Jonas B.

Eigen

`#include <iostream>; #include <Eigen/Dense>; using namespace Eigen;`

Matrix
- MatrixNt ≙ Matrix<type, N, N>
- MatrixXt ≙ Matrix<type, Dynamic, Dynamic>

Vector
- VectorNt ≙ Matrix<type, N, 1>

Types
- Types: i ≙ int / f ≙ float / d ≙ double / cf ≙ complexfloat / cd ≙ complex double

constructor
- Constructor: MatrixXd a(10,15) => 10x15 dynamically allocated
  VectorXd b(10) => 10x1 dynamically allocated

Access
- Access: a(0,0) = 3
  b(0) = 3
- Filling: MatrixXi C; C << 1, 2, 3, 4, 5, 6, 7, 8, 9 (1. Row, 2. Row, 3. Row)

Size
- Size: a.rows() => 10 / a.cols() => 15 / C.size() => 9
- Resizing: · Not allowed for fixed-size
  · a.resize(100, 150)

Template
- Templates: · Matrix3d m = Matrix3d::Random() (size) || Zero() (size) || Ones() (size) || Constant(5) (size, 5)
  m.setZero() || m.setOnes || m.setConstant(5)
  · Identity: Matrix3d::Identity

Functions
- Transpose: m.transpose()
- View vector as diagonal-matrix: matrix1 = vector1.asDiagonal();
- Diagonal of a matrix to vector: vector1 = matrix1.diagonal();
- Triangular view: · matrix1 = matrix2.triangularView<XXX>();
  · XXX: Upper, Lower, StrictlyUpper, StrictlyLower
- Inverse: m.inverse()

Effort
- Inner product / dot product / scalar-product: scal = vec1.transpose() · vec2 O(n)
- Outer product / tensor product: mat = vec1 · vec2.transpose() O(m·n)
- Matrix-vector product: vec1 = mat1 · vec2 O(n²)
- Matrix-matrix product: mat1 = mat2 · mat3 O(m·n·k)
- Matrix-diagonal-matrix product: mat1 = mat2 · mat3 O(n)

Array cwise
- ArrayNMt ≙ Array<type, N, M> (= Good for componentwise product)
- Matrix to array: mat1.array() || Array to matrix: arr.matrix()
- Cwise product: mat1.array() * mat2.array / mat1.cwiseProduct(mat2)

Blocks
- i^th row: mat.row(i) || i^th column: mat.col(i)
- First n elements: vec.head(n) || Last n elements: vec.tail(n)

Implicit inverse
- R^-1 · (b - v): R.triangularView<Upper>().solve(b - v) <=> .solve(IdentityMatrix) · (b - v)

col / rowwise
- mat.colwise() - vector / mat.rowwise() - vector.transpose()
- Norm: vector.norm() / matrix.colwise().norm()

Min/Max
- Find min/max coefficients: vector.minCoeff(minValueInd)
  matrix.maxCoeff(maxX, maxY)
  double minValue = vector.cwiseAbs().minCoeff(minValueInd)

C++

- using namespace std: cout << "Hallo ich " << name << endl;
- #include <math.h>: pow(10,5) (= 10^5) || cos(3) || abs(-10)
- class Rectangle { int width, height; public: void set_values(int, int){}};
- Iterator of vector: for(auto temp: matrice.list of vector)
- Lambda-function: · auto lambda = [&f, int x] (double y) { return f(x+y); }
  (lambda = function; [&f, int x] = what from outside should be visible from inside; (double y) = Argument)
  · double output = lambda(5, 0);
- Pi: #define _USE_MATH_DEFINES; #include <math.h>; : M_PI
- Arccos: acos(x) (acos(cos(x)) = x) / f(x) = arctan(x) <=> f'(x) = 1/(x²+1)
- Struct: struct QuadRule { VectorXd nodes, weights; };
- double eps = numeric_limits<double>::epsilon(); + #include <limits>

→ Absolute error: |x - x̃|
→ Relative error: |x - x̃| / |x|

- |-5| => abs(-5)
- ||mat||₂ => mat.norm()

Glossar

- Linear systems of equation: Gauss Elimination / LU-Decomposition
- Linear least squares: Normal equation / QR-Decomposition → Gram-Schmidt, Householder
  Singular-Value-Decomposition
- Total least squares: A/B is perturbed, use SVD-k-Approximation
- Constrained least-squares: Lagrange Multipliers
- Filtering: Discrete (periodic) convolution / Fourier Transformation
- Interpolation: Piecewise linear / Lagrange: Normal, Barycentric-Formula, Update-friendly
  Newton Basis / Cubic-Spline-Interpolation
- Approximation: Taylor-Polynom / Bernstein-Polynom
  Lagrange: Equidistant / Chebychev / Piecewise
- Quadrature: Newton-Cotes / Gauss-Quadrature: Gauss-Legendre / Composite
- Nonlinear systems of equations: Bisection / Fixed-Point / Newton: Normal, Secant, Damped
- Optimization: Convex / Newton C^2 / Golden-Section / Gradient-Descend / BFGS quasi Newton

Sparse-Matrices

```
#include <Eigen/Sparse>
#include <vector>
```

- Number of nonzero-elements: nnz(A)
- Memory only depend on nnz(A)
- COO: Triplet format: (row, column, value), 2 locations are the same → sum
- CRS: Compressed row storage
- CCS: Compressed column storage

Create fill sparse Matrix

- `SparseMatrix<double, RowMajor> matrice;`
  `std::vector<Triplet<double>> list;`
  `list.push_back(Triplet<double>(i,j,v))`
  `SparseMatrix<double> matrice;`
  `matrice.setFromTriplets(list.begin(), list.end())`

$$A = \begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix}$$

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

Access elements

- `list.size()`
  `Triplet<double> temp = list[i]`
  `temp.row() /.col() /.value()`
  - `Matrix COO temp`
    `Triplet<double> list = temp.list`
  - `Matrix CRS A`
    `A.makeCompressed()`
    `double* valPtr = A.valuePtr()`
    `index_t* innPtr = A.innerIndexPtr()`
    `index_t* outPtr = A.outerIndexPtr()`

- ValuePtr: Stores coefficient
- InnerIndices: Stores the column
- OuterStarts: Stores for each row the index to the first non-zero column
- InnerNNZ: Stores the number of non-zeros of each row (only for uncompressed)

↔ Compress (delete all Platzhalter) to CRS/CCS

Horner Scheme

$$p(t) = +(\ldots +(+(\alpha_n t + \alpha_{n-1}) + \alpha_{n-2}) + \ldots + \alpha_1) + \alpha_0$$

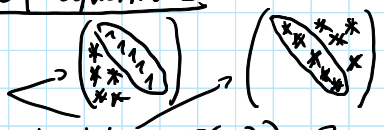
(coefficients)

Code 5.2.7

- `px = c[0]; for(int i=1; i<n; i++) {px = x*px + c[i];}`
- `dpx = (n-1)c[0]; for(int i=1; i<n-1; i++) {dpx = x*dpx + (n-i-1)*c[i];}`
  ↳ $c = [\alpha_n, \alpha_{n-1}, \ldots, \alpha_0] \Leftrightarrow f(x) = \alpha_n \cdot x^n + \ldots + \alpha_1 \cdot x^1 + \alpha_0$
- `px = c[0]*pow(x, n-1); for(int i=1; i<n; i++) {px += c[i]*pow(x, n-i-1);}`
- `dpx = (n-1)*c[0]*pow(x, n-2); for(int i=1; i<n-1; i++) {dpx += (n-1-i)*c[i]*pow(x, n-2-i);}`

2. Linear systems of equations

LU-Decomposition



$Ax=b$ ①
$LUx=b$ ②
$Ly=b$ ③
$Ux=y$

1. LU-Decomposition: $A=LU$ $O(n^3)$
2. Forward-Substitution: $Lz=b$ $O(n^2)$
3. Backward-Substitution: $Ux=z$ $O(n^2)$

} $O(n^3)$ same as Gaussian, but we can solve for different right hand sides in $O(n^2)$

- FullPivLU<MatrixXd> lu(A)
  MatrixXd L = MatrixXd::Identity(A.cols())
  L = L + lu.matrixLU().triangularView<StrictlyLower>()
  MatrixXd U = lu.matrixLU().triangularView<Upper>()
  VectorXd x = lu.solve(b)
- VectorXd x = A.fullPivLu().solve(b)

} Reuse A for more right sides

} only one right side

LU-Decomposition with low rank modification

- After solving $Ax=b$, we solve $\tilde{A}\tilde{x}=b$, with 1 element in $\tilde{A}$ changed
- Rand-1-modification: $\tilde{A}=A+u\cdot v^T$

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

① $\begin{cases}\tilde{A}\tilde{x}+u\xi=b \\ v^T\cdot\tilde{x}-\xi=0\end{cases} \Rightarrow A\tilde{x}+u\cdot v^T\cdot\tilde{x}=b \Rightarrow (A+u\cdot v^T)\cdot\tilde{x}=b$, $\xi=v^T\cdot\tilde{x}$

② $\tilde{x}=A^{-1}\cdot(b-u\cdot\xi) \Rightarrow (v^T)^{-1}\xi=A^{-1}\cdot(b-u\cdot\xi) \Rightarrow v^T\cdot A^{-1}\cdot(b-u\cdot\xi)-\xi=0 \Rightarrow (v^T\cdot A^{-1}\cdot u+1)\cdot\xi=v^T\cdot A^{-1}\cdot b$

$\tilde{x}=A^{-1}\cdot b-A^{-1}\cdot u\cdot\xi \Rightarrow \tilde{x}=A^{-1}\cdot b-A^{-1}\cdot u\cdot\frac{v^T\cdot A^{-1}\cdot b}{v^T\cdot A^{-1}\cdot u+1}$ $\Rightarrow \xi=\frac{v^T\cdot A^{-1}\cdot b}{v^T\cdot A^{-1}\cdot u+1}$

Block LU-Decomposition (for arrow-matrices)

$$\begin{bmatrix}R & \overleftarrow{v}\\ u^T & \alpha\end{bmatrix}\cdot\begin{bmatrix}x\\ \xi\end{bmatrix}=\begin{bmatrix}B\\ \beta\end{bmatrix}$$

(R: diagonal; α: scalar)

1. $\begin{pmatrix}I & 0\\ u^T\cdot R^{-1} & 1\end{pmatrix}\cdot\begin{pmatrix}y\\ \eta\end{pmatrix}=\begin{pmatrix}B\\ \beta\end{pmatrix}$
   - y.head(n) = b.head(n)
   - $\eta = b(n) - u.\text{transpose}() * \text{backSub}(R, y.\text{head}(n))$
2. $\begin{pmatrix}R & v\\ 0 & u^T\cdot R^{-1}\cdot v\end{pmatrix}\cdot\begin{pmatrix}x\\ \xi\end{pmatrix}=\begin{pmatrix}y\\ \eta\end{pmatrix}$
   - $u \ll R$
   - VectorXd::zero(n), $v - u.\text{transpose}() * R.\text{inverse}() \cdot v$
   - $x = \text{backSubs}(u, z)$

BackSub: for(int i = n-1; i >= 0; --i) { x(i) = B(i); for(int j = n-i; j > i; --j) {
x(i) = x(i) - A(i,j) * x(j) } x(i) /= R(i,i) }

Block Gaussian Decomposition (for arrow-matrices)

$$\left[\begin{array}{cc|c}A_{11} & A_{12} & b_1\\ A_{21} & A_{22} & b_2\end{array}\right] \rightarrow \left[\begin{array}{cc|c}A_{11} & A_{12} & b_1\\ 0 & A_{22}-A_{21}\cdot A_{11}^{-1}A_{12} & b_2-A_{21}A_{11}^{-1}b_1\end{array}\right]$$

$$\rightarrow \left[\begin{array}{cc|c}1 & 0 & A_{11}^{-1}(b_1-A_{12}\cdot S^{-1}\cdot b_S)\\ 0 & 1 & S^{-1}\cdot b_S\end{array}\right] \quad S=A_{22}-A_{21}\cdot A_{11}^{-1}A_{12},\ b_S=b_2-A_{21}A_{11}^{-1}b_1$$

$x_1 = A_{11}^{-1}(b_1 - A_{12}S^{-1}\cdot b_S)$ } vector

$x_2 = S^{-1}\cdot b_S$ } constant

Normal equation

- $Ax=b \quad /\cdot A^T \Rightarrow A^T\cdot Ax = A^T\cdot b \Rightarrow x=(A^T\cdot A)^{-1}\cdot A^T\cdot b$

3. Linear least-squares problem $\begin{bmatrix} A \end{bmatrix}\begin{bmatrix} x \end{bmatrix} = \begin{bmatrix} b \end{bmatrix} \Leftrightarrow x^* = \operatorname{argmin} \|Ax-b\|_2$

$(\alpha, \beta)^* = \operatorname{argmin} \|T_{\alpha,\beta} \cdot z - c\|_2$

↳ Reformulate: Ax, b

Normal equations

$\cdot Ax = b \;/\cdot A^T \Rightarrow A^T \cdot A \cdot x = A^T \cdot b \;/(A^T A)^{-1} \Rightarrow x = (A^T A)^{-1} \cdot b$

Sparse Approximate Inverse

- $Inv = \operatorname{argmin} \|I - A \cdot X\|_F \;/ Inv = \begin{pmatrix} | & | & | \\ x_1 & x_2 & x_3 \cdots \\ | & | & | \end{pmatrix} \;/ Inv_n = x_n$
- Each column is an own least-squares problem: $(A) \cdot (x_n) = (I_n)$
- To get better performance, delete columns of A_i when element $A(n,m) = 0$, because it is not used for computation
- We can do that column-wise, because of the Frobenius-Norm
  ↳ $\|A\|_F = \sqrt{(\text{Each element})^2 + (\text{other Element})^2} \triangleq Inv_i = \operatorname{argmin} \|I_i - Ax_i\|$

QR-Decomposition

1. Compute $A \rightsquigarrow Q$ (orthonormal vectors of A) via Cholesky/Householder-Reflection
2. $A = QR \;/\cdot Q^T \Rightarrow Q^T A = \underbrace{Q^T \cdot Q}_{I} \cdot R \Rightarrow R = Q^T \cdot A$
2. $z = Q^T \cdot b$
3. $Rx = z$ via Backsubstitution

Cholesky:
- R: upper triangular / Q: Orthogonal
- `MatrixXd AtA = A.transpose() * A`
  `LLT<MatrixXd> chol = AtA.llt()`
- $chol_{ij} = \begin{cases} 0 & \text{für } j < i \\ \sqrt{a_{ii} - \sum_{k=1}^{i-1} chol_{ik}^2} & \text{für } i = j \\ \frac{1}{chol_{jj}} \cdot \left(a_{ij} - \sum_{k=1}^{j-1} chol_{ik} \cdot chol_{jk}\right) & \text{für } i > j \end{cases}$

```
-> MatrixXd L = chol.matrixL()
   R = L.transpose()
Q = L.triangularView<Lower>().solve(
    A.transpose()).transpose()
```

Householder:
- `HouseholderQR<MatrixXd> QR = A.householderQr();` ← Economical
  `Q = QR.householderQ() * MatrixXd::Identity(m, min(m,n))`
  `R = MatrixXd::Identity(min(m,n), m) * QR.matrixQR().triangularView<Upper>()`
  `x = QR.solve(B)`
- $v_1 = a_{i1} - \alpha e_1$ — $Q_1 = \bar{I} - 2 \cdot \frac{v_1 \cdot v_1^T}{v_1^T \cdot v_1}$ — $\alpha = sign(a_{11}) \cdot \|a_i\|_2$
- $A = \begin{pmatrix} 1 & 1 & 2 \\ 2 & -3 & 0 \\ 2 & 4 & -4 \end{pmatrix} \Rightarrow Q_1 \cdot A = \begin{pmatrix} -3 & -1 & 0 \\ 0 & -4 & 0 \\ 0 & 3 & -4 \end{pmatrix}$ → This region is new A

Constrained Least Squares (Least squares with side-conditions)

- Minimize $\|Ax-b\|$ and $Cx = d$ exact (x is the same)

1. Lagrange Multiplier $L(x, m) := \frac{1}{2} \cdot \|Ax-b\|_2^2 + m^T(Cy - d)$ (m = Multiplier)
2. $\frac{\partial f}{\partial x}(x,m) = A^T(Ax-b) + C^T \cdot m = 0$ } Augmented normal equation
3. $\frac{\partial f}{\partial m}(x,m) = Cx - d = 0$ }

$$\underbrace{\begin{bmatrix} A^TA & C^T \\ C & 0 \end{bmatrix}}_{A} \cdot \underbrace{\begin{bmatrix} x \\ m \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} A^Tb \\ d \end{bmatrix}}_{B}$$

- E.g. Minimize $\|M\|_F$: $\begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} m_{11} \\ m_{12} \\ \vdots \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

Singular Value Decomposition

Code 3.4.13

$A = U \cdot \Sigma \cdot V^T$ ← Left singular vectors, ← Right singular vectors

- Σ: Singular-values in ascending diagonal form (singular value = Eigenvalue!)
- Code: 3.4.15 (Full: $\Sigma = (\cdots)$ / Eco: $\Sigma = (\cdots)$)
- Low-rank-approximation: $A^k_{compressed} = U^{m \times k} \cdot \Sigma^{k \times k} \cdot V^{T\, k \times n}$

- Commutativity: $AB^T \Rightarrow (Q_A \cdot R_A) \cdot (Q_B \cdot R_B)^T \Rightarrow Q_A \cdot \underbrace{R_A \cdot R_B^T}_{U \cdot \Sigma \cdot V^T} \cdot Q_B^T \Rightarrow \underbrace{Q_A \cdot U}_{U} \cdot \underbrace{\Sigma}_{\Sigma} \cdot \underbrace{(Q_B \cdot V)^T}_{V^T}$

Principle component Analysis

Face recognition

1. Get some data $\begin{bmatrix} d_1 & d_2 & d_3 \\ | & | & | \end{bmatrix}$
2. Compute the mean of the data and subtract it $\begin{bmatrix} d_1\text{-mean} & d_2\text{-mean} \\ | & | \end{bmatrix}$
3. Covariance-Matrix: $A \cdot A^T$ (Eigenvector of $A \cdot A^T$ same as left singular vectors of A)
   (Eigenvalues of $A \cdot A^T$ / $A^T \cdot A$ are the same)
4. Compute SVD of A extract U → left singular vectors
5. projFaces = U.transpose() * (data-mean-matrix)
6. projNewFace = U.transpose() * (newFace - meanFace)

} 3. Smallest difference norm is the most identical

4. Filtering

- Impulse response: $h = (h_0, \dots, h_{n-1})^T$
- Properties of filter: · finite length input → Finite length output
  · Time invariant / · Linearity / · Output depends on past/present
- Input signal x: Output $y_k = \sum_{j=0}^{n-1} h_{k-j} \cdot x_j = \sum_{j=0}^{n-1} h_j \cdot x_{k-j}$ / $h_j = 0$ for $j < 0$
  ↳ Discrete convolution
- n-periodic: When input signal repeats itself after n inputs → Also output periodic
  · We must zeropadd to match: length of x = length of h
  · $y_k = \sum_{j=0}^{m} p_{k-j} \cdot x_j = \sum_{j=0}^{m} x_{k-j} \cdot p_j$ / $p_j = \sum_{l \in \mathbb{N}} h_{j+l \cdot n}$, $j \in \{0, 1, \dots, n-1\}$
  ↳ periodic impulse response
- Periodic signal can be represented as Circulant-Matrix $\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} p_0 & p_{n-1} & p_{n-2} & \dots \\ p_1 & p_0 & p_{n-1} & \dots \\ p_2 & p_1 & p_0 & \dots \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$

Fourier Transformation

$\underbrace{\begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}}_{C} = \underbrace{\begin{bmatrix} w^0 & w^0 & w^0 \\ w^0 & w^1 & w^2 \\ w^0 & w^2 & w^4 \end{bmatrix}}_{F_n} \cdot \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix}}_{y}$ $F_n(x,y) = w_n^{x \cdot y}$

- $w_N = e^{-\frac{2\pi \cdot i}{N}}$ / $w_N^{i \cdot j} = \left(e^{-\frac{2\pi \cdot i}{n}}\right)^{i \cdot j} = e^{-\frac{2\pi i \cdot i \cdot j}{n}}$
- We get the spectrum vector C by calculating $C = F_N \cdot y$
- $c_k = \sum_{j=0}^{n-1} y_j \cdot w_n^{k \cdot j} \iff y_i = \frac{1}{n} \cdot \sum_{k=0}^{n-1} c_k \cdot w_n^{-k \cdot i}$ $\left(F_n^{-1} = \frac{1}{n} \cdot \overline{F_n}\right)$
- Solve $Ax = b$: If $A = F_n$ and you want to solve/least squares x: $F_n \cdot x = b \Rightarrow x = \frac{1}{n} \cdot b \cdot F_n$
- DFT: C.fft.fwd(y) ⟺ inverse DFT: Y = fft.inv(c)
- One dimensional data ≙ signal / two dimensional data ≙ image

→ FFT<double> fft;
VectorXcd fft_coeff = fft.fwd(y);

fft_b = fft.fwd(b);
x = fft_b.real()/n

Code 4.2.48

- 2D Fourier Transformation: 2 nested 1D DFTs: $(c)_{k_1,k_2} = \sum_{j_1=0}^{m-1} w_m^{j_1 k_1} \left(\sum_{j_2=0}^{m-1} w_n^{j_2 \cdot k_2} \cdot y_{j_1, j_2} \right)$
- 2D Fourier Inverse: $Y = F_m^{-1} \cdot C \cdot F_n^{-1}$ ⇒ $C = F_m \cdot (F_n \cdot Y^T)^T = F_m \cdot Y \cdot F_n$
  2D inverse

Code 4.2.55

- Discrete periodic convolution
  ↳ VectorXcd tmp = (fft.fwd(h)).cwiseProduct(fft.fwd(x)) 2D } ifft(fft(h) · fft(x)) ≙ $h *_n x$
  y = fft.inv(tmp)
  Both vectors same size

Code 4.2.26

Discrete convolution

```
int m = x.size(); int n = y.size(); int l = m+n-1; VectorXd out = Vector::Zero(l);
for(int i=0; i<l; i++) { for(int k = max(0, i+1-m); k < min(i+1, n); k++)
  out(i) += x(i-k) * y(k);
```

5. Interpolation

Lagrange-Interpolation $p(t) = \sum_{i=0}^{n} y_i \cdot L_i(t)$ $\quad L_i(t) := \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{t - t_j}{t_i - t_j}$

- Evaluating $L_i(t)$ is $O(n)$ with Horner-Scheme
- Evaluating $p(t)$ is $O(n^2)$, because we have to evaluate n polynomials
- Reusing computation for different y-components: Barycentric-Formula

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

$$\lambda_i = \frac{1}{(t_i - t_0) \cdot \ldots \cdot (t_i - t_{i-1}) \cdot (t_i - t_{i+1}) \cdot \ldots \cdot (t_i - t_n)}$$

$$1 = \prod_{j=0}^{n} (t - t_j) \cdot \sum_{i=0}^{n} \frac{\lambda_i}{t - t_i} \quad / : \sum_{i=0}^{n} \frac{\lambda_i}{t - t_i} \Rightarrow \prod_{j=0}^{n} (t - t_i) = \frac{1}{\sum_{i=0}^{n} \frac{\lambda_i}{t - t_i}} \quad / \prod_{j=0}^{n} (t - t_j) = \sum_{j=0}^{n} \frac{\lambda_i}{t - t_i}$$

$$p(t) = \frac{\sum_{i=0}^{n} \frac{\lambda_i}{t - t_i} \cdot y_i}{\sum_{i=0}^{n} \frac{\lambda_i}{t - t_i}}$$

Barycentric-Formula
↳ λ_i is independant of y_i ⇒ precompute $O(n^2)$
↳ cost of every new set of y $O(n)$

- Update friendly for adding/removing nodes ⇒ Aitken-Neville

Code 5.2.35

$p_{k,k}(x) = y_k \qquad p_{k,l}(x) = \frac{(x - t_k) \cdot p_{k+1,l}(x) - (x - t_l) \cdot p_{k,l-1}(x)}{t_l - t_k}$

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

$[x \cdot p_{k+1,l}(x)]' = x' \cdot p_{k+1,l}(x) + x \cdot p'_{k+1,l}(x)$

$p'_{k,k}(x) = 0 \quad / \quad p'_{k,l}(x) = \frac{p_{k+1,l}(x) + (x - t_k) \cdot p'_{k+1,l}(x) - p_{k,l-1}(x) - (x - t_l) \cdot p'_{k,l-1}(x)}{t_l - t_k}$

Code 5.2.55, 5.2.58

Newton Basis $N_0(t) = 1 \quad / \quad N_1(t) = (t - t_0) \quad / \quad N_n(t) = \prod_{i=0}^{n} (t - t_i) \Rightarrow p(t) = \sum_{i=0}^{n} a_i \cdot N_i(t)$ ← unknowns

- Good for many interpolation points, because Lagrange is numerically unstable
- Find interpolant $p(t)$ by lower triangular system of equations
  ↳ $p(t_0) = a_0 + 0 + 0 + \ldots + 0 \quad / \quad p(t_1) = a_0 + a_1 N_1(t_1) + 0 + 0 \ldots + 0$
  $p(t_2) = a_0 + a_1 N_1(t_2) + a_2 N_2(t_2) + 0 + 0 \ldots + 0$

Cubic spline interpolation (each subinterval has polynomial with degree 3)

- For each subinterval: $s_j := a_j + b_j t + c_j t^2 + d_j t^3$ (4n coeff. to determine)

LSE

1. $s_j(t_{j-1}) = y_{j-1}$ & $s_j(t_j) = y_j$: $a_j + b_j t_{j-1} + c_j \cdot t_{j-1}^2 + d_j \cdot t_{j-1}^3 = y_{j-1}$
2. Match 1. derivative: $s'_j(t_j) = s'_{j+1}(t_j)$: $b_j + 2c_j t_j + 3d_j \cdot t_j^2 = b_{j+1} + 2c_{j+1} \cdot t_j + 3d_{j+1} \cdot t_j^2$
3. Match 2. derivative: $s''_j(t_j) = s''_{j+1}(t_j)$: $2c_j + 6d_j \cdot t_j = 2c_{j+1} + 6d_{j+1} \cdot t_j$ $\quad j = 1, \ldots, n-1$
4. Natural/Simple cubic spline: $s''_1(t_0) = 0$ und $s''_n(t_n) = 0$

1. Solve tridiagonal system for σ_j 2. Compute $a_j = y_{j-1}$ 3. Insert into $b_j / c_j / d_j$

$$\begin{bmatrix} \frac{h_1 + h_2}{3} & \frac{h_2}{6} & & \\ \frac{h_2}{6} & \frac{h_2 + h_3}{3} & \frac{h_3}{6} & \\ & \ddots & \ddots & \ddots \\ & & \frac{h_{n-1}}{6} & \frac{h_{n-1} + h_n}{3} \end{bmatrix} \cdot \begin{bmatrix} \sigma_1 \\ \vdots \\ \sigma_{n-1} \end{bmatrix} = \begin{bmatrix} r_1 \\ \vdots \\ r_{n-1} \end{bmatrix}$$

- $r_j := \frac{y_{j+1} - y_j}{h_{j+1}} - \frac{y_j - y_{j-1}}{h_j}$
- $h_j := t_j - t_{j-1} \qquad (t \hat{=} x)$
- $\sigma :=$ to be determined $\qquad (\sigma_0 \overset{!}{=} 0 \overset{!}{=} \sigma_n)$

- $a_j := y_{j-1}$ - $b_j := \frac{y_j - y_{j-1}}{h} - \frac{h_j (2 \cdot \sigma_{j-1} + \sigma_j)}{6}$ - $c_j := \frac{\sigma_{j-1}}{2}$ - $d_j := \frac{\sigma_j - \sigma_{j-1}}{6 h_j}$

6. Approximation (Given a function, find a simple-approx function $\tilde{f}$)

- Algebraic convergence: $p>0 : \exists p>0 : T(n) \leq n^{-p}$ => Linear in log-log-scale
- Exponential convergence: $\exists 0<p<1 : T(n) \leq q^n$ ⟹ linear in lin-log-scale

Taylor-polynom

- $f(t) \approx \sum_{j=0}^{n} \frac{f^{(j)}(t_0)}{j!} \cdot (t-t_0)^j$ (Only approximates $f(t)$ in neighbourhood t_0)

Uniform Bernstein Polynomials

- $p_n(t) = \sum_{j=0}^{n} f(j/n) \cdot \binom{n}{j} \cdot t^j (1-t)^{n-j}$ (Inefficient -> need higher degree)

Equidistant Lagrange Approximation

- Take some nodes with same distance to each other and do Lagrange-Interpolation
  ↳ As more points we use, the more precision we get ↯ => Runges-Experiment

Chebychev-Interpolation

Code 6.1.104 / 6.1.112

- Points from Chebychev-Polynomial, interpolation with Lagrange
- $T_n(t) = \cos(n \cdot \arccos(t))$ $(-1 \leq t \leq 1)$
- The zeros from the polynomial we use as points
  ↳ $t_k = \cos\left(\frac{2k+1}{2n} \cdot \pi\right)$
- Recursive-definition: $T_0 = 1$ / $T_1(t) = t$ / $T_{n+1}(t) = 2t \cdot T_n(t) - T_{n-1}(t)$
- We can write the approximation as Chebychev-expansion: $p = \sum_{j=0}^{n} \alpha_j \cdot T_j$

Piecewise polynomial interpolation

- Make a mesh M out of the interval $[a,b]$
- x_j: nodes / $[x_{j-1}, x_j]$: cell / h_M: max. mesh width / n_j: degree of poly.
- We need to stratisfy:

Error

- Taylor Approximation: 6.1.26)
- L^∞ polynomial best approximation estimate: 6.1.15 / 6.1.17 / 6.1.17 (Best infimum)
- Polynomial best approximation on general intervals: 6.1.31
- Interpolation error estimate for n-degree Lagrange approximation: 6.1.50
- Lebesgue constant: 6.1.57
- Supremum norm interpolation error Chebychev on $[-1,1]$: $\|f - L_T f\|_{L^\infty([-1,1])} \leq \frac{2^{-n}}{(n+1)!} \cdot \|f^{(n+1)}\|_{L^\infty}$
- Error estimates for piecewise polynomial Lagrange-interpolation: 6.5.12 ↳ week 10 page 16
- Interpolation error chebychev for $[a,b]$: 6.1.88

Types of convergences (6.1.38)

- $\|f - L_{T_n} f\| \leq C \cdot T(n)$: Algebraic: $\|f - L_T f\| = O(n^{-p})$ / Exponential: $\|f - L_T f\| = O(q^n)$
- Determining: Algebraic: linear in log-log scale / Exponential: linear in lin-log-scale

Affine Pullback $[-1,1] \to [a,b]$

- $\Phi : [-1,1] \to [a,b]$
- $\Phi(t) = a + \frac{1}{2}(t+1) \cdot (b-a)$

7. Numerical Quadrature

$\int_a^b f(t)\,dt \approx Q_n(f) := \sum_{j=1}^{n} \underbrace{w_j^n}_{\text{weight}} \cdot f(\underbrace{c_j^n}_{\text{nodes}}) \in [a,b] \doteq x_j$

Affine Pullback

- $\int_a^b f(t)\,dt \approx \frac{1}{2}(b-a) \cdot \sum_{j=1}^{n} \hat{w}_j \cdot f(\hat{c}_j)$
- $\hat{c}_j = a + \frac{1}{2}(c_j+1)(b-a)$
- $\hat{w}_j = \frac{1}{2}(b-a) \cdot w_j$

Polynomial quadrature by approximation scheme

- $\int_a^b f(t)\,dt \approx \int_a^b \underbrace{p_{n-1}}_{\text{Lagrange-Interpolation}}(t)\,dt = \sum_{i=0}^{n-1} \underbrace{f(t_i)}_{\text{Nodes}} \cdot \underbrace{\int_a^b L_i(t)\,dt}_{w_i} = \sum_{i=0}^{n-1} f(t_i) \cdot w_i$

Newton-Cotes formula

- Lagrange interpolation with equidistant nodes: $t_i := a + \frac{b-a}{n-1} \cdot i$: 7.2.5
- $n=2$: Trapezoidal-Rule: $\int_a^b f(t)\,dt \approx \underbrace{\frac{b-a}{2}(f(a)+f(b))}_{\text{Nodes}} \cdot \underbrace{\frac{1}{n}}_{\text{weight}}$
- $n=3$: Simpson-Rule: 7.2.6

Gauss-Legendre Quadrature

Code 7.3.36
- Evaluation points from Legendre-polynomial-roots: t_j (Gauss points)
- Weights for points from Lagrange: $w_j = \int_a^b L_{j-1}(t)\,dt$ (Gauss weights)

↳ The gauss points and gauss weights can be computed by Golub-Welsch

Order

- Quality measurement, for how good the quadrature-rule is
- Max-degree of approximation +1 of polynomials, for with quadrature rule is guaranteed to be correct
- The maximal order of an n-point-rule is $2n \Rightarrow$ Unique QF
  ↳ With n points one can integrate a polynomial with max degree $2n-1$
  - When does a QF with n points have order $\geq n$: 7.3.5: with Lagrange
  - When does a QF with n points have order $2n$: 7.3.22: Legendre

Error

7.3.39
- $E_n(f) := \left| \int_a^b f(t)\,dt - Q_n(f) \right|$ - $q :=$ Order - $r :=$ $f \in C^r$

7.3.42
- In case $q \geq r$: $E_n(f) \leq C \cdot q^{-r} \cdot |b-a|^{r+1} \cdot \|f^{(r)}\|_{L^\infty([a,b])}$
- In case $q < r$: $E_n(f) \leq \frac{|b-a|^{q+1}}{q!} \cdot \|f^{(q)}\|_{L^\infty([a,b])}$
- If $f \in C^R([a,b])$: algebraic convergence with rate r
- If $f \in C^\infty([a,b])$: exponential convergence

Code for $[a,b]$

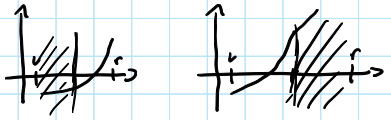
```
double quadrature(Function& f, VectorXd w, VectorXd x, double a, double b){
    double I=0;
    for(int i=0; i<w.size(); i++){
        I += f((x(i)+1)*(b-a)/2+a) * (1/2*(b-a)*w(i));
    }
    return I;
}
```

8. Iterative methods for nonlinear systems of equations

Code 8.3.2

- How to solve $f(x^*)=0$

Bisection Algorithm



- Iterative error: $e^{(k)} = x^{(k)} - x^*$
- Rate of convergence: linear type by factor $\frac{1}{2}$

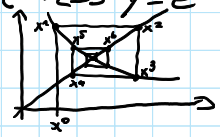
$$\left.\begin{array}{l} \cdot\, |e^{(1)}| \le \frac{1}{2}(a-b) \\ \cdot\, |e^{(2)}| \le \frac{1}{2^2}(a-b) \end{array}\right\}$$

Fixed point Iteration in 1D

Code 8.1.16

- $f(x)=0 \Leftrightarrow \Phi(x) := f(x)+x$ (=finding fixed point)
- $\Phi(x^*)=x^* \Rightarrow f(x^*)=0$
- Requirements: Φ is Lipschitz-stetig on $[a,b]$: $\exists L<0\ \forall x,y\in[a,b]: |\Phi(x)-\Phi(y)| \le L\cdot|x-y|$

1. Rewrite $f(x)=0 \Rightarrow \Phi(x)=x$: $x\cdot e^x - 1 = y \Leftrightarrow x\cdot e^x - 1 = 0 \Leftrightarrow x = e^{-x} \Leftrightarrow y = e^{-x}$
2. Start with initial guess $x^{(0)}$
3. Use fixed point iteration: $x^{(k+1)} = \Phi(x^{(k)}) : x^{(k+1)} = e^{-x^{(k)}}$



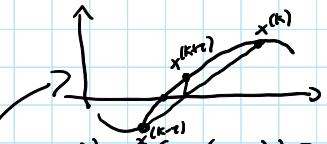
Newton's method in 1D

Code 8.4.2 / 8.4.51

- Gives us quadratic convergence, if $f'(x^*) \neq 0$
- Requirement: $f \in C^1$

1. start with initial guess
2. $x^{(k+1)} = x^{(k)} + f(x^{(k)}) / f'(x^{(k)})$

- Costly to compute $f'(x^{(k)})$ for each iteration



Quasi-Newton: Secant method

Code 8.3.25

- Replaces $f'(x^{(k)})$ by approximation: $f'(x^{(k)}) \approx \dfrac{f(x^{(k)}) - f(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}$ } 2 point method

Fixed point Iteration in higher dimensions

- Φ converges at least local linearly, if $\|D\Phi(x^*)\| < 1$ ($D\Phi(x)$ = Jacobian Matrix)
- Φ converges at least linearly, if $\|D\Phi(x)\| < 1$ everywhere $(\forall x)$
- Termination criteria: $\|x^{(k+1)} - x^{(k)}\| \le \tau \cdot \|x^{(k+1)}\|$ or $\|x^{(k+1)} - x^{(k)}\| \le$ absolute tolerance
  ↳ $\|f(x^{(k)}) - f(x^*)\|$ small $\not\Rightarrow$ $\|x^{(k)} - x^*\|$ small!
- Better termination criteria: A priori: 8.2.21 / posteriori: 8.2.21 $\dots \le \tau$
  (a priori: compute at beginning; posteriori: compute at every step)

Newton's method in higher dimensions

- Requirement: Invertibility of Jacobian-Matrix

1. Start with initial guess $x^{(0)}$
2. $x^{(k+1)} = x^{(k)} - DF(x^{(k)})^{-1} \cdot F(x^{(k)})$

- Newton correction: $:= -DF(x^{(k)})^{-1} \cdot F(x^{(k)})$: LSE : $DF(x^{(k)}) \cdot y = -F(x^{(k)})$
- Termination criteria: $\|Df(x^{(k-1)})^{-1} \cdot F(x^{(k)})\| \le \tau \cdot \|x^{(k)}\|$
- Costly to compute Jacobian for each iteration

Quasi-Newton: Broyden's method // Sherman-Morrison-Woodbury

↳ 8.4.75

Problem: Overshooting / Solution: Damped Newton's method

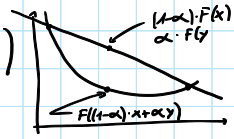
Code 8.4.58

- $x^{(k+1)} := x^{(k)} - \lambda^{(k)} \cdot DF(x^{(k)})^{-1} \cdot f(x^{(k)})$ — $\lambda^{(k)}$:= Damping factor

1. Set $\lambda^{(k)} = 1$
2. Check natural monotonicity test: 8.4.57
3. Repeatedly take $\lambda^{(k)} = \lambda^{(k)}/2$ until NMT passes for the first time and calculate $x^{(k)}$ with the calculated damping-factor

Newton's method oscillation → no convergence: criteria $x^{(1)} = x^{(0)} - \frac{f(x^{(0)})}{f'(x^{(0)})} = -x^{(0)}$

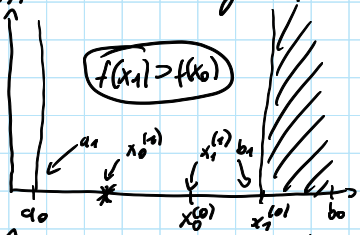
8.5 Unconstrained Optimization (How to find minimum of f(x))



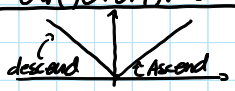
- Maximizing $f(x) \hat{=}$ minimizing $-f(x)$
- Convex function: $F((1-\alpha)\cdot x+\alpha y) \leq (1-\alpha)\cdot F(x)+\alpha F(y)$: local min. $\hat{=}$ global min.

Newton's method for C^2 functions in 1D

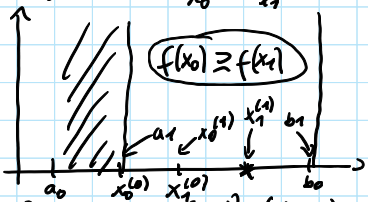
- Applied to $f'(x)$ to search for $f'(x)=0$
1. Start with initial guess x^0
2. $x_{k+1} = x_k - f'(x)/f''(x)$



Golden section search for non-differentiable functions



- Function must be unimodal
- Linear type convergence
- In each iteration: one step is computed



while $|b-a| >$ tolerance { if $f_1 > f_0$: $b = x_1$; $x_1 = x_0$; $f_1 = f_0$; $x_0 = a + [1-\lambda]\cdot(b-a)$; $f_0 = f(x_0)$;
if $f_0 \geq f_1$: $a = x_0$; $x_0 = x_1$; $f_0 = f_1$; $x_1 = a + \lambda(b-a)$; $f_1 = f(x_1)$; }

Methods in higher dimensions

Gradient descend

1. Start with initial guess x^0
2. while $\|\nabla f(x)\| <$ tol
3. Find stepsize $t^{(k)}$ in each iteration through line search
4. $x^{(k+1)} = x^{(k)} - t^{(k)} \cdot \nabla F(x^{(k)})$

Helper: Backtracking Line search

- $F(x - t\cdot\nabla F(x)) \leq F(x) - t\cdot\|\nabla F(x)\|^2 < F(x) - \alpha\cdot t\cdot(\|\nabla F(x)\|^2$
1. Iterate, until good decrease is reached
2. start with $t=1$, fix $\alpha \in (0, 0.5)$, $\beta \in (0, 1)$
3. while $F(x - t\cdot\nabla F(x)) > F(x) - \alpha\cdot t\cdot\|\nabla F(x)\|^2$
4. $t = \beta\cdot t$

Newton's method for C^2 functions in higher dimensions

- $x^{(k+1)} = x^{(k)} - [H_F(x^{(k)})]^{-1}\cdot\nabla F(x^{(k)})$

Helper: BFGS-method (Quasi Newton)

- Approximate $H_F(x^{(k)})$ by B_k
- B_{k+1} is obtained from a simple update of B_k

↳ $B_{k+1}\cdot\underbrace{(x^{(k+1)} - x^{(k)})}_{s^{(k)}} = \underbrace{\nabla F(x^{(k+1)}) - \nabla F(x^{(k)})}_{y^{(k)}}$

- $B_{k+1} = B_k + \frac{y^k\cdot y^{kT}}{y^{kT}\cdot s^k} - \frac{B_k\cdot s^k\cdot s^{kT}\cdot B_k^T}{s^{kT}\cdot B_k\cdot s^k}$

↳ For B_{k+1}^{-1} use Sherman-Morrison-Woodbury