NumCSE - Complete (Ch. 1-3)

Additional files	
Class	NumCSE
Date	@Oct 25, 2020
PPT	NumCSE_Lecture_Document.pdf
Topic	Lecture Document Ch. 1-3
Type	Book

0. Table of Content

1. Computing with Matrices and Vectors

1.1 Fundamentals

1.1.1 Notations

In this lecture we denote *column vectors* with x and *row vectors* with x^T. For example, $\mathbb{K}^n$ denotes the vector space of column vectors whereas $\mathbb{K}^{1,n}$ denotes the vector space of row vectors. We denote *sub vectors* with $(x)_{k:l} = [x_k,\ ...,\ x_l]^T,\ 1 \leq k \leq l \leq n$.

We denote *matrices* with X. In this lecture, matrices are two-dimensional arrays of real or complex numbers:

$$A := \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix} \in \mathbb{K}^{n,m}$$

Here, $\mathbb{K}^{n,m}$ denotes the vector space of $n \times m$-matrices, where n is the number of rows and m is the number of columns. We denote *matrix blocks* with $(A)_{k:l,r:s} := [a_{ij}]_{i=k,\dots l;\ j=r,\dots,s}$. We define the *(hermitian) transposed matrix* as:

$$A^H := \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix}^H := \begin{bmatrix} \overline{a}_{11} & \cdots & \overline{a}_{n1} \\ \vdots & & \vdots \\ \overline{a}_{1m} & \cdots & \overline{a}_{mn} \end{bmatrix}$$

1.2 Software and Libraries

1.2.1 Eigen

`Eigen` is a header-only C++ template library designed to enable easy, natural and efficient numerical linear algebra: it provides data structures and a wide range of operations for matrices and vectors.

A generic matrix data type is given by the templated class:

```
Eigen::Matrix<typename Scalar, int RowsAtCompileTime, int ColsAtCompileTime>
```

Here `Scalar` is the underlying scalar type of the matrix entries, usually either double, float or complex<>.

Matrix and vector data types in Eigen

```
#include <Eigen/Dense>

template <typename Scalar>
void eigenTypeDemo(unsigned int dim) {
  // General dynamic matrices
  using dynMat_t = Eigen::Matrix<Scalar, Eigen::Dynamic, Eigen::Dynamic>;
  // Dynamic column vectors
  using dynColVec_t = Eigen::Matrix<Scalar, Eigen::Dynamic, 1>;
  using index_t = typename dynMat_t::Index;
  using entry_t = typename dynMat_t::Scalar;
  // Declare vectors of size 'dim', not yet initialized
  dynColVec_t colvec(dim);
  // Initialisation through component access
  for(index_t i = 0; i < colvec.size(); ++i) colvec[i] = (Scalar)i;
}
```

The following convenience data types are provided by Eigen:

- `MatrixXd` is a generic variable size matrix with *double precision* entries
- `VectorXd, RowVectorXd` is a dynamic column or row vector
- `MatrixNd` with $N = 2, 3, 4$ for small fixed size square $N \times N$-matrices
- `VectorNd` with $N = 2, 3, 4$ for small column vector with fixed length N

The `d` in the type name may be replaced with `i` for `int`, `f` for `float`, and `cd` for `complex<double>` to select another basic scalar type. All matrix type feature the methods `cols(), rows()`, and `size()` telling the number of columns, rows, and total number of entries.

Initialization of dense matrices in Eigen

```
#include <Eigen/Dense>
// Just allocate space for matrix, no initialisation
Eigen::MatrixXd A(rows, cols);
// Zero matrix
Eigen::MatrixXd O = MatrixXd::Zero(rows, cols);
// Ones matrix
Eigen::MatrixXd B = MatrixXd::Ones(rows, cols);
// Matrix with all entries same as value
Eigen::MatrixXd C = MatrixXd::Constant(rows, cols, value);
// Random matrix, entries uniformly distributed in [0,1]
Eigen::MatrixXd E = MatrixXd::Random(rows, cols);
// Identity matrix
Eigen::MatrixXd I = MatrixXd::Identity(rows, cols);
std::cout << "size of A = (" << A.rows() << ',' << A.cols() ')' << std:endl;
```

Access to sub-matrices in Eigen

The method `block(int i, int j, int p, int q)` returns a reference to the sub-matrix with upper left corner at position (i, j) and size $p \times q$. The methods `row(int i)` and `col(int j)` provide a reference to the corresponding row and column of the matrix.

Remark

If you want a C++ code built using the Eigen library to run fast, for instance, for large computations or runtime measurements, you should compile in *release mode*, that is, with the compiler switches `-O2 -DNDEBUG` (for gcc or clang).

1.2.3 Dense Matrix Storage Formats

All numerical libraries store the entries of a (generic = *dense*) matrix $A \in \mathbb{K}^{m,n}$ in a linear array of length mn. Accessing entries entails suitable index computations.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$
$$\text{Row major: } [1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9]$$
$$\text{Column major: } [1,\ 4,\ 7,\ 2,\ 5,\ 8,\ 3,\ 6,\ 9]$$

1.4 Computational Effort

Large scale numerical computations require immense resources and execution time of numerical codes often becomes a central concern. Therefore, much emphasis has to be putt on

1. designing algorithms that produce a desired result with nearly minimal computational effort
2. exploit possibilities for parallel and vectorised execution
3. organising algorithms in order to make them fit memory hierarchies
4. implementing codes that make optimal use of hardware resources and capabilities

Definition: The *computational effort* required by a numerical code amounts to the number of elementary operations (additions, subtractions, multiplications, divisions, square roots) executed in a run.

1.4.1 Asymptotic Computational Complexity

Definition: The *asymptotic (computational) complexity* of an algorithm characterises the worst-case dependence of it's computational effort on one or more problem size parameters when these tend to ∞.

1.4.2 Cost of Basic Linear-Algebra Operations

Performing elementary BLAS-type operations through simple (nested) loops, we arrive at the following complexity bounds:

Computational Cost of Basic Numerical Linear Algebra Operations

Operation	Description	#mul/div	#add/sub	asymptotic complexity
dot product	$(x \in \mathbb{R}^n, y \in \mathbb{R}^n) \to x^H y$	n	n-1	O(n)
tensor product	$(x \in \mathbb{R}^m, y \in \mathbb{R}^n) \to xy^H$	nm	0	O(mn)
matrix * vector	$(x \in \mathbb{R}^n, A \in \mathbb{R}^{m,n}) \to Ax$	mn	(n-1)m	O(mn)
matrix product	$(A \in \mathbb{R}^{m,n}, B \in \mathbb{R}^{n,k}) \to AB$	mnk	mk(n-1)	O(mnk)

1.4.3 Improving Complexity in Numerical Linear Algebra

Efficient associative matrix multiplication

Given $a \in \mathbb{K}^m,\ b \in \mathbb{K}^n,\ x \in \mathbb{K}^n$ we may compute the vector $y = ab^T x$ in two ways:

1. $y = (ab^T)x$, `T = (a*b.transpose())*x;` → complexity $O(mn)$
2. $y = a(b^T x)$, `t = a*b.dot(x)` → complexity $O(m+n)$

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

$$A \otimes B := \begin{bmatrix} (A)_{11}B & (A)_{12}B & \cdots & (A)_{1n}B \\ (A)_{21}B & (A)_{22}B & \cdots & (A)_{2n}B \\ \vdots & \vdots & & \vdots \\ (A)_{m1}B & (A)_{m2}B & \cdots & (A)_{mn}B \end{bmatrix} \in \mathbb{K}^{ml,\, nk}$$

Efficient multiplication of Kronecker product with vector in Eigen

```
template <class Matrix, class Vector>
void kronmultv(const Matrix &A, const Matrix &B, const Vector &x, Vector &y) {
  unsigned int m = a.rows(); unsigned int n = A.cols();
  unsigned int l = B.rows(); unsigned int k = B.cols();
  Matrix t = B * Matrix::Map(x.data(), k, n) * A.transpose();
  y = Matrix::Map(t.data(), m*l, 1);
}
```

1.5 Machine Arithmetic and Consequences

1.5.2 Machine Numbers

The reason, why computers must fail to execute exact computations with real numbers is clear: computers are finite automatons, which therefore can only handle *finitely many number*, not $\mathbb{R}$. This is an essential property: $\mathbb{M}$, the set of *machine numbers*, is a finite, discrete subset of $\mathbb{R}$.

> The set of machine numbers $\mathbb{M}$ cannot be close under elementary arithmetic operations $+,\ -,\ \cdot,\ /$, that is, when adding, multiplying, etc., two machine numbers the result may not belong to $\mathbb{M}$. The results of elementary operations with operands in $\mathbb{M}$ have to be mapped back to $\mathbb{M}$, an operation called **rounding.** This leads to the fact, that **roundoff errors** are *inevitable*.

1.5.3 Roundoff Errors

Definition: Let $\tilde{x} \in \mathbb{K}$ be an approximation of $x \in \mathbb{K}$. Then the **absolute error** is given by

$$\epsilon_{\text{abs}} := |x - \tilde{x}|,$$

and its **relative error** is defined as

$$\epsilon_{\text{rel}} := \frac{|x - \tilde{x}|}{|x|}.$$

The *number of correct digits* of an approximation $\tilde{x}$ of $x \in \mathbb{K}$ is defined through the relative error: If $\epsilon_{\text{rel}} \leq 10^{-l}$, then $\tilde{x}$ has l correct digits, $l \in \mathbb{N}_0$.

Definition: Correct rounding is given by the function

$$\text{rd}: \mathbb{R} \to \mathbb{M},\ x \to \max \operatorname{argmin}_{\tilde{x} \in \mathbb{M}} |x - \tilde{x}|$$

1.5.4 Cancellation

We define the term *cancellation* as the subtraction of almost equal numbers (with both having some relative error), which leads to an extreme amplification of the relative errors.

Example: Assume two numbers 2.0 and 1.9, both with a relative error of 5%, i.e. 2.1 and 1.81. The subtraction of both number should yield 0.1, but due to the relative errors it yields 0.29, almost three times as big as expected.

2. Direct Methods for Square Linear Systems of Equations

2.1 Introduction: Linear Systems of Equations (LSE)

The problem: solving a linear system

We are given the following input and are looking for the output as shown below:

- Input/data: *square* matrix $A \in \mathbb{K}^{n,n}$, vector $b \in \mathbb{K}^n$
- Output/result: solution vector $x \in \mathbb{K}^n$, such that $Ax = b$

We call A the *system matrix* or *coefficient matrix* and b the *right hand side vector*.

2.2 Theory: Linear Systems of Equations (LSE)

2.2.1 LSE: Existence and Uniqueness of Solutions

Definition: We say that $A \in \mathbb{K}^{n,n}$ is **invertibel/regular** iff. $\exists B \in \mathbb{K}^{n,n} : AB = BA = I$. B is called the **inverse** of A and is noted as $B = A^{-1}$.

Definition: Given $A \in \mathbb{K}^{m,n}$, the **range/image** (space) of A is the subspace of $\mathbb{K}^m$ spanned by the columns of A

$$\mathcal{R}(A) := \{Ax,\ x \in \mathbb{K}^n\} \subset \mathbb{R}^m.$$

The **kernel/nullspace** of A is

$$\mathcal{N}(A) := \{z \in \mathbb{R}^n : Az = 0\}.$$

Definition: The **rank** of a matrix $M \in \mathbb{K}^{m,n}$, denoted by $\text{rank}(M)$, is the maximal number of linearly independent rows/columns of M. Equivalently, $\text{rank}(M) = \dim \mathcal{R}(A)$.

> **Theorem:** A square matrix $A \in \mathbb{K}^{n,n}$ is **invertible/regular** if one of the following *equivalent* conditions is satisfied:
> 1. $\exists B \in \mathbb{K}^{n,n} : BA = AB = I$
> 2. the columns or rows of A are linearly independent
> 3. $\det(A) \neq 0$
> 4. $\text{rank}(A) = n$

2.3 Gaussian Elimination (GE)

2.3.1 Basic Algorithm

The idea of Gaussian elimination is the transformation of a linear system of equations into a "simpler", but equivalent LSE by means of successive *row transformations*. We define row transformations as a *left multiplication with a transformation matrix*.

Obviously, left multiplication with a regular matrix **does not affect the solution of an LSE**, formally: for any *regular* $T \in \mathbb{K}^{n,n}$

$$Ax = b \Rightarrow A'x = b', \text{ if } A' = TA,\ b' = Tb.$$

Gaussian elimination: algorithm

```
void gausseliminationsolve(const MatrixXd &A, const VectorXd& b, VectorXd& x) {
  int n = A.rows();
  MatrixXd Ab(n, n++);
  Ab << A, b;
  // Forward elimination
  for(int i = 0; i < n-1; ++i) {
    double pivot = Ab(i, i);
    for(int l = i+1; k < n; ++k) {
      double fac = Ab(k, i) / pivot;
      Ab.block(k, i+1, 1, n-i) -= fac * Ab.block(i, i+1, 1, n-1);
    }
  }
  // Back substitution
  Ab(n-1, n) = Ab(n-1, n) / Ab(n-1, n-1);
  for(int i = n-2; i >= 0; --i) {
    for(int l = i+1; l < n; ++l) {Ab(i, n) -= Ab(l, n) * Ab(i, l);}
    Ab(i, n) /= Ab(i, i);
  }
  x = Ab.rightCols(1); // Solution in the right most column!
}
```

The computational cost of Gaussian elimination is given by

- forward elimination: $n(n-1)(\frac{2}{3}n + \frac{7}{6})$ Ops.
- backward elimination: n^2 Ops.

which yields a total **asymptotic complexity for GE** without pivoting for a generic LSE: $\frac{2}{3}n^3 + O(n^2) = O(n^3)$.

2.3.2 LU-Decomposition

A *matrix factorization* expresses a general matrix A as the product of two special matrices. Requirements for these special matrices define the matrix factorization.

We can perform **LU-Decomposition** by performing the known Gaussian elimination algorithm, but keeping track of the *negative multipliers* and let them take the places of matrix entries mate to vanish:

$$\begin{bmatrix}1 & & \\ & 1 & \\ & & 1\end{bmatrix}\begin{bmatrix}1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1\end{bmatrix}\begin{bmatrix}4 \\ 1 \\ -3\end{bmatrix} \Rightarrow \begin{bmatrix}1 & & \\ 2 & 1 & \\ & & 1\end{bmatrix}\begin{bmatrix}1 & 1 & 0 \\ 0 & -1 & -1 \\ 3 & -1 & -1\end{bmatrix}\begin{bmatrix}4 \\ -7 \\ -3\end{bmatrix} \Rightarrow \begin{bmatrix}1 & & \\ 2 & 1 & \\ 3 & & 1\end{bmatrix}\begin{bmatrix}1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & -4 & -1\end{bmatrix}\begin{bmatrix}4 \\ -7 \\ -15\end{bmatrix} \Rightarrow \begin{bmatrix}1 & & \\ 2 & 1 & \\ 3 & 4 & 1\end{bmatrix}\begin{bmatrix}1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & 3\end{bmatrix}\begin{bmatrix}4 \\ -7 \\ 13\end{bmatrix}$$

After performing the above Gaussian elimination, we get the following decomposition

$$A = LU \Rightarrow \begin{bmatrix}1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1\end{bmatrix} = \begin{bmatrix}1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 4 & 1\end{bmatrix}\begin{bmatrix}1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & 3\end{bmatrix}$$

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

LU-factorization

```
std::pair<MatrixXd, MatrixXd> lufac(const MatrixXd &A) {
  int n = A.rows();
  assert(n == A.cols());
  MatrixXd L{MatrixXd::Identity(n, n)};
  MatrixXd U{MatrixXd::Zero(n, n)};
  for(int k = 0; k < n; ++k) {
    for(int j = k; j < n; ++j) {
      U(k, j) = A(k, j) - (L.block(k, 0, 1, k) * U.block(0, j, k, 1))(0, 0);
    }
    for(int i = k + 1; i < n; ++i) {
      L(i, k) = (A(i, k) - (L.block(i, 0, 1, k) * U.block(0, k, k, 1))(0, 0)) / U(k, k);
    }
  }
  return { L, U };
}
```

The **asymptotic complexity for LU-factorization** of $A \in \mathbb{R}^{n,n}$ is given by $\frac{2}{3}n^3 + O(n^2) = O(n^3)$ if $n \to \infty$.

2.3.3 Pivoting

When doing *pivoting* in numerical methods we usually choose the **relatively larges pivot**, defined as

$$j \in \{k, ..., n\} \text{ such that } \frac{|a_{ij}|}{\max\{|a_{jl}|,\ l = k, ..., n\}} \to \max$$

for $k = j,\ k \in \{i, ..., n\}$, also called **partial pivoting.**

> **Lemma:** For any regular $A \in \mathbb{K}^{n,n}$ there is a permutation matrix $P \in {}^{n,n}$, a normalized lower triangular matrix $L \in \mathbb{K}^{n,n}$, and a regular upper triangular matrix $U \in \mathbb{K}^{n,n}$, such that $PA = LU$.

Performing explicit LU-factorization in Eigen

```
const Eigen::MatrixXd::Index n = A.cols();
assert(n == A.rows());
Eigen::PartialPivLU<MatrixXd> lu(A);
MatrixXd L = MatrixXd::Identity(n, n);
L.triangularView<StrictlyLower>() += lu.matrixLu();
MatrixXd U = lu.matrixLU().triangularView<Upper>();
MatrixXd P = lu.permutationP();
```

2.6 Exploiting Structure when Solving Linear Systems

By *structure* of a linear system we mean prior knowledge that

- either certain entries of the systems vanish,
- or the system matrix is generated by a particular formula.

Triangular linear systems

Triangular linear systems are linear systems of equations whose system matrix is a triangular matrix. They can be solved by backward/forward elimination with $O(n^2)$ asymptotic computational effort compared to an asymptotic complexity of $O(n^3)$ for a generic dense matrix.

Linear Systems with arrow matrices

From $n \in \mathbb{N}$, a diagonal matrix $D \in \mathbb{K}^{n,n},\ c \in \mathbb{K}^n,\ b \in \mathbb{K}^n$, and $\alpha \in \mathbb{K}$, we can build an $(n+1) \times (n+1)$ **arrow matrix**

$$A = \begin{bmatrix} D & c \\ & \\ b & \alpha \end{bmatrix} \Rightarrow \begin{bmatrix} \ddots & & \vdots \\ & \ddots & \vdots \\ \cdots & \cdots & \cdot \end{bmatrix}$$

In this case we have, for a LSE of the form $Ax = b$, that

$$Ax = \begin{bmatrix} D & c \\ b^T & \alpha \end{bmatrix} \begin{bmatrix} x_1 \\ \zeta \end{bmatrix} = y := \begin{bmatrix} y_1 \\ \eta \end{bmatrix}$$
$$\Rightarrow \zeta = \frac{\eta - b^T D^{-1} y_1}{\alpha - b^T D^{-1} c},\ x_1 = D^{-1}(y_1 - \zeta c).$$

Solving an arrow system in Eigen

```
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);
  if(std::abs(den) < std::numeric_limits<double>::epsilon() * (b.norm() + std::abs(alpha))) {
    throw std::runtime_error("Nearly singular system");
  }
  constt double xi = (y(n) - b.dot(w)) / den;
  return (VectorXd(n+1) << w - xi * z, xi).finished();
}
```

This code snippet yields an **asymptotic complexity for solving arrow systems** of $O(n)$ for $n \to \infty$.

Solving LSE subject to low-rank modification of system matrix

Given a regular matrix $A \in \mathbb{K}^{n,n}$, let us assume that at some point in a code we are in a position to solve any linear system $Ax = b$ "fast" because

- either A has a favorable structure, eg. triangular,
- or an LU-decomposition of A is already available

Now, a $\tilde{A}$ is obtained by changing a *single entry* of A:

$$A, \tilde{A} \in \mathbb{K}^{n,n} : \tilde{a}_{ij} \text{ if } (i,j) \neq (i^*, j^*),\ z + a_{ij} \text{ otherwise},\ i^*, j^* \in \{1, ..., n\}$$
$$\Rightarrow \tilde{A} = A + z \cdot e_{i^*} e_{j^*}^T.$$

We may also consider a matrix modification affecting a single row: *Changing a single row*: given $z \in \mathbb{K}^n$

$$A, \tilde{A} \in \mathbb{K}^{n,n} : \tilde{a}_{ij} = a_{ij} \text{ if } i \neq i^*,\ (z)_j + a_{ij} \text{ otherwise},\ i^*, j^* \in \{1, ..., n\}$$
$$\Rightarrow \tilde{A} = A + e_{i^*} z^T$$

Both of the above mentioned modifications represent so-called **rank-1-modifications** of A. A generic rank-1-modification reads

$$A \in \mathbb{K}^{n,n} \to \tilde{A} := A + uv^H\ ,\ u, v \in \mathbb{K}^n.$$

We consider the block partitioned linear system

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

The *Schur complement* system after elimination of ζ reads $(A + uv^H)\tilde{x} = b \Leftrightarrow \tilde{A}\tilde{x} = b$. We do block elimination again, now getting rid of $\tilde{x}$ first, which yields the other Schur complement system

$$\begin{aligned} &(1 + v^H A^{-1}u)\zeta = v^H A^{-1}b \\ &\Rightarrow A\tilde{x} = b - \frac{uv^H A^{-1}}{1 + v^H A^{-1}u} b. \end{aligned}$$

2.7 Sparse Linear Systems

We start with a rather fuzzy classification of matrices according to their number os zero:

Notion: $A \in \mathbb{K}^{n,n},\ m, n \in \mathbb{N}$ is said to be **sparse**, if

$$\mathrm{nnz}(A) := \#\{(i, j) \in \{1, ..., m\} \times \{1, ..., n\} : a_{ij} \neq 0\} << mn.$$

The matrix is said to be **dense** otherwise.

2.7.1 Sparse Matrix Storage Formats

Sparse matrix storage formats for storing a sparse matrix $A \in \mathbb{K}^{m,n}$ are designed to achieve two objectives:

1. Amount of memory required is only slightly more than $\mathrm{nnz}(A)$ scalars.
2. Computational effort for matrix $\times$vector multiplication is proportional to $\mathrm{nnz}(A)$.

Triplet/coordinate list (*COO*) format

This format stores triplets $(i,\ j,\ \alpha),\ 1 \leq i \leq m,\ 1 \leq j \leq n$:

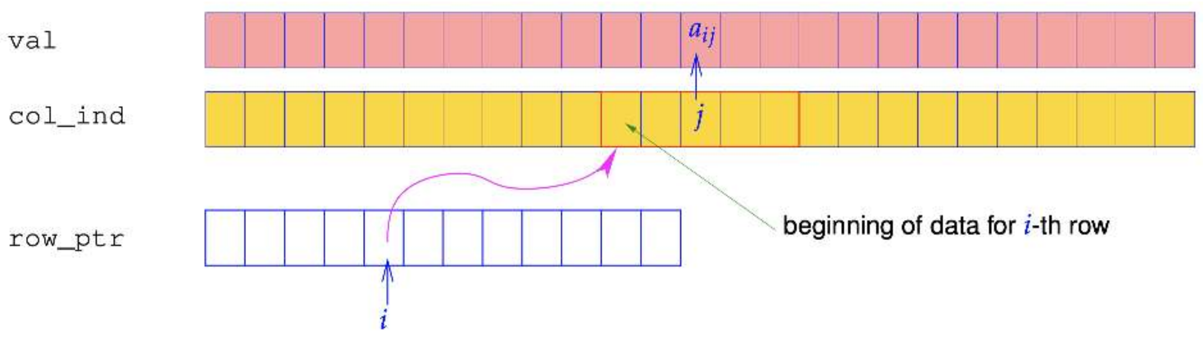
```
struct Triplet {
  size_t i;      // row index
  size_t j;      // column index
  scalar_t a;    // additive contribution to matrix entry
};
using TripletMatrix = sttd::vector<Triplet>;
```

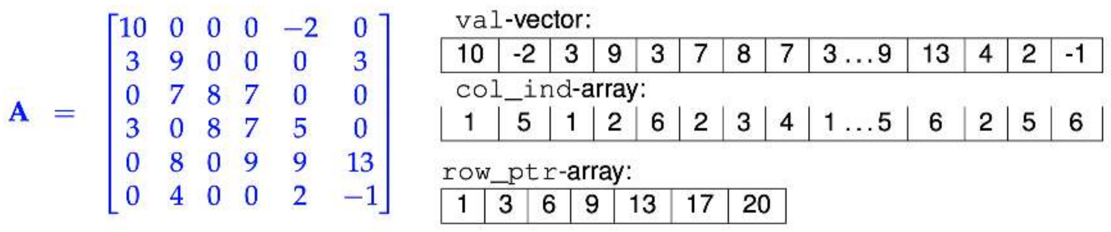
The vector of triplets in a `TripletMatrix` has size $\geq \text{nnz}(A)$. We write $\geq$ because *repetitions* of index pairs (i, j) are allowed. The matrix entry $(A)_{ij}$ is defined to be the sum of all values α_{ij} associated with the index pair (i, j).

Compressed row-storage (*CRS*) format

The CRS format for a sparse matrix $A \in \mathbb{K}^{n,n}$ keeps the data in three contiguous arrays:

- `std::vector<scalar_t> val` → size $\text{nnz}(A)$
- `std::vector<size_t> col_ind` → size $\text{nnz}(A)$
- `std:vector<size_t> row_ptr` → size $n + 1$ and `row_ptr[n+1]` = $\text{nnz}(A) + 1$




$\mathbf{A} = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}$
val-vector:
10 | -2 | 3 | 9 | 3 | 7 | 8 | 7 | 3 . . . 9 | 13 | 4 | 2 | -1
col_ind-array:
1 | 5 | 1 | 2 | 6 | 2 | 3 | 4 | 1 . . . 5 | 6 | 2 | 5 | 6
row_ptr-array:
1 | 3 | 6 | 9 | 13 | 17 | 20

2.7.2 Sparse Matrices in Eigen

Eigen can handle sparse matrices in the standard Compressed Row Storage (*CRS*) and Compressed Column Storage (*CCS*) format:

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

Matrices should first be assembled in *triplet format*, from which as sparse matrix is built. Eigen offers special data types and facilities for handling triplets:

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

Furthermore, a triplet object can be initialized as demonstrated in the following example:

```
unsigned int row_idx = 2;
unsigned int col_idx = 4;
double value = 2.5;
Eigen::Triplet<double> triplet(row_idx, col_idx, value);
std::cout <<'(' triplet.row() << ',' << triplet.col()
          <<',' triplet.value() << ')' << std::endl;
```

2.7.3 Direct Solution of Sparse Linear Systems of Equations

The standard sparse solver in Eigen is `SparseLU` :

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

> When solving linear systems of equations directly, **dedicated sparse elimination solver** from numerical libraries have to be used! System matrices are passed to these algorithms in sparse storage formats to convey information about zero entries.
> *Never ever* even think about implementing a general sparse elimination solver by yourself!

3. Direct Methods for Linear Least Squares Problems

In this chapter we study numerical methods for **overdetermined (*OD*)** linear systems of equations, that is, a linear system with a "tall" rectangular system matrix:

$$"Ax = b" : x \in \mathbb{R}^n,\ b \in \mathbb{R}^m,\ A \in \mathbb{R}^{m,n},\ m \geq n$$

$$\begin{bmatrix} \\ \\ A \\ \\ \\ \end{bmatrix} \begin{bmatrix} \\ x \\ \\ \end{bmatrix} = \begin{bmatrix} \\ \\ b \\ \\ \\ \end{bmatrix}$$

Note that the quotation marks indicate that this is not a well-defined problem in the sense of that $Ax = b$ does not define a mapping $(Ax) \to b$ because

- such a vector $x \in \mathbb{R}^n$ may not exist,

- and, even if it exists, it may not be unique.

3.1 Least Squares Solution Concepts

Throughout this chapter we consider the possibly overdetermined linear system of equations

$$x \in \mathbb{R}^n : \text{"}Ax = b\text{"},\ b \in \mathbb{R}^m,\ A \in \mathbb{R}^{m,n},\ m \geq n.$$

Recall from linear algebra that $Ax = b$ has a solution, if and only if the right hand side vector b lies in the image of the matrix A:

$$\exists x \in \mathbb{R}^n : Ax = b \Leftrightarrow b \in \mathcal{R}(A).$$

Following the notation for important subspaces associated with a matrix $A \in \mathbb{K}^{m,n}$:

- *image/range*: $\mathcal{R}(A) := \{Ax,\ x \in \mathbb{K}^n\} \subset \mathbb{K}^m$,
- *kernel/nullspace*: $\mathcal{N}(A) := \{x \in \mathbb{K}^n : Ax = 0\}$.

3.1.1 Least Squares Solutions: Definitions

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

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

In other words, a least squares solution is any vector x that minimizes the Euclidean norm of the **residual** $r = b - Ax$.

We write $\operatorname{lsq}(A, b)$ for the set of least squares solutions of the linear system of equations $Ax = b,\ A \in \mathbb{R}^{m,n},\ b \in \mathbb{R}^m$:

$$\operatorname{lsq}(A, b) := \{x \in \mathbb{R}^n : x \text{ is a least squares solution of } Ax = b\} \subset \mathbb{R}^n.$$

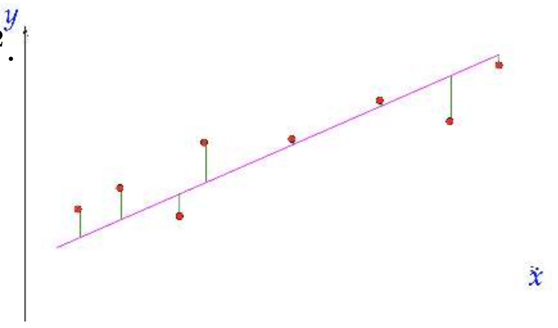
Example: Linear regression

- Given: measured data points $(x_i,\ y_i),\ x_i \in \mathbb{R},\ i = 1, \dots, m,\ m \geq n + 1$
- Known: without measurement errors, data would satisfy *affine linear relationship* $y = a^T x + \beta$, for some parameters $a \in \mathbb{R}^n,\ \beta \in \mathbb{R}$.

Solving the overdetermined linear system of equations in least squares sense we obtain a *least squares estimate* for the parameters a and β

$$(a, \beta) = \operatorname{argmin}_{a \in \mathbb{R}^n,\, \beta \in \mathbb{R}} \sum_{i=1}^{m} |y_i - a^T x_i - \beta|^2.$$

In statistics, solving this equation is known as a **linear regressions.**



Theorem: For any $A \in \mathbb{R}^{m,n},\ b \in \mathbb{R}^m$ a least squares solution of $Ax = b$ exists.

3.1.2 Normal Equations

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

$$A^T A x = A^T b.$$

Theorem: For $A \in \mathbb{R}^{m,n},\ m \geq n$, holds

$$\mathcal{N}(A^T A) = \mathcal{N}(A),$$
$$\mathcal{R}(A^T A) = \mathcal{R}(A).$$

Lemma: For any matrix $A \in \mathbb{K}^{m,n}$ holds

$$\mathcal{N}(A) = \mathcal{R}(A^H)^\perp$$
$$\mathcal{N}(A)^\perp = \mathcal{R}(A^H).$$

We define the **orthogonal complement** of a subspace $V \subset \mathbb{K}^k$:

$$V^\perp := \{x \in \mathbb{K}^k : x^H y = 0\ \forall y \in V\}.$$

Corollary: If $m \geq n$ and $\mathcal{N}(A) = \{0\}$, then the linear system of equations $Ax = b,\ A \in \mathbb{R}^{m,n},\ b \in \mathbb{R}^m$, has a **unique least squares solution**

$$x = (A^T A)^{-1} A^T b,$$

that can be obtained by solving the *normal equations*.

The assumption that $\mathcal{N}(A) = 0$ is also called a **full-rank condition (*FRC*),** because $\mathcal{N}(A) = 0 \Leftrightarrow \operatorname{rank}(A) = n.$

3.1.3 Moore-Penrose Pseudoinverse

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

$$x^\dagger := \operatorname{argmin}\{||x||_2 : x \in \operatorname{lsq}(A, b)\}.$$

In other words, the generalized solution is the least squares solution with *minimal norm*.

Theorem: Given $A \in \mathbb{R}^{m,n},\ b \in \mathbb{R}^m$, the generalized solution $x^\dagger$ of the linear system of equations $Ax = b$ is given by

$$x^\dagger = V(V^TA^TAV)^{-1}(V^TA^Tb),$$

where V is any matrix whose columns form a basis of $\mathcal{N}(A)^\perp$.

The matrix $A^\dagger := V(V^TA^TAV)^{-1}V^TA^T \in \mathbb{R}^{n,m}$ is called the **Moore-Penrose pseudoinverse** of A. Note, that the Moore-Penrose pseudoinverse does *not depend* on the choice of V.

3.2 Normal Equation Methods

We can give a simple algorithm for the normal equation method for solving *full-rank* least squares problems $Ax = b$:

1. Compute *regular* matrix $C := A^TA \in \mathbb{R}^{n,n}$.
2. Compute right hand side vector $c := A^Tb$.
3. Solve symmetric positive definite (*s.p.d.*) linear system of equations $Cx = c$.

Definition: $M \in \mathbb{K}^{n,n}$ is **symmetric (Hermitian) positive definite (*s.p.d.*),** if

$$M = M^H \text{ and } \forall x \in \mathbb{K}^n : x^HMx > 0 \Leftrightarrow x \neq 0.$$

If $x^HMx \geq 0$ for all $x \in \mathbb{K}^n$, we say that M is *positive semi-definite*.

Solving a linear least squares problem via 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;
}
```

The asymptotic complexity of the normal equation method is given by $O(n^2m + n^3)$ for $m, n \to \infty$.

3.3 Orthogonal Transformation Methods

3.3.1 Transformation Idea

In this chapter we consider the full-rank linear least squares problem $A \in \mathbb{R}m, n,\ b \in \mathbb{R}^m$ given and we try to find $x = \operatorname{argmin}_{y \in \mathbb{R}^n} ||Ay - b||_2$. We furthermore know that $m \geq n$ and A has full rank: $\operatorname{rank}(A) = n$.

Furthermore it is to note, that we call two overdetermined linear systems $Ax = b$ and $\tilde{A}x = \tilde{b}$ equivalent, if both have the same set of least squares solutions: $\operatorname{lsq}(A, b) = \operatorname{lsq}(\tilde{A}, \tilde{b})$.

The idea is that if we have a transformation matrix $T \in \mathbb{R}^{m,m}$ satisfying $||Ty||_2 = ||y||_2\ \forall y \in \mathbb{R}^m$, then

$$\operatorname{argmin}_{y \in \mathbb{R}^n} ||Ay - b||_2 = \operatorname{argmin}_{y \in \mathbb{R}^n} ||\tilde{A}y - \tilde{b}||_2,$$

where $\tilde{A} = TA$ and $\tilde{b} = Tb$.

3.3.2 Orthogonal/Unitary Matrices

Definition: Unitary and **orthogonal matrices**

- $Q \in \mathbb{K}^{n,n},\ n \in \mathbb{N}$, is *unitary*, if $Q^{-1} = Q^H$
- $Q \in \mathbb{K}^{n,n},\ n \in \mathbb{N}$, is *orthogonal*, if $Q^{-1} = Q^T$

Theorem: A matrix is *unitary/orthogonal,* if and only if the associated linear mapping preserves the 2-norm:

$$Q \in \mathbb{K}^{n,n} \text{ unitary } \iff ||Qx||_2 = ||x||_2\ \forall x \in \mathbb{K}^n.$$

From the above theorem we can directly state the following conclusions. If a matrix $Q \in \mathbb{K}^{n,n}$ is unitary/orthogonal, then

- all rows/columns have Euclidean norm $= 1$
- all rows/columns are pairwise orthogonal (w.r.t Euclidean inner product)
- $|\det Q| = 1,\ ||Q||_2 = 1$, and all eigenvalues $\in \{z \in \mathbb{C} : |z| = 1\}$
- $||QA||_2 = ||A||_2$ for any matrix $A \in \mathbb{K}^{n,m}$

3.3.3 QR-Decomposition

3.3.3.1 QR-Decomposition: Theory

We first recall the **Gram-Schmidt orthogonalization** algorithm as follows:

- Input: $\{a^1, ..., a^n\} \subset \mathbb{K}^m$

- Output: $\{q^1, ...q^n\}$

The algorithm is given by:

```
- $q^1 := \frac{a^1}{||a^1||_2}$
- for $j = 2, ..., n$ do {
    - $q^j := a^j$;
    - for $l = 1, 2, ..., j-1$ do{
        - $q^j \leftarrow q^j - < a^j, q^l > q^l$; }
    - if $(q^j = 0)$ then STOP
    - else{ $q^j \leftarrow \frac{q^j}{||q^j||_2}$ }}
```

Theorem: If $\{a^1, ..., a^n\} \subset \mathbb{R}^m$ is linearly independent, then the Gram-Schmidt algorithm computes **orthogonal vectors** $q^1, ...q^n \in \mathbb{R}^m$ satisfying

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

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

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

1. a unique matrix $Q_0 \in \mathbb{R}^{n,k}$ that satisfies $Q_0^H Q_0 = I_k$, and a unique *upper triangular* Matrix $R_0 \in \mathbb{K}^{k,k}$ with $(R)_{ii} > 0,\ i \in \{1, ..., k\}$, such that

$$A = Q_0 \cdot R_0 \text{ (”economical” QR-decomposition)}$$

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

$$A = Q \cdot R \text{ (full QR-decomposition)}$$

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

3.3.3.2 Computation of QR-Decomposition

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

The following so called **Householder matrices (*HHM*)** effect the reflection of a vector into a multiple of the first unit vector with the same length:

$$Q = H(v) := I - 2\frac{vv^T}{v^T v} \text{ with } v = a \pm ||a||_2 e_1$$

where e_1 is the first Cartesian basis vector.

Suitable **successive Householder transformations** determined by the left most column of shrinking bottom right matrix blocks can be used to achieve upper triangular form R. Writing Q_l for the Householder matrix used in the l-th factorization yields for the **QR-decomposition** of $A \in \mathbb{C}^{n,n},\ A = QR$:

$$Q_{n-1} \cdot Q_{n-2} \cdots Q_1 A = R \text{ and } Q := Q_1^T \cdots Q_{n-1}^T.$$

The following orthogonal transformation, a **Givens rotation,** annihilates the k-th component of a vector $a = [a_1, ..., a_n]^T \in \mathbb{R}^n$. Here γ stands for $\cos\phi$ and σ for $\sin\phi$, ϕ the angle of rotation:

$$G_{1k}(a_1, a_k)a := \begin{bmatrix} \gamma & \cdots & \sigma & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots \\ -\sigma & \cdots & \gamma & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} a_1 \\ \vdots \\ 0 \\ \vdots \\ a_n \end{bmatrix},$$
$$\gamma = \frac{a_1}{\sqrt{|a_1|^2 + |a_k|^2}},\ \sigma = \frac{a_k}{\sqrt{|a_1|^2 + |a_k|^2}}$$

The QR-decomposition by successive Householder transformations has asymptotic complexity $O(mn^2)$ for $m, n \to \infty$.

Definition: For $A = (a_{ij})_{i,j} \in \mathbb{K}^{m,n}$ we call

$$\overline{bw}(A) := \min\{k \in \mathbb{N} : j - i > k \Rightarrow a_{ij} = 0\} \text{ the upper bandwidth,}$$
$$\underline{bw}(A) := \min\{k \in \mathbb{N} : i - j > k \Rightarrow a_{ij} = 0\}\text{the lower bandwidth.}$$

Furthermore we define $\mathrm{bw}(A) := \overline{bw}(A) + \underline{bw}(A) + 1$ as the **bandwidth** of A.

Theorem: If $A = QR$ is the QR-decomposition of a regular matrix, then $A \in \mathbb{R}^{n,n}$, then $\mathrm{bw}(R) \leq \mathrm{bw}(A)$.

3.3.3.3 QR-Decomposition: Stability

It is important to note, that unitary/orthogonal transformations do **not** involve any *amplification of relative errors* in data vectors.

Theorem: Let $\tilde{R} \in \mathbb{R}^{m,n}$ be the R-factor of the QR-decomposition of $A \in \mathbb{R}^{m,n}$ computed by means of successive Householder reflections. Then there exists an orthogonal $Q \in \mathbb{R}^{m,m}$ such that

$$A + \Delta A = Q\tilde{R} \text{ with } ||\Delta A||_2 \leq \frac{cmn\,\mathrm{EPS}}{1 - cmn\,\mathrm{EPS}}||A||_2,$$

where EPS is the machine precision and $c > 0$ a small constant independent of A.

3.3.3.4 QR-Decomposition in Eigen

```
#include <Eigen/QR>

std::pair<MatrixXd, MatrixXd> qr_decomp_eco(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);
}
```

3.3.4 QR-Based Solver for Linear Least Squares Problems

We consider the full-rank linear least squares problem: Given $A \in \mathbb{R}^{m,n},\ m \geq n,\ \mathrm{rank}(A) = n$, seek $x \in \mathbb{R}^n$ such that $||Ax - b||_2 \to \min$. We assume that we are *given a QR-decomposition*: $A = QR,\ Q \in \mathbb{R}^{m,m}$ orthogonal, $R \in \mathbb{R}^{m,n}$ regular upper triangular matrix.

We then apply the orthogonal 2-norm preserving transformation encoded in Q to $Ax - b$:

$$||AX - b||_2 = ||QRx - b||_2 = ||Q(Rx - Q^T b)||_2 = ||Rx - \tilde{b}||_2,\ \tilde{b} := Q^T b.$$

Eigen's built-in QR-based linear least squares solver

```
double lsqsolve_eigen(const MatrixXd& A, const VectorXd& b, VectorXd& x) {
  x = A.householderQr().solve(b);
  return ((A*x - b).norm());
}
```

Remark:

- Computing generalized QR-decomposition $A = QR$ by means of Householder reflections or Givens rotations is *numerically stable* for any $A \in \mathbb{C}^{m,n}$.
- For any regular system matrix an LSE can be solved by means of "QR-decomposition + orthogonal transformation + backward substitution" in a *stable manner*.

Normal equations vs. orthogonal transformations methods

- Use *orthogonal transformation methods* for least squares problems, whenever $A \in \mathbb{R}^{m,n}$ is dense and n is small.
- Use *normal equations* in the expanded form, when $A \in \mathbb{R}^{m,n}$ is sparse and m, n are big.

3.3.5 Modification Techniques for QR-Decomposition

3.5.5.1 Rank-1 Modifications

For $A \in \mathbb{R}^{m,n},\ m \geq n,\ \text{rank}(A) = n$, we consider the **rank-1 modification**

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

Given a full QR-decomposition $A = QR = \begin{bmatrix} R_0 \\ 0 \end{bmatrix},\ Q \in \mathbb{R}^{m,m}$ orthogonal, $R \in \mathbb{R}^{m,n}$ and $R_0 \in \mathbb{R}n, n$ upper triangular, the goal is to find an efficient algorithm that yields a QR-decomposition of $\tilde{A} = \tilde{Q}\tilde{R},\ \tilde{Q} \in \mathbb{R}^{m,m}$ a product of orthogonal transformations, $\tilde{R} \in \mathbb{R}^{n,n}$ upper triangular:

1. Compute $w = Q^T u \in \mathbb{R}^m$
2. Orthogonally transform $w \to ||w||e_1,\ e_1 \in \mathbb{R}^m$, this can be done by applying $m-1$ ivens rotations from bottom to top.
3. Convert $R_1 + ||w||_2 e_1 v^T \in \mathbb{R}^{n,n}$ into upper triangular form by $n-1$ Givens rotations .

$$\tilde{A} = A + uv^T = \tilde{Q}\tilde{R} \text{ with } \tilde{Q} = QQ_1^T G_{12}^T G_{23}^T \cdots G_{n-1,n-2}^T G_{n,n-1}^T$$

3.4 Singular Value Decomposition (SVD)

3.4.1 SVD: Definition and Theory

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

$$A = U\Sigma V^H.$$

Definition: The decomposition $A = U\Sigma V^H$ is called the **singular value decomposition (*SVD*)** of A. The diagonal entries of σ_i of Σ are the *singular values* of A. The columns of U/V are the left/right *singular vectors* of A.

Remark: As in the case of QR-decomposition we can also drop the bottom zero rows of Σ and the corresponding columns of U in the case of $m > n$. Thus we end up with an **economical singular value decomposition,** also called **thin SVD** in literature.

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

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

- $\text{rank}(A) = r$ (the number of non-zero singular values)

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

3.4.2 SVD in Eigen

Computing SVDs 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);
}
```

It holds that

- Eigen's algorithm for computing SVD is numerically stable
- The asymptotic complexity for the economical SVD is $O\left(\min\{m,n\}^2 \cdot \max\{m,n\}\right)$

Computing rank of matrix through SVD

```
MatrixXd::Index rank_by_svd(const MatrixXd &A, double tol = EPS) {
  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;
}
```

Computation using rank() in Eigen

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

3.4.3 Solving General Least-Squares Problems by SVD

In this chapter we consider the most general setting

$$Ax = b \in \mathbb{R}^m \text{ with } A \in \mathbb{R}^{m,n},\ \text{rank}(A) = r \leq \min\{m,n\}.$$

We can use the invariance of the 2-norm of a vector with respect to multiplication with $U := [U_1\ U_2]$ together with the fact that U is unitary:

$$||Ax-b||_2 = \left|\left|[U_1\ U_2]\begin{bmatrix}\Sigma_r & 0\\ 0 & 0\end{bmatrix}\begin{bmatrix}V_1^T\\ V_2^T\end{bmatrix}x - b\right|\right|_2 = \left|\left|\begin{bmatrix}\Sigma_r V_1^T x\\ 0\end{bmatrix} - \begin{bmatrix}U_1^T b\\ U_2^T b\end{bmatrix}\right|\right|_2.$$

With this equation we arrive at the generalized solution

$$x^\dagger = V_1\Sigma_r^{-1}U_1^T b,\ ||r||_2 = ||U_2^T b||_2.$$

Computing generalized solution of $Ax = b$ via SVD

```
#include <Eigen/SVD>

VectorXd lsqsvd(const MatrixXd &A, const VectorXd &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));
}
```

Computation via solve() method

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

Theorem: If $A \in \mathbb{K}^{m,n}$ has the SVD decomposition $A = U\Sigma V^H$ then its *Moore-Penrose pseudoinverse* is given by $A^\dagger = V_1\Sigma_r^{-1}U_1^H$.

3.4.4 SVD-Based Optimization and Approximation

3.4.4.1 Norm-Constrained Extrema of Quadratic Forms

We consider the following problem of finding the extrema of quadratic forms on the Euclidean unit sphere $\{x \in \mathbb{K}^n : ||x||_2 = 1\}$:

$$\text{given } A \in \mathbb{K}^{m,n},\ m \geq n,\ \text{find } x \in \mathbb{K}^n,\ ||x||_2 = 1,\ \ ||Ax||_2 \to \min.$$

This problem can be solved with SVD with the minimizer $x^* = Ve_n = (V)_{:,n}$ from which we can obtain the minimal value $||Ax^*||_2 = \sigma_n$.

Solving the minima problem with SVD in Eigen

```
double minconst(VectorXd &x, const MatrixXd &A) {
  MatrixXd::Index m = A.rows(), n = A.cols();
  if(m < n) throw std::runtimer_error("A must be tall matrix");
  Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeThinV);
  x.resize(n); x.setZero(); x(n-1) = 1.0;
  x = svd.matrixV() * x;
  return (svd.singularValues())(n-1);
}
```

Lemma: If $A \in \mathbb{K}^{m,n}$ has singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0,\ p := \min\{m, n\}$, then its Euclidean matrix norm is given by $||A||_2 = \sigma_1(A)$. If $m = n$ and A is regular/invertible, then its 2-norm condition number is $\text{cond}_2(A) = \sigma_1/\sigma_n$.

3.4.4.2 Best Low-Rank Approximation

Thomas: TLDR for the best k-rank approximation you turn the sigma into $k \times k$ matrix (cut everything else away) then you take away the columns in U and V accordingly.

3.4.4.3 Principal Component Data Analysis (PCA)

whatever

3.6 Constrained Least Squares

We define **linear least squares problems with linear constraints** as follows:

- Given:
  - $A \in \mathbb{R}^{m,n},\ m \geq n,\ \text{rank}(A) = n,\ b \in \mathbb{R}^m$
  - $C \in \mathbb{R}^{p,n},\ p < n,\ \text{rank}(C) = p,\ d \in \mathbb{R}^p$
- Find $x \in \mathbb{R}^n$ such that:

$$||Ax - b||_2 \to \min,\ \text{and } Cx = d.$$

This problem can be solved via SVD the following way:

1. Compute an orthonormal basis of $\mathcal{N}(C)$ using SVD

$$C = U[\Sigma\ 0]\begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix},\ U \in \mathbb{R}^{p,p},\ \Sigma \in \mathbb{R}^{p,p},\ V_1 \in \mathbb{R}^{n,p},\ V_2 \in \mathbb{R}^{n,n-p}$$
$$\Rightarrow \mathcal{N}(C) = \mathcal{R}(V_2)$$

   and the particular solution $x_0 \in \mathcal{N}(C)^T = \mathcal{R}(V_1)$ of the constraint equation

$$x_0 := V_1\Sigma^{-1}U^Td.$$

   This gives us a representation of the solution x of the form

$$x = x_0 + V_2 y,\ y \in \mathbb{R}^{n-p}.$$

2. Insert this representation into the LSQ problem. This yields s standard linear least squares problem with coefficient matrix $AV_2 \in \mathbb{R}^{m,n-p}$ and a right hand side vector $b - Ax_0 \in \mathbb{R}^m$:

$$||A(x_0 + V_2 y) - b||_2 \to \min \iff ||AV_2 y - (b - Ax_0)||_2 \to \min.$$

4. Midterm Prep-Questions

4.1 HS 2019

4.1.1 Rank-1 Modifications

▼ A rank-1 modification of $A \in \mathbb{R}^{n,n}$ affects at most $2n - 1$ entries of the matrix.

False - Choosing $u = (1,\ 1, ..., 1)^T$ and $v = (1,\ 1, ...,\ 1)^T$ will add 1 to *every* entry in A.

▼ If $\tilde{A}$ is a rank-1 modification of $A \in \mathbb{R}^{n,n}$, then $\text{rank}(A) - 1 \leq \text{rank}(\tilde{A}) \leq \text{rank}(A) + 1$.

True - The outer product of two vectors (i.e. uv^T) always produces a rank-1 matrix. Furthermore it holds in general, that $\text{rank}(A + B) \leq \text{rank}(A) + \text{rank}(B)$.

▼ For every matrix $A \in \mathbb{R}^{n,n}$ there is an invertible $\tilde{A}$ arising from a rank-1 modification of A.

False - A matrix A is only invertible if it has full rank. For any matrix A with $\text{rank}(A) < n - 1$ we therefore cannot reach a full rank by a rank-1 modification.

▼ By rank-1 modification every matrix $A \in \mathbb{R}^{n,n}$ can be converted into a singular (non-invertible) matrix.

True - Take for example $u := (A)_{:,1},\ v := -e_1$. Then the first column of A will vanish in $\tilde{A}$, and this will result in $\tilde{A}$ not being a full-rank matrix (and therefore also not invertible)

▼ Let $\tilde{A}$ be the matrix arising from $A \in \mathbb{R}^{m,n}$ by replacing it's k-th row $(A)_{k,:}$ with w^T, where $w \in \mathbb{R}^n$ is a given vector. What rank-1 modification of A spawns $\tilde{A}$. Give u and v such that $\tilde{A} = A + uv^T$.

We can choose $u = e_k$ and $v = w - ((A)_{k,:})^T$.

4.1.2 Computational cost of numerical linear algebra operations

```
double sumtrv1(const Eigen::MatrixXd &A, const Eigen::VectorXd &b) {
  const int n = A.cols();
  assert((A.rows() == n) && (b.size() == n));

  return b.transpose() *
         A.triangularView<Eigen::Upper>().solve(
         Eigen::MatrixXd::Identity(n,n)) * b;
}
```

▼ Asymptotic complexity for $n \to \infty$: $O(n),\ O(n^2),\ O(n^3)$ or $O(n^4)$?

$O(n^3)$ - We are solving n linear systems of equations with an $n \times n$ upper triangular system matrix. This amounts to n backward substitutions, each of which costs $O(n^2)$ operations.

```
double sumtrv2(const Eigen::MatrixXd &A, const Eigen::VectorXd &b) {
  const int n = A.cols();
  assert((A.rows() == n) && (b.size() == n));

  return b.transpose() * A.triangularView<Eigen::Upper>().solve(b);
}
```

▼ Asymptotic complexity for $n \to \infty$: $O(n),\ O(n^2),\ O(n^3)$ or $O(n^4)$?

$O(n^2)$ - We solve a single $n \times n$ upper triangular linear system of equations. Solving such a linear system of equations takes $O(n^2)$ operations. (The vector multiplications do not matter since they require $O(n)$ operations)

```
Eigen::VectorXd diagmodsolve1(Eigen::MatrixXd A, const Eigen::VectorXd &b) {
  const int n = A.cols();
  assert((A.rows() == n) && (b.size() == n));
  Eigen::VectorXd x{Eigen::VectorXd::Zero(n)};
  double tmp = A(0, 0);
  for(int i = 0; i < n; ++i) {
    if(i > 0) {
      A(i-1, i-1) = tmp;
    }
    tmp = A(i, i);
    A(i, i) *= 2.0;
    x += A.lu().solve(b);
  }
  return x;
}
```

▼ Asymptotic complexity for $n \to \infty$: $O(n),\ O(n^2),\ O(n^3)$ or $O(n^4)$?

$O(n^4)$ - We solve a $n \times n$ dense linear system of equations n times.

```
Eigen::VectorXd diagmodsolve2(const Eigen::MatrixXd &A, const Eigen::VectorXd &b) {
  const int n = A.cols();
  assert ((A.rows() == n) && (b.size() == n));
  const auto Alu = A.lu();
  const auto z = Alu.solve(b);
  const auto W = Alu.solve(Eigen::MatrixXd::Identity(n, n));
  const Eigen::VectorXd alpha = Eigen::VectorXd::Constant(n, 1.0) +
                                A.diagonal().cwiseProduct(W.diagonal());
  if((alpha.cwiseAbs().array() < 1E-12).any()) {
    throw std::runtime_error("Tiny pivot!");
  }
  return n * z - W * z.cwiseProduct(A.diagonal().cwiseQuotient(alpha));
}
```

▼ Asymptotic complexity for $n \to \infty$: $O(n),\ O(n^2),\ O(n^3)$ or $O(n^4)$?

$O(n^3)$ - Due to the LU-decomposition of an $n \times n$ densely populated matrix in Line 4.

4.1.3 Cancellation

```
double f1(double x) { return std::log(std::sqrt(x * x + 1) - x); }
```

▼ No cancellation or cancellation? If there is cancellation, give $x \simeq$??? and a cancellation-free implementation of the function.

Cancellation for $x \simeq +\infty$. Use the equation $a - b = \frac{a^2-b^2}{a+b}$ to prevent cancellation. Cancellation-free implementation:

```
double f1(double x) {
  return (x > 0.0) ? -std::log((std::sqrt(x * x + 1) + x)) :
                     std::log((std::sqrt(x * x + 1) - x));
}
```

```
double f2(double x) {
  assert(x > 0);
  return std::log(x * x + 1) - 2 * std::log(x);
}
```

▼ No cancellation or cancellation? If there is cancellation, give $x \simeq$??? and a cancellation-free implementation of the function.

Cancellation for $x \simeq +\infty$. Use the equation $\log a - \log b = \log \frac{a}{b}$ to prevent cancellation. Cancellation-free implementation:

```
double f2(double x) {
  assert(x > 0);
  const double y = 1.0 / x;
  return std::log(y * y + 1);
}
```

```
double f3(double x) {
  assert((x >= -1) && (x <= 1));
  return 1 - std::sqrt(1 - x * x);
}
```

▼ No cancellation or cancellation? If there is cancellation, give $x \simeq ???$ and a cancellation-free implementation of the function.

Cancellation for $x \simeq 0.$ Use $a - b = \frac{a^2-b^2}{a+b}$ to prevent cancellation. Cancellation-free implementation:

```
double f3(double x) {
  assert((x >= -1) && (x <= 1));
  const double s = x * x;
  return s / (1 + std::sqrt(1 - s));
}
```

```
double f4(double x) {
  const double s = std::cos(x);
  return std::sqrt(1 - s * s);
}
```

▼ No cancellation or cancellation? If there is cancellation, give $x \simeq ???$ and a cancellation-free implementation of the function.

Cancellation for $x \simeq \cdots, -\pi,\ 0, \pi,\ 2\pi,\ \cdots$. Use the trigonometric identity $\cos^2 x + \sin^2 x = 1$ to prevent cancellation. Cancellation-free implementation:

```
double f4(double x) {
  return std::abs(std::sin(x));
}
```

4.2 HS 2018

4.2.1 Singular Value Decomposition

Let $A \in \mathbb{R}^{3,2}$ be defined as

$$A = \begin{pmatrix} 0 & 2 \\ 1 & 0 \\ 0 & 0 \end{pmatrix}$$

▼ What are the non-zero singular values of A?

We first calculate AA^T get the eigenvalues of A:

$$A^TA = \begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$$

We then form the characteristic polynomial A with $\det A^T A - \lambda I$:

$$\det A^T A - \lambda I = (1-\lambda)\cdot(4-\lambda) - 0\cdot 0 = (1-\lambda)\cdot(4-\lambda).$$

This polynomial has roots 1 and 4 and therefore the two singular values are given by:

$$\sigma_1 = \sqrt{1} = 1 \text{ and } \sigma_2 = \sqrt{4} = 2.$$

▼ Consider the *full* singular value decomposition $A = U\Sigma V^T$ of A. Determine $a,\ b,\ \alpha,\ \beta \in \mathbb{N}$ such that $U \in \mathbb{R}^{a,b}$ and $V \in \mathbb{R}^{\alpha,\beta}$.

Per definition of the full singular value decomposition, $U \in \mathbb{K}^{m,m}$ and $V \in \mathbb{K}^{n,n}$. We therefore have:

$$U \in \mathbb{R}^{3,3} \text{ and } V \in \mathbb{R}^{2,2}.$$

▼ Consider the *reduced* singular value decomposition $A = \tilde{U}\tilde{\Sigma}\tilde{V}^T$ of A. Determine $a,\ b,\ \alpha,\ \beta \in \mathbb{N}$ such that $\tilde{U} \in \mathbb{R}^{a,b}$ and $\tilde{V} \in \mathbb{R}^{\alpha,\beta}$.

Since we only have 2 singular values, $\Sigma \in \mathbb{R}^{2,2}$. From this it follows, that $U \in \mathbb{R}^{3,2}$ and $\tilde{V} \in \mathbb{R}^{2,2}$.

▼ Let $\tilde{A} \in \mathbb{R}^{3,2}$ be t he best rank-1 approximation of A. Let $||\cdot||_F$ denote the Frobenius norm. What is the value $||A - \tilde{A}||_F$? *unfinished*

We first recall the definition of the Frobenius norm:

$$||A||_F = \sqrt{\Sigma_{i=1}^m \Sigma_{j=1}^n |a_{ij}|_2}$$

Furthermore, we recall that the best rank-1 approximation is defined as

4.2.2 Asymptotic Complexity

Consider the following Eigen/C++ code:

```
MatrixXd A = MatrixXd::Zero(n, n);

A(0, 0) = 1.0; A(1, 0) = 1.0;
for(int j = 1; j < n-1; ++j) {
  for(int i = j-1; i < j + 2; ++i) {
    A(i, j) = 1.0;
  }
}
A(n-2, n-1) = 1.0; A(n-1, n-1) = 1.0;

MatrixXd Q = A.householderQr().householderQ();
cout << Q;

for(int i = 0; i < n*n; ++i) {
  VectorXd b = VectorXd::Random(n);
  VectorXd M = A.fullPivLu().solve(b);
  cout << M;
}

FullPivLu<MatrixXd> lu = A.fullPivLu();
for(int i = 0; i < n*n; ++i) {
  VectorXd b = VectorXd::Random(n);
  VectorXd M = lu.solve(b);
  cout << M;
}
```

unfinished

4.2.3 Cancellation

Which side of the equations below should be preferred in order to minimize the impact of cancellation?

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

▼ RHS or LHS?

  RHS

$$x >> 1: \ \frac{1}{\sqrt{x^2+1} + x} = \sqrt{x^2+1} - x$$

▼ RHS or LHS?

  LHS

$$\text{small } x > 0: \ \frac{2x^2}{(1+2x)(1+x)} = \frac{1}{1+2x} - \frac{1-x}{1+x}$$

▼ RHS or LHS?

  LHS

$$\text{small } x > 0: \ (1-x)^2 - 1 = x^2 - 2x$$

▼ RHS or LHS?

  RHS

4.2.4 Householder reflections

The Householder matrix for a reflection about the hyper-plane with the normal vector v is defined as

$$H_v := I_m - 2\frac{vv^T}{v^Tv} = I_m\tilde{v}\tilde{v}^T,$$

where $\tilde{v} = \frac{v}{||v||_2}$ is a unit vector. Note that H_v is symmetric and orthogonal. We want to reduce a matrix $A \in \mathbb{R}^{3,3}$ to an upper triangular form R using successive Householder transformations

$$H_{v^2}H_{v^1}A = R,$$

where

$$A = \begin{bmatrix} -3 & 20 & 1 \\ 4 & -20 & -1 \\ 0 & 3 & 2 \end{bmatrix}.$$

▼ Find the unit vector $\tilde{v}^1 \in \mathbb{R}^3$ such that the first element of $\tilde{v}^1$ is negative and the second is positive.

The reflecting vector can be obtained by:

$$\begin{aligned} v^1 &= a^1 + \text{sign}(a^1_1) \cdot ||a^1||_2 \cdot e^1 \\ &= \begin{bmatrix} -3 \\ 4 \\ 0 \end{bmatrix} - 5 \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -8 \\ 4 \\ 0 \end{bmatrix}. \end{aligned}$$

Now we have to make v^1 into a unit vector by dividing it by it's length:

$$\tilde{v}^1 = \frac{1}{\sqrt{80}} \cdot \begin{bmatrix} -8 \\ 4 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{5}} \cdot \begin{bmatrix} -2 \\ 1 \\ 0 \end{bmatrix}.$$

▼ Find the unit vector $\tilde{v}^2 \in \mathbb{R}^3$ such that the second and third element of $\tilde{v}^2$ are both positive.

The corresponding Householder matrix can be computed as:

$$\begin{aligned} H_{v^1} &= I_3 - 2\frac{v^1(v^1)^T}{(v^1)^Tv^1} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - 2\begin{bmatrix} \frac{4}{5} & -\frac{2}{5} & 0 \\ -\frac{2}{5} & \frac{1}{5} & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} -\frac{3}{5} & \frac{4}{5} & 0 \\ \frac{4}{5} & \frac{3}{5} & 0 \\ 0 & 0 & 1 \end{bmatrix}. \end{aligned}$$

Premultiplying A by H_{v^1} gives:

$$H_{v^1}A = \begin{bmatrix} 5 & -28 & -\frac{7}{5} \\ 0 & 4 & \frac{1}{5} \\ 0 & 3 & 2 \end{bmatrix}.$$

Now we can obtain v^2 as follows:

$$v^2 = \begin{bmatrix} 0 \\ 4 \\ 3 \end{bmatrix} + 5 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 9 \\ 3 \end{bmatrix}.$$

To get the unit vector $\tilde{v}^2$ we divide v^2 by its length:

$$\tilde{v}^2 = \frac{1}{\sqrt{90}} \begin{bmatrix} 0 \\ 9 \\ 3 \end{bmatrix} = \frac{1}{\sqrt{10}} \begin{bmatrix} 0 \\ 3 \\ 1 \end{bmatrix}.$$