = std::abs(y[i]-y[i-1]);
    if (errest < rtol*std::abs(y[i-1]) || errest < atol)
      break;
  }
  return y[0];
}
```

Compared to directly evaluating (especially for computing derivatives as illustrated in the example), on the one hand, this enables much higher accuracy. But on the other hand, it also allows for much better control over the error so that we can stop early if we don't require a high accuracy but can also keep going for quite long if we want the best possible accuracy.

Newton Basis and Divided Differences This may be the most general case of all.

- extendable set of nodes $\{t_0, \ldots, t_n\}$ with data values $\{y_0, \ldots, y_n\}$ for $i = 0, \ldots, n$
- arbitrary arguments x for evaluation

Listing 41: Newton Basis and Divided Differences Interpolation Interface in C++

```
class PolyEval {
  private:
  public:
    PolyEval();
    void addPoint(t,v);
    Eigen::VectorXd operator() (const Eigen::VectorXd &x) const;
}
```

So far we considered two approaches. Aitken-Neville only enabled evaluation at one point. The Barycentric interpolation formula requires recomputing all coefficients, i.e., repeat the entire setup phase, if an additional data point is added. To avoid this, we introduce a new basis for the space of polynomials which is update friendly: The Newton basis.

- $N_0(t) = 1$

- $N_k(t) = \prod_{i=0}^{k-1}(t - t_i)$ for $k = 1, \ldots, n$

$\{N_0, \ldots, N_n\}$ is a basis of $\mathcal{P}_n$. The polynomials of the basis are linearly independent, because all polynomials have different degree. We consider this basis to be update friendly because of its property $N_k(t_j) = 0$ for $j = 0, \ldots, k-1$. The interpolating conditions (IC) for this basis (as for any basis) are expressed through:

$$
\begin{aligned}
&a_0N_0(t_j) + a_1N_1(t_j) + \cdots + a_nN_n(t_j) = y_j, \quad j = 0, ..., n \\
\implies &\begin{bmatrix} N_0(t_0) & N_1(t_0) & \ldots & N_n(t_0) \\ N_0(t_1) & N_1(t_1) & \ldots & N_n(t_1) \\ \vdots & \vdots & & \vdots \\ N_0(t_n) & N_1(t_n) & \ldots & N_n(t_n) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_0 \\ \vdots \\ y_n \end{bmatrix} \\
\implies &\begin{bmatrix} 1 & 0 & \ldots & 0 \\ 1 & (t_1 - t_0) & \ldots & \vdots \\ \vdots & \vdots & \vdots & 0 \\ 1 & (t_n - t_0) & \ldots & \prod_{i=0}^{n-1}(t_n - t_i) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}
\end{aligned}
$$

From this we can directly see that a_k only depends on (t_j, y_j) with $0 \leq j \leq k$. There is no change of $a_0, \ldots, a_n$ if (t_{n+1}, y_{n+1}) is added.

Efficient Evaluation of a Polynomial in Newton Form If we want to work with this basis, one task we can't avoid is evaluating the polynomial for some x given a set of coefficients. Evaluation is possible similar to the Horner scheme in an analogous special form:

$$
\begin{aligned}
p(t) &= a_0N_0(t) + a_1N_1(t) + \cdots + a_nN_n(t) \\
&= a_0 + a_1(t - t_0) + a_2(t - t_0)(t - t_1) + \cdots + a_n \prod_{j=0}^{n-1}(t - t_j), t \in \mathbb{R} \\
&= a_0 + (((a_n(t - t_n) + a_{n-1})(t - t_{n-1}) + a_{n-2}) \cdots + a_1)(t - t_1)
\end{aligned}
$$

Evaluation for one point is $\mathcal{O}(n)$. Evaluating k points takes $\mathcal{O}(nk)$.

Listing 42: Evaluating Polynomial in Newton Form

```
Eigen::VectorXd evalNewtonForm(
  const Eigen::VectorXd &t,
  const Eigen::VectorXd &a,
  const Eigen::VectorXd &x
) {
  const unsigned int n = a.size() - 1;
  const Eigen::VectorXd ones = VectorXd::Ones(x.size());
  Eigen::VectorXd p{a[n] * ones*}
  for (int j = n-1; j>=0; --j) {
    p = (x- t[j]*ones).cwiseProduct(p) + a[j]*ones;
  }
  return p;
}
```

Divided Differences Algorithm Another unavoidable task is to compute the coefficients of the polynomial in Newton basis given some data points and to extend the coefficients when given additional data points. We need to find a_j so that $p(t) = \sum_{j=0}^{n} a_j N_j(t)$.

This algorithm employs an approach similar to partial interpolation as discussed before. Notice that N_j has degree j and leading coefficient 1. No other N_i has degree j. Thus, a_j is the leading coefficient of the partial interpolating polynomial $p_{0,j}$. If we continue and add more points, we have already argued that a_j will remain unchanged. So its safe to compute a_j as the leading coefficient of $p_{0,j}$. We will use the following notation to denote leading coefficients from now on:

$$\mathbf{y}\left[t_l, \ldots, t_m\right] \text{ for the leading coefficient of } p_{l,m}$$
$$\text{specifically: } a_j = \mathbf{y}\left[t_0, ..., t_j\right]$$

Partial interpolation reminds us of the Aitken-Neville scheme. We will use its approach to compute the leading coefficients.

- $y[t_i] = y_i$
- $y[t_i, ..., t_{i+k}] = \frac{y[t_{i+1},...,t_{i+k}] - y[t_i,...,t_{i+k-1}]}{t_{i+k} - t_i}$

  This follows swiftly from $\frac{(x-t_k)p_{k+1,l}(x) - (x-t_l)p_{k,l-1}(x)}{t_l - t_k}$ as derived as part of the Aitken-Neville scheme.

This is called the divided differences recursion for obvious reason. The divided differences a_j can be computed using this scheme/recusion. The following implementation does so slightly different from the Atiken-Neville scheme. It starts at the bottom instead of at the top so that the divided differences/quotients are in the vector in the end.

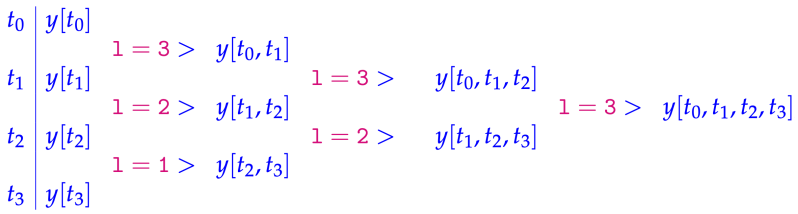


Figure 37: Divided Differences Evaluation Order

Listing 43: Divided-Differences

```
void divdiff(const VectorXd &t, VectorXd &y) {
  const int n = y.size() - 1;
  for (int l = 1; l<n; ++l) {
    for (int j = n-l; j<n; ++j) {
      y[j+1] = (y[j+1] - y[j]) / (t[j+1]-t[n-l]);
    }
  }
}
```

This implementation is slightly different than in the lecture document as it seemed flawed.

We can use this only to compute the initial coefficients. Extending the data set is not possible due to the order of performing the scheme. It overwrites values we would need for extension. Extending the dataset is considered in the next section.

This evaluation runs in $\mathcal{O}(n^2)$ - just as Aitken-Neville did.

Implementation in Eigen Here, we will put everything together to enable evaluating in many points as well as adding data points. Remember the LSE we introduced earlier for the coefficients a_j. We will use is now to extend the dataset.

$$\begin{bmatrix} 1 & 0 & \dots & 0 \\ 1 & (t_1 - t_0) & \dots & \vdots \\ \vdots & \vdots & \vdots & 0 \\ 1 & (t_n - t_0) & \dots & \prod_{i=0}^{n-1}(t_n - t_i) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

$$\begin{aligned} a_n &= \prod_{i=0}^{n-1}(t_n - t_i)^{-1}\left(y_n - \sum_{k=0}^{n-1}\prod_{i=0}^{k-1}(t_n - t_i)a_k\right) \\ &= \prod_{i=0}^{n-1}(t_n - t_i)^{-1}y_n - \sum_{k=0}^{n-1}\prod_{i=k}^{n-1}(t_n - t_i)^{-1}a_k \\ &= (\dots(((y_n - a_0)/(t_n - t_0) - a_1)/(t_n - t_1) - a_2)/\dots - a_{n-1})/(t_n - t_{n-1}) \end{aligned}$$

Listing 44: Divided Differences Extension & Evaluation in C++

```
class PolyEval {
  private:
    std::vector<double> t;  // Interpolation nodes
    std::vector<double> y;  // Coefficients in Newton
       representation
  public:
    PolyEval(); // Idle constructor
    void addPoint(double t, double y); // Add another data point
    // evaluate value of current interpolating polynomial at \
       Blue{$x$},
    double operator() (double x) const;
};

void PolyEval::addPoint(double td, double yd) {
  t.push_back(td);
  y.push_back(yd);
  int n = t.size();
  for(int j = 0; j<n-1; j++)
    y[n-1] = ((y[n-1]-a[j]) / (t[n-1]-t[j]));
}

double PolyEval::operator() (double x) const {
  double s = y.back();
```

```
22    for (int i = y.size()-2; i>=0; --i)
23      s = s * (x-t[i]) + y[i];
24    return s;
25  }
```

Both functions run in $\mathcal{O}(n)$ each.

7 Iterative Methods for Non-Linear Systems of Equations

We have already discussed solving linear systems of equations when they are regular as well as overdetermined. But we did not yet look at NON-linear systems of equations at all. This section is dedicated to such non-linear systems of equations (NLSE). There is no general solution scheme for NLSE. Hence, we can only discuss heuristics.

An abstract NLSE can be written as $F(x) = 0$ where $F : D \subset \mathbb{R}^n \to \mathbb{R}^n$ ($n \in \mathbb{N}$) is an arbitrary function. n defines the number of (scalar) equations as well as the number of (scalar) unknowns. We always have $\#unknowns = \#equations$. As NLSE also contain LSE as a subset, we can express those in this formalism. $\mathbf{Ax} = \mathbf{b}, \mathbf{A} \in \mathbb{R}^{n,n}, \mathbf{b} \in \mathbb{R}^n$ would be expressed as $F(x) = 0$ with $F(x) = \mathbf{b} - \mathbf{Ax}$.

Although $F(x)$ may look very simple, it may contain an arbitrarily complex expression. Thus, there is no general theory for solving such problems. Only for specific examples/instances, a theory can be developed. Furthermore, F may often be only available in procedural form. Meaning, we can evaluate F for a specific argument but don't know its functional expression.

7.1 Iterative Methods

7.1.1 Fundamental Concepts

Iterative methods for (approximately) solving a non-linear system of equations $F(x) = 0$ are algorithms, which generate an arbitrarily long sequence $\left(x^{(k)}\right)_k$ of approximate solutions. Those approximate solutions may be vectors if $\mathbf{x} \in \mathbb{R}^n, n > 1$. $x^{(k)}$ is the k-th iterate.

A fundamental question is, whether this series converges to our target value x^*, i.e., whether $x^* := \lim_{k\to\infty} x^{(k)}$ exists. For the iterative method to be any good, this is necessary.

m-point iteration computes the next iterate based on m previous iterates.

$$x^{(k+1)} = \phi(x^{(k)}, \ldots, x^{(k-m+1)})$$

To use this approach, we need some initial vectors to start the estimation. Those vectors are called initial guesses. For an m-point iterative guesses, we need m initial guesses: $x^{(0)}, \ldots, x^{(m-1)}$.

1-point iteration is the special case with $m = 1$ and $x^{(k+1)} = \phi(x^{(k)})$.

One very important property of iterative methods is consistency. It is necessary to make many meaningful statements about iterative methods. Usually, we only consider iterative methods which satisfy consistency. Besides, its a very reasonable property.

Definition Consistency of Iterative Methods: A stationary m-point iteratie method

$$x^{(k+1)} = \phi_F(x^{(k)}, \ldots, x^{(k-m+1)}), m \in \mathbb{N}$$

is consistent with the non-linear system of equations $F(x) = 0$, if and only if

$$\phi_F(x^*, \ldots, x^*) = x^* \Leftrightarrow F(x^*) = 0$$

If some iterative method ϕ_F (a) converges, (b) ϕ_F is continuous, and (c) ϕ_F is consistent, then it must converge to our target value x^*. Unfortunately, the problem mostly is that for many methods there is no guarantee that it converges. The convergence heavily depends on where the convergence starts, i.e., the initial guesses. If one starts close enough to a solution, the sequence might converge nicely. But if one starts further away, the sequence may go anywhere. This is a

challenge, because a-priori one doesn't know how close a guess is to the solution. Other important questions are: How fast do the sequences converge to solutions? What is the region of (local) convergence?

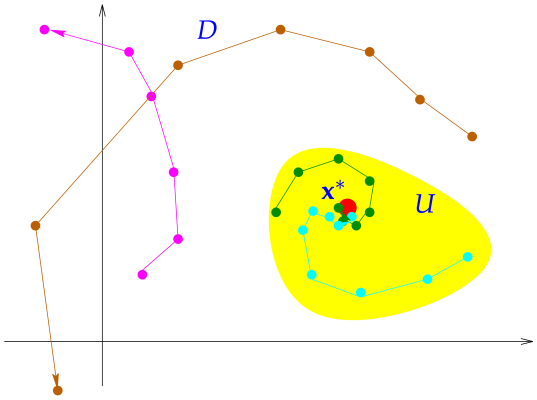


Figure 38: Local Convergence

7.1.2 Speed of Convergence

If some iterative method converges, we want to know how fast it converges. Hence, we classify iterative methods according to their speed of convergence.

With the speed of convergence we measure how fast the error goes to zero: $\epsilon_k = \|x^{(k+1)} - x^*\| \to 0$.

Definition Linear Convergence: A sequence $x^{(k)}, k = 0, 1, 2, ...$ in $\mathbb{R}^n$ converges linearly to $x^* \in \mathbb{R}^n$ if and only if

$$\exists 0 < L < 1 : \|x^{(k+1)} - x^*\| \leq L\|x^{(k)} - x^*\|, \forall k \in \mathbb{N}_0$$

The smallest possible L is called the **rate of convergence**.

Here, $\|x^{(k)} - x^*\| \to 0$ is guaranteed by a geometric decay of the error.

Detecting Linear Convergence If we want to identify linear convergence in a numerical experiment, we assume sharpness of the convergence. This means, instead of $\|x^{(k+1)} - x^*\| \leq L\|x^{(k)} - x^*\|$ we assume $\|x^{(k+1)} - x^*\| \approx L\|x^{(k)} - x^*\| \Leftrightarrow \epsilon_{k+1} \approx L\epsilon_k$. When looking at tabulated data, we can compute $\frac{\epsilon_{k+1}}{\epsilon_k}$ for all k. If this quotient is $\approx L < 1$ for some L for all values, we have an indication for linear convergence.

There is also a graphical approach. If we have linear convergence, then $\epsilon_k \approx L^k\epsilon_0 \Rightarrow \log \epsilon_k \approx k \log L + \log \epsilon_0$. Thus, in a lin-log plot the points should be on a line. The slope of the line gives us the rate of convergence. Note that the slope of the line must be negative as $L < 1$.

Example: We consider the scalar 1-point iteration $x^{(k+1)} := x^{(k)} + \frac{\cos(x^{(k)})+1}{\sin(x^{(k)})}, k \in \mathbb{N}_0$. If $x^{(k)}$ converges, it must converge to a value so that cos of that value is -1.
We can compute the sequence for different initial guesses and plot the computed values with the associated error. This reveals a rate of convergence of ≈ 0.5.

k	$x^{(0)} = 0.4$		$x^{(0)} = 0.6$		$x^{(0)} = 1$	
	$x^{(k)}$	$\frac{\lvert x^{(k)} - x^{(15)}\rvert}{\lvert x^{(k-1)} - x^{(15)}\rvert}$	$x^{(k)}$	$\frac{\lvert x^{(k)} - x^{(15)}\rvert}{\lvert x^{(k-1)} - x^{(15)}\rvert}$	$x^{(k)}$	$\frac{\lvert x^{(k)} - x^{(15)}\rvert}{\lvert x^{(k-1)} - x^{(15)}\rvert}$
2	3.3887	0.1128	3.4727	0.4791	2.9873	0.4959
3	3.2645	0.4974	3.3056	0.4953	3.0646	0.4989
4	3.2030	0.4992	3.2234	0.4988	3.1031	0.4996
5	3.1723	0.4996	3.1825	0.4995	3.1224	0.4997
6	3.1569	0.4995	3.1620	0.4994	3.1320	0.4995
7	3.1493	0.4990	3.1518	0.4990	3.1368	0.4990
8	3.1454	0.4980	3.1467	0.4980	3.1392	0.4980

Figure 39: Determining the Rate of Convergence with Linear Convergence

In the error computation, x^* has been replaced by $x^{(15)}$ for convenience.

Linear convergence is just a special case of general order p convergence.

Definition Order of Convergence: A convergent sequence $x^{(k)}, k = 0, 1, 2, ...$ in $\mathbb{R}^n$ with limit $x^* \in \mathbb{R}^n$ converges with order p (usually $p \geq 1$) if

$$\exists C > 0 : \|x^{(k+1)} - x^*\| \leq C\|x^{(k)} - x^*\|^p, \forall k \in \mathbb{N}_0$$

If $p = 1$ (linear convergence) we additionally require that $C < 1$.

C is called the **rate of convergence**. According to our definition of the order of convergence: $C := \sup_{k\geq 0} \frac{\epsilon_{k+1}}{\epsilon_k^p}$. However, in the literature it is common to also define it as $\lim \frac{\epsilon_{k+1}}{\epsilon_k^p}$.

Detecting Order $p > 1$ Convergence Again, we assume sharpness of the convergence, i.e., $\|x^{(k+1)} - x^*\| \approx C\|x^{(k)} - x^*\|^p \Leftrightarrow \epsilon_{k+1} \approx C\epsilon_k^p$ for some C and p.

$$\Rightarrow \log \epsilon_{k+1} \approx \log C + p \log \epsilon_k$$

$$\Longrightarrow \log \epsilon_{k+1} \approx \log C + p \log \epsilon_k \quad \text{and} \quad \log \epsilon_k \approx \log C + p \log \epsilon_{k-1}$$

$$\Rightarrow p \approx \frac{\log \epsilon_{k+1} - \log \epsilon_k}{\log \epsilon_k - \log \epsilon_{k-1}}$$

We then create a table and print the computed quotient for many k and observe whether it is approximately constant. But notice for large k this quotient will become unreliable due to roundoff as $\epsilon_k \approx \epsilon_{k-1} \approx \epsilon_{k+1}$.

In the below example one can see that with quadratic convergence the number of correct digits doubles with each iterative step. This is very fast convergence. m correct digits mean that the relative error is below 10^{-m}. If we denote the relative error with δ_k: $x^{(k)} = x^*(1 + \delta_k)$.

$$|x^{(k+1)} - x^*| \approx C|x^{(k)} - x^*|^2 \Rightarrow |\delta_{k+1}x^*| \approx C|\delta_k x^*|^2 \Rightarrow \delta_{k+1} \approx C|x^*|\delta_k^2$$

assuming $x^* \neq 0$ so that the relative error exists

This has to hold for all iterations of order 2 as can be seen from $\epsilon_{k+1} \leq C\epsilon_k^p$. Similar statements also hold for order 3 convergence etc.

Example: This is a famous example of converges to the square root of some number $a > 0$.

$$x^{(k+1)} = \frac{1}{2}(x^{(k)} + \frac{a}{x^{(k)}}) \Rightarrow |x^{(k+1)} - \sqrt{a}| = \frac{1}{2x^{(k)}}|x^{(k)} - \sqrt{a}|^2$$

This shows that the error in some step is the squared error of the previous step multiplied with some factor. And we can give the upper bound $\frac{1}{2\sqrt{a}}$ for that factor as $x^{(k)} \geq \sqrt{a}$. Thus, this is order two convergence, where we can also give the constant: $p = 2$ and $C = \frac{1}{2\sqrt{a}}$.

k	$x^{(k)}$	$e^{(k)} := x^{(k)} - \sqrt{2}$	$\log \frac{\lvert e^{(k)}\rvert}{\lvert e^{(k-1)}\rvert} : \log \frac{\lvert e^{(k-1)}\rvert}{\lvert e^{(k-2)}\rvert}$
0	2.00000000000000000	0.58578643762690485	
1	1.50000000000000000	0.08578643762690485	
2	1.41666666666666652	0.00245310429357137	1.850
3	1.41421568627450966	0.00000212390141452	1.984
4	1.41421356237468987	0.00000000000159472	2.000
5	1.41421356237309492	0.00000000000000022	0.630

Figure 40: Square Root Iteration Error & Order

7.1.3 Termination Criteria

If we designed ϕ well, it will converge to the solution x^*. However, almost always none of the sequence elements will correspond to the solution perfectly. So, we must decide when to stop computing next elements in the sequence as we can't do infinitely many iterations.

Let's assume the iterative method produces the sequence $(x^{(k)})_{k\in\mathbb{N}}$. We then want that the current iterate is sufficiently close to the solution if we stop in step K. We consider either an absolute or a relative tolerance.

- $\|x^{(K)} - x^*\| \leq \tau_{abs}$, τ_{abs} being the prescribed (absolute) tolerance
- $\|x^{(K)} - x^*\| \leq \tau_{rel}\|x^*\|$, τ_{rel} being the prescribed (relative) tolerance

The ideal stopping rule is to stop at step $K = \text{argmin}\left\{k \in \mathbb{N}_0 : \|x^{(k)} - x^*\| \leq \begin{cases}\tau_{abs} \\ \tau_{rel}\|x^*\|\end{cases}\right\}$.
But this is not accessible, because we don't know x^* as it is the value we try to compute. Hence, we must consider practical stopping rules.

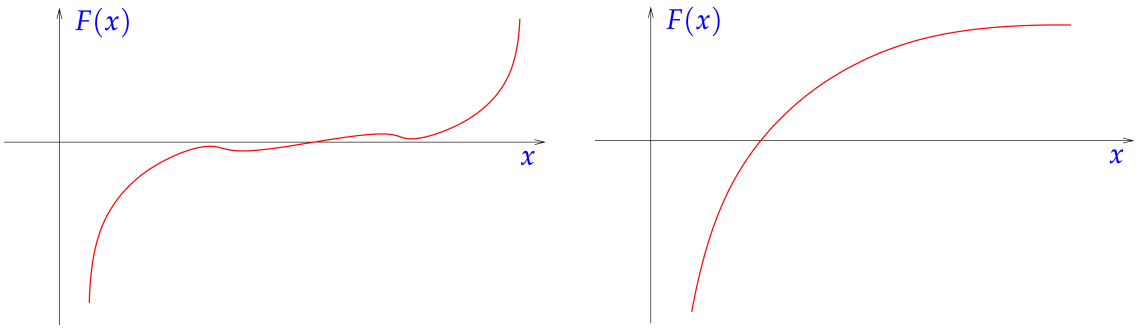
- A priori termination rules set some fixed step K after which we stop.

  This works if we have precise information on the convergence. For instance, we may know that it converges linearly and it's rate of conversion. Then, we can easily predict when accuracy requirements are met.

- A posteriori termination rules rely on the iterates while they are comptued.

  This is relevant, because we rarely have precise information beforehand. An example is residual-based termination as discussed next.

Residual-Based Termination We step as soon as $\|F(x^{(k)})\| \leq \tau$ for some prescribed tolerance. But unfortunately, this tells us little about the error norm $\|x^{(k)} - x^*\|$.



$\|F(\mathbf{x}^{(k)})\|$ small $\not\Rightarrow |x - x^*|$ small $\qquad$ $\|F(\mathbf{x}^{(k)})\|$ small $\Rightarrow |x - x^*|$ small

Figure 41: Problem with Residual-Based Termination

Correction-Based Termination The heuristic is to stop the iteration when the computed $x^{(k)}$ only change very little between iterations. We stop the convergent iteration $\{x^{(k)}\}_{k\in\mathbb{N}_0}$ when

$$\left\|x^{(k+1)} - x^{(k)}\right\| \leq \begin{cases} \tau_{abs} \\ \tau_{rel}\|x^{(k+1)}\| \end{cases}$$

$\|x^{(k+1)} - x^{(k)}\|$ is often called the correction. This is especially justified for order p iterations where $p > 1$, because we have seen that the iteration will converge very fast. Meaning, $x^{(k+1)} \approx x^*$ as the error shrinks geometrically.

Stationary in $\mathbb{M}$ Termination With real numbers, most iterations will never settle on one value. But remember that machine arithmetics is discrete. So once we are close enough to the target value we can't get any closer in machine numbers and will settle on one machine number. This has a guaranteed relative error equal to EPS (machine precision).

This is suitable (only) for special functions. In many cases it is 'overkill' and not so relevant.

A Posteriori Termination for Linear Convergence This shows that the usefulness of correction based termination can also be theoretically justified for linear convergent sequences.

$$\begin{aligned} \|x^{(k)} - x^*\| = \|x^{(k)} - x^{(k+1)} + x^{(k+1)} - x^*\| &\leq \|x^{(k+1)} - x^*\| + \|x^{(k+1)} - x^{(k)}\| \\ &\leq L\|x^{(k)} - x^*\| + \|x^{(k+1)} - x^{(k)}\| \\ \Longrightarrow (1-L)\|x^{(k)} - x^*\| &\leq \|x^{(k+1)} - x^{(k)}\| \\ \Longrightarrow \|x^{(k)} - x^*\| &\leq \frac{1}{1-L}\|x^{(k+1)} - x^{(k)}\| \\ \Longrightarrow \|x^{(k+1)} - x^*\| &\leq \frac{L}{1-L}\|x^{(k+1)} - x^{(k)}\| \end{aligned}$$

With the final formula we can always compute an upper bound for the error. And although L is usually not known, we can just use a coarse upper estimation for L and still get a good termination criterion. Due to such estimations and the estimations in the derivation we may compute a value much more precise than required, but it will have at least the required precision in any case.

Example: Consider again $x^{(k+1)} = x^{(k)} + \frac{\cos x^{(k)}+1}{\sin x^{(k)}} \Rightarrow x^{(k)} \to \pi$, for $x^{(0)}$ close to π. From numerical experiments, we can conclude that $L \approx \frac{1}{2}$.
The following table shows the actual error and error estimation. This showcases that the actual error may be much lower, but the error estimation still is accurate.

k	$\lvert x^{(k)} - \pi \rvert$	$\frac{L}{1-L} \lvert x^{(k)} - x^{(k-1)} \rvert$	slack of bound
1	2.191562221997101	4.933154875586894	2.741592653589793
2	0.247139097781070	1.944423124216031	1.697284026434961
3	0.122936737876834	0.124202359904236	0.001265622027401
4	0.061390835206217	0.061545902670618	0.000155067464401
5	0.030685773472263	0.030705061733954	0.000019288261691
6	0.015341682696235	0.015344090776028	0.000002408079792
7	0.007670690889185	0.007670991807050	0.000000300917864
8	0.003835326638666	0.003835364250520	0.000000037611854
9	0.001917660968637	0.001917665670029	0.000000004701392
10	0.000958830190489	0.000958830778147	0.000000000587658
11	0.000479415058549	0.000479415131941	0.000000000073392
12	0.000239707524646	0.000239707533903	0.000000000009257
13	0.000119853761949	0.000119853762696	0.000000000000747
14	0.000059926881308	0.000059926880641	0.000000000000667
15	0.000029963440745	0.000029963440563	0.000000000000181

7.2 Fixed-Point Iterations

We already discussed that fixed-point iteration is a special case of m-point iteration.

Definition Fixed-Point Iteration: A fixed point iteration is defined by iteration function $\phi : U \subset \mathbb{R}^n \to \mathbb{R}^n$. With an initial guess $x^{(0)} \in U$, the iteration is

$$(x^{(k)})_{k\in\mathbb{N}_0} : \quad x^{(k+1)} := \phi(x^{(k)})$$

Let $x^* = \lim_{k\to\infty} x^{(k)}$. If ϕ is continuous, we can take the limit into ϕ and get $x^* = \phi(x^*)$. So, x^* is a so called fixed point of ϕ, which is where the name of this special case comes from.

7.2.1 Consistent Fixed-Point Iterations

Definition Consistency of Fixed Point Iterations: A fixed point iteration $x^{(k+1)} = \phi(x^{(k)})$ is consistent with $F(x) = 0$ if, for $x \in U \cap D$,

$$F(x) = 0 \iff \phi(x) = x$$

I.e., every fixed point is a solution.

Of course the question rises, how to find a consistent fixed point iteration given some F. The general construction follows two steps:

1. Rewrite equivalently $F(x) = 0 \Leftrightarrow \phi(x) = x$
2. Use the fixed point iteration $x^{(k+1)} := \phi(x^{(k)})$

3. Hope that ϕ converges.

Definition Contractive Fixed-Point Iteration: $\phi : D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ is contractive (w.r.t. norm $\|\cdot\|$), if for some $L < 1$ and for all $x, d \in D$

$$\|\phi(x) - \phi(y)\| \leq L\|x - y\|$$

Theorem Banach's Fixed Point Theorem: If $\phi : \mathbb{K}^n \to \mathbb{K}^n$ is contractive, then $(x^{(k)})$ converges to a unique fixed point from any $x^{(0)}$ at least linearly.
(Not discussed or proven in the lecture but probably relevant.)

Example: We consider the function $F(x) = xe^x - 1, x \in [0, 1]$ of which we want to find the root. From this, we can find many different fixed-point forms, i.e. $\phi_i(x) = x \Leftrightarrow F(x) = 0$. Some are:

- $\phi_1(x) = e^{-x}$
- $\phi_2(x) = \frac{1+x}{1+e^x}$
- $\phi_3(x) = x + 1 - xe^x$

Those functions and their iterations behave very differently.

k	$x^{(k+1)} := \Phi_1\left(x^{(k)}\right)$	$x^{(k+1)} := \Phi_2\left(x^{(k)}\right)$	$x^{(k+1)} := \Phi_3\left(x^{(k)}\right)$
0	0.500000000000000	0.500000000000000	0.500000000000000
1	0.606530659712633	0.566311003197218	0.675639364649936
2	0.545239211892605	0.567143165034862	0.347812678511202
3	0.579703094878068	0.567143290409781	0.855321409174107
4	0.560064627938902	0.567143290409784	-0.156505955383169
5	0.571172148977215	0.567143290409784	0.977326422747719
6	0.564862946980323	0.567143290409784	-0.619764251895580
7	0.568438047570066	0.567143290409784	0.713713087416146
8	0.566409452746921	0.567143290409784	0.256626649129847
9	0.567559634262242	0.567143290409784	0.924920676910549
10	0.566907212935471	0.567143290409784	-0.407422405542253

Figure 42: Iteration Values

k	$\left\|x_1^{(k+1)} - x^*\right\|$	$\left\|x_2^{(k+1)} - x^*\right\|$	$\left\|x_3^{(k+1)} - x^*\right\|$
0	0.067143290409784	0.067143290409784	0.067143290409784
1	0.039387369302849	0.000832287212566	0.108496074240152
2	0.021904078517179	0.000000125374922	0.219330611898582
3	0.012559804468284	0.000000000000003	0.288178118764323
4	0.007078662470882	0.000000000000000	0.723649245792953
5	0.004028858567431	0.000000000000000	0.410183132337935
6	0.002280343429460	0.000000000000000	1.186907542305364
7	0.001294757160282	0.000000000000000	0.146569797006362
8	0.000733837662863	0.000000000000000	0.310516641279937
9	0.000416343852458	0.000000000000000	0.357777386500765
10	0.000236077474313	0.000000000000000	0.974565695952037

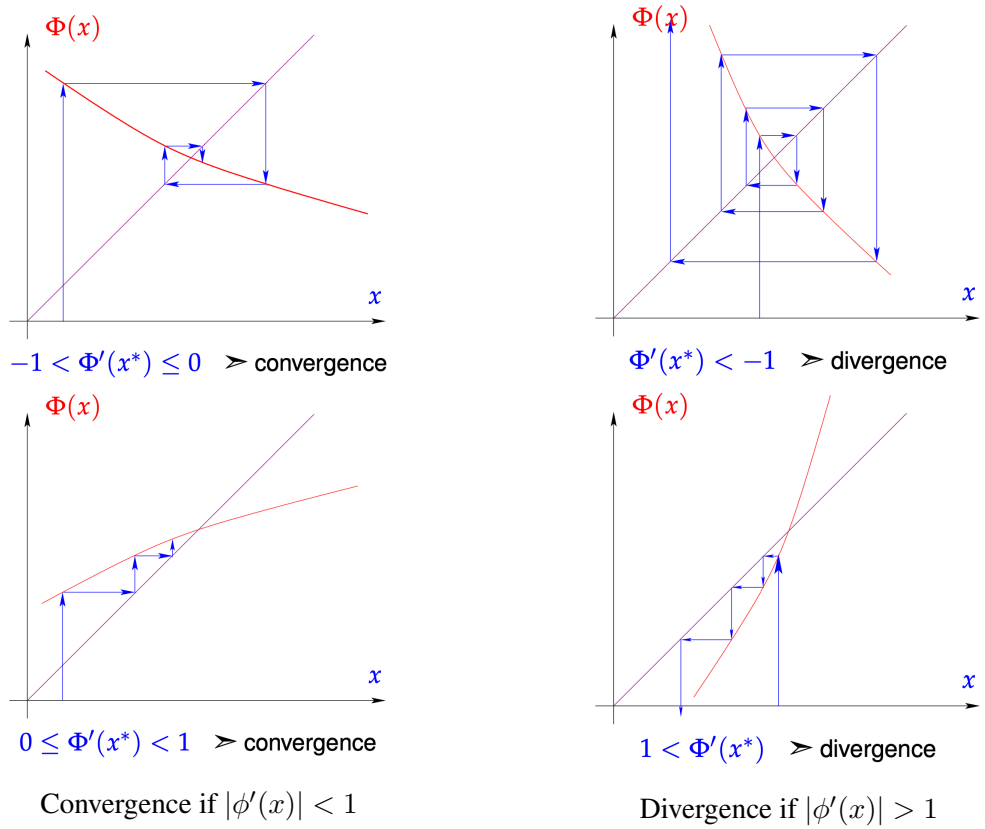
Figure 43: Iteration Errors

ϕ_1 converges linearly, ϕ_2 converges very fast (approximately quadratic), and ϕ_3 is erratic/-does not converge

7.2.2 Convergence of Fixed-Point Iterations

In this subsection we will discuss whether we can make predictions regarding the convergence of an iteration based on its functional form/without a numerical experiment.

The following figure illustrates how the derivative is crucial for convergence for initial guesses close to x^*. Note that this is not definitive but just an indicator for initial guesses close to x^*. If we start far away, anything may happen as we don't know the function behavior there and in between our target point.



Convergence if $|\phi'(x)| < 1$

Divergence if $|\phi'(x)| > 1$

Figure 44: Convergence of Fixed-Point Iteration based on Derivative

Lemma Sufficient Condition for Local Linear Convergence of Fixed Point Iteration:
If $\phi : U \subset \mathbb{R}^n \to \mathbb{R}^n, \phi(x^*) = x^*$, ϕ differentiable in x^*, and $\|D\phi(x^*)\|_2 < 1$, then the fixed point iteration

$$x^{(k+1)} := \phi(x^{(k)})$$

converges locally and at least linearly that is

$$\exists 0 \leq L < 1 : \|x^{(k+1)} - x^*\| \leq L\|x^{(k)} - x^*\|, \forall k \in \mathbb{N}_0$$

provided that the initial guess $x^{(0)}$ belongs to some neighborhood of x^*.

As justification (not a proof!) consider the Taylor approximation. If we are sufficiently close, the remainder can be neglected and this is guaranteed to converge. But if we are not sufficiently close, the remainder may prevent convergence.

$$\|x^{(k+1)} - x^*\| = \|\phi(x^{(k)}) - \phi(x^*)\| = \|D\phi(x^*)(x^{(k)} - x^*) + \mathcal{O}(\|x^{(k)} - x^*\|)$$

Lemma Higher Order Local Convergence of Fixed Point Iterations: If $\phi : U \subset \mathbb{R} \to \mathbb{R}$ is $m+1$ times continuously differentiable, $\phi(x^*) = x^*$ for some x^* in the interior of U, and $\phi^{(l)}(x^*) = 0$ for $l = 1, ..., m$, $m \geq 1$, then the fixed point iteration converges locally to x^* with order $\geq m + 1$.

This can be understood with the same Taylor polynomial argument as for the previous lemma. It follows from the fact that from $\phi^{(l)}(x^*) = 0$ we get that the next summands in the Taylor polynomial are zero and the remainder is $\mathcal{O}(\|x^{(k)} - x^*\|^{m+1})$.

7.3 Finding Zeros of Scalar Functions

This considers the simplest case of NLSEs with $F : U \subset \mathbb{R} \to \mathbb{R}$. We want to find $x^* \in U$ so that $F(x^*) = 0$.

7.3.1 Bisection

We assume that F is continuous and use the intermediate value theorem from Analysis. Remember it as: If $a, b \in U$ so that $F(a)F(b) < 0$ (i.e., a and b have different signs) then $\exists x^* \in]\min\{a, b\}, \max\{a, b\}[$ so that $F(x^*) = 0$.

We want to enclose x^* in geometrically decreasing intervals $[a^{(k)}, b^{(k)}]$ on which f changes sign: $f(a^{(k)})f(b^{(k)}) \leq 0$ (equality with 0 iff we have found the root/zero exactly with one border). Because this approach is very simple we won't explain it in detail but just give its implementation as explanation:

Listing 45: Root of Scalar Function with Bisection

```
template <typename Func, typename Scalar>
Scalar bisect(Func&& F, Scalar a, Scalar b, Scalar tol) {
  if(a>b)
    std::swap(a,b);
  if(F(a)*F(b)>0)
    throw "f(a) and f(b) have same sign";
  static_assert(std::is_floating_point<Scalar>::value,
    "Scalar must be a floating point type");
  int v = F(a)<0 ? 1 : -1;
  Scalar x = (a+b)/2; // determine midpoint
  while(b-a > tol) { // if tol=0, termination relies on machine
     arithemtic
    assert(a<=x && x<=b);
    if (v*F(x)>0)
      b=x;
    else
      a=x;
    x = (a+b)/2;
  }
  return x;
}
```

This is not quite an iterative method in a strict sense, because it does not compute a sequence of approximate zeroes but intervals. But we can just set $x^{(k)}$ to one of the end points ($:= a^{(k)}$ for example) or even their average, which will generate a sequence.

This has a "linear-type" convergence with the rate $\frac{1}{2}$, because $|b^{(k)} - a^{(k)}| \leq 2^{-k}|b - a|$. As our sequence element and the root have to lie in the interval $[a^{(k)}, b^{(k)}]$, we get:

$$|x^{(k)} - x^*| \leq 2^{-k}|b - a|$$

Despite its simplicity, this method has various advantages:

- Because it guarantees convergence, we call this a robust method. Also, the convergence rate is known.
- This method does not require any information on differentiability. Instead we only need evaluation of F at different points. We say, it is a "derivative-free" method.
- We can apply this method to many functions. Specifically, all continuous methods.

The price we pay is that this method is not very fast.

7.3.2 Model Function Methods

The most important iterative method is the Newton method. We dedicate the entire next subsection to study it. But we already introduce it in a limited form for one dimension here. The Newton method itself is a special case of a certain approach: Model function.

The general idea of model function methods is as follows: Given recent iterates (approximate zeroes) $x^{(k)}, x^{(k-1)}, \ldots, x^{(k-m+1)}, m \in \mathbb{N}$:

1. replace F with a k-dependent model function $\tilde{F}_k$ which is based on the function values $F(x^{(k)}), F(x^{(k-1)}), \ldots, F(x^{(k-m+1)})$ and, possibly, derivative values.
2. Then, $x^{(k+1)} :=$ zero of $\tilde{F}_k$ is found analytically given $\tilde{F}_k$.

   This means that locally around $x^{(k)}$ we approximate $F(x)$ with some $F_k(x)$.

Newton Method We assume that $F \in C^1$ (continuously differentiable) and that we have the derivative F' of F. Here, the model function $\tilde{F}$ is based on the tangent function G_f in $(x^{(k)}, F(x^{(k)}))$. So we get $\tilde{F}(x) = F(x^{(k)}) + F'(x^{(k)})(x - x^{(k)})$. Then:

$$x^{(k+1)} = x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})}$$

So that this works properly we have to assume that $F'(x^{(k)}) \neq 0$.

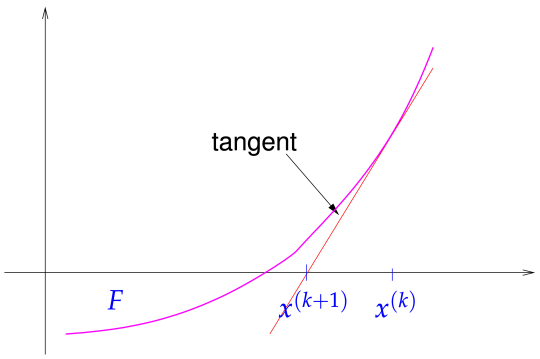


Figure 45: Newton Method

Listing 46: Scalar Newton Method in Eigen

```
template <typename FuncType, typename DervType, typename Scalar>
Scalar newton1D(
  FuncType &&F,
  DervType &&DF,
  const Scalar &x0,
  double rtol,
  double atol
) {
  Scalar s, x = x0;
  do {
    s = F(x)/DF(x);
    x -= s;
  } while((std::abs(s)>rtol*std::abs(x)) && (std::abs(s)>atol));
  return (x);
}
```

Example: Consider $F(x) = x^2 - a, a > 0$, which as $\sqrt{a}$ as its root. From the function we get $F'(x) = 2x$. Using the Newton iteration leads to the famous square root iteration formula.

$$x^{(k+1)} = x^{(k)} - \frac{(x^{(k)})^2 - a}{2x^{(k)}} = \frac{1}{2}(x^{(k)} - \frac{a}{x^{(k)}})$$

Theorem Convergence of Newton's Method in 1D: Newton's method locally converges quadratically to a zero x^* of F, if $F'(x^*) \neq 0$.

Proof. A Lemma from above told us how to find a lower bound for the order of convergence using ϕ and its derivatives. (We reasoned using the Taylor polynomial.) Here, we have $\phi'(x) = 1 - \frac{F'(x)^2 - F(x)F''(x)}{F'(x)^2} = \frac{F(x)F''(x)}{F'(x)^2}$. With $F(x^*) = 0$ and $F'(x^*) \neq 0$ we see that $\phi'(x^*) = 0$. From the mentioned Lemma follows directly that we must have at least order 2 convergence for sufficiently close initial guesses. □

Example:

- $F(x) = xe^x - 1$ with $F'(x) = e^x(1 + x)$ has:

  $$x^{(k+1)} = x^{(k)} - \frac{x^{(k)}e^{x^{(k)}} - 1}{e^{x^{(k)}}(1+x^{(k)})} = \frac{(x^{(k)})^2 + e^{-x^{(k)}}}{1+x^{(k)}}$$

- $F(x) = x - e^{-x}$ with $F'(x) = 1 + e^{-x}$ has:

  $$x^{(k+1)} = x^{(k)} - \frac{x^{(k)} - e^{-x^{(k)}}}{1+e^{-x^{(k)}}} = \frac{1+x^{(k)}}{1+e^{x^{(k)}}}.$$

Observe that those two F have the same roots x^* but their Newton iterations look very different and may also behave very differently. This suggests that is may be worth to recast some function into an equivalent F with the same roots that is easier to analyze.

We may be given F in functional form but still need to compute F' ourselves. Generally, we should apply our knowledge from Analysis to do so. Additionally, one should remember **implicit differentiation**. Often, it is the only way to apply the Newton method.
With implicit differentiation one can find the derivative of a function that is not explicitly solved for one variable in terms of another. It is particularly useful when dealing with equations where it is difficult or impossible to solve for one variable in terms of the other. In this process, you differentiate both sides of an equation with respect to a chosen variable, treating other variables as functions of that variable, and then solve for the derivative of the function of interest. This technique is widely used to find the derivatives of functions defined implicitly.

Example: Consider $F(\mathbf{x}) = \omega^\top(\mathbf{A} + \mathbf{xI})^{-1}\mathbf{b} - 1$. We identify $u(\mathbf{x}) = (\mathbf{A} + \mathbf{xI})^{-1}\mathbf{b}$ which we need to differentiate. We can write that as $(\mathbf{A} + \mathbf{xI})u(\mathbf{x}) = \mathbf{b}$ and apply the product rule to get:

$$\mathbf{I} \cdot u(\mathbf{x}) + (\mathbf{A} + \mathbf{xI})u'(\mathbf{x}) = 0 \Rightarrow u'(\mathbf{x}) = -(\mathbf{A} + \mathbf{xI})^{-1}u(\mathbf{x})$$

When substituting $u(\mathbf{x})$, we have an expression for $u'(\mathbf{x})$.

Preconditioning of Newton's Method In this paragraph we want to replace $F(x)$ with some other function $G(x)$ so that G has approximately the same roots but offers a larger range of convergence. We do so by defining $g(x) = \hat{F}^{-1}(F(x)) \approx x$ with the assumption that some $\hat{F}^{-1} \approx F^{-1}$ is available. G then is $G(x) := g(x) - \hat{F}^{-1}(0) = 0 \Leftrightarrow F(x) = 0$. G is an "almost linear" function. Thus, we expect a very large convergence range (global?) from the Newton method. Furthermore, we can expect quadratic convergence.

Let's compute the Newton iteration. For that we need to know $G'(x)$ which we get as $g'(x)$:

$$(\hat{F}^{-1})'(y) = \frac{1}{\hat{F}'(\hat{F}^{-1}(y))} \Rightarrow g'(x) = \frac{1}{\hat{F}'(g(x))}F'(x)$$

$$\Longrightarrow x^{(k+1)} = x^{(k)} - \frac{G(x^{(k)})\hat{F}'(g(x^{(k)}))}{F'(x^{(k)})}$$

Example: Consider $F(x) = \frac{1}{(x+1)^2} + \frac{1}{(x+0.1)^2} - 1$ with $x > 0$. We approximate $F(x) \approx \hat{F}(x) = \frac{2}{x^2} - 1 \Rightarrow \hat{F}'(y) = \frac{1}{\sqrt{y+1}}$. With $x^{(0)} = 0$ we then get nice convergence:

k	$x^{(k)}$	$F\left(x^{(k)}\right)$	$x^{(k)} - x^{(k-1)}$	$x^{(k)} - x^*$
1	0.91312431341979	0.24747993091128	0.91312431341979	-0.13307818147469
2	1.04517022155323	0.00161402574513	0.13204590813344	-0.00103227334125
3	1.04620244004116	0.00000008565847	0.00103221848793	-0.00000005485332
4	1.04620249489448	0.00000000000000	0.00000005485332	-0.00000000000000

Multi-Point Methods Now, we want to create a more elaborate model of the function compared to the simple linear approximation of the Newton method. We want to replace F with an interpolating polynomial. On the one hand, this allows us to use more points to create more accurate approximations and, one the other hand, we don't rely on the the derivative of the function F.

Secant Method We model F by the linear function connecting $(x^{(k-1)}, F(x^{(k-1)}))$ and $(x^{(k)}, F(x^{(k)}))$. The iteration becomes:

$$\tilde{F}(x) = F(x^{(k)}) + \frac{F(x^{(k)}) - F(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}(x - x^{(k)})$$
$$x^{(k+1)} = x^{(k)} - \left(\frac{F(x^{(k)}) - F(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}\right)^{-1} F(x^{(k)})$$

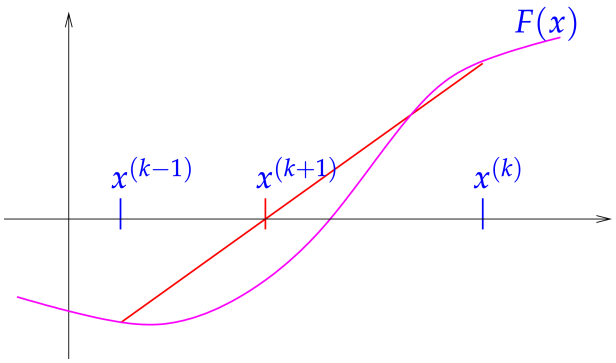


Figure 46: Secant Method

While we don't need the derivative of F for this method, we now need two initial guesses. But after those, we only need to do one evaluation of F per iterative step.

Listing 47: Secant Method in C++

```
template <typename Func>
double secant(double x0, double x1, Funct &&F, double rtol,
    double atol, unsigned int maxit) {
  double fo = F(x0);
  for(unsigned int i = 0; i<maxit; ++i) {
    double fn = F(x1);
    double s = fn * (x1-x0) / (fn-fo);
    x0 = x1;
    x1 = x1 - s;
    if (abs(s)<max(atol, rtol*min(abs(x0),abs(x1))))
      return x1;
    f0 = fn;
  }
  return x1;
}
```

It can be mathematically shown that this converges with order $p \approx 1.62$. Fractional orders of convergence are typical for two-point iterations. One can extend this approach to more-point iterations by modeling a polynomial of higher degree with more points and more initial guesses.

Multi-point iterations (and especially the secant method) usually only converge locally!

Inverse Interpolation We assume that $F : I \subset \mathbb{R} \to \mathbb{R}$ is one-to-one, i.e., monotone. Thus: $F(x^*) = 0 \Rightarrow F^{-1}(0) = x^*$. We now interpolate F^{-1} by a polynomial p of degree $m-1$ degermined by the points/interpolation conditions:

$$p(F(x^{(k-j)})) = x^{(k-j)}, \quad j = 0, \ldots, m-1$$

The iteration then is given through evaluation of p at 0: $x^{(k+1)} := p(0)$. With $m = 2$ we just get the secant method. But with $m = 3$ we get quadratic inverse interpolation. The iteration breaks down if previous points agree as the root of the polynomial then can be arbitrary. For the case that the polynomial is well-defiend:

$$x^{(k+1)} = \frac{F_0^2 (F_1 x_2 - F_2 x_1) + F_1^2 (F_2 x_0 - F_0 x_2) + F_2^2 (F_0 x_1 - F_1 x_0)}{F_0^2 (F_1 - F_2) + F_1^2 (F_2 - F_0) + F_2^2 (F_0 - F_1)}$$

with:

$$F_0 := F\left(x^{(k-2)}\right), F_1 := F\left(x^{(k-1)}\right), F_2 := F\left(x^{(k)}\right), x_0 := x^{(k-2)}, x_1 := x^{(k-1)}, x_2 := x^{(k)}$$

The 3-point iteration also converges with a fractional order, but with ≈ 1.8 it is slightly higher than for the secant method.

Example: $F(x) = xe^x - 1$ with initial guesses $x^{(0)} = 0, x^{(1)} = 2.5, x^{(2)} = 5$.

k	$x^{(k)}$	$F\left(x^{(k)}\right)$	$e^{(k)} := x^{(k)} - x^*$	$\frac{\log\lvert e^{(k+1)}\rvert - \log\lvert e^{(k)}\rvert}{\log\lvert e^{(k)}\rvert - \log\lvert e^{(k-1)}\rvert}$
3	0.0852039005817	-0.90721814294134	-0.48193938982803	
4	0.1600925262258	-0.81211229637354	-0.40705076418392	3.3379115437883
5	0.7987938181639	0.77560534067946	0.23165052775411	2.2874048891220
6	0.6309463675284	0.18579323999999	0.06380307711864	1.8249466728971
7	0.5610775099102	-0.01667806436181	-0.00606578049951	1.8732326421421
8	0.5670694103310	-0.00020413476766	-0.00007388007872	1.7983293698045
9	0.5671433170709	0.00000007367067	0.00000002666114	1.8484126152709
10	0.5671432904098	0.00000000000003	0.00000000000001	

Figure 47: Numerical Experiment for 3-Point Method

7.3.3 Asymptotic Efficiency

We have discussed different iterative method to solve (one-dimensional) NLSEs. But so far we did not yet formally compare them. We only looked at numerical experiments and orders of convergence. But for a proper comparison we must consider not only the order of convergence but also the computational cost of each iterative step.

We say that a method is efficient if the fraction $\frac{\text{gain}}{\text{work}}$ is high. As a measure of gain we will use ρ. It measure the relative reduction of the error by a factor of ρ: $|e^{(k)}| \leq \rho |e^{(0)}|$ with $0 < \rho < 1$. $|\log \rho|$ then corresponds to the number of digits gained.

As a measure for work we will use $W \cdot k(\rho)$. W is the work/computational effort per step as the number of evaluation of F and and its derivatives in that step. $k(\rho)$ is the number of iteration steps required to reduce the relative error by a factor of ρ.

$$\text{efficiency} := \frac{\#\text{digits gained}}{\text{total work required}} = \frac{|\log \rho|}{k(\rho) \cdot W}$$

We now compute $k(\rho)$ for order p convergence.

$$\begin{aligned}\left|e^{(k)}\right| &\leq C\left|e^{(k-1)}\right|^p \leq C^{1+p}\left|e^{(k-2)}\right|^{p^2} \leq C^{1+p+p^2}\left|e^{(k-3)}\right|^{p^3} \leq \cdots \\ &\leq C^{1+p^2+p^3+\cdots+p^{k-1}}\left|e^{(0)}\right|^{p^k} = C^{\frac{p^k-1}{p-1}}\left|e^{(0)}\right|^{p^k-1}\left|e^{(0)}\right| \\ &= \left(\underbrace{C^{\frac{1}{p-1}}\left|e^{(0)}\right|}_{\text{assume } |e^{(0)}| \text{ so small so that } <1}\right)^{p^k-1}\left|e^{(0)}\right|, \quad k \in \mathbb{N}\end{aligned}$$

$$\begin{aligned}\implies |e^{(k)}| \leq \rho|e^{(0)}| \quad \text{if} \quad \left(C^{\frac{1}{1-p}}|e^{(0)}|\right)^{p^k-1} \leq \rho \\ \Rightarrow (p^k-1)\log(C^{\frac{1}{1-p}}|e^{(0)}|) \leq \log\rho \\ \Rightarrow p^k - 1 \geq \frac{\log\rho}{\log L_0} \quad \text{with } L_0 := C^{\frac{1}{1-p}}|e^{(0)}| < 1 \\ \Rightarrow p^k \geq 1 + \frac{\log\rho}{\log L_0} \\ \Rightarrow k \geq \frac{\log(1+\frac{\log\rho}{\log L_0})}{\log p} = \log_p(1+\frac{\log\rho}{\log L_0})\end{aligned}$$

Asymptotic Perspective For very small ρ: $\rho \ll 1 \Rightarrow \log\rho < 0, \log L_0 < 0, |\log\rho| \gg 1$ and get an approximation by the simplification $\frac{\log|\log\rho|}{\log p}$.

$$\text{efficiency} = \begin{cases}\frac{|\log\rho|}{k(\rho)\cdot W} = \frac{\log C}{\log\rho}\frac{|\log\rho|}{W} = -\frac{\log C}{W}, & p = 1 \\ \frac{|\log\rho|}{k(\rho)\cdot W} = \frac{\log p}{W}\frac{|\log\rho|}{\log|\log\rho|}, & p > 1\end{cases}$$

For $p > 1$, only the factor $\frac{\log p}{W}$ is specific to one iterative method and relevant for the asymptotic analysis.

Newton Method vs. Secant Method

- $W_{Newton} = 1 \cdot F$-evaluation $+ \cdot F'$-evluation
  $p_{Newton} = 2$
- $W_{secant} = 1 \cdot F$-evaluation
  $p_{secant} = 1.62$

$$\frac{\frac{\log p_{Newton}}{W_{Newton}}}{\frac{\log p_{secant}}{W_{secant}}} \approx 0.71$$

We see that the secant method has higher efficiency. This means that the reduced work per iteration step pays off although the convergence order is lower.

7.4 Newtons Method for Rn

We have been introduced to the Newton method in 1D already. Now we generalize to arbitrarily many dimensions. The goal is to find $x^* \in D \subset \mathbb{R}^n$ where $F(x^*) = 0$ and F is some function $F : D \to \mathbb{R}^n$. For the Newton method to be applicable, F must be continuously differentiable.

7.4.1 Multi-Dimensional Differentiaton

Before we actually get started we will discuss multi-dimensional differentation. This is a prerequisite for this section, because the Newton method in multiple dimensions relies on the Jacobian - the equivalent of the differential in multiple dimensions. This has mostly been discussed at some point in Analysis 2.

Derivatives Derivation can be understood as local linearization. $F(x+h) = F(x)+DF(x)h+\mathcal{O}(\|h\|^2)$, which is a good approximation for small $h/h \to 0$. $DF(x)$ is a linear mapping $\mathbb{R}^n \to \mathbb{R}^n$. The following definition of the derivative/Jacobian and Hessian are different from Analysis 2. But they boil down to the same concepts. So understanding the (properly explained) definition from Analysis 2 should suffice.

Definition Multi-Dimensional Derivatives: Let V, W be finite dimensional vector spaces and $F : D \subset V \to W$ a sufficiently smooth mapping. The derivative (differential) $DF(x)$ of F in $x \in V$ is the unique linear mapping $DF(x) : V \to W$ such that there is $\delta > 0$ and a function $\epsilon : [0, \delta] \to \mathbb{R}^+$ satisfying $\lim_{\gamma \to 0} \epsilon(\gamma) = 0$ such that

$$\|F(x+h) - F(x) - DF(x)h\| = \epsilon(\|h\|), \forall h \in V, \|h\| < \delta \tag{1}$$

Definition Gradient and Hessian: For sufficiently smooth $F : D \subset \mathbb{R}^n$ the gradient $gradF : D \to \mathbb{R}^n$, and the Hessian (matrix) $HF(x) : D \to \mathbb{R}^{n,n}$ are defined as

$$grad\ F(x)^\top h := DF(x)h \tag{2}$$

$$h_1^\top HF(x)h_2 := D(DF(x)(h_1))(h_2), \forall h, h_1, h_2 \in \mathbb{R}^n \tag{3}$$

High-Level Differentiation Rules If $F : V \to W$ is a linear map, we just have $DF = F$!

Chain rule For $F : V \to W, G : U \to V$ sufficiently smooth

$$D(F \circ G)(x)h = DF(G(x))(DG(x)h), h \in V, x \in D$$

Product Rule The generalization of the product in 1D are bilinear combinations $b : W \times U \to Z$, i.e., mappings that are linear in each argument. Meaning, they are linear on each argument as usual when the other argument is held fixed.

$T(\mathbf{x}) := b(F(\mathbf{x}), G(\mathbf{x})) \Rightarrow \mathrm{D}T(\mathbf{x})\mathbf{h} = b(\mathrm{D}F(\mathbf{x})\mathbf{h}, G(\mathbf{x})) + b(F(\mathbf{x}), \mathrm{D}G(\mathbf{x})\mathbf{h})$

$$\begin{aligned} T(x+h) &= b(F(x+h), G(x+h)) \\ &= b(F(x) + DF(x)h + o(\|h\|), G(x) + DG(x) + o(\|h\|)) \\ &\overset{(*)}{=} b(F(x), G(x)) + \underbrace{b(DF(x)h, G(x)) + b(F(x), DG(x)h)}_{DT(x)h} + o(\|h\|) \end{aligned}$$

Example: **Derivative of a Bilinear Form/Product Rule**
$\psi : \mathbb{R}^n \to \mathbb{R}, \psi(x) := x^\top Ax, A \in \mathbb{R}^{n,n}$. We use the product rule with $F, G : id : \mathbb{R}^n \to \mathbb{R}^n$ and $b(x, y) = x^\top Ax$.

$$D\psi(x)h = h^\top Ax + x^\top Ah = (x^\top A^\top + x^\top A)h$$
$$\Rightarrow grad\ \psi(x) = (A^\top + A)x \in \mathbb{R}^n$$

Example: **Derivative of Euclidean Norm**
We can compute the Euclidean norm manually by computing partial derivatives. But that is very cumbersome. Instead, we can also write the Euclidean norm $F : \mathbb{R}^n \backslash \{0\} \to \mathbb{R}$, $F(x) = \|x\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$ by splitting it: $F = G \circ H, G(\zeta) := \sqrt{\zeta}, H(x) = x^\top x$. Those components have the derivatives $H(x) = x^\top x$ and $DH(x)h = 2x^\top h$.

$$DF(x)h = DG(H(x))DH(x)h$$
$$= \frac{1}{2\sqrt{x^\top x}} 2x^\top h$$

7.4.2 The Newton Iteration

Just as in one dimension we approximate our function with a linear function and the compute the root of that approximation as the next iterate. With what is known from Analysis we get the model function $\tilde{F}$:

$$\tilde{F}(x) := F(x^{(k)}) + DF(x^{(k)})(x - x^{(k)})$$

In 1D we required that the differential is $\neq 0$. Here we require that $\det DF \neq 0$, i.e., that the Jacobian DF is invertible. This is required so that we have a unique root of the approximation. Then:

$$\tilde{F}(x^{(k+1)}) = 0 : x^{(k+1)} = x^{(k)} - DF(x^{(k)})^{-1} F(x^{(k)})$$

$S := -DF(x^{(k)})^{-1} F(x^{(k)})$ is called the Newton correction. Notice that in practice we clearly don't compute the inverse of the Jacobian but instead solve the $n \times n$ LSE $DF(x^{(k)})S = -F(x^{(k)})$.

Listing 48: Newton Method in C++

```
template <typename FuncType, typename JacType, typename VecType>
VecType newton(
  FuncType &&F,
  JacType &&DFinv,
  VecType x,
  const double rtol,
  const doulbe atol
) {
  VecType s(x.size());
  do {
    s = DFinv(x,F(x));
    x -= s;
  } while((s.norm() > rtol * x.norm()) && (s.norm() > atol));
```

```
14    return x;
15 }
```

Affine Invariance of the Newton Method Consider some function F. Looking for a zero of F is equivalent to seeking a zero of G_A, i.e., $G_A(x) = 0$ where $G_A(x) = A \cdot F(x)$ and $A \in \mathbb{R}^{n,n}$ is regular, because only every root of $F(x)$ is a root of G_A as $DG_A(x) = A \cdot DF(x)$.

$$\begin{aligned} x^{(k+1)} &= x^{(k)} - [A \cdot DF(x^{(k)})]^{-1} AF(x^{(k)}) \\ &= x^{(k)} - DF(x^{(k)})^{-1} A^{-1} AF(x^{(k)}) \\ &= x^{(k)} - DF(x^{(k)}) F(x^{(k)}) \end{aligned}$$

Affine invariance is important, because also the stopping criteria are then affine invariant.

A "Quasi-Linear" System of Equations We consider $A(x) \cdot x = b$ with $b \in \mathbb{R}^n$, $A : \mathbb{R}^n \to \mathbb{R}^n$ continuously differentiable. If A would just be some matrix, this would be a LSE. But as A depends on the solution x, this is a NLSE. With $F(x) := b - A(x)x$ this is equivalent to solving $F(x) = 0$. To apply the Newton method, we must know $DF(x)$, which we determine via the product rule.

$$DF(x)h = (DA(x)h)x + A(x)h$$

To solve the Newton method we need to solve this for s.

$$\begin{gathered} DF(x^{(k)})s = -F(x^{(k)}) \\ \Rightarrow (DA(x^{(k)})s)x^{(k)} + A(x^{(k)})s = A(x^{(k)})x^{(k)} - b \end{gathered}$$

A Special Quasi-Linear System of Equations We consider $A(x)x = b$ with

$$A(x) := \begin{bmatrix} \gamma(x) & 1 & & & & & & \\ 1 & \gamma(x) & 1 & & & & & \\ & 1 & \gamma(x) & 1 & & & & \\ & & \cdots & \cdots & \cdots & & & \\ & & & \cdots & \cdots & \cdots & & \\ & & & & 1 & \gamma(x) & 1 & \\ & & & & & 1 & \gamma(x) & 1 \end{bmatrix} \in \mathbb{R}^{n \times n}$$

$$\gamma(x) = 3 + \|x\|_2$$

$$\Longrightarrow A(x)x = Tx + x\|x\|_2, \quad T := \begin{bmatrix} 3 & 1 & & & & \\ 1 & 3 & 1 & & & \\ & \cdots & 3 & \cdots & & \\ & & \cdots & \cdots & \cdots & \\ & & & 1 & 3 & 1 \\ & & & & 1 & 3 \end{bmatrix}$$

From $F(x) = Tx + x\|x\|_2 - b$ we can compute $DF(x)h$.

$$\begin{aligned} DF(x)h &= D\{x \mapsto Tx\}(x)h + D\{x \mapsto x \cdot \|x\|_2\}(x)h \\ &= Th + h \cdot \|x\|_2 + x\frac{x^\top}{\|x\|_2}h \\ &= (A(x) + \frac{xx^\top}{\|x\|_2})h \end{aligned}$$

Now we can compute the Newton correction using $DF(x)h$:

$$\left(A(x^{(k)}) + \frac{x^{(k)}(x^{(k)})^\top}{\|x^{(k)}\|_2}\right) s = b - A(x^{(k)})x^{(k)}$$

The system matrix is a rank-1 modification of $A(x^{(k)})$, which we have seen in the section on LSEs to be solvable in $\mathcal{O}(n)$ with the Sherman-Morrison-Woodbury Formula as $A(x^{(k)})$ is a triadiagonal matrix and itself easy to solve.

Matrix Inversion by Means of Newton Method We construct a special $F : \mathbb{R}^{n,n} \to \mathbb{R}^{n,n}$ so that the root of F corresponds to the inverse of $A \in \mathbb{R}^{n,n}$. Let $\begin{cases} F : \mathbb{R}^{n,n}_* \to \mathbb{R}^{n,n} \\ F(X) := A - X^{-1} \end{cases}$, with $\mathbb{R}^{n,n}_*$ being the invergible $n \times n$ matrices. As the derivative of F we have $DF(x)H = X^{-1}HX^{-1}, X \in \mathbb{R}^{n,n}_*$. See the next subparagraph to understand deriving matrix inversion.

$$\begin{gathered} x^{(k+1)} = x^{(k)} - s, \quad s := DF(x^{(k)})^{-1}F(x^{(k)}) \\ \Rightarrow DF(x^{(k)})s = (x^{(k)})^{-1}s(x^{(k)})^{-1} = F(x^{(k)}) = A - (x^{(k)})^{-1} \\ \Rightarrow s = x^{(k)}(A - (x^{(k)})^{-1})x^{(k)} = x^{(k)}Ax^{(k)} - x^{(k)} \end{gathered}$$

$$\Rightarrow x^{(k+1)} = x^{(k)} - (x^{(k)}Ax^{(k)} - x^{(k)}) = x^{(k)}(2I - Ax^{(k)})$$

Derivative of Matrix Inversion Before we used the derivative of the matrix inversion operator as part of our computation. Now we show how to get the derivative of matrix inversion, i.e., the derivative of $inv : \begin{cases} \mathbb{R}^{n,n}_* \to \mathbb{R}^{n,n} \\ X \mapsto X^{-1} \end{cases}$. Here we use a technique introduced earlier: implicit differentiation. We differentiate the expression $\mathbf{X}\mathbf{X}^{-1} = \mathbf{I}$ on both sides and then extract the derivative of matrix inversion.

$$\begin{gathered} D(X \cdot inv(X)) = DI \\ \Rightarrow H \cdot inv(X) + X \cdot Dinv(X)H = 0 \end{gathered}$$

$$\Longrightarrow Dinv(X)H = -X^{-1}HX^{-1}$$

Simplified Newton Method This approach trades speed of convergence for less work per interation step. We do so using this iteration:

$$x^{(k+1)} = x^{(k)} - DF(x^{(0)})^{-1}F(x^{(k)}), k \in \mathbb{N}$$

This always considers the same $DF(x^{(0)})^{-1}$ instead of $DF(x^{(k)})^{-1}$. This allows us to reuse the same LU-Decomposition for solving the LSE to get the Newton correction. Hence, the LSE

can be solved in $\mathcal{O}(n^2)$ instead of $\mathcal{O}(n^3)$ as we only need to do the elimination but not the factorization. However, the approximation is also worse. Hence, the speed of convergence is lower.

Listing 49: Simplified Newton Method

```
template <typename Func, typename Jac, typename Vec>
void simpnewton(Vec& x, Func F, Jac DF, double rtol, double atol
    ) {
  auto lu = DF(x).lu();
  Vec s;
  double ns, nx;
  do {
    s = lu.solve(F(x));
    x -= s;
    ns = s.norm(); nx = x.norm();
  } while((ns > rtol*nx) && (ns > atol));
}
```

7.4.3 Convergence of Newton's Method

We justify that the Newton method converges locally quadratic.

The Newton iteration is the fixed point iteration $x^{(k+1)} = \Phi(x^{(k)}), \Phi(x) = x - DF(x)^{-1}F(x)$ as introduced above. Through some transformations based on the Taylor approximation of the error $\|x^{(k+1)} - x^*\|$ we see that it is quadratic in the previous error $\|x^{(k)} - x^*\|$.

$$\begin{aligned}\|x^{(k+1)} - x^*\| &= \|\Phi(x^{(k)}) - \Phi(x^*)\| \\ &= \|D\Phi(x^*)(x^{(k)} - x^*) + \mathcal{O}(\|x^{(k)} - x^*\|^2)\| \\ &= \mathcal{O}(\|x^{(k)} - x^*\|^2) \text{ for } x^{(k)} \text{ close to } x^*\end{aligned}$$

It remains to show why $\|D\Phi(x^*)(x^{(k)} - x^*)\| = 0$ for $x^{(k)}$ close to x^*.

$$\begin{aligned}D\Phi(x)h &= h - \left(D\{x \mapsto DF(x)^{-1}\}(x)h \cdot F(x) + DF(x)^{-1}DF(x)h\right) \\ &= h - D\{x \mapsto DF(x)^{-1}\}(x)h \cdot F(x) - h \\ &= -D\{x \mapsto DF(x)^{-1}\}(x)h \cdot F(x)\end{aligned}$$

$$\begin{aligned}D\Phi(x^*)h &= -D\{x \mapsto DF(x)^{-1}\}(x)h \cdot F(x) \\ &= -D\{\dots\}(x)h \cdot 0 = 0\end{aligned}$$

Local quadratic convergence only holds when the initial guess is sufficiently close to the target value. Unfortunately, it is very difficult to say a priori how close is close enough.

When only looking at one dimension, this is a special case of a more general theorem considered for the 1D case.

Example: As an example for the convergence of the Newton method we will again look at matrix inversion. We have already derived the Newton iteration as $x^{(k+1)} = x^{(k)} - x^{(k)}(x^{(k)}Ax^{(k)} - x^{(k)}) = x^{(k)}(2I - Ax^{(k)})$. The error of this iteration is given by $e^{(k)} = x^{(k)} - A^{-1}$.

$$\begin{aligned} e^{(k+1)} &= x^{(k+1)} - A^{-1} \\ &= x^{(k)}(2I - Ax^{(k)}) - A^{-1} \\ &= (e^{(k)} + A^{-1})(2I - A(e^{(k)} + A^{-1})) - A^{-1} \\ &= (e^{(k)} + A^{-1})(I - Ae^{(k)}) - A^{-1} = -e^{(k)}Ae^{(k)} \end{aligned}$$

For the Euclidean matrix norm we have submultiplicativity: $\|AB\| \leq \|A\| \cdot \|B\|$. Then: $\|e^{(k+1)}\| \leq \|e^{(k)}\|^2\|A\|$. This exhibits local quadratic convergence.
In this special case we can even give a bound for when we have local convergence. Specifically, the error converges to 0 if $\|e^{(0)}A\| = \|x^{(0)}A - I\| < 1$.

7.4.4 Termination of Newton Iteration

$\|x^{(k+1)} - x^*\| \ll \|x^{(k)} - x^*\| \Rightarrow \|x^{(k)} - x^*\| \approx \|x^{(k)} - x^{(k+1)}\|$ explains why the difference between different iterations, i.e., the correction is a good indicator for the error. With $\Delta x^{(k)}$ we denote the absolute correction between $x^{(k)}$ and $x^{(k+1)}$. When working with $\Delta x^{(k)}$ we may stop based on an absolute or relative tolerance: $\|\Delta x^{(k)}\| \leq \tau_{rel}\|x^{(k)}\|$ or $\|\Delta x^{(k)}\| \leq \tau_{abs}$.

One shall remember that the Newton method converges very fast (locally quadratic if it converges). Thus, we often only need a few iteration steps to reach a sufficient solution. If $x^{(k)}$ is sufficient, just computing one more iteration, i.e., $x^{(k+1)}$, corresponds to a significant higher (relative) cost. Each iteration of the Newton method takes $\mathcal{O}(n^3)$ due to the LSE solve. Thus, we want to avoid computing any unnecessary step. This is a problem, because to consider $\Delta x^{(k)}$ for the termination criteria, we must compute $x^{(k+1)}$. But computing that comes at a high cost considering that we arent't actually interested in the value if our current solution is already sufficient.

The idea to avoid the costly operation of computing $x^{(k+1)}$ is to compute an approximation by replacing $DF(x^{(k)})$ with $DF(x^{(k-1)})$. This allows us to reuse the LU-Decomposition already computed for $DF(x^{(k-1)})$ and the solve can be done in $\mathcal{O}(n^2)$.

Listing 50: Simplified/Economical Newton Correction based Newton Method

```
template <typename FuncType, typename JacType, typename VecType>
void newton_src(const FuncType &F, const JacType &DF, VecType &x
    , double rtol, double atol) {
  using scalar_t = typename VecType::Scalar;
  scalar_t sn;
  do {
    auto jacfac = DF(x).lu();
    x -= jafac.solve(F(x));
    sn = jacfac.solve(F(x)).norm();
  } while ((sn>rtol*x.norm()) && (sn>atol));
}
```

$\Delta\tilde{x}$ (the approximation of Δx through the approximation of $x^{(k+1)}$) is affine invariant. It does not change when replacing F with AF where A is some regular matrix.

7.4.5 Damped Newton Method

Remember that the Newton iteration converges locally quadratic. But unfortunately, our initial guess has to be quite close to the result so that we are guaranteed convergence. With the damped newton method we intend to address this challenge of rather limited convergence range. The goal is to enlarge the domain of convergence.

We look at some examples to understand why the Newton Method may fail. The observations lead to introducing damping afterwards.

Example: $F(x) = xe^x - 1$. Notice that $F'(-1) = 0$.

- $x^{(0)} < -1 \Rightarrow x^{(k)} \to -\infty$ (as we have Newton correction in the wrong direction)
- $x^{(0)} > -1 \Rightarrow x^{(k)} \to x^*$

This issue cannot be corrected for as it is caused by the core idea of the Newton correction.

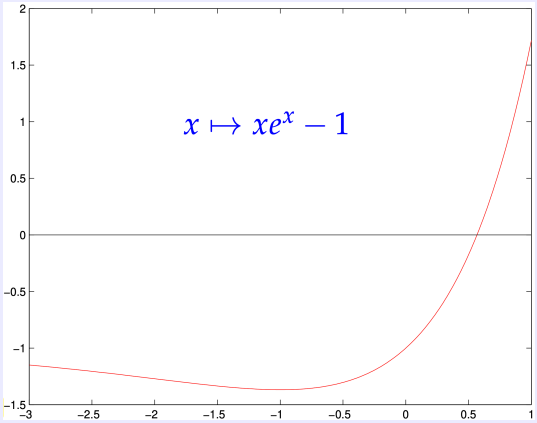


Figure 48: Newton Method Diverging due to 'wrong' Derivative

Example: $F(x) = \arctan(ax), a > 0, x \in \mathbb{R}$ has $x^* = 0$.
Here, we observe overshooting. This can be best understood by looking at the graph.

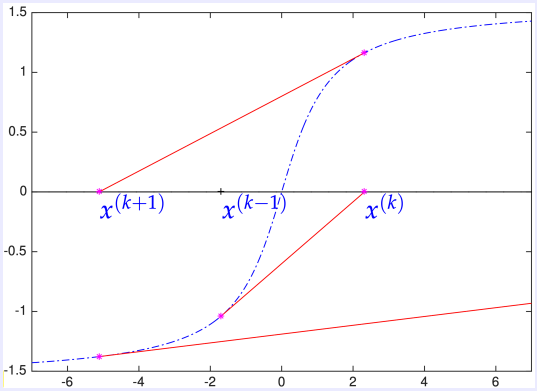


Figure 49: Newton Method Diverging due to Overshooting

The problem of overshooting can be fixed. We can trade speed of convergence for an increased range of convergence. We add a damping factor $\lambda^{(k)}0 \in]0, 1]$ to the iteration. Choosing the

damping factor is non-trivial, because functions behave very differently.

$$x^{(k+1)} := x^{(k)} - \lambda^{(k)} DF(x^{(k)})^{-1} F(x^{(k)})$$

The following is a heuristic. There is no guarantee that this works. To understand the different components of this, consider the explanation afterwards.

Definition Affine Invariant Damping Strategy: We choose the damping factor based on the affine invariant natural monotonicity test (NMT). Specifically, we choose $\lambda^{(k)}$ maximal from $0 < \lambda^{(k)} \leq 1$ such that the following inequality still holds.

$$\|\Delta\overline{x}(\lambda^{(k)})\|_2 \leq (1 - \frac{\lambda^{(k)}}{2})\|\Delta x^{(k)}\|_2$$

- $\Delta x^{(k)} := DF(x^{(k)})^{-1} F(x^{(k)})$ is the current Newton correction
- $\Delta\overline{x}(\lambda^{(k)}) := DF(x^{(k)})^{-1} F(x^{(k)} - \lambda^{(k)}\Delta x^{(k)})$ is the simplified Newton correction
  $x^{(k)} - \lambda^{(k)}\Delta x^{(k)}$ is the tentative next iterate

$\Delta x^{(k)}$ is the Newton correction from the current $x^{(k)}$ to $x^{(k+1)}$. $\Delta\overline{x}(\lambda^{(k)})$ approximates the Newton correction from $x^{(k+1)}$ to $x^{(k+2)}$. The precise Newton correction for that step would be $DF(x^{(k+1)})^{-1} F(x^{(k+1)}) = DF(x^{(k+1)})^{-1} F(x^{(k)} - \Delta x^{(k)})$. Thus, the approximation differs in that (a) we use the simplified Newton correction with $DF(x^{(k)})^{-1}$ instead of $DF(x^{(k+1)})^{-1}$, which we do to avoid having to compute $DF(x^{(k+1)})^{-1}$, and (b) add the damping factor $\lambda^{(k)}$.

The inequality then tests whether the Newton correction decreases in size. Wanting to choose the maximum $\lambda^{(k)}$ corresponds to wanting to avoid damping. But we only avoid damping if it allows us to shrink the Newton correction. Otherwise we introduce damping to such an extend that we still shrink the Newton correction reasonably, which is expressed through the factor $(1 - \frac{\lambda^{(k)}}{2})$.

For sufficiently close $x^{(k)}$, quadratic convergence clearly allows us to choose $\lambda^{(k)} = 1$ as $\|\Delta\overline{x}(\lambda^{(k)})\|_2 = \|\Delta x^{(k+1)}\|_2 \leq \frac{1}{2}\|\Delta x^{(k)}\|_2 = (1 - \frac{\lambda^{(k)}}{2})\|\Delta x^{(k)}\|_2$. This means we require no damping for quadratic convergence. But if The inequality doesn't hold, we introduce the damping factor by decreasing $\lambda^{(k)}$.

- NMT fails: $\lambda^{(k)} \mapsto \frac{\lambda^{(k)}}{2}$ & retry
- NMT passed: do the update according to damping formula, increment $k \mapsto k + 1$, and use less damping initially in the next step with $\lambda \mapsto 2\lambda$.

Listing 51: Damped Newton Method

```
template <typename FuncType, typename JacType, typename VecType>
void dampnewton(const FuncType &F, const JacType &DF, VecType &x
    , double rtol, double atol) {
  using index_t = typename VecType::Index;
  using scalar_t = typename VecType::Scalar;
  const index_t n = x.size(); // number of unknowns
  const scalar_t lmin = 1E-3; // minimal damping factor
  scalar_t lambda = 1.0; // initial and actual damping factor
  VecType s(n), st(n); // newton corrections
  VecType xn(n); // tentative new iterate
  scalar_t sn, stn; // norms of newton corrections

```

```
  do {
    auto jacfac = DF(x).lu(); // LU-factorize Jacobian
    s = jacfac.solve(F(x)); // newton correction
    sn = s.norm(); // norm of newton correction
    lambda *= 2.0;
    do {
      lambda /= 2; // reduce damping factor
      if (lambda < lmin) throw "No convergence: lambda -> 0";
      xn = x-lambda*s; // tentative next iterate
      st = jacfac.solve(F(xn)); // simplified newton correction
      stn = st.norm();
    } while(stn > (1-lambda/2)*sn); // natural monotonicity test
    x = xn; // xn accepted as new iterate
    lambda = std::min(2.0*lambda, 1.0); // try to mitigate
       damping
  } while((stn>rtol*x.norm()) && (stn>atol));
}
```

Notice that this entire heuristic is affine invariant (as its name already suggests). This can be easily seen as $\Delta x^{(k)}$ and $\Delta\overline{x}(\lambda^{(k)})$ are not affected by replacing F with AF, A being an invertible matrix.

Example: $F(x) = \arctan(x)$ was already considered in the beginning to motivate damping. Now we will see that applying the Heuristic actually increases the range of convergence. We consider $x^{(0)} = 20$ (a point that is rather far away from the target $x^* = 0$) and a lower bound for damping of 0.001.

k	$\lambda^{(k)}$	$x^{(k)}$	$F\left(x^{(k)}\right)$
1	0.03125	0.94199967624205	0.75554074974604
2	0.06250	0.85287592931991	0.70616132170387
3	0.12500	0.70039827977515	0.61099321623952
4	0.25000	0.47271811131169	0.44158487422833
5	0.50000	0.20258686348037	0.19988168667351
6	1.00000	-0.00549825489514	-0.00549819949059
7	1.00000	0.00000011081045	0.00000011081045
8	1.00000	-0.00000000000001	-0.00000000000001

Figure 50: Dampted Newton Method on $\arctan$

We see that initially, the damping factor is very small to accommodate for much overshooting. But within several iterations, we are sufficiently close to the target so that the algorithm goes back to quadratic convergence.

But with $F(x) = xe^x - 1$ and $x^{(0)} = -1.5$, this wouldn't work. The damping factor would continue to go down until we reach the minimum. Then, one could terminate/bail out.

7.5 Quasi-Newton Methods

One drawback of the Newton method is that it requires the derivative of the function F. In many applications that may not be available. Thus, we now consider derivative-free variations of the

Newton method.

For one dimension we already discussed the secant method, where we replace the derivative with an approximation based on the last two iterates to interpolate a line.

$$x^{(k+1)} = x^{(k)} - \frac{x^{(k)} - x^{(k-1)}}{F(x^{(k)}) - F(x^{(k-1)})} F(x^{(k)})$$

For higher dimensions $n > 1$ we generalize by replacing $DF(x^{(k)})$ with an approximation of the Jacobian $J_k \in \mathbb{R}^{n,n}$. One basic idea for this is to approximate all partial derivatives of F by means of difference quotient. But that is computationally extremely expensive, because computing n^2 difference quotients requires many evaluations of F.

Now, we specify two conditions for J_k. First, the **secant condition** so that $J_k \approx DF(x^{(k)})$.

$$J_k(x^{(k)} - x^{(k-1)}) = F(x^{(k)}) - F(x^{(k-1)})$$

This leads to the iteration $x^{(k+1)} = x^{(k)} - J_k^{-1} F(x^{(k)})$. But the secant condition does not uniquely determine J_k, because we only specify its incline in one direction. But in a multi-dimensional space also all other directions must be considered. The idea for the remaining dimensions is to reuse J_{k-1}. **Broyden's condition** requires that J_k and J_{k-1} are identical in all directions not covered by the secant condition.

$$J_k z = J_{k-1} z, \forall z : z \perp (x^{(k)} - x^{(k-1)})$$

Broyden's condition and the secant condition together uniquely determine J_k.

Definition Broyden's Quasi-Newton Method:

$$x^{(k+1)} := x^{(k)} + \Delta x^{(k)}, \Delta x^{(k)} := -J_k^{-1} F(x^{(k)})$$
$$J_{k+1} := J_k + \frac{F(x^{(k+1)})(\Delta x^{(k)})^\top}{\|\Delta x^{(k)}\|_2^2}$$

That this satisfies the conditions specified earlier can be seen when multiplying with $\Delta x^{(k)}$ or with a vector defined to be orthogonal. In the former case, $J_k \Delta x^{(k)} = -F(x^{(k)})$ and the right fraction becomes $F(x^{(k+1)})$. In the latter case, the right fraction becomes zero as the dot product between $\Delta x^{(k)}$ and a vector orthogonal to it is zero.

Notice that we require an initial guess for J_0. If we can compute the Jacobian, we can take $J_0 := DF(x^{(0)})$.

Convergence of Broyden's Method We only considered anecdotal evidence. The concluson is that th 'pure' Newton method converges locally quadratically. Broyen's method also converegs nicely but not quite quadratic. But it performs still noticeably better than the simplified Newton method.

Implementation of Boryden's Method In this paragraph we consider a smart way to actually compute J_k. The idea relies on the observation that according to the above definition J_{k+1} is a rank-1 modification of J_k. In the section on solving LSEs, we discussed the special case of updating the solution of a LSE given a rank-1 modification of the system matrix. We will apply that here.

The Sherman-Morrison-Woodburry formula was used to update the solutoins given some low-rank modification. For a rank-1 modification the formula is:

$$(A + uv^\top)^{-1} = \left(I - \frac{A^{-1}uv^\top}{1 + v^\top A^{-1}u}\right) A^{-1}, \quad \text{assuming } 1 + v^\top A^{-1}u \neq 0$$

When choosing $u := F(x^{(k+1)})$ and $v := \frac{\Delta x^{(k)}}{\|\Delta x^{(k)}\|_2^2}$ we get an update formula for J_k^{-1}:

$$J_{k+1}^{-1} = \left(I - \frac{J_k^{-1}F(x^{(k+1)})(\Delta x^{(k)})^\top}{\|\Delta x^{(k)}\|_2^2 + \Delta x^{(k)} \cdot J_k^{-1}F(x^{(k+1)})}\right) J_k^{-1} = \left(I - \frac{\Delta\overline{x}^{(k+1)}(\Delta x^{(k)})^\top}{\|\Delta x^{(k)}\|_2^2 + \Delta x^{(k)} \cdot \Delta\overline{x}^{(k+1)}}\right) J_k^{-1}$$

where $\Delta\overline{x}^{(k+1)} := J_k^{-1}F(x^{(k+1)})$ is called the simplified quasi-newton correction. This vector is very helpful for simplified termination and damping. Thus, computing it is not wasteful.

In an algorithmic implementation we of course have to check that we never divide by zero. For that we test whether $|(\Delta x^{(k)})^\top \Delta\overline{x}^{(k+1)}| < 1$. This can be expected, because those vectors should be rather small in the asymptotic phase if the method converges nicely.

For the k-th update of our iteration we get:

$$\Delta x^{(k)} = -J_k^{-1}F(x^{(k)}) = -\prod_{l=0}^{k-1}\left(I + \frac{\Delta\overline{x}^{(l+1)}(\Delta x^{(l)})^\top}{1 + (\Delta x^{(l)})^\top \Delta\overline{x}^{(l+1)}}\right) \cdot J_0^{-1}F(x^{(k)})$$

But clearly we don't always perform k matrix multiplications. Instead, we only take the previous correction and multiply one rank-1 matrix to get the next correction. With $\Delta x^{(k)} := -t_k$:

$$t_{l+1} := t_l + \Delta\overline{x}^{(l+1)}\frac{(\Delta x^{(l)})^\top t_l}{1 + (\Delta x^{(l)})^\top \Delta\overline{x}^{(l+1)}}, l = 0, ..., k-1$$

For the entire algorithm we first compute the LU-Decomposition of J_0 so that we can compute $J_0^{-1}F(x^{(k)})$ for all k in $\mathcal{O}(n^2)$ after this initial $\mathcal{O}(n^3)$ investion. Then, we must do the above iteration for each k up to our 'target k', because we always have a different initial $F(x^{(k)})$. For each iteration, we do two triangular solves to get t_0 in $\mathcal{O}(n^2)$. Afterwards, each recursive step is done in $\mathcal{O}(n)$. So for k steps we have $\mathcal{O}(kn)$. For N steps, i.e., different k values, in total, we get $\mathcal{O}(n^3 + Nn^2 + N^2n)$.

The quality of the result can be adversely affected by roundoff due to the use of the Sherman-Morrison-Woodbury formula. Thus, this algorithm may not be very stable. One could use updates with QR instead of LU. That would increase complexity and cost.