Yannick Müller muelleya@student.ethz.ch
yajm.ch

ETH MADE EASY

Computer Science
Block 3

January 2020

Numerical Methods for CSE - Lecture

Prof. R. Hiptmair

Contents

The following was presented in the lecture on 19. September 2019.

1 Computing with Matrices and Vectors

Use the C++ library Eigen for faster numerical methods computations.
Eigen is a Header-only C++ template library.
Matrices can be fixed, dynamically or sparsed. The size is known at compile time if it is fixed otherwise not.

Definition 1. The Asymptotic complexity of an algorithm characterizes the worst-case dependence of its computational effort on one or more problem size parameters when these tend to ∞.

Note 1. This is not a very good indicator for the runtime. As it is much more dependent by the memory access pattern. Nevertheless the asymptotic complexity is important to predict the scalability of an algorithm.

Note 2. It is very important to always check in which order one multiplies the matrices.

$$AB^T = \sum_{l=1}^{p} (A)_{i,l} \cdot (B)^T_{i,l} \tag{1}$$

Definition 2. Kronecker Product $A \otimes B$

$$A \otimes B \overset{\text{def}}{=} \begin{pmatrix} (A)_{1,1}B & (A)_{1,2}B & \dots & (A)_{1,n}B \\ \dots & & & \\ (A)_{m,1}B & (A)_{m,2}B & \dots & (A)_{m,n}B \end{pmatrix} \tag{2}$$

Note 3. Computers have difficulty handling real numbers.

The following was presented in the lecture on 20. September 2019.

1.1 Roundoff Errors

Computers cannot compute in $\mathbb{R}$ instead they compute in $\mathbb{M}$ (set of machine numbers)

$$op: \quad \mathbb{M} \times \mathbb{M} \nRightarrow \mathbb{M} \tag{3}$$

Definition 3. Correct rounding (rounding up) is given by the function

$$rd: \begin{cases} \mathbb{R} & \to & \mathbb{M} \\ x & \mapsto & \max \arg \min_{\tilde{x} \in \mathbb{M}} |x - \tilde{x}| \end{cases} \tag{4}$$

Note 4. Float numbers should never be checked equal to zero, instead check relative smallness with another float number/matrix times an epsilon. Absolute tests can sometimes also fail.

Note 5. Cancellation effect: Roundoff errors can sum up to big errors. This is specifically true, when subtracting two big numbers and it equals a small number. This is also true for calculating the derivative, when h cannot be made too small.

Note 6. Instead of evaluating with cancellation one sometimes should use the Taylor approximation which is cancellation-free

2 Direct Methods for Linear Systems of Equations

Note 7. The Gaussian Elimination is stable and is not affected by roundoff in practice.

Definition 4. A sparse matrix is a matrix where most entries are 0 and this is worth exploiting

Note 8. If it is a sparse matrix one can save the matrix in the COO / Triplet format with (row index, column index, value)

Definition 5. CRS-format: We have a value vector with all the non-zero values, an column index array the index at which index in the column the value from the value vector is. The row ptr-array points to the value and index value.

Note 9. How to create an array:

1. Intermediate COO format
2. Convert to CRS

Alternative: One can reserve the space and insert

The following was presented in the exercise on 23. September 2019.

3 Introduction to C++

```
#include<iostream>
int main() {
    std::cout << "Hello World!\n"; // This is a comment << endl; is equivalent to
     \n
    return 0; // Declares the end of the function.
}
```

```
int x;
x=5;
return &x; // Returns the address in memory of x
```

A pointer is an address which points to another cell in memory. The pointer is declared as follows:

```
char t='a';
char *y;
y = &t;
cout << y; // Returns the address
cout << *y; // Return the value behind the address
```

```
void f1(int x) {x++;} // Creates a new x;
void f2(int &x) {x++;} // Edits the variable the function is called with.
```

Note 10. std::vector is an array which resizes itself dynamically

Note 11. delete [] array deletes the array.

```cpp
#include <iostream>
#include <Eigen/Dense>
using Eigen::MatrixXd;
int main()
{
  MatrixXd m(2,2);
  m(0,0) = 3;
  m(1,0) = 2.5;
  m(0,1) = -1;
  m(1,1) = m(1,0) + m(0,1);
  std::cout << m << std::endl;
}
```

The following was presented in the lecture on 10. October 2019.

3.1 Solving a Sparse Matrix in Eigen

```cpp
Eigen::SparseLU<SparseMatrix> solver(A);
x = solver.solve(b);
```

4 Direct Methods for Linear Least Squares Problems

Theorem 4.1

A linear system of equations is solvable if and only if it is solvable for the normal equations (NEQ)

$$A^T Ax = A^T b \tag{5}$$

Algorithmus 4.1: Normal equation method to solve LSE

1. Compute regular matrix $C = A^T A$
2. Compute right hand side vector $c = A^T b$
3. Solve linear system of equations $Cx = c$

```cpp
VectorXd x = (A.transpose()*A).llt().solve(A.transpose()*b) // llt() ~ lu()
```

Runtime: $\mathcal{O}(n^2 \cdot m + n^3)$

Note 12. Normal equations are vulnerable to roundoff errors. Another problem is the loss of sparsity. One way to solve this is by using the extended normal equation

$$k = b - Ax \text{ and } A^T(b - Ax) = 0 \Leftrightarrow A^T k = 0 \text{ and } k + Ax = b$$

$$\begin{bmatrix} A^T & 0 \\ I & A \end{bmatrix} \begin{bmatrix} k \\ x \end{bmatrix} = \begin{bmatrix} 0 \\ b \end{bmatrix}$$

4.1 Orthogonal Transformation Methods

Note 13. The idea is to put the matrix into a triangular matrix. $\Rightarrow$ QR-Decomposition

The following was presented in the lecture on 11. October 2019.

4.2 Householder QR decomposition

$$H = I - \frac{2vv^T}{v^T v} \tag{6}$$

where v is the first column vector with the first entry subtracted by the norm of this column vector.

```
using index_t = MatrixXd::Index;
const index_t m = A.rows(), n = A.cols();
Eigen::HouseholderQR<MatrixXd> qr(A);
MatrixXd Q = (qr.householderQR()*MatrixXd::Identity(m,n));
MatrixXd R = qr.matrixQR().block(0,0,n,n).template triangluarView<Eigen::Upper>()
    ;
```

4.3 Least Squares Solver

```
x = A.householderQR().solve(b);
(A*x-b).norm()
```

Cost HouseholderQR: $\mathcal{O}(mn^2)$ and Cost solve: $\mathcal{O}(mn + n^2)$

Note 14. Use orthogonal transformations methods for least squares whenever it is dense
Use normal equations in the expanded form when it is sparse

4.4 Singular Value Decomposition

Theorem 4.2

U and V are unitary matrices. And Σ is a diagonal Matrix, so that:

$$A = U\Sigma V^H \tag{7}$$

Lemma 4.1

The squares σ_i^2 of the non-zero singular values of A are the non-zero eigenvalues of $A^H A$, AA^H

4.5 Rewriting the economical SVD

$$A = \sum_{l=1}^{rank(1)} \sigma_l \cdot (U)_{i,l} \cdot (V)_{i,l}^H \tag{8}$$

4.6 Economical SVD in Eigen

```
Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeThinU | Eigen::ComputeThinV);
```

Cost of SVD is $\mathcal{O}(n^3)$

The following was presented in the exercise on 14. October 2019.

Theorem 4.3: Kronecker Law

$$(A \otimes B)(C \otimes D) = (AC) \otimes (BD) \tag{9}$$

The following was presented in the lecture on 17. October 2019.

4.7 Computing rank() in Eigen

```
A.jacobiSvd().setThreshold(tol).rank();
```

4.8 Computing generalized solution of $Ax = b$ via SVD

$$y = \sum_r^{-1} U_1^T b \tag{10}$$

$$x = V_1 \sum_r^{-1} U_1^T b \tag{11}$$

```
Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeThinU | Eigen::ComputeThinV);
svd.solve(b);
```

4.9 SVD-Based Optimization & Approximation

Find $x \in \mathbb{R}^n$ so that $\|Ax\|_2$ is maximal.

$$y \overset{\text{def}}{=} v^T x \Rightarrow \|Ax\|_2^2 = \sum_{l=1}^{n} \sigma_l^2 y_l^2 \rightarrow \max \tag{12}$$

The maximal value is $\sigma_1 = \|A\|_2$ where $x = (V)_{i,1}$ The minimal value is σ_n where $x = (V)_{i,n}$

4.10 Fitting of Hyperplans

```
MatrixXd R = A.householderQr().matrixQR().template triangluarView<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::Uper>().solve(R.block(0, p-dim,p-
    dim, dim)) * n)(0);
```

Theorem 4.4: Best low rank approximation

$$\|A - A_k\|_2 \le \|A - F\|_2 \quad \forall F \in R_k(m,n \overset{\text{def}}{=} \{M \in \mathbb{R}^{m,n} : rank(M) \le k\} \tag{13}$$

A_k is the rank-k best approximation.

The following was presented in the lecture on 18. October 2019.

4.11 SVD-based low rank matrix compression

```
const Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeThinU | Eigen::ComputeThinV
    );
(svd.matrixU().leftCols(k)) * (svd.singularValues().head(k).asDiagonal()) * (svd.
    matrixV().leftCols(k).transpose());
```

Note 15. Singular Value Decomposition can be used to compress images. It is especially useful for machine learning, as the input data can be made smaller.

4.12 Principal Component Analysis (PCA)

Can be either used for trend detection and data classification.
One can find the most important data in the first column.

Theorem 4.5

The subspace $\mathcal{U}_k$ spanned by the first k left singular vector of A solves the Proper Orthogonal Decomposition Problem (POD)

$$\mathcal{U}_k = \mathcal{R}\left((U)_{:,1,k}\right) \quad \text{with } A = U\Sigma V^T \text{ the SVD of } A \tag{14}$$

4.13 Constrained Least Squares

The following was presented in the lecture on 24. October 2019.

5 Data Interpolation and data fitting

Definition 6. In a one-dimensional interpolation there are data points given and the goal is to reconstruct a function:

$$f(t_i) = y_i \tag{15}$$

```
interpolate(const vector<double> &t, const vector<double> &y);
double operator (double t) const;
```

5.1 Horner Scheme

$$p(t) = t(\cdots t(t(a_n \cdot t + a_{n-1}) + a_{n-2}) + \cdots + a_1) + a_0 \tag{16}$$

```
Eigen::VectorXd horner(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;
}
```

Cost: $\mathcal{O}(n)$ mit n zeigt das Grad vom Polynomial an.

5.2 Lagrange Polynomial

$$L_i(t) = \frac{(t-t_0)\cdot\ldots\cdot(t-t_{i-1})\cdot(t-t_{i+1})\cdot\ldots\cdot(t-t_n)}{(t_i-t_0)\cdot\ldots\cdot(t_i-t_{i-1})\cdot(t_i-t_{i+1})\cdot\ldots\cdot(t_i-t_n)} \tag{17}$$

always has a unique solution. The cost is $\mathcal{O}(n^2 N)$

5.3 Barycentric interpolation formula

$$p(t) = \frac{\sum_{i=0}^{n} \frac{\lambda_i}{t-t_i} y_i}{\sum_{i=0}^{n} \frac{\lambda_i}{t-t_i}} \tag{18}$$

where λ_i is the denominator of the Lagrange Polynomial.

Algorithmus 5.1: Aitken-Neville

```
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);
}
```

The following was presented in the lecture on 31. October 2019.

Theorem 5.1

The cubic Hermite interpolation polynomial with slopes provides a local monotonicity preserving C^1 - Interpolant (The function is smooth).

Definition 7. Given an interval and a knot set/mesh, the vector space of the spline functions of degree d is defined by

$$S_{d,M} \overset{\text{def}}{=} \{s \in C^{d-1}(I) : \quad s_j \overset{\text{def}}{=} s_{t_{j-1},t_j} \in P_d \forall j = 1,..,n\} \tag{19}$$

is $d-1$ times continuously differentiable.

The following was presented in the lecture on 1. November 2019.

5.4 Cardinal cubic spline

A cardinal cubic spline has global support, but exponential decay.

Algorithmus 5.2: Least Square Fitting

The goal is to find a continuous function so that the difference of the actual point and the function squared is minimal.

$$x = \arg\min \|Az - y\|_2^2 \tag{20}$$

Definition 8. Vandermonde matrix

$$(A)_{ij} = t_i^{j-1} \tag{21}$$

Algorithmus 5.3: Polynomial fitting

```
Vector polyfit(const VectorXd &t, const VectorXd &y, const unsigned &order) {
    Eigen::MatrixXd A = Eigen::MatrixXd::Ones(t.size().order + 1);
    for (unsigned j=1; j<order +1; j++) {
        A.col(j) = A.col(j-1).cwiseProduct(t);
    }
    Eigen::VectorXd coeffs = A.householderQr().solve(y);
    return coeffs.reverse();
}
```

Definition 9. Overfitting is fitting data with functions from a large space; it often produces poorer results.

6 Approximation of Functions in 1D

Simple bound for the approximation norm

$$\inf_{p \in P_n} \|f - p\|_{L^\infty([-1,1])} \leq C(r) n^{-r} \left\| f^{(r)} \right\|_{L^\infty([-1,1])'} \tag{22}$$

The following was presented in the lecture on 8. November 2019.

Definition 10. The Lagrangian interpolation approximation scheme is defined by

$$L_\tau \overset{\text{def}}{=} I_\tau(y) \in \mathcal{P}_n \tag{23}$$

with $y \overset{\text{def}}{=} (f(t_0), ..., f(t_n))^T$

Definition 11. Algebraic convergence $T(n) \leq n^{-p}$ mit $p > 0$

Exponential Convergence $T(n) \leq q^n$ mit $0 < q < 1$

Theorem 6.1

$$f(t) - L_\tau(f)(t) = \frac{f^{(n+1)}(\tau_t)}{(n+1)!} \cdot \prod_{j=0}^{n} (t - t_j) \tag{24}$$

Note 16. Quantitative interpolation error estimates rely on smoothness.

Definition 12. The n^{th} Chebychev polynomial is $T_n(t) \overset{\text{def}}{=} \cos(n \arccos t)$ mit $-1 \leq t \leq 1$

Note 17. It is recommended to use Chebychev nodes for approximation by polynomial interpolation.

6.1 Approximation by piecewise polynomials

The idea is to use piecewise polynomials with respect to a grid/mesh to approximate a function.

The following was presented in the lecture on 8. November 2019.

7 Numerical Quadrature

Definition 13. An n-point quadrature formula/quadrature rule provides an approximation:

$$\int_a^b f(t)dt \approx Q_n(f) \overset{\text{def}}{=} \sum_{j=1}^{n} w_j^n f(c_j^n) \tag{25}$$

The following was presented in the lecture on 15. November 2019.

Definition 14.

$$order(Q_n) \overset{\text{def}}{=} \max\left\{m \in \mathbb{N}_0 : \quad Q_n(p) = \int_a^b p(t)dt \quad \forall p \in P_m\right\} + 1 \tag{26}$$

Theorem 7.1

$$Q_n(f) \overset{\text{def}}{=} \sum_{j=1}^{n} w_j f(c_j) \tag{27}$$

has order $\geq n$ if and only if

$$w_j = \int_a^b L_{j-1}(t)dt \tag{28}$$

Theorem 7.2

The maximal order of an n-point quadrature is $2n$

Theorem 7.3

The quadrature error satisfies:

$$E_n(f) \overset{\text{def}}{=} \left|\int_a^b f(t)dt - Q_n(f)\right| \leq 2|b-a| \inf_{p \in P_{q-1}} ||f-p||_{L^\infty(|a,b|)} \tag{29}$$

The following was presented in the lecture on 15. November 2019.

7.1 Adaptive numerical Quadrature

A priori Fix the nodes before the evaluation

a posteriori The nodes are chosen or improved during the computation

7.2 Adaption loop for numerical quadrature

1. Estimate
2. Check Termination
3. Mark
4. Refine

The following was presented in the lecture on 21. November 2019.

8 Iterative Methods for Non-Linear Systems of Equations

Definition 15. An iterative method for approximately solving the non-linear equation $F(x) = 0$ is an algorithm generating an arbitrarily long sequence $\left(x^{(k)}\right)_k$ of approximative solutions.

Definition 16. A stationary m-point iterative method converges locally, if there is a neighborhood such that

$$x^{(0)}, \dots, x^{(m-1)} \in U \Rightarrow x^{(k)} \text{well defined} \wedge \lim_{k\to\infty} x^{(k)} = x^\star \tag{30}$$

If $U = D$, the iterative method is globally convergent.

Definition 17. A sequence $x^{(k)}$ converges linearly to $x^\star$

$$\exists 0 < L < 1 : \quad \left\| x^{(k-1)} - x^\star \right\| \leq L \cdot \left\| x^{(k)} - x^\star \right\| \tag{31}$$

Definition 18. A sequence $x^{(k)}$ converges with order p to $x^\star$

$$\exists C > 0 : \quad \left\| x^{(k-1)} - x^\star \right\| \leq C \cdot \left\| x^{(k)} - x^\star \right\|^p \tag{32}$$

Example 1.

$$x^{(k+1)} = \frac{1}{2}\left(x^{(k)} + \frac{a}{x^{(k)}}\right) \Rightarrow \left| x^{(k+1)} - \sqrt{a} \right| = \frac{1}{2x^{(k)}} \cdot \left| x^{(k)} - \sqrt{a} \right|^2 \tag{33}$$

Example 2.

$$x^{(k+1)} = x^{(k)} + \frac{\cos x^{(k)} + 1}{\sin x^{(k)}} \tag{34}$$

Note 18. Be careful, because the estimation error is much smaller than the iteration error

The following was presented in the lecture on 22. November 2019.

8.1 Newton's Model

$$\tilde{F}(x) \overset{\text{def}}{=} F\left(x^{(k)}\right) + DF\left(x^{(k)}\right)\left(x - x^{(k)}\right) \overset{!}{=} 0 \tag{35}$$

$$x^{(k+1)} = x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})} \tag{36}$$

Algorithmus 8.1: Newton's method

```
VecType s(x.size());
do {
    s = DFinv(x, F(x));
    x -= s;
}
while ((s.norm() > rtol * x.norm()) && (s.norm() > atol));
return x;
```

The following was presented in the lecture on 28. November 2019.

8.2 Newton correction

$$S = -X^{(k)}AX^{(k)} + X^{(k)} \tag{37}$$

Algorithmus 8.2: Efficient implementation of simplified Newton method

```cpp
auto lu = DF(x).lu();
Vec s;
double ns, nx;
do {
     s = lu.solve(F(x)); // O(n^2)
     x = x - s;
     ns = s.norm();
     nx = x.norm();
}
while((ns > rtol*nx) && (ns > atol));
```

Stop as soon as

$$\left\|\nabla\tilde{x}^{(k)}\right\| \le \tau_{rel}\left\|x^{(k)}\right\| \le \tau_{abs} \tag{38}$$

with

$$\nabla\tilde{x}^{(k)} \overset{\text{def}}{=} DF\left(x^{(k-1)}\right)^{-1}F\left(x^{(k)}\right) \tag{39}$$

Note 19. With a damping strategy one can lead the Newton correction to the right direction.

The following was presented in the lecture on 29. November 2019.

8.3 Quasi-Newton Method

$$J_k \overset{\text{def}}{=} J_{k-1} + \frac{F\left(x^{(k)}\right)\left(x^{(k)} - x^{(k-1)}\right)^T}{\left\|x^{(k)} - x^{(k-1)}\right\|_2^2} \tag{40}$$

The final form of Broyden's quasi-Newton method for solving $F(x) = 0$:

$$x^{(k+1)} \overset{\text{def}}{=} x^{(k)} + \nabla x^{(k)} \qquad \nabla x^{(k)} \overset{\text{def}}{=} -J_k^{-1}F\left(x^{(k)}\right) \tag{41}$$

$$J_{k+1} \overset{\text{def}}{=} J_k + \frac{F\left(x^{(k+1)}\right)\left(\nabla x^{(k)}\right)^T}{\left\|\nabla x^{(k)}\right\|_2^2} \tag{42}$$

8.4 Non-Linear Least Squares

Definition 19. The non-linear least squares solution is defined as

$$x^\star = \arg\min_{x \in D} \|F(x)\|^2 \tag{43}$$

Algorithmus 8.3: Gauss-Newton Iteration

$$x^{(k+1)} \overset{\text{def}}{=} \arg\min_{x \in \mathbb{R}^n} \left\| F\left(x^{(k)}\right) + DF\left(x^{(k)}\right)\left(x - x^{(k)}\right) \right\|_2 \tag{44}$$

```
Eigen::VectorXd x = init;
Eigen::VectorXd s = J(x).househodlerQr().solve(F(x));
x = x -s;
while ((s.norm() > rtol * x.norm()) && (s.norm() > atol)) {
    s = J(x).housholderQr().solve(F(x));
    x = x - s;
}
return x;
```

The following was presented in the lecture on 5. December 2019.

11 Numerical Integration - Single Step Methods

Definition 20. An autonomous ordinary differential equation (ODE) is a function that does not depend on time but only on state.

Theorem 11.1: Peano & Picard-Lindelöf

If f is continuous differentiable then for all initial conditions the initial value problem (IVP) has a solution with maximal domain.

The following was presented in the lecture on 6. December 2019.

Definition 21. Given a discrete evolution ψ and initial state y_0 and a Mesh M the recursion:

$$y_{k+1} \overset{\text{def}}{=} \psi(t_{k+1} - t_k, y_k) \tag{45}$$

defines a single-step method (SSM) for the autonomous IVP $\dot{y} = f(y)$

11.1 Runge-Kutta Methods

Goal: Construct explicit SSM of higher order.

The following was presented in the lecture on 12. December 2019.

11.2 Bootstrap Construction

$$y(t_1) = y_0 + \int_{t_0}^{t_1} f(\tau, y(\tau))d\tau \tag{46}$$

Definition 22. For an s-stage explicit Runge-Kutta single step method for the ODE $\dot{y} = f(t, y)$ is defined by

$$k_i \overset{\text{def}}{=} f(t_0 + c_i h, y_0 + h\sum_{j=1}^{i-1} a_{ij}k_j), \quad i = 1, ..., s \quad y_1 \overset{\text{def}}{=} y_0 + h\sum_{i=1}^{s} b_i k_i \tag{47}$$

The vectors k_i are called increments.

11.3 Adaptive Stepsize Control

Note 20. There is no global error control through local-in time adaptive timestepping. The absolute/relative tolerances imposed for local-in-time adaptive timestepping do not allow to predict the accuracy of a solution.

The following was presented in the lecture on 13. December 2019.

12 Single-Step methods for Stiff Initial-Value Problems

Theorem 12.1

The discrete evolution Ψ_λ^h of an explicit s-stage Runge-Kutta step method with Butcher scheme for the ODE $\dot{y} = \lambda y$ amounts to a multiplications with the number

$$\Psi_\lambda^h = S(\lambda h) \Leftrightarrow y_1 = S(\lambda h) y_0 \tag{48}$$

where S is the stability function

$$S(z) = \overset{\text{def}}{=} 1 + zb^T(I - z\mathfrak{A})^{-1} = \det(I - z\mathfrak{A} + zIb^T) \tag{49}$$

Corollary 12.1

For a consistent s-stage explicit Runge-Kutta single step method the stability function S is a non-constant polynomial of degree $\leq s$ that is $S \in P_s$

Lemma 12.1

Let S denote the stability function of an s-stage explicit Runge-Kutta single step method of order $q \in \mathbb{N}$, then:

$$|S(z) - \exp(s)| = \mathcal{O}\left(|z|^{q+1}\right) \quad \text{for } |z| \to 0 \tag{50}$$

The following was presented in the lecture on 19. December 2019.

Theorem 12.2

The sequence of approximations generated by an explicit RK-SSM with stability function S applied to the linear autonomous ODE $\dot{y} = MY$, $M \in C^{d,d}$ with uniform timestep $h > 0$ decays exponentially for every initial state $y_0 \in C^d$, if and only if $|S(\lambda, h)| < 1$ for all eigenvalues λ_i of M.

Definition 23. Let the discrete evolution Ψ for a single step method applied to the scalar linear ODE $\dot{y} = \lambda y$ and $\lambda \in \mathbb{C}$ be of the form

$$\Psi^h y = S(z)y, \;\; y \in \mathbb{C}, \; h > 0 \quad z \overset{\text{def}}{=} h\lambda \tag{51}$$

The region of absolute stability of the single step method is given by

$$S_\Psi = \{z \in \mathbb{C} : \; |S(x)| < 1\} \subset \mathbb{C} \tag{52}$$

Note 21. An initial value problem is called stiff, if stability imposes much tighter timestep constraints on explicit single step methods than than the accuracy requirements. The discrete evolution of the RK-SSM for $\dot{y} = f(y)$ in the state $y^\star$ is close to the discrete evolution of the same RK-SSM applied to the linearization of the ODE in $y^\star$
For small timestep the behavior of an explicit RK-SSM applied to $\dot{y} = f(y)$ close to the state $y^\star$ is determined by the eigenvalues of the Jacobian $D\,f(y^\star)$
An initial value problem for an autonomous ODE will probably be stiff if

$$\min\{Re\lambda : \lambda \in \omega(D\,f(y(t)))\} << 0 \tag{53}$$

$$\max\{Re\lambda : \lambda \in \omega(D\,f(y(t)))\} \approx 0 \tag{54}$$

where $\omega(M)$ is the spectrum of the matrix M.

Note 22. For any timestep the implicit Euler method generates exponentially solution decaying solution sequences $(Y_k)_{k=0}^{\infty}$ for $\dot{y} = My$ with diagonalizable matrix $M \in \mathbb{R}^{d,d}$ with eigenvalues $\lambda_1, ..., \lambda_d$ if $Re\ \lambda_i < 0$

The following was presented in the lecture on 20. December 2019.

Definition 24. A Runge-Kutta single step method with stability function S is A-stable, if

$$C^{\sim} \overset{\text{def}}{=} \{z \in C : \; Re\ z \leq 0\} \subset S_\Psi \quad \text{where } \subset S_\Psi \text{ is the Region of Stability} \tag{55}$$

Note 23. A stable Runge-Kutta single step method will not be affected by stability induced timestep constraints when applied to stiff IVP.

Definition 25. A Runge-Kutta method is L-stable/asymptotically stable, if its stability function satisfies:

1. $Re\ z < 0 \Rightarrow |S(z)| < 1$
2. $\lim_{Re\ z \to -\infty} S(z) = 0$

0 C++ / Eigen 14

```
g++ -std=c++11 -I /usr/include/eigen3
-Wno-deprecated-declarations input.cpp -o output
```

```
#include <Eigen/Dense> #include <Eigen/Sparse>
#include <iostream>
using namespace Eigen; using Matrix = Eigen::
    SparseMatrix<double, Eigen::RowMajor>;
std::cout << std::setw(15) << "Hello\n"; // Output Width
```

```
#include "matplotlibcpp.h"
namespace plt = matplotlibcpp;
plt::figure(); plt::plot(x, y, "+", {{"label", "approx IVP"
 }}); plt::savefig("save.png"); plt::show();
```

1 Computing with Matrices and Vectors 32

Definition Kronecker Product $A \otimes B$

$$A \otimes B \overset{\text{def}}{=} \begin{pmatrix} (A)_{1,1}B & (A)_{1,2}B & \dots & (A)_{1,n}B \\ \vdots & & & \vdots \\ (A)_{m,1}B & (A)_{m,2}B & \dots & (A)_{m,n}B \end{pmatrix}$$

Theorem Kronecker Law

$$\boxed{(A \otimes B)(C \otimes D) = (AC) \otimes (BD)}$$

```
MatrixXd kron(const MatrixXd &A, const MatrixXd &B){
  MatrixXd C(A.rows() * B.rows(), A.cols() * B.cols());
  for(unsigned int i=0; i<A.rows(); i++)
    for(unsigned int j=0; j<A.cols(); j++)
      C.block(i * B.rows(), j * B.cols(), B.rows(),
      B.cols()) = A(i, j) * B;
  return C;}
```

2 Direct Methods for Linear Systems of Equations 96

Note The Gaussian Elimination is stable and is not affected by roundoff in practice.

Definition A sparse matrix is a matrix where most entries are 0 and this is worth exploiting.

Note If it is a sparse matrix one can save the matrix in the COO/Triplet format with (row index, column index, value).

Definition CRS-format: We have a value vector with all the non-zero values, an column index array the index of which index in the column the value from the value vector is. The row ptr-array points to the value and index value.

Note How to create an array:

1. Intermediate COO format   2. Convert to CRS

Alternative: One can reserve the space and insert.

2.7.2 Solving a Sparse Matrix in Eigen 145

```
Eigen::SparseLU<SparseMatrix> solver(A);
x = solver.solve(b);
```

Solve $X_j^T a = y_j$

```
FullPivLU<MatrixXd> lu_m = (X.transpose()*X).fullPivLu();
VectorXd q_m = lu_m.solve(X.transpose()*z);
solve {
    VectorXd p = lu_m.solve(xm);
    VectorXd w = qm_m + ym * p;
    double xi = xm.dot(p);
    return w - xm.dot(w) / (1.+xi) * p;
}
```

3 Direct Methods for Linear Least Squares Problems

176

Theorem

A linear system of equations is solvable if and only if it is solvable for the normal equations (NEQ): $A^TAx = A^Tb$

Algorithmus Normal equation method to solve LSE

1. Compute regular matrix $C = A^TA$
2. Compute right hand side vector $c = A^Tb$
3. Solve linear system of equations $Cx = c$

```
VectorXd x= (A.transpose()*A).llt().solve(
    A.transpose()*b)   // llt() ~ lu()
```

Runtime $O(n^2 \cdot m + n^3)$

190

Normal equations are vulnerable to roundoff errors. Another problem is the loss of sparsity. One way to solve this is by using the extended normal equations.

$k = b - Ax$ and $A^T(b - Ax) = 0$ $\quad \begin{pmatrix} A^T & 0 \\ I & A \end{pmatrix}\begin{pmatrix} k \\ x \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix}$

$\Leftrightarrow A^Tk = 0$ and $k + Ax = b$

3.3 Householder QR decomposition

195

$H = I - \frac{2vv^T}{v^Tv}$ where v is the first column vector with the first entry subtracted by the norm of this column vector.

3.3 Least Squares Solver

209

```
x = A.householderQR().solve(b); return (A*x-b).norm();
```

Cost HouseholderQR: $O(mn^2)$ and Cost solve $O(mn + n^2)$

Note Use orthogonal trasformations methods for least squares whenever it is dense. Use normal equations in the expanded form when it is sparse.

3.4 Singular Value Decomposition

217

Theorem

U and V are unitary matrices. Σ is a diagonal matrix, so that:

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

Solution of $Ax = b$: $y = \Sigma_r^{-1} U_1^H b \qquad x = V_1 \Sigma_r^{-1} U_1^H b$

```
JacobiSVD<MatrixXd> svd(A, computeThinU | computeThinV);
svd.solve(b);
```

Theorem Best low rank approximation

$\|A - A_k\|_2 \leq \|A - F\|_2 \quad \forall F \in R_k(m,n := \{M \in \mathbb{R}^{m,n} : \text{rank}(M) \leq k\})$

A_k is the rank-k best approximation

In Principal Component Analysis (PCA) one can find the most important data in the first column.

3.4 SVD of AB^T

```
HouseholderQR<MatrixXd> QRA = A.householderQr();
MatrixXd QA = QRA.householderQ()*MatrixXd::Identity(
                  std::min(m, k));
MatrixXd RA = MatrixXd::Identity(std::min(m, k), m)*QRA.
         matrixQR().triangularView<Upper>(); //Same for B
JacobiSVD<MatrixXd> svd (RA*RB.transpose(),
         ComputeFullU | ComputeFullV);
VectorXd s = svd.singularValues();
MatrixXd U = svd.matrixU(); MatrixXd V= svd.matrixV();
U = QA * U;  V= QB * V;
return std::make_tuple(U, s.asDiagonal(), V);
```

5 Data Interpolation and data fitting 312

Definition In a one-dimensional interpolation there are data points given and the goal is to reconstruct a function:

$$f(t_i)=y_i$$

5.2 Lagrange Polynomial 321

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

always has a unique solution. The cost is $O(n^2N)$

5.2 Barycentric interpolation formula 325

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

where λ_i is the denominator of the Lagrange Polynomial.

Theorem

The cubic Hermite interpolation polynomial with slopes provide a local monotonicity preserving C^1-Interpolant (The function is smooth)

Definition Given a interval and a knot set/mesh, the vector space of the spline functions of degree d is defined by $S_{d,M}:=\{s\in C^{d-1}(I):$ $s_j:=s_{|[t_{j-1},t_j]}\in P_d\ \forall j=1,\ldots,n\}$ is $d-1$ times continuously differentiable.

5.5 Cardinal cubic spline 352

A cardinal cubic spline has global support, but exponential decay.

Algorithms Least Square Fitting

The goal is to find a continuous function so that the difference of the actual point and the function squared is minimal: $x=\arg\min\|Az-y\|_2^2$ 371

Definition Vandermonde matrix $(A)_{ij}=t_i^{j-1}$ $V=\begin{pmatrix}1 & t_0 & t_0^2 & \cdots & t_0^n\\ \vdots & & & & \\ 1 & t_n & t_n^2 & \cdots & t_n^n\end{pmatrix}$

5.3 Linear Interpolant 344

```
LI::LI(const data &i_points_) { i_points = i_points_;
  auto Ordering = [] (const pair &P, const pair &Q) ->
     bool {return P.first < Q.first;};
  std::sort(i_points.begin(), i_points.end(), Ordering);}
double LI::operator() (double x) {
  auto Compare = [] (const pair &P, double V) -> bool {
  return P.first < V;};
  auto it = std::lower_bound(i_points.begin(),
            i_points.end(), x, Compare);
  if ((it == i_points.begin() && it->first != x)
      || it == i_points.end()) return 0;
  double dist_r=(x-(it-1)->first)/(it->first-(it-1)->first);
  return (it-1)->second*(1-dist_r)+it->second*dist_r;}
```

5.5 Quadratic Spline 360

```
MatrixXd quadSpline(const VectorXd &x, const VectorXd &y) {
int N=x.size()-1; VectorXd dx = x.tail(N) - x.head(N);
VectorXd dx2 = dx.cwiseProduct(dx); SparseMatrix<double>
    A(3*N,3*N); A.reserve(VectorXd::Constant(3*N,3));
for (int i=0; i<N; i++) A.insert(i,2*N+i)=1; //f(x_i)=y_i
// require f(x_{i+1}) = y_{i+1}:
for (int i=0; i<N; i++) { A.insert(N+i, i) = dx2(i);
   A.insert(N+i,N+i)= dx(i); A.insert(N+i,2*N+i)=1;}
// require f' to be continuous at x_i
for (int i=0; i<N; i++) { A.insert(2*N+i, i) =2*dx(i);
   A.insert (2*N+i,N+i)=1; A.insert(2*N+i, N+((i+1)%N))=-1;}
A.makeCompressed(); VectorXd b(3*N);
b<< y.head(N), y.tail(N), VectorXd::Zero(N);
SparseLU<SparseMatrix<double>> solver;
solver.compute(A); VectorXd out = solver.solve(b);
return Map<MatrixXd>(out.data(), N, 3);
```

6 Approximation of Functions in 1D 377

Simple bound for the approximation norm:

$$\inf_{p \in P_n} \|f-p\|_{L^\infty([-1,1])} \leq C(r) n^{-r} \|f^{(r)}\|_{L^\infty([-1,1])}$$

Definition The Lagrangian interpolation approximation scheme is defined by

$$L_\mathcal{T} := I_\mathcal{T}(y) \in P_n$$

with $y := (f(t_0), \dots, f(t_n))^T$

Definition Algebraic convergence $T(n) \leq n^{-p}$ mit $p > 0$
Exponential convergence $T(n) \leq q^n$ mit $0 < q < 1$

Theorem Representation of interpolation error

$$\boxed{f(t) - L_\mathcal{T}(f)(t) = \frac{f^{(n+1)}(\tau_t)}{(n+1)!} \cdot \prod_{j=0}^{n} (t - t_j)}$$

Note Quantitative interpolation error estimates rely on smoothness.

Definition The n^{th} Chebychev polynomial is:

$$T_n(t) := \cos(n \arccos t) \text{ mit } -1 \leq t \leq 1$$

Chebychev nodes: $\mathcal{T} = \{\cos(2k + \frac{1}{2}(n+1) * \pi)\}$

Note It is recommended to use Chebychev nodes for approximation by polynomial interpolation.

Note Clenshaw algorithm should be used for the evaluation of Chebychev expansion.

6.5 Approximation by piecewise polynomials 431

The idea is to use piecewise polynomials with respect to a grid/mesh to approximate a function.

6.1 Gauss-Chebyshev 397

```
double q=0.0;
for(int i=0; i<n; i++)
    q+=f(std::cos((2*i+1)*M_PI/(2*n)));
return q * M_PI/n;
```

5.2 Aitken-Neville to evaluate the derivative 328

```
for(int i=0; i<x.size(); i++){
    VectorXd p(y);
    VectorXd dP = VectorXd::Zero(y.size());
    for(int im=1; im<y.size(); im++)
        for(int i0 = im - 1; i0 >= 0; i0--){
            dP(i0)=(p(i0+1)+(x(i)-t(i0))*dP(i0+1)
            -p(i0)-(x(i)-t(im))*dP(i0))/(t(im)-t(i0));
            p(i0)=((x(i)-t(i0))*p(i0+1)-(x(i)-t(im))
            *p(i0))/(t(im)-t(i0));
    } ret(i)=dP(0);}
return ret;
```

2.7 Triplets 146

```
std::vector<Eigen::Triplet<double>>triplets;
triplets.reserve(rows*cols);
triplets.push_back(Eigen::Triplet<double>(i,j,v));
Eigen::SparseMatrix<double,Eigen::RowMajor>sparseMatrix(rows,cols);
sparseMatrix.setFromTriplets(triplets.begin(), triplets.end());
```

7 Numerical Quadrature 441

Definition An n-point quadrature rule provides an approximation:

$$\int_a^b f(t)\,dt \approx Q_n(f) := \sum_{j=1}^{n} w_j^n \cdot f(c_j^n)$$

The maximal order of an n-point quadrature is 2n

Theorem Quadrature error

The quadrature error satisfies

$$E_n(f) := \left| \int_a^b f(t)\,dt - Q_n(f) \right| \leq 2|b-a| \inf_{p \in P_{n-1}} \|f-p\|_{L^\infty([a,b])}$$

7.5 Adaptive Numerical Quadrature 470

A priori: Fix the nodes before the evaluation

A posterior: The nodes are chosen are improved during the comput.

Adaption loop: Estimate, Check Termination, Mark, Refine

7.3 Integrate Gauss Quadrature 449

```
double integrate(const Function &f, unsigned n, double a,
                   double b){
      double I=0; QuadRule qr; gaussquad(n, qr);
      VectorXd &nodes = qr.nodes;
      VectorXd &weights = qr.weights;
      for (unsigned i=0; i<nodes.size(); i++){
            double x= (b+a)/2 +(b-a)/2.*nodes(i);
            I += f(x)* weights(i);
      }
      I*=(b+a)/2;
      return I;}
```

```
double getLambda(const Function &f, double atol, double
                    rtol, unsigned n=7, unsigned maxit=10){
      double lambda = 0.;
      auto f_exp_f = [&] (double x){return f(x)*
              std::exp(lambda* f(x));};
      auto exp_f =[&] (double x){
                 return std::exp(lambda * f(x));};
      for (unsigned i=0; i<maxit; i++){
            double DF = integrate(f_exp_f, n, 0, 1) + 1.;
            double F = integrate(exp_f, n, 0, 1) + lambda;
            double lambda_new = lambda - F/DF;
            double step = std::abs(lambda_new - lambda);
            if(step<atol && step < rtol*std::abs(lambda)) break;
            lambda = lambda_new;
      }
      return lambda;
}
```

7.3 Gauss-Legendre Quadrature rule 455

```
double quad(Function &&f, const VectorXd &w,
              const 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)*w(i);
      return I*(b-a)/2.;
}
double quadInf (const int n, Function &&f){
      VectorXd w, x;
      golubwelsh(n, w, x);
      auto ftilde = [&f] (double x) {
         return f(std::cos(x)/std::sin(x))/std::pow(std::sin(x),2);};
      return quad(ftilde, w, x, 0, PI);
}
```

8 Iterative Methods for Non-Linear Systems of Equations 493

Definition An iterative method for approximately solving the non-linear $F(x)=0$ is an algorithm generating an arbitrarily long sequence $(x^{(k)})_k$ of approximative solutions.

Definition A sequence $x^{(k)}$ converges linearly to x^*
$\exists 0<L<1$: $\|x^{(k+1)}-x^*\| \le L \cdot \|x^{(k)}-x^*\|$
A sequence $x^{(k)}$ converges with order p to x^*
$\exists C>0$: $\|x^{(k+1)}-x^*\| \le C \cdot \|x^{(k)}-x^*\|^p$

8.4 Newton's Method 514

$$\tilde{F}(x) := F(x^{(k)}) + DF(x^{(k)})(x-x^{(k)}) \overset{!}{=} 0$$

$$x^{(k+1)} = x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})} \qquad x^{(k+1)} = x^{(k)} - (DF(x^{(k)}))^{-1} F(x^{(k)})$$

Algorithms Newton's method

```
VecType s(x.size());
do { s = DFinv(x, F(x)); x -= s; }
while ((s.norm() > rtol*x.norm()) && (s.norm() > atol));
return x;
```
515

8.4 Newton correction 523

$$s = -X^{(k)} A X^{(k)} + X^{(k)}$$

Stop as soon as

$$\|\nabla \tilde{x}^{(k)}\| \le \tau_{rel} \|x^{(k)}\| \le \tau_{abs} \quad \text{with } \nabla \tilde{x}^{(k)} := DF(x^{(k-1)})^{-1} F(x^{(k)})$$

Note With a damping strategy one can lead the Newton correction to the right direction.

8.4 Quasi-Newton Method 531

$$J_k := J_{k-1} + \frac{F(x^{(k)})(x^{(k)} - x^{(k-1)})^T}{\|x^{(k)} - x^{(k-1)}\|_2^2}$$

The final form of Broyden's quasi-Newton method for solving $F(x)=0$:

$$x^{(k+1)} := x^{(k)} + \nabla x^{(k)} \qquad \nabla x^{(k)} := -J_k^{-1} F(x^{(k)})$$

$$J_{k+1} := J_k + \frac{F(x^{(k+1)})(\nabla x^{(k)})^T}{\|\nabla x^{(k)}\|_2^2}$$

8.5 Non Linear Least Squares 536

Definition The non-linear least squares solution is defined as

$$x^* = \arg\min_{x \in D} \|F(x)\|^2$$

Algorithms Gauss-Newton Iteration

$$x^{(k+1)} := \arg\min_{x \in \mathbb{R}^n} \|F(x^{(k)}) + DF(x^{(k)})(x - x^{(k)})\|_2$$

```
VectorXd x = init; VectorXd s;
do {
    s = J(x).householderQr().solve(F(x));
    x -= s;
} while ((s.norm() > rtol*x.norm()) && (s.norm() > atol));
return x;
```
540

11 Numerical Integration - Single Step Methods 657

Definition An autonomous ordinary differential equation (ODE) is a function that does not depend on time but only on state.

Theorem Peano & Picard-Lindelöf

If f is continuous differentiable then for all initial conditions the initial value problem (IVP) has a solution with maximal domain.

Definition Given a discrete evolution Ψ and initial state y_0 and a mesh M the recursion:

$$y_{k+1} := \Psi(t_{k+1} - t_k, y_k)$$

defines a single-step method (SSM) for the autonomous IVP $\dot{y} = f(y)$

11.4 Bootstrap Construction 663

$$y(t_1) = y_0 + \int_{t_0}^{t_1} f(\tau, y(\tau))\, d\tau$$

Definition For an s-stage explicit Runge-Kutta single step method for the ODE $\dot{y} = f(t,y)$ is defined by:

$$k_i := f(t_0 + c_i h, y_0 + h\sum_{j=1}^{i-1} a_{ij} k_j) \text{ for } i = 1,\dots,s \qquad y_1 := y_0 + h\sum_{i=1}^{s} b_i k_i$$

The vectors k_i are called increments.

11.4 SSPRK

```
SSPRK::SSPRK(const std::function<VectorXd(VectorXd)> &f,
             const MatrixXd &alphas, const MatrixXd &betas)
    : f(f), alphas(alphas), betas(betas) {
    assert(alphas.rows() == betas.rows() && "Size");
    assert(alphas.cols() == betas.cols() && "Size");
    s = alphas.rows() - 1;
}
```

```
std::vector<VectorXd> SSPRK::solve(const VectorXd & y0,
        double T, unsigned int N) {
    std::vector<VectorXd> ret(N+1);
    double h = T/N; ret.push_back(y0); ret.reserve(N+1);
    std::vector<VectorXd> K(s+2);
    for (int k=0; k<N; k++) {
        K.at(0) = ret.back();
        for (int i=1; i<=s+1; i++) {
            K.at(i) = VectorXd::Zero(K.at(0).size());
            for (int l=0; l<i; l++)
                K.at(i) += alphas(i-1, l) * K.at(l)
                    + betas(i-1, l) * h * f(K.at(l));
        } ret.push_back(K.back());
    } return ret; }
```

11.4 RK-SSM 665

```
template<class State> class RKIntegrator {
public: // Same as ImplicitRKIntegrator
private:
    void step(const Function &f, double h, const
              State &y0, State &y1) const {
        y1 = y0; std::vector<State> k; k.reserve(s);
        for (int i=0; i<s; i++) {
            State incr = y0;
            for (int j=0; j<i; j++)
                incr += h * A(i, j) * k.at(j);
            k.push_back(f(incr));
            y1 += h * b(i) * k.back();
        }
    }
    const MatrixXd A; const VectorXd b; unsigned int s;
};
```

11.1 Initial Value Problem 643

```
Vector2d Newton(Vector2d x, double h, int n, double tol){
    const Vector2d zk=x;
    for(int i=0; i<n; i++){
        if(F(x, zk, h).squaredNorm()<=tol*tol) return x;
        x -= DF(x, h).fullPivLu().solve(F(x,zk,h));
    } return x;}
Vector2d QuasiNewton(Vector2d x, double h, int n, double
    tol){
    const Vector2d zk =x;
    Matrix2d J_inr = DF(x,h).inverse();
    for(int i=0; i<n; i++){
            Vector2d dx = -J_inr * F(x, zk,h);  x+=dx;
            if(F(x, zk, h).squaredNorm()<=tol*tol) return x;
            Vector2d T = J_inr * F(x,zk, h);
            J_inr += (-T * dx.transpose()/(dx.squaredNorm()
            +dx.transpose()*T))* J_inr;
    } return x;}
Vector2d ImplicitEuler(Vector2d z, double h, int N) {
    for(int k=0; k<N; k++) z=Newton(z, h, 10, 1.0e-8);
    return z;}
```

5.2 Piecewise Lagrange Interpolant and Horner scheme 320
to evaluate

```
void Newton::Interpolate(const Matrix &y){
    int n =nodes.rows(); int M = nodes.cols();
    for (int l=0; l<M; l++){
        MatrixXd A(n, n);  A.setZero();
        for (int i=0; i<n; i++) {A(i,0)= l;
            for(int j=1; j<=i; j++)
    }       A(i,j)=A(i,j-1)*(nodes(i, l)-nodes{j-1, l));
        alpha.col(l)= A.triangularView<Eigen::Lower>().
            solve(y.col(l));}}

double Newton::Evaluate (const int L, double x){
    int n = alpha.rows(); double y = alpha(n-1, L);
    for (int i=n-2; i>=0; i--) //Horner Scheme
        y=y*(x-nodes(i,L))+alpha(i,l);
    return y;}
```

8.4 Quasi Linear Newton System 532

```
VectorXd newton_step (const VectorXd &x, const VectorXd &b) {
    const int n=x.size(); double nrm = x.norm();
    SparseMatrix<double> A(n, n);  A.reserve(3*n);
    for (int i=0; i< n; i++){ if (i>0){
        A.insert(i, i-1) = 1;} A.insert(i, i) = 3+nrm;
        if(i<n-1) A.insert (i, i+1)= 1;
    }
    SparseLU<SparseMatrix<double>> Ax_lu(A);
    VectorXd Axinv_b = Ax_lu.solve(b);
    VectorXd Axinv_x = Ax_lu.solve(x);
    return Axinv_b + Ax_lu.solve (x*X.transpose()
    *(x-Axinv_b))/(x.norm()+ x.dot(Axinvx));
}
VectorXd solveQLSystemNewton(double rtol, double atol,
    VectorXd &b){
    int n= b.size(); VectorXd x(n); x=b; VectorXd x_new(n);
    for(int itr=0; itr< 100; itr++){
        x_new = newton_step(x, b);
        double err = (x-x_new).norm(); x=x_new;
        if (err< atol || err< rtol * x_new.norm())
            break;
    }
    return x;}
```

12 Single-Step methods for Stiff Initial-Value Problems
GM

Definition Let the discrete evolution Ψ for a single step method applied to the scalar linear ODE $\dot{y} = \lambda y$ and $\lambda \in \mathbb{C}$ be of the form

$$\Psi^h y = S(z) y, \quad y \in \mathbb{C}, \ h > 0 \qquad z := h\lambda$$

The region of absolute stability of the single step method is given by

$$S_\Psi = \{z \in \mathbb{C} : |S(z)| < 1\} \subset \mathbb{C}$$

A IVP is called stiff, if stability imposes much tighter time step contraints on explicit single step methods than the accuracy requirements.

Note For any timestep the implicit Euler method generates exponentially decaying solution sequences $(y_k)_{k=0}^{\infty}$ for $\dot{y} = My$ with diagonalizable matrix $M \in \mathbb{R}^{d,d}$ with eigenvalues $\lambda_1, \dots, \lambda_d$ if $\mathrm{Re}\,\lambda_i < 0$

Definition A Runge-Kutta single step method with stability function S is A-stable if

$$\mathbb{C}^- := \{z \in \mathbb{C} : \mathrm{Re}\, z \le 0\} \subset S_\Psi \qquad \text{where } S_\Psi \text{ is the Region of Stability}$$

A Runge-Kutta method is L-stable/asymptotically stable, if its stability function satisfies:

1. $\mathrm{Re}\; z < 0 \Rightarrow |S(z)| < 1$
2. $\lim_{\mathrm{Re}\, z \to -\infty} S(z) = 0$

Note A stable Runge-Kutta single step method will not be affected by stability induced timestep constraints when applied to stiff IVP.

12.3 Implicit RK-Integrator

```
class implicit_RKIntegrator { public:
  implicit_RKIntegrator(const MatrixXd &A, const
  VectorXd &b) : A(A), b(b), s(b.size()) {
    assert(A.cols() == A.rows() && "Square");
    assert(A.cols() == b.size() && "Incompatible"); }
  std::vector<VectorXd> solve(const Function &f, const
   Jacobian &Jf, double T, const VectorXd &y0, unsigned
   int N) const {
   double h=T/N; std::vector<VectorXd> res;
   res.reserve(N+1); res.push_back(y0);
   VectorXd ytemp1 = y0; VectorXd ytemp2 = y0;
   VectorXd *yold = &ytemp1; VectorXd *ynew= &ytemp2;
   for(int k= 0; k<N; k++) {
     step(f, Jf, h, *yold, *ynew);
     res.push_back(*ynew); std::swap(yold, ynew);
  } return res;}
private:
  void step(const Function &f, const Jacobian &Jf,
      double h, const VectorXd &y0, VectorXd &y1) const{
    int d=y0.size(); auto eye= MatrixXd::Identity(d,d);
    auto F=[&y0, h, d, this, &f, &eye] (VectorXd gv){
    VectorXd Fv=gv;
    for(int j=0; j<s; j++)
      Fv-=h*kron(A.col(j), eye)*f(y0,gv.segment(j*d,d));
    return Fv;};
    auto JF=[&y0, h, d, &Jf, this, &eye] (VectorXd gv){
      MatrixXd DF(s*d, s*d);
      for(int j=0; j<s; j++)
          DF.block(0, j*d, s*d, d)= kron(A.col(j), eye)
          * Jf(y0+gv.segment(j*d,d));
      DF = MatrixXd::Identity(s*d, s*d) - h*DF;
      return DF;};
    VectorXd gv = VectorXd::Zero(s*d);
    dampnewton(F, JF, gv); MatrixXd K(d,s);
    for(int j=0 j<s; j++)
      K.col(j) = f(y0+gv.segment(j*d,d));
    y1=y0+h*K*b;
  }const MatrixXd A; const VectorXd b; unsigned int s;};
```

11.2 Linear Implicit Mid-Point 652

```
std::vector<VectorXd> lin_mid(Function &&f, Jacobian &&Jf,
        double T, const VectorXd &y0, unsigned int N) {
    std::vector<VectorXd> res(N+1);
    double h = T/N; int d = y0.size(); res.push_back(y0);
    VectorXd ytemp1 = y0; VectorXd ytemp2 = y0;
    VectorXd *yo = &ytemp1; VectorXd *ynew = &ytemp2;
    MatrixXd eye = MatrixXd::Identity(3,3);
    for(int k= 0; k<N; k++) {
        *ynew = *yo + h*(eye - h/2*Jf(*yo)).lu().solve(f(*yo));
        res.push_back(*ynew); std::swap(yo, ynew);
    } return res;}
```

Implicit mid-point method

```
std::vector<VectorXd> imp_mid(Function &&f, Jacobian &&Jf,
        double T, const VectorXd &y0, unsigned int N){
    unsigned int s=1; MatrixXd A(s,s); VectorXd b(s);
    A << 1./2.; b << 1.; implicit_RKIntegrator RK(A,b);
    return RK.solve(f, Jf, T, y0, N); }
```

11.1 Cross product ODE 638

```
void tab_crossprod(void) {
    double T= 10.; int N=128; double c=1.; Vector3d y0;
    y0<<1.,1.,1.; Vector3d a; a<<1.,0.,0.;
    auto f=[&a,&c] (const Vector3d &y) -> Vector3d {
        return a.cross(y)+ c*y.cross(a.cross(y)); };
    auto Jf=[&a,&c](const Vector3d) {
        Matrix3d temp << -c*(a(1)*y(1) + a(2)*y(2)); //More
        return temp; };
    std::vector<VectorXd> res_imp = imp_mid(f,Jf,T,y0,N);
    for(int i=0; i<N+1; i++) //Same for lin_mid
        cout<<setw(10)<<T*i/N<<setw(15)<<res_imp[i].norm();
```

12.4 Rosenbrock 717

```
std::vector<StateType> solveRosenbrock(const Func &f,
        const DFunc & df, const StateType &y0, int N, double T){
    cd h=T/N; cd a=1./(std::sqrt(2)+2.); //cd=const double
    std::vector<StateType> res (N+1);
    res.at(0) = y0; StateType k1, k2; Matrix2d J, W;
    for(int i=1; i<=N; i++){
        StateType &yprev = res.at(i-1);
        J=df(yprev); W= Matrix2d::Identity() - a*h*J;
        auto W_lu = W.partialPivLu();
        k1 = W_lu.solve(f(yprev));
        k2 = W_lu.solve(f(yprev+0.5*h*k1)-a*h*J*k1);
        res.at(i) = yprev + h*k2; } return res; }

double cvgRosenbrock(void) {
    cd T=10.; const ArrayXd K= ArrayXd::LinSpaced(7,4,10);
    Vector2d y0<<1,1; cd C=1.; Matrix2d R<<0,-1,1,0;
    auto f=[&R, &C] (const Vector2d &y) {
        return R*y + C*(1.-y.squaredNorm())*y; };
    auto df=[&C] (const Vector2d &y) {
        double x= 1 - y.squaredNorm();
        Matrix2d J<< C*x - 2*C*y(0)*y(0), -1-2*C*y(1)*y(0),
            1-2*C*y(1)*y(0), C*x-2*C*y(1)*y(1); return J; };
    const int N_ref = 10*std::pow(2,12);
    auto solref = solveRosenbrock(f, df, y0, N_ref, T);
    ArrayXd Error(K.size());
    for(int i=0; i<K.size(); i++){
        int N= T*std::pow(2,K[i]);
        auto sol = solveRosenbrock(f, df, y0, N, T);
        double maxerr =0;
        for(int j=0; j<sol.size(); j++)
            maxerr = std::max(maxerr, (sol.at(j)
                - solref.at(j*N_ref)/(N)).norm());
        Error[i] = maxerr;
    }
    return -polyfit(std::log(2)*K, Error.log(), 1)(0); }
```