Summary for Numerical Methods for Computational Science and Engineering

Contents

Computing with Matrices and Vectors

Machine Arithmetic

As machines can't compute in real numbers but compute with machine numbers, there are some constraints. Over- and underflow exist, the amount of numbers is finite. Thus, instead of checking for equality, we check if the difference of two numbers is smaller than some maximal relative error.

Cancellation

Cancellation occurs when Substrachting we attempt to:

- Substract numbers of the same size.
- Divide by a small number close to zero.

To avoid cancellation, we simply avoid such kinds of substractions and divisions by recastion expressions or use tricks like taylor approximations.

Increasing Speed of Algorithms

For the following algorithms and formulas described, take into consideration that in many cases, the execution speed of a program can be increased significantly by simply precomputing things that would be computed multiple times, for example in a `for`-loop.

Direct Methods for Square Linear Systems of Equations

Gaussian Elimination

A is *invertible* or *regular* if and only if there exists an unique solution x for $Ax = b$, that is $x = A^{-1}b$.

In Eigen, we use the following code to solve $Ax = b$, and remember to exploit structure by using the appropriate function:

```
MatrixXd A;
VectorXd b, x;
...
// A has no special form
x = A.lu().solve(b);
// A is lower triangular
x = A.triangularView<Eigen::Lower>().solve(b);
// A is upper triangular
x = A.triangularView<Eigen::Upper>().solve(b);
// A is hermitian and positive definite
x = A.selfadjointView<Eigen::Upper>().llt().solve(b);
// A is hermition and positive or negative definite
x = A.selfadjointView<Eigen::Upper>().ldlt().solve(b);
// Solving based on householder transformation
```

```
x = A.HouseholderQR<MatrixXd>().solve(b);
// Solving based on singular value decomposition
x = A.jacobiSvd<MatrixXd>().solve(b);
```

The generic asymptotic complexity is $O(n^3)$, but triangular linear systems can be solved in $O(n^2)$.

Solving $Ax = b$ by computing A^{-1} and calculating $x = A^{-1}b$ is unstable. Instead we use gaussian elimination, which is stable and not affected by roundoff in practice.

Low-Rank Modifications

We assume a generic rank-one midification is applied to A:

$$\tilde{A} = A + uv^H$$

By using *block elimination*, we get:

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

From this follows that:

$$\begin{gathered} A\tilde{x} + u\zeta = b \\ \tilde{x} = A^{-1}(b - u\zeta) \\ \dots \\ (1 + v^H A^{-1} u)\zeta = v^H A^{-1} b \\ \dots \\ A\tilde{x} = b - \frac{uv^H A^{-1}}{1 + v^H A^{-1} u} b \end{gathered}$$

Sparse Matrices

A matrix A is called *sparse*, if the number of non-zero entries $nnz(A)$ is much smaller than the size of A.

Storage Formats

Sparse matrices should be solved with `x = A.SparseLU<SparseMatrixType>().solve(b)` to further reduce the asymptotic complexity.

Also note that the product of sparse matrices isn't necessarily sparse.

Triplet/Coordinate List (COO) Format

Stores triplets with row indices, column indices and the associated values.

Compressed Row-Storage (CRS) Format

Stores the values, the corresponding column indices and row pointers that indicate the start of a new row.

```
SparseMatrix<double, RowMajor> A(rows, cols);
```

Compressed Column-Storage (CCS) Format

Same as CRS, but with column pointers and row indices instead.

```
SparseMatrix<double, ColMajor> A(rows, cols);
```

Direct Methods Linear Least Square Problems

Example: Linear Parameter Estimation

Assume we take m measurements and find the pairs (x_i, y_i). We can express these measurements in the following overdetermined linear system:

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

Solution Concepts

For given A, b, the vector x is a least squares solution of $Ax = b$, if:

$$x \in argmin_y \|Ay - b\|_2^2$$

For a least squares solution x, the vector Ax is the unique orthogonal projection of b onto the image of $Image(A)$. From this follows that:

$$b - Ax \perp Image(A)$$
$$A^T(b - Ax) = 0$$
$$A^T Ax = A^T b$$

If $m \geq n$ and $Kernel(A) = 0$, then the linear system of equations $Ax = b$ has a unique least squares solution $x = (A^T A)^{-1} A^T b$.

We want to solve the *normal equation* $A^T Ax = A^T b$. To do this, we use the following procedure:

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

```
MatrixXd C = A.transpose() * A;
VectorXd c = A.transpose() * b;
VectorXd x = C.llt().solve(c);
```

The asymptotic complexity is $O(n^2 m + n^3)$.

Moore-Penrose Pseudoinverse

As there are many potential least square solutions, we define the *generalized solution* x^+ of a linear system of equations as:

$$x^+ = argmin\{\|x\|_2, x \in lsq(A, b)\}$$

The generalized solution x^+ of the linear systems of equations $Ax = b$ is given by:

$$x^+ = V(V^T A^T AV)^{-1}(V^T A^T b)$$

Where the matrix $V(V^T A^T AV)^{-1}V^T$ is called the *Moore-Penrose pseudoinverse* of A.

Orthogononal Transformation Methods

Transform $Ax = b$ to $\tilde{A}\tilde{x} = \tilde{b}$ such that $lsq(A, b) = lsq(\tilde{A}, \tilde{b})$ and $\tilde{A}\tilde{x} = \tilde{B}$ is easy to solve.

OR-Decomposition

Householder

Data Interpolation and Data Fitting in 1D

Given some data points (t_i, y_i), $i = 0, ..., n$, find an interpolant function f satisfying $f(t_i) = y_i$ and belonging to a set V of specific functions.

Note that we can use the *Horner scheme* to evaluate polynomials efficiently for the following polynomial evalutations:

$$\alpha_k t^k + \alpha_{k-1}t^{k-1} + ... + \alpha_0 = t(...t(t(\alpha_n t + \alpha_{n-1}) + \alpha_{n-2}) + ... + \alpha_1) + \alpha_0$$

Global Polynomial Interpolation

The set of functions is $V = \mathcal{P}_k = \{f \mid f(t) = \alpha_k t^k + ... + \alpha_0 t^0, \alpha_j \in \mathbb{R}\}$.

Polynomials are usefull because ...

- ... they span a finite-dimensional vector space.
- ... they are smooth.
- ... they are easy to evaluate, differentiate and integrate.

Notice that $dim\mathcal{P}_k = k + 1$. For n data points, we construct a $n + 1$-dimensional Polynomial from $\mathcal{P}_n$. The polynomial will be unique.

We introduce the Lagrange polynomials as a helpful tool:

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

Note that:

$$L_i(t_j) = \begin{cases} 1 & , j = i \\ 0 & , \text{else} \end{cases}$$

The polynomial is built by summing up the parts:

$$p(x) = \sum_{j=0}^{n} y_j L_j(x)$$

We can use the *Aitken-Neville* algorithm for increased efficiency, where l is the first and k is the last point included in the interpolation, such that $p_{0,n} = p$.

$$p_{k,k}(x) = y_k$$

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

Another advantage is the update-friendliness of this algorithm, because there is no need to recompute the whole formula if another point is added.

Because adding another point affects all Lagrange polynomials, we introduce the new *Newton basis* with the purpose of being update-friendlier.

$$N_0(t) = 1, N_1(t) = (t - t_0), ..., N_n(t) = \prod_{i=0}^{n-1}(t - t_i)$$

$$p(t) = \sum_{i=0}^{i \leq n} a_i N_i$$

To find a_i, we have to solve the following equations:

$$p(t_i) = y_i$$

Which leads us a triangular linear system to solve:

$$\begin{bmatrix} 1 & 0 & \dots & 0 \\ 1 & (t_1 - t_0) & & \vdots \\ \vdots & \vdots & & 0 \\ 1 & (t_n - t_0) & \dots & \prod_{i=0}^{n-1}(t_n - t_i) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

We observe a similar recursion like before, and we can define analogous to $p_{l,m}$:

$$a_{l,m} = \frac{a_{l+1,m} - a_{l,m-1}}{t_m - t_l}$$

Instead of solving the triangular linear system, we use the *divided differences* algoritm.

$$y[t_i] = y_i$$

$$y[t_i, ..., t_{i+k}] = \frac{y[t_{i+1}, ..., t_{i+k}] - y[t_i, ..., t_{i+k-1}]}{t_{i+k} - t_i}$$

$$a_i = y[t_0, ..., t_i]$$

Shape Preserving Interpolation

Data (t_i, y_i) is called *monotonic* if $y_i \geq y_{i-1}$ or $y_i \leq y_{i-1}$ for $i = 1, ..., n$.

Data (t_i, y_i) is called *convex* (*concave*) if $s_j \leq (\geq) s_j + 1$ for $i = 1, ..., n-1$, where $s_j = \frac{y_j - y_{j-1}}{t_j - t_{j-1}}$.

Interpolation is *shape preserving* if the shape features of the interpolant are the same as the shape features for the data.

If the shape is preserved (positivity/negativity, increasing/decreasing, convexity/concavity) for every peace of the interpolation, we say that the interpolant is *locally shape preserving*.

Cubic Hermite Interpolation

Given some data points $(t_j, y_j), j = 0, ..., n$ and slopes $c_j \in \mathbb{R}$, find a piecewise cubic hermite interpolant s such that $s_{[t_{i-1}, t_i]} \in \mathcal{P}_3, i = 1, ..., n$ and further $s(t_i) = y_i, s'(t_i) = c_i, i = 0, ..., n$.

s is representated locally by $s(t) = y_{i-1}H_1(t) + y_i H_2(t) + c_{i-1}H_3(t) + c_i H_4(t)$, where $t \in [t_{i-1}, t_i]$ and the basis polynimials defined as follows:

$$h_i = t_i - t_{i-1}$$

TODO

The next task is to find c_i such that monotonicity is preserved.

$$d_j = \frac{y_j - y_{j_1}}{t_j - t_{j-1}}, j = 1, ..., n$$

$$c_i = \begin{cases} 0 & , sgn(d_i) \neq sgn(d_{i+1}) \\ \text{some average of } d_i, d_{i+1} & , \text{otherwise} \end{cases}$$

Splines

The vector space of *spline functions* of degree d and order $d+1$ is defined by $\mathcal{S}_{d,\mathcal{M}} = \{s \in C^{d-1}(I) : s_j = s_{[t_{j-1}, t_j]} \in \mathcal{P}_d, j = 1, ..., n\}$ on an interval $I = [a, b]$ and a *mesh* $\mathcal{M}$.

Linear Interpolation

Linear interpolation is just a spline interpolation $\mathcal{S}_{1,\mathcal{M}}$, where the interpolation nodes are $\mathcal{M}$.

Cubic Spline Interpolation

Interpolation into $\mathcal{S}_{3,\mathcal{M}}$. The natural cubic spline interpolant minimizes the elastic curvature energy among all interpolating functions, but it is locally weak.

We reuse the local representation of a cubic spline s through a cubic Hermite cardinal basis polynomials from above. We just need to finde the slopes $s(t_j)$ in the knots of the mesh $\mathcal{M}$.

Because $s \in C^2$, $s''_{[t_{j-1}, t_j]}(t_j) = s''_{[t_j, t_{j+1}]}(t_j)$.

Complete Cubic Spline Interpolation

Natural Cubic Spline Interpolation

Periodic Cubic Spline Interpolation

Least Squares Data Fitting

Given some data points $(t_i, y_i), i = 1, ..., m$, we need to find a continuous function f in some set $S \subset C^0$ satisfying $f \in argmin_{g \in S} \sum_{i=1}^{m} \|g(t_i) - y_i\|_2^2$. The function f is called the *best least squares fit* for the data in S.

Approximation of Functions in 1D

Given a function f, often in *procedural form*, e.g. `double f(double)`, find a "simple" (for example polynomials) function $\tilde{f}$ such that the approximation error $f - \tilde{f}$ is small.

$f \rightarrow_{sampling} (t_i, y_i) \rightarrow_{interpolation} \tilde{f}$

We now have the freedom to select the nodes t_i.

Approximation by Global Polynomials

Taylor polynomials can be used to approximate a function $f(t) \approx \sum_{l=0}^{k} \frac{f^{(l)}(t_0)}{l!}(t - t_0)^l, f \in C^k$.

We can also use Lagrange interpolation as an approximation method, which gives us the error bound $O(n^{-r})$ on the Interval $[1, -1]$ for polynomial approximation, we can use this to our advantage and do a *pullback* from the interval $[a, b]$ to $[-1, 1]$ by using the function $\Phi(\hat{t}) = a + \frac{1}{2}(\hat{t} + 1)(b - a), -1 \leq \hat{t} \leq 1$.

Types of Asymptotic Convergence

We distinguish the following types of asymptotic behavior for a bound $T(n)$:

Algebraic Convergence with rate $p > 0$	$\forall n \in \mathbb{N},\ \exists p > 0 : T(n) \leq n^{-p}$	Data points on a line on a log-log plot
Exponential Convergence	$\forall n \in \mathbb{N},\ \exists 0 < q < 1 : T(n) \leq q^n$	Data points on a line on a lin-log plot

Some examples:

	$f \in C^\infty$, $f(t) = t^{\frac{3}{2}}, \Omega = [1, 2]$, $f(t) = 2t^{\frac{3}{2}}, \Omega = [2, 4]$	$f \in C^r$, $f(t) = t^{\frac{5}{2}}, \Omega = [0, 1]$, $f(t) = 2t^{\frac{5}{2}}, \Omega = [0, 2]$, $f(t) = \lvert t\rvert, \Omega = [-1, 1]$, $f(t) = \lvert t\rvert, \Omega = [-2, 2]$
Chebychev Interpolation	Exponential	Algebraic
Gauss-Legendre Quadrature	Exponential	Algebraic
Composite Trapezoidal Quadrature	Algebraic	Algebraic
Composite Simpson Quadrature	Algebraic	Algebraic
Composite 2-point Gauss Quadrature	Algebraic	Algebraic

Chebychev Interpolation

For equidistant linear interpolation points we observe a blowup of the error at the endpoints due to oscillation. With Chebychev interpolation, our goal is to avoid this by choosing our nodes $t_0, ..., t_n$ such that this blowup is minimal. To reach this goal, we introduce the n-th Chebychev polynomial:

$$T_n(t) := cos(n\ arccos(t))$$

We choose our sampling data points at the n positions where the n-th Chebychev polynomial intersects the x-axis, which gives us the points:

$$t_k := a + \frac{1}{2}(b-a)(cos(\frac{2k+1}{2(n+1)}\pi) + 1),\ k = 0, ..., n$$

Approximation by Piecewise Polynomials

The idea is to use piecewise polynomials with respect to a mesh $\mathcal{M} := \{a = x_0 < x_1 < ... < x_{m-1} < x_m = b\}$, wich gives us the advantage of locality.

Piecewise Lagrange Interpolation

We use Lagrange Interpolation on each mesh cell.

Numerical Quadrature

The task is to approximate an Integral $\int_a^b f(t)dt$.

Quadrature Formulas

We approximate the Integral by a weighted sum of point values of the integrand:

$$\int_a^b f(t)dx \approx \sum_{j=1}^{n} w_j f(c_j)$$

Because we have the weights and evaluation points only on the interval $[-1, 1]$, we again use a transformation to the target interval $[a, b]$:

$$\Phi(\hat{t}) = a + \frac{1}{2}(\hat{t}+1)(b-a), -1 \leq \hat{t} \leq 1$$

Given some weights and nodes $(\hat{w}_j, \hat{c}_j)$ on the interval $[-1, 1]$. Using Φ as a transformation function we get:

$$\int_a^b f(t)dx \approx \frac{1}{2}(b-a)\sum_{j=1}^{n} \hat{w}_j \hat{f}(\hat{c}_j)$$

$$c_j = \frac{1}{2}(1-\hat{c}_j)a + \frac{1}{2}(1+\hat{c}_j)b$$

$$w_j = \frac{1}{2}(b-a)\hat{w}_j$$

Polynomial Quadrature Formulas

The idea is to replace the integrand f with a Lagrange interpolant of f:

$$\int_a^b f(x)dt \approx \int_a^b p_{n-1}(t)dt = \sum_{i=0}^{n-1} f(t_i) \int_a^b L_i(t)dt$$

This leads us to:

$$c_i = t_{i-1}$$

$$w_i = \int_a^b L_{i-1}(t)dt$$

Midpoint Rule

If we use $n = 1$ in the formula above, we get:

$$\int_a^b f(t)dt \approx (b-a)f(\frac{1}{2}(a+b))$$

Trapezoidal Rule

For $n = 2$, we get:

$$\int_a^b f(t)dt \approx \frac{b-a}{2}(f(a)+f(b))$$

For $n > 2$, we call these rules the *Newton-Cotes formulas* on an argument $m = n - 1$.

Gauss Quadrature

The *order* of a quadrature rule $\mathcal{Q}_n$ is defined as the maximal degree plus one of polynomials for which the quadrature rule is guaranteed to be exact:legendre

$$order(\mathcal{Q}_n) := max\{m \in \mathbb{N} : \mathcal{Q}_n(p) = \int_a^b p(t) \forall p \in \mathcal{P}_m\} + 1$$

A n-point polynomial quadrature rule has order n. The maximal order of an n-point quadrature rule is $2n$. In fact, for all non-zero polynomials $\bar{P}_n$ that satisfy

- $\bar{P}_n \in \mathcal{P}_n$
- $\int_{-1}^{1} q(t)$

TODO

The n-point Quadrature formulas whose nodes, the Gauss points, are given by the zeros of the n-th Legendre polynomial, and whose weights are chosen according to the rule seen above are called *Gauss-Legendre quadrature formulas.* The legendre polynomial $P_n \in \mathcal{P}_n$

TODO

Note that the gauss points are equal to the zeros of the Legendre polynomial.

Composite Quadrature

The idea of composite quadrature is to apply the quadrature formulas from above locally on some mesh intervals and then sum up the result.

Adaptive Quadrature

For and adaptive quadrature approximate, we choose the nodes depending on the integrand f. We distinguish between two types of adaptive quadrature:

1. *A priori* adaptive quadrature, where the nodes are fixed before the evaluation of the quadrature formula. For example, we choose the mesh such that the cell errors are equally distributed.
2. *A posteriori* adaptive quatrature, where the node positions can be chosen or improved based on information recieved during the computation of the formula. It terminates as soon as sufficient accuracy has been reached. An example for is if we in each loop iteration add a node inside the mesh intervals with the largest error contributions.