Introduction to Computational Science - Zusammenfassung

Dino Wernli, Fabian Hahn

15. März 2009

1 Nonlinear Equations

1.1 Bisection

Let $f(x)$ be a continuous function and assume we know an Interval $[a, b]$, for which $f(a) \cdot f(b) < 0$ holds.
Bisection algorithm to compute the zero $s \in [a, b]$:

1. compute the function value $f(m)$ in the middle of $[a, b]$
2. decide whether to proceed in the left or right half
3. repeat steps 1 and 2 until s is narrowed down to an arbitrary precision

The algorithm can be made machine independant by the termination criterion $(a < m) \wedge (m < b)$. The computation stops as soon as there are no more machine numbers in the interval. The error in the k-th step is:

$$|m_k - s| \leq \frac{1}{2^k}(b - a) \to 0, \quad k \to \infty$$

The number k of steps for an error smaller than *tol* is:

$$k > \ln\left(\frac{b-a}{tol}\right) / \ln 2$$

1.2 Fixed Point Iteration

General Iteration Concept

Given 2 defined, continuous functions $f(x)$ and $g(x)$, the task is to find s, so that:

$$f(s) = g(s)$$

We start with an initial value x_0 and iterate according to the following pattern:

$$f(x_1) = g(x_0)$$
$$x_1 = f^{-1}(g(x_0))$$

If we can produce the special case $f(x) = x$, the iteration becomes much simpler, because $f^{-1}(x) = x$:

$$x_1 = f^{-1}(g(x_0)) = g(x_0)$$

Algorithm

Let $f(x)$ be a defined, continous function on $[a, b]$. $f(s) = 0$ must hold for the zero $s \in [a, b]$. Through algebraic transformation, we can express the same condition as:

$$x = F(x)$$

Finding a zero s is now equivalent to finding a fixed point, so that $s = F(s)$ holds. The equation suggests the following fixed point iteration:

$$x_0 \text{ initial guess}, \quad x_{k+1} = F(x_k)$$

Depending on the algebraic transformations and our initial guess x_0, the algorithm might or might not produce a converging sequence $x_k \to s$. If it does, we found a zero of $f(x)$.

Banach's Fixed Point Theorem

A Banach Space $\mathcal{B}$ is a complete, normed vector space over some number field $\mathcal{K}$. Complete means, that every Cauchy sequence converges in $\mathcal{B}$.
Let $A \subset \mathcal{B}$ be a closed subset and F be a mapping $A \to A$ where $A = [a, b]$. F is called Lipschitz continuous on A, if:

$$\exists L < \infty \forall x, y \in A : ||F(x) - F(y)|| \leq L||x - y||$$

F is called a contraction, if L can be chosen with $L < 1$.
The Theorem of Banach states, that for a contraction F on $A \subset \mathcal{B}$ the following must hold:

1. F has a unique fixpoint s, the only solution of $x = F(x)$
2. The sequence $x_{k+1} = F(x_k)$ converges to s for any $x_0 \in A$
3. $||s - x_k|| \leq \frac{L^{k-l}}{1-L}||x_{l+1} - x_l||$, for $0 \leq l \leq k$

A first consequence is that setting $l = 0$ gives us an a priori estimate for the error after k steps:

$$||s - x_k|| \leq \frac{L^k}{1 - L}||x_1 - x_0||$$

Using this estimate, we can make a prediction depending on L and ε for how many steps we will need to obtain $||s - x_k|| < \varepsilon$:

$$k > \frac{\ln \frac{\varepsilon(1-L)}{||x_1 - x_0||}}{\ln L}$$

By setting $l = k - 1$ we can also get an a posteriori estimate of the error:

$$||s - x_k|| \leq \frac{L}{1 - L}||x_k - x_{k-1}||$$

If $||x_k - x_{k-1}|| < \varepsilon$ then:

$$||s - x_k|| \leq \frac{L}{1 - L}\varepsilon$$

Note that the bound on the right side can get very large, if L goes towards 1. The intuitive reason for this, is that you are then trying to find the intersection between two functions, which almost have the same incline. This is a typical reason for precision loss.

For practical purposes we need L to check if a given fixed point iteration $x = F(x)$ will converge. If $|F'(s)| < 1$ then there exists a starting value x_0 for which the iteration will converge. However, since we usually don't know s in advance, we perform a worst-case analysis and use the following formula for F:

$$L = \max_{a \leq \xi \leq b} |F'(\xi)|$$

1.3 Convergence Rates

From the definition of fixed point iterations, we know $s = F(s)$ and $x_{k+1} = F(x_k)$. Subtraction and Taylor expansion of $F(x_k)$ around s gives us the following formula for the error:

$$\begin{aligned} e_{k+1} = x_{k+1} - s &= -s + x_{k+1} = -F(s) + F(x_k) \\ &= -F(s) + F(s) + F'(s)\underbrace{(x_k - s)}_{=e_k} + \frac{F''(s)}{2!}e_k^2 \end{aligned}$$

Now we divide e_{k+1} by e_k and take the limit:

$$\begin{aligned} \lim_{k\to\infty} \frac{e_{k+1}}{e_k} = F'(s), \quad F'(s) \neq 0 \\ \Rightarrow e_{k+1} \sim F'(s) \cdot e_k \end{aligned}$$

Asymptotically, the error is reduced by a constant factor at each step. This is called linear convergence. More generally we define conversions as:

- linear: $F'(s) \neq 0 \ \wedge \ |F'(s)| < 1$
- quadractic: $F'(s) = 0 \ \wedge \ F''(s) \neq 0$
- of order m: $F'(s) = F''(s) = \cdots = F^{(m-1)}(s) = 0 \wedge F^{(m)}(s) \neq 0$

1.4 Construction of One Point Iterations

Geometric Construction

The idea is to approximate f by a function h in the neighbourhood of the zero. You then solve $h(x) = 0$ analytically and hope that the zero of h is also a zero of f.
The only question left, is how to chose $h(x)$. Newton's idea was to take a linear function with: $f(x) = h(x), \ f'(x) = h'(x)$. This means, that $h(x)$ is the first Taylor Approximation of f:

$$h(x) = f(x_k) + f'(x_k)(x - x_k)$$

Solving $h(x) = 0$ analytically, we obtain the iteration:

$$h(x) = 0 \ \leftrightarrow \ x_{k+1} = F(x_k) = x_k - \frac{f(x_k)}{f'(x_k)}$$

Halley proposed a different approach for the approximation:

$$h(x) = \frac{a}{x+b} + c$$

The values a, b, c are to be determined, so that: $f(x) = h(x), \ f'(x) = h'(x), \ f''(x) = h''(x)$ After solving the equation system for a, b and c, you obtain the iteration:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \cdot \frac{1}{1 - \frac{1}{2}\frac{f(x_k)f''(x_k)}{f'(x_k)^2}}$$

Generally, you can reduce the number of steps in the iteration by making your approximation function h better, but this implies a longer computation for each step, thus the resulting tradeoff.

Algebraic Construction

To solve $f(x) = 0$, a possibility for iteration is always: $x = F(x) = x + f(x)$. However, we can not expect convergence for arbitrary f, since the condition is: $|F'(s)| = |1 + f'(s)| < 1$.

An idea for a solution is to premultiply $f(x) = 0$ with some function $h(x)$:

$$x = F(x) = x + \underbrace{f(x)}_{=0} h(x)$$

Our goal is to chose h in such a way, that $|F'(x)| < 1$. For quadratic convergence, we need:

$$F'(s) = 1 + h'(s)f(s) + h(s)f'(s) = 1 + h(s)f'(s) = 0$$

If $f'(s) \neq 0$ holds, we can solve the equation and obtain:

$$h(s) = -\frac{1}{f'(s)}$$

This leads us to Newton's Iteration, thus proving that it converges quadratically:

$$x = F(x) = x + f(x)h(x) = x - \frac{f(x)}{f'(x)}$$

Every fixed point iteration $x = F(x)$ can be regarded as the Newton Iteration for some function g. Solving the differential equation $x - \frac{g(x)}{g'(x)} = F(x)$, we obtain:

$$|g(x)| = exp\left(\int \frac{1}{x - F(x)} dx\right)$$

In similar way, we can conclude, that Halley's Iteration for $f(x) = 0$ is Newton's Iteration for:

$$\frac{f(x)}{\sqrt{f'(x)}} = 0$$

Halley-like Iterations

Halley-Iteration is a form of "more precise" Newton-Iteration. A general form for such an iteration looks like:

$$x = F(x) = x - \frac{f(x)}{f'(x)} H(x)$$

The question is how to chose $H(x)$, so that the iteration yields a quadratically or even cubically converging sequence. For the analysis, we first introduce the abbreviation:

$$u(x) = \frac{f(x)}{f'(x)}$$

We then obtain for the derivatives of $F(x)$:

$$\begin{aligned} F &= x - uH \\ F' &= 1 - u'H - uH' \\ F'' &= -u''H - 2u'H' - uH'' \end{aligned}$$

and

$$\begin{aligned} u' &= 1 - \frac{ff''}{f'^2} \\ u'' &= -\frac{f''}{f'} + 2\frac{ff''^2}{f'^3} - \frac{ff'''}{f'^2} \end{aligned}$$

Since $f(s) = 0$, the following must hold for the fixed point:

$$u(s) = 0; \quad u'(s) = 1; \quad u''(s) = -\frac{f''(s)}{f'(s)}$$

so we conclude for the derivatives of F:

$$F'(s) = 1 - H(s)$$
$$F''(s) = \frac{f''(s)}{f'(s)}H(s) - 2H'(s)$$

We obtain the condition for quadratic convergence:

$$F'(s) = 0 \;\rightarrow\; H(s) = 1$$

and in addition for cubic convergence:

$$F''(s) = 0 \;\rightarrow\; H'(s) = \frac{1}{2}\frac{f''(s)}{f'(s)}$$

The problem is, that we don't know the fixed point in advance, so this is not a constructive method for iterations. We need to chose $H(x)$ as a function G of what we have:

$$H(x) = G(f(x), f'(x), \dots)$$

For example, $H(x) = 1 + f(x) \;\rightarrow\; H(s) = 1$ would converge quadratically. If we chose $H(x) = G(t(x))$ with:

$$t(x) = \frac{f(x)f''(x)}{f'(x)^2}$$

then we can conclude:

$$\begin{aligned} t(x) = 1 - u'(x) \;\rightarrow\; t(s) &= 1 - 1 = 0 \\ \rightarrow H(s) &= G(t(s)) = G(0) \\ H'(x) = G'(t(x))t'(x) &= -G'(t(x))u''(x) \\ \rightarrow H'(s) &= G'(0)\frac{f''(s)}{f'(s)} \end{aligned}$$

Now we apply the conditions for cubic convergence. We can conclude that, for any simple zero s of a function f, for any function G with $G(0) = 1, G'(0) = \frac{1}{2}, |G''(0)| < \infty$, the following iteration will produce a cubic convergence:

$$x = F(x) = x - \frac{f(x)}{f'(x)}G\left(\frac{f(x)f''(x)}{f'(x)^2}\right)$$

Some examples of well known cubic iterations for simple zeros:

$$\begin{aligned} \text{Halley:} \quad & G(t) = \frac{1}{1 - \frac{1}{2}t} \\ \text{Euler:} \quad & G(t) = \frac{2}{1 + \sqrt{1 - 2t}} \\ \text{Inverse Interpolation:} \quad & G(t) = 1 + \frac{1}{2}t \end{aligned}$$

1.5 Multiple Zeros

The quadratic convergence in Newton's Iteration was based on the assumption of a single zero. We now assume that $f(x)$ has a zero of multiplicity n at $x = s$. Therefore:

$$f(x) = (x - s)^n g(x), \quad g(s) \neq 0$$

If we inspect the limit of the derivative $F'(x)$ towards s, we can see that the quadratic convergence is lost:

$$\lim_{x \to s} F'(x) = \frac{n-1}{n} \neq 0$$

A possible remedy for the problem would be to take the iteration:

$$x_{k+1} = x_k - n\frac{f(x_k)}{f'(x_k)}$$

But since one doesn't know the multiplicity in advance, Schröder proposed (1870) to use the estimate:

$$n = \frac{f'^2}{f'^2 - ff''}$$

The resulting iteration would then be:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}\frac{1}{1 - \frac{f(x_k)f''(x_k)}{f'(x_k)^2}}$$

By looking at Schröder's Iteration as a form of Newton's Iteration and solving the appropriate differential equation, we can see that Schröder's Iteration for $f(x)$ is equivalent to Newton's Iteration for $g(x) = \frac{f(x)}{f'(x)}$. The quadratic convergence is restored, since g does not have multiple zeros.
This becomes clear by looking at the example $f(x) = x^4$. We get $g(x) = \frac{1}{4}x$, which clearly does not have multiple zeros.

1.6 Multi-Point Iteration

As opposed to the previous One-Point-Iterations, we want to use more information about the last iteration steps for the new point:

$$x_{k+1} = F(x_k, x_{k-1}, \dots)$$

Like in Newton's Iteration, the technique is to interpolate the function f by using the given points. You then have an approximation h, of which you can compute the zero. You then hope that the zero you computed is an approximation of the actual zero, and start again with your new point.

Secant Method

This is the simplest case. You lay a secant through two points $(x_0, f(x_0)), (x_1, f(x_1))$ and take its zero as the next point. With the abbreviations $f(x_0) = f_0$ and $f(x_1) = f_1$ we get the iteration:

$$x_{k+1} = x_k - f_k\frac{x_k - x_{k-1}}{f_k - f_{k-1}}$$

This iteration method converges superlinearly (which is better than linear):

$$e_{k+1} \sim \frac{f''(s)}{2f'(s)}e_k^{1.618}$$

Regula Falsi

Like in bisection, you start with 2 points in which the function values have different signs. After connecting the values, you compute the zero and replace the point with the same sign as the new zero. The advantage is that the zero is always between the 2 points, so convergence is guaranteed.
The convergence is only linear (not superlinear).

Müller's Method

The idea here is to take 3 given points $(x_0, f_0), (x_1, f_1), (x_2, f_2)$ and compute a function

$$h(x) = ax^2 + bx + c$$

which passes through all the points. You compute the zero of h analytically and use the new point for the iteration of h.

1.7 Zeros of Polynomials

The fundamental theorem of algebra states, that a polynomial of degree $n > 0$

$$P_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0, \quad a_n \neq 0$$

always has exactly n zeros $z_1, \ldots, z_n \in \mathbb{C}$. The polynomial can then be factored:

$$P_n(x) = P_{n-1}(x) \cdot (x - z_1) = \cdots = a_n \prod_{i=1}^{n}(x - z_i)$$

where $P_{n-1}(x)$ is a polynomial of degree $n-1$, which has the remaining zeros of $P_n(x)$ as zeros.
Theorem 1.4 states that all the zeros of $P_n(x)$ lie within the disk around the origin of $\mathbb{C}$ with radius:

$$r = 2 \cdot \max_{1 \leq k \leq n} \sqrt[k]{\left|\frac{a_{n-k}}{a_n}\right|}$$

The Galois Theorem states that an explicit formula for the zeros of a polynomial only exists for $n \leq 4$.

Condition of the Zeros

A computation problem is called ill-conditioned when a small deviation from the correct input generates a large offset from the correct output. Computing the zeros of polynomials is found to be ill-conditioned in many cases.
A general description for zero-computation of $P_n(x)$ on a finite precision computer is:

$$h(x) = P_n(x) + \varepsilon g(x) = 0$$

The zero z of this polynomial becomes a function $z(\varepsilon)$ of the disturbance ε. We know that $P_n(s) = P_n(z(0)) = 0$. Since $z(\varepsilon)$ is a function, we can expand it into its Taylor polynomial:

$$z(\varepsilon) = z(0) + \sum_{k=1}^{\infty} p_k(\varepsilon - 0)^k = z + \sum_{k=1}^{\infty} p_k \varepsilon^k$$

For small ε, higher powers of ε go towards 0 very quickly. Hence we will only use p_1 for the analysis, since it is the most important coefficient. Computing p_1 is done as follows:

$$p_1 = -\frac{g(z(0))}{P_n'(z(0))}$$

We also obtain for the error:

$$\delta_{z_r} = z(\varepsilon) - z(0) = p_1\varepsilon + p_2\varepsilon^2 + \ldots$$
$$\delta_{z_r} \approx p_1\varepsilon$$

Multiple roots are always ill-conditioned. If you make minor changes, a n-fold root will become non-existant in the perturbed polynomial and will explode into n single roots.

The Companion Matrix

The companion matrix of the polynomial

$$Q_n(x) = 1 \cdot x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots a_0$$

is defined as:

$$A = \begin{pmatrix} -a_{n-1} & -a_{n-2} & \cdots & -a_1 & -a_0 \\ 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

By developing $\det(A - \lambda I)$ in A's first row, we can see that the eigenvalues of the companion matrix are exactly the roots of the polynomial.

Horner's Scheme

Dividing a polynomial $P_n(x)$ by the factor $(x - s)$, we obtain:

$$\frac{P_n(x)}{x - s} = P_{n-1}(x) + \frac{r}{x - s}$$

where $r = 0$ holds exactly if s is a zero of $P_n(x)$.
The result

$$P_{n-1}(x) = b_{n-1}x^{n-1} + \cdots + b_0$$

of this polynomial can be generalized through the recursion:

$$\begin{aligned} b_{n-1} &= a_n \\ b_{i-1} &= s \cdot b_i + a_i, \quad i = n-1, n-2, \ldots, 1 \\ r &= s \cdot b_0 + a_0 \end{aligned}$$

Multiplying the original equation by $x - s$, we get:

$$P_n(x) = P_{n-1}(x)(x - s) + r$$

now we set $x = s$ and obtain $P_n(s) = r$. This shows that we can use the division algorithm to evaluate the polynomial. We can also use it to calculate the first derivative of $P_n(s)$:

$$\frac{P_n(x)}{x - s} = P_{n-1}(x) + \frac{P_n(s)}{x - s} \Rightarrow P_{n-1}(x) = \frac{P_n(x) - P_n(s)}{x - s}$$

This implies:

$$\lim_{x \to s} \frac{P_n(x) - P_n(s)}{x - s} = P_n'(s) = P_{n-1}(s)$$

careful: this does NOT mean that P_{n-1} is the first derivative of P_n. By factoring we can reduce the number of multiplications from $\frac{n(n+1)}{2}$ to n:

$$P_n(s) = \underbrace{(\underbrace{(\underbrace{a_n}_{b_{n-1}} s + a_{n-1})}_{b_{n-2}} s + a_{n-2})s + \cdots + a_1) s + a_o}_{b_0}$$

Horner's Scheme can also be used to decompose polynomials:

$$\begin{aligned} P_n(x) &= P_n(s) + (x - s)P_{n-1}(x) \\ P_{n-1}(x) &= P_{n-1}(s) + (x - s)P_{n-2}(x) \\ &\vdots \\ P_n(x) &= \sum_{k=0}^{n} P_{n-k}(s) \cdot (x - s)^k \end{aligned}$$

We conclude that the Horner Scheme can be used to express the same polynomial around the starting point s instead of 0:

$$P_n(x) = \sum_{k=0}^{n} a_k x^k = \sum_{k=0}^{n} P_{n-k}(s) \cdot (x - s)^k$$

The coefficients $P_{n-k}(s)$ of the new expression for the polynomial are all computed as the residues r_i of the Horner Scheme.

Number Conversion

To convert a b-ary number of length n into decimal you just have to evaluate the polynomial:

$$a_{n-1}b^{n-1} + a_{n-2}b^{n-2} + \cdots + a_0$$

where, a_i are the digits of the b-ary number.
To convert a decimal number w to a b-ary number, you divide your number by b:

$$\frac{w}{b} = t_0 + r_0, \quad 0 \leq r_0 < b$$

you then divide t_0 by b and get r_1. Repeating this procedure until $t_i = 0$ (let's say k times), you obtain the new number:

$$[w]_{10} = [r_k \ r_{k-1} \ \dots \ r_0]_b$$

Algorithmic Differentiation

Assume we have a polynomial $P(x)$, for which we know an algorithm, which evaluates the polynomial recursively for a given x, like Horner's Scheme. It is easy to extend the algorithm, making it capable of computing the derivative $P'(x)$ for that same x. What we have to do:

- for an initialization a of the polynomial evaluation, we initialize the derivative evaluation to $b = \frac{da}{dx}$
- we replace all the update expressions $p = f(m, n, \dots)$ by their derivations $ps = \frac{d}{dx}f = h(m, ms, n, ns, \dots)$.

Very often, the computation of h is based on the current value of the variable describing the polynomial iteration. In such a case, we must make sure to compute the value of the derivative before overwriting the value of the polynomial.

Newton's Method

Version 1 Applying Newton's iteration, we can compute all the zeros of a polynomial $P(x)$ with the following algorithm:

1. compute a zero s using classic Newton. Start with the initial guess $1 + i$
2. deflate the polynomial by dividing it by $x - s$ and repeat step 1 with s as starting point

Since we want the loop for one Newton Iteration to terminate upon reaching machine precision, we get the exit condition

$$|P_n(x)| \approx \varepsilon \sum_{i=0}^{n} |a_i x^i| \ \Leftrightarrow \ |P_n(x)| \approx \varepsilon \cdot |P_n(x)|$$

Version 2 This method is based on re-developing the polynomial at new points in the complex plane. Generally, the development of $P_n(x)$ around the point x_0 looks like this:

$$j^n_{x_0}(P_n(x)) =$$
$$P_n(x_0) + P'_n(x_0) \cdot (x - x_0) + \cdots + \frac{1}{n!}P^{(n)}_n(x_0) \cdot (x - x_0)^n$$

Looking at x_0 as the current iteration point, we take the same development, but we set $x = x_0 + h$, thus getting a function of the offset h from x_0:

$$P_n(x_0 + h) = P_n(x_0) + P'_n(x_0) \cdot \underbrace{h}_{h=x-x_0} + \dots$$
$$= a_0 + a_1 h + a_2 h^2 + \cdots + a_n h^n$$

The development above shows us that $a_0 = P_n(x_0)$ and that $a_1 = P'_n(x_0)$. This leads us to the new algorithm:

1. initialize x_0 to some guess for a zero
2. develop the polynomial around x_0 as shown above and perform the Newton step $x_0 = x_0 - a_0/a_1$. This will have the effect that a_0 decreases at each iteration.
3. when x_0 finally becomes a root, we can say: $a_0 = 0 \rightarrow P_n(x_0+h) = h(a_1+a_2h+\cdots+a_nh^{n-1})$ and the remaining roots are the zeros of:

$$P_{n-1}(x_0 + h) =$$
$$\frac{h(a_1 + a_2 h + \cdots + a_n h^{n-1})}{h - 0} = a_1 + a_2 h + \cdots + a_n h^{n-1}$$

The disadvantages here are that you have to compute the whole Horner's scheme at every iterationn and that the problem is ill-conditioned, since deflation with an approximate root will deliver inaccurate coefficients.

Version 3 - "Newton-Maehly" Here we assume that we do not have the polynomial in explicit form, but that we are only given an algorithm to compute its evaluation at an arbitrary point. This method also requires all the roots to be real. We proceed as follows:

1. use algorithmic differentiation to compute the differential evaluation in parallel to the polynomial evaluation
2. compute one zero after the other using Newton's iteration
3. proceed with step 1 using a technique called suppression to prevent finding the same zero again

Since deflation only works for explicit polynomials, this algorihm has to use suppression:

$$P_{n-1}(x) = \frac{P_n(x)}{x - x_1}, \quad \text{for } P_n(x_1) = 0$$
$$P'_{n-1}(x) = \frac{P'_n(x)}{x - x_1} - \frac{P_n(x)}{(x - x_1)^2}$$

so the usual Newton correction becomes:

$$-\frac{P_{n-1}(x)}{P'_{n-1}(x)} = -\frac{P_n(x)}{P'_n(x) - \frac{P_n(x)}{x - x_1}}$$

applying the same logic, but assuming we have computed the first k zeros:

$$P_{n-k}(x) = \frac{P_n(x)}{(x - x_1) \dots (x - x_k)}$$

After derivating and finding the Newton correction, we obtain the iteration step for k known zeros:

$$x_{i+1} = x_i - \frac{P_n(x_i)}{P'_n(x_i)} \cdot \frac{1}{1 - \frac{P_n(x_i)}{P'_n(x_i)} \cdot \sum_{l=1}^{k} \frac{1}{x_i - x_l}}$$

While examining the implementation details of the algorithm, Stoer and Bulirsch found that the performance was suboptimal, due to:

$$x_{i+1} = x - \frac{P_n(x_i)}{P'_n(x_i)} \approx x_i - \frac{1}{n}x_i \approx 1 \cdot x_i$$

To fight this inefficiency, they proposed double steps:

1. start with double steps from the right of the first zero: $x := y, y := x - 2\frac{P(x)}{P'(x)}$
2. as soon as $y \geq x$ holds, this means that our new value is to the right of our old value, so we have changed direction. This is when we continue iterating with single steps
3. we continue from step 1, suppressing the new zero

Laguerre's Method

Let $P(x)$ be a polynomial of degree n. The idea is to approximate P near a zero s by the special polynomial $g(x) = a(x-b)(x-c)^{n-1}$. The parameters a, b, c are chosen so that these conditions hold:

$$P(x) = g(x)$$
$$P'(x) = g'(x)$$
$$P''(x) = g''(x)$$

Then we compute the two zeros of $g(x)$ (which can be done by solving a quadratic equation) in the hope that the first one of them is a better approximation of the zero s. This gives us the iteration (for P, P', P'' evaluated at x_i):

$$x_{i+1} = F(x_i) = x_i - \frac{P}{P'} \cdot \frac{n}{1 + \sqrt{(n-1)^2 - n(n-1)\frac{PP''}{P'^2}}}$$

In an exercise, we proved that for $P(s) = 0$:

$$F'(s) = 0$$
$$F''(s) = 0$$

so Laguerre is cubically convergent.

1.8 Nonlinear Equations in Several Variables

For a given funtion $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^n$ we want to compute a vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ such that $\boldsymbol{f}(\boldsymbol{x}) = 0$. For the iteration of such equations, we need norms for vectors and matrices.

Vector Norms

A vector norm $||\boldsymbol{x}|| : \mathbb{R}^n \to \mathbb{R}^+$ must fulfill the following conditions for all $\boldsymbol{x}$:

- $||\boldsymbol{x}|| \geq 0$ and $||\boldsymbol{x}|| = 0 \Leftrightarrow \boldsymbol{x} = 0$
- $||\alpha\boldsymbol{x}|| = |\alpha| \cdot ||\boldsymbol{x}||, \quad \alpha \in \mathbb{R}$
- $||\boldsymbol{x} + \boldsymbol{y}|| \leq ||\boldsymbol{x}|| + ||\boldsymbol{y}||$

The following are popular examples of vector norms:

Spectral norm or 2-norm:

$$||\boldsymbol{x}||_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$$

Infinity norm:

$$||\boldsymbol{x}||_\infty = \max_{1 \leq i \leq n} |x_i|$$

1-norm:

$$||\boldsymbol{x}||_1 = \sum_{i=1}^{n} |x_i|$$

Matrix Norms

A matrix norm $||\boldsymbol{A}|| : \mathbb{R}^{m \times n} \to \mathbb{R}^+$ must fulfill the following conditions for all $\boldsymbol{A}$:

- $||\boldsymbol{A}|| \geq 0$ and $||\boldsymbol{A}|| = 0 \Leftrightarrow \boldsymbol{A} = 0$
- $||\alpha\boldsymbol{A}|| = |\alpha| \cdot ||\boldsymbol{A}||, \quad \alpha \in \mathbb{R}$
- $||\boldsymbol{A} + \boldsymbol{B}|| \leq ||\boldsymbol{A}|| + ||\boldsymbol{B}||$

The following are popular examples of matrix norms:

Spectral norm or 2-norm:

$$||\boldsymbol{A}||_2 = \sup_{||\boldsymbol{x}||_2 = 1} ||\boldsymbol{A}\boldsymbol{x}||_2 = \sigma_{max}(\boldsymbol{A})$$

Infinity norm or maximum row sum norm:

$$||\boldsymbol{A}||_\infty = \sup_{||\boldsymbol{x}||_\infty = 1} ||\boldsymbol{A}\boldsymbol{x}||_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^{n} |a_{ij}|$$

1-norm or maximum column sum norm:

$$||\boldsymbol{A}||_1 = \sup_{||\boldsymbol{x}||_1 = 1} ||\boldsymbol{A}\boldsymbol{x}||_1 = \max_{1 \leq j \leq n} \sum_{j=1}^{n} |a_{ij}|$$

Frobenius norm:

$$||\boldsymbol{A}||_F = \sqrt{\sum_{i,j=0}^{n} a_{ij}^2} = \sqrt{\sum_{i=0}^{n} \sigma_i^2(A)}$$

Fixed Point Iterations

In order to solve $\boldsymbol{f}(\boldsymbol{x}) = 0$, we want to iterate according to the scheme:

$$\boldsymbol{x} = \boldsymbol{F}(\boldsymbol{x})$$

The iteration itself is the same as its one-dimensional analogon, except for the exit criterion:

```
while norm(old_x - new_x) > tol*norm(new_x)
```

This criterion makes the iteration stop, when the value update changes less than *tol* relative to the solution.
The Banach fixed point theorem also applies here, which means that we have convergence when $\boldsymbol{F}(\boldsymbol{x})$ is a contraction.

Newton's Method

Like for the one-dimensional problem, we use the first degree approximation of $\boldsymbol{f}(\boldsymbol{x})$ around the current iteration point $\boldsymbol{x_k}$ for our next value $\boldsymbol{x_{k+1}}$. The approximation is:

$$\boldsymbol{f_1}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x_k}) + \boldsymbol{J}(\boldsymbol{x_k})\boldsymbol{h}, \quad \boldsymbol{h} = \boldsymbol{x} - \boldsymbol{x_k}$$

where $\boldsymbol{J}$ is the Jacobian Matrix of $\boldsymbol{f}$:

$$\boldsymbol{J} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}$$

The new Newton's method is:

$$\boldsymbol{x_{k+1}} = \boldsymbol{F}(\boldsymbol{x_k}) = \boldsymbol{x_k} - \boldsymbol{J}^{-1}(\boldsymbol{x_k})\boldsymbol{f}(\boldsymbol{x_k})$$

This formula basically implies having to solve a complete linear system at each iteration step.

Continuation Methods

A continuation method considers the problem:

$$\boldsymbol{h}(\boldsymbol{x},t) = t\boldsymbol{f}(\boldsymbol{x}) + (1-t)\boldsymbol{g}(\boldsymbol{x}) = 0$$

where $\boldsymbol{h}(\boldsymbol{x},0)$ is the problem $\boldsymbol{g}(\boldsymbol{x}) = 0$ which has a known solution and $\boldsymbol{h}(\boldsymbol{x},1)$ is a difficult problem $\boldsymbol{f}(\boldsymbol{x}) = 0$ which we want to solve. The idea is to take small steps $t := t + \Delta t$, thus getting closer and closer to the result.
More systematically, we are looking for the function $\boldsymbol{x}(t)$. If we have this function, we can compute $\boldsymbol{x}(1)$ and obtain what we are looking for. By looking at the derivative:

$$\frac{d}{dt}\boldsymbol{h}(\boldsymbol{x}(t),t) = \boldsymbol{h_x}(\boldsymbol{x}(t),t)\dot{\boldsymbol{x}}(t) + \boldsymbol{h_t}(\boldsymbol{x}(t),t) = 0$$
$$\dot{\boldsymbol{x}} = -(\boldsymbol{h_x}(\boldsymbol{x},t))^{-1} \cdot \boldsymbol{h_t}(\boldsymbol{x},t)$$

Note that $\boldsymbol{h_x}$ is the Jacobian matrix of $\boldsymbol{h(x,t)}$ with respect to $\boldsymbol{x}$.
After solving this system of differential equations generally, you can insert the starting condition, which is the known solution $\boldsymbol{x}(0)$. You then get a concrete function $\boldsymbol{x}(t)$. $\boldsymbol{x}(1)$ is the solution to $\boldsymbol{f}(x) = 0$.

1.9 Dynamical Systems

The idea here is that fixed point iterations can do much more than just converge or diverge. Lets look at the iteration of $f(x) = ax^2 - ax + x$:

$$x = F(x) = ax - ax^2, \quad a > 0$$

Since $f(x)$ is a very simple function, it is obvious that it has 2 zeros:

$$s_1 = 0, \quad s_2 = \frac{a-1}{a}$$

If we take a look at the convergence properties of our iteration, we see that:

$$F'(s_1) = a, \quad F'(s_2) = 2 - a$$

Since convergence can only occur when the function is a contraction, we can only hope for convergence to s_1 for $0 < a < 1$ and to s_2 for $1 < a < 3$.
The interesting observation is that for the case $a \geq 3$ the trajectory will not be attracted to any of the zeros. Instead, it will go back and forth until it finds itself in a permanent period-2 orbit. This means, that every second step will give the same value, but it will not actually converge.
Now the idea is to always jump a step while iterating, thus taking the iteration of the iteration. This gives:

$$x = F(F(x)) = a^2x(1-x)(1-ax+ax^2)$$

This new iteration now has, in addition to our original fixed points, two more zeros which represent our period-2 solutions. By calculating the derivative, we can again determine, for which values of a the iteration has hope of converging to which of the 4 zeros.
Again we observe that if we chose a within none of those ranges, we will get a period-8 result. Let b_i be the a-value of the i-th bifurcation point. Then the Feigenbaum constant is:

$$\delta = \lim_{n\to\infty} \frac{b_n - b_{n-1}}{b_{n+1} - b_n} = 4.66920166$$

Since the intervals will approach this ratio, they will get smaller and smaller and an infinite period will be reached. The system then behaves chaotically.
Another interesting observation is that every chaotic system caused by period doubling will approach the Feigenbaum Constant.

2 Interpolation and Extrapolation

2.1 Basic Idea

Interpolation and extrapolation are forms of blending in a missing value. Assuming we only know some tabulated values $x_1, x_2, \ldots, x_n$ and $f(x_1), f(x_2), \ldots, f(x_n)$ about a function $f(x)$, we are wondering, what $f(z)$ is for a given z.
If z is within $[x_1, x_n]$, the process is called interpolation. If z is outside, it's called extrapolation.

2.2 Interpolation Principle

If only a list of values for x and $f(x)$ is given (and no additional information about the function), the problem is called ill-posed.
To interpolate the data given in a table for the value z, we use an auxliary function $g(x)$. Typically, g should be easy to evaluate and it should also be a reasonable approximation of f. After g is known, $g(z)$ is taken as an approximation for $f(z)$ in the hope that the interpolation error $|g(z) - f(z)|$ is small.
The next question would be how to chose g. Here are some examples:

- $g(x) = \frac{a}{x-b}$
- $g(x) = ax + b$

In both examples you compute the parameters a and b by requiring that $f(x_i) = g(x_i)$ for both x_i around z.

2.3 The Interpolation Polynomial

Given $n+1$ points of the function $f(x)$, we are looking for a polynomial $P(x)$ such that:

$$\forall i \in \{1, 2, \ldots, n+1\} : \; f(x_i) = P(x_i)$$

Since $n+1$ data points are sufficient information for $n+1$ parameter, our polynomial looks as follows:

$$P(x) = a_0 + a_1x + a_2x^2 + \ldots a_nx^n$$

To determine the coefficients $\{a_i\}$, we solve the system of equations:

$$\begin{bmatrix} 1 & x_0 & \ldots & x_0^n \\ 1 & x_1 & \ldots & x_1^n \\ \vdots & \vdots & & \vdots \\ 1 & x_n & \ldots & x_n^n \end{bmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{pmatrix}$$

The matrix containing the powers of the nodes (usually referred to with V) is called a Vandermonde Matrix. If all the nodes x_i are different, V is regular and the problem has a unique solution.
On the other hand, Vandermonde matrices tend to be ill-conditioned, so we will be looking for other ways to compute the polynomial.

Lagrange Polynomials

Instead of interpolating the whole polynomial, we would like to have a representation of $P(x)$ of the form:

$$P(x) = \sum_{j=0}^{n} f(x_j) \cdot l_j(x)$$

where the factors l_j are all functions in x of degree n. To make sure that $P(x_j) = f(x_j)$ at all nodes, we want to choose the l_j such that:

$$l_j(x_j) = 1, \quad l_j(x_i) = 0, \quad i \neq j$$

For a specific j, the polynomial with this exact behaviour can be written directly in its factored form:

$$l_j(x) = \prod_{i=0, i\neq j}^{n} \frac{x - x_i}{x_j - x_i} =$$

$$\frac{(x - x_0)(x - x_1)\dots(x - x_{j-1})(x - x_{j+1})\dots(x - x_n)}{(x_j - x_0)(x_j - x_1)\dots(x_j - x_{j-1})(x_j - x_{j+1})\dots(x_j - x_n)}$$

Clearly, the evaluation $l_j(x_i)$ yields 0 for $i \neq j$, due to the fact that one factor in the numerator becomes 0, whereas the whole numerator can not be 0, since $x_i \neq x_j$. On the other hand, the evaluation $l_i(x_i)$ will always yield 1, since the numerator becomes equal to the denominator. These polynomials are called Lagrange Polynomials. Obviously, the $n + 1$ Lagrange polynmials can only exist if all the x_i are different (else the denominator would be zero in some polynomials).

This is how we show that the polynomial computed with the Vandermonde matrix (let's call it $Q(x)$) is the same as our polynomial $P(x)$. What we know, is that for all our $n + 1$ points:

$$P(x_i) = Q(x_i)$$

We define the polynomial $d(x) = P(x) - Q(x)$ which must have degree $\leq n$, since P and Q both have degree n. Since for all the $n + 1$ points

$$d(x_i) = P(x_i) - Q(x_i) = 0$$

we know that d has $n + 1$ zeros, making the function: $d(x) = const = 0$. Thus, proving the following Theorem:

Given $n + 1$ distinct nodes x_j, the Vandermonde matrix is regular and there exists a unique interpolation polynomial $P(x)$ with:

$$\forall j \in [0, n] : P(x_j) = f(x_j)$$

The Interpolation Error

Assuming that the function f has continuous derivatives (within the interpolation range), we get an interpolation error:

$$R(x) = f(x) - P(x) = \frac{(x - x_0)(x - x_1)\dots(x - x_n)}{(n + 1)!} f^{(n+1)}(\xi)$$

$$\xi \in [x_0, x_n]$$

That the error is correct has been proven in the script. For further reference we define:

$$L(x) = (x - x_0)(x - x_1)\dots(x - x_n)$$

Sometimes, a more general approximation is made by inserting the maximum for $L(x)$ on the interpolation interval instead of a specific one.
There are 3 main problems with this interpolation method:

- in general, you do not have $f^{(n+1)}$, since you do not know f. So error estimation is usually only possible, if a bound is known for $f^{(n+1)}(\xi)$.
- the error tends to grow very rapidly toward the ends of the interpolation interval, since $L(x)$ becomes pretty big. A high number of given points and/or equidistant points intensify this effect.
- it requires $O(n^2)$ operations for each interpolation (after setup): n evaluations of l_i which each need $O(n)$ operations.

The Barycentric Formula

This method only requires $O(n)$ operations for each interpolation (after setup):

$$P(x) = \frac{\sum_{i=0}^{n} \frac{\lambda_i}{x - x_i} f(x_i)}{\sum_{i=0}^{n} \frac{\lambda_i}{x - x_i}}$$

for

$$\lambda_i = \frac{1}{(x_i - x_0)(x_i - x_1)\dots(x_i - x_{i-1})(x_i - x_{i+1})\dots(x_i - x_n)}$$

It is derived directly from the Lagrange method. It's improved runtime comes from the fact that the λ_i stay constant for each interpolation, so they only need to be calculated once during the setup in $O(n^2)$. From then on, every interpolation is $O(n)$.

Newton's Interpolation Formula

As a basis for this interpolation method, we use the Newton polynomials:

$$\pi_k(x) = \prod_{j=0}^{k-1} (x - x_j), \qquad \text{for } 0 \leq k \leq n$$

We can then express our interpolation polynomial as:

$$P(x) = d_0\pi_0(x) + d_1\pi_1(x) + \dots + d_n\pi_n(x)$$

To find the coefficients d_k we require that $P(x_i) = f(x_i)$ at all our given data points. First, we define the matrix U^T:

$$U^T = \begin{pmatrix} \pi_0(x_0) & \pi_1(x_0) & \dots & \pi_n(x_0) \\ \pi_0(x_1) & \pi_1(x_1) & \dots & \pi_n(x_1) \\ \vdots & \vdots & \vdots & \vdots \\ \pi_0(x_n) & \pi_1(x_n) & \dots & \pi_n(x_n) \end{pmatrix}$$

$$= \begin{pmatrix} 1 & & & & \\ 1 & x_1 - x_0 & & & \\ 1 & x_2 - x_0 & (x_2 - x_0)(x_2 - x_1) & & \\ \vdots & \vdots & \vdots & \ddots & \\ 1 & x_n - x_0 & (x_n - x_0)(x_n - x_1) & \dots & \prod_{j=0}^{n-1}(x_n - x_j) \end{pmatrix}$$

As we can see, solving the system

$$U^T \cdot \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{pmatrix}$$

can easily be done with forward substitution. So we obtain the formula:

$$d_0 = f(x_0), \quad d_i = \frac{f(x_i) - \sum_{j=0}^{i-1} d_j \pi_j(x_i)}{\pi_i(x_i)}, \quad 1 \leq i \leq n$$

There is another way of computing these coefficients. It uses the notion of divided differences, which are defined recursively:

$$f[x_i] = f(x_i)$$
$$f[x_i, x_{i+1}, \ldots, x_{i+k}] = \frac{f[x_{i+1}, x_{i+2}, \ldots, x_{i+k}] - f[x_i, x_{i+1}, \ldots, x_{i+k-1}]}{x_{i+k} - x_i}$$

We observe, that

$$f[x, x_0] = \frac{f[x_0] - f[x]}{x_0 - x} \;\Rightarrow\; f(x) = f[x_0] + (x - x_0) f[x, x_0]$$

and that

$$f[x, x_0, x_1] = \frac{f[x_0, x_1] - f[x, x_0]}{x_1 - x}$$
$$\Rightarrow\; f[x, x_0] = f[x_0, x_1] + (x - x_1) f[x, x_0, x_1]$$

This leads to:

$$\begin{aligned} f(x) &= f(x_0) + (x - x_0) f[x_0, x_1] + (x - x_0)(x - x_1) f[x, x_0, x_1] \\ &= \ldots \\ &= Q(x) + (x - x_0)(x - x_1) \ldots (x - x_n) f[x, x_0, x_1, \ldots, x_n] \end{aligned}$$

for the function $Q(x)$ defined as:

$$Q(x) = \sum_{k=0}^{n} \left(f[x_0, x_1, \ldots, x_k] \cdot \prod_{q=0}^{k-1} (x - x_q) \right)$$

We notice that for all our data points x_i, the second term (after $Q(x)$) becomes zero. Therefore the following property must hold:

$$\forall i : f(x_i) = Q(x_i) \quad \Rightarrow \quad Q(x) \equiv P(x)$$

Simply comparing the expressions shows that:

$$d_i = f[x_0, x_1, \ldots, x_i], \quad 0 \leq i \leq n$$

Using horners scheme, the polynomial Q can be evaluated at an arbitrary point z as follows:

$$Q(z) = f[x_0] + (z - x_0) \left(f[x_0, x_1] + (z - x_1) \right.$$
$$\left. (f[x_0, x_1, x_2] + \cdots + (z - x_{n-1})(f[x_0, x_1, \ldots, x_n]) \ldots) \right)$$

This gives us the following recurrence:

$$\begin{aligned} p_0 &= f[x_0, x_1, \ldots, x_n] \\ p_{k+1} &= (z - x_i) \cdot p_k + f[x_0, x_1, \ldots, x_i], \quad i = n-1, n-2, \ldots, 0 \end{aligned}$$

As we can see, the approximation of f through divided differences has many parallels to Taylor expansion. The basic difference is that you have a finite number of different function values instead of an infinite number of derivatives which have to be correct at each level.
If we now compare the exact expressions for f:

$$\begin{aligned} f(x) &= P(x) + \prod_{k=0}^{n} (x - x_k) \frac{f^{(n+1)}(\xi)}{(n+1)!} \\ &= P(x) + \prod_{k=0}^{n} (x - x_k) f[x, x_0, \ldots, x_n] \end{aligned}$$

We obtain the following lemma. There exists a ξ so that the divided difference for $x_0, \ldots, x_n$ becomes:

$$f[x_0, x_1, \ldots, x_n] = \frac{f^{(n)}(\xi)}{(n)!}$$

Obviously, symmetry is another property of divided differences. This becomes clear if you look at the following explicit formula for its computation:

$$f[x_i, x_{i+1}, \ldots, x_{i+k}] = \sum_{j=i}^{i+k} \frac{f(x_j)}{\prod_{p=i, p \neq j}^{i+k} (x_j - x_p)}$$

Interpolation by Orthogonal Polynomials

Let $p_j(x)$ be a polynomial of degree j and $\langle ., . \rangle$ be any scalar product. A set $\{p_j(x)\}$ is called orthogonal, if:

$$\forall u, v, u \neq v : \; \langle p_u(x), p_v(x) \rangle = 0$$

The norm of a polynomial is also defined as:

$$||p(x)||^2 = \langle p(x), p(x) \rangle$$

The following algorithm generates k polynomials $p_j(x)$, which are orthogonal in respect to a scalar product, which is defined by the $n + 1$ given interpolation points $x_0, \ldots, x_n$. It is important that the scalar product be linear in its second factor. In the script, we used the product:

$$\langle p_a, p_b \rangle = \sum_{i=0}^{n} p_a(x_i) \cdot p_b(x_i)$$

The initialization of the Lanczos algorithm is $p_{-1}(x) = 0$, $p_0(x) = 1, \beta_0 = 0$. We then compute recursively:

$$p_{k+1}(x) = (x - \alpha_{k+1}) p_k(x) - \beta_k \cdot p_{k-1}(x)$$
$$\alpha_{k+1} = \frac{\langle x \cdot p_k(x), p_k(x) \rangle}{||p_k(x)||^2}$$
$$\beta_k = \frac{||p_k(x)||^2}{||p_{k-1}(x)||^2}$$

For our specific scalar product, this means:

$$\alpha_{k+1} = \frac{\sum_{i=0}^{n} x_i \cdot p_k(x_i)^2}{\sum_{i=0}^{n} p_k(x_i)^2}$$
$$\beta_k = \frac{\sum_{i=0}^{n} p_k(x_i)^2}{\sum_{i=0}^{n} p_{k-1}(x_i)^2}$$

We now consider the following approximation:

$$b_0 p_0(x_j) + b_1 p_1(x_j) + \cdots + b_k p_k(x_j) \approx f(x_j)$$

In matrix notation $\boldsymbol{Pb} \approx \boldsymbol{f}$ we obtain:

$$\begin{pmatrix} p_0(x_0) & p_1(x_0) & \ldots & p_k(x_0) \\ p_0(x_1) & p_1(x_1) & \ldots & p_k(x_1) \\ \vdots & \vdots & \ldots & \vdots \\ p_0(x_n) & p_1(x_n) & \ldots & p_k(x_n) \end{pmatrix} \begin{pmatrix} b_0 \\ \vdots \\ b_k \end{pmatrix} \approx \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{pmatrix}$$

For $n = k$ we have a complete system of equations which we can solve exactly. For $k < n$ we can perform a least squares fit. This task is simplified by the fact that $\boldsymbol{P}$ ist orthogonal:

$$\boldsymbol{P}^T \boldsymbol{Pb} = \boldsymbol{P}^T \boldsymbol{f}$$

We notice that the $\boldsymbol{P}^T\boldsymbol{P}$ is a diagonal matrix, so the computation becomes:

$$b_j = \frac{\sum_{i=0}^{n} p_j(x_i) f_i}{\sum_{i=0}^{n} p_j(x_i)^2} = \frac{\boldsymbol{p}_j^T \boldsymbol{f}}{||\boldsymbol{p}_j||^2}$$

A common application is to assume that a measurement error δ occured and we now want to approximate $\boldsymbol{f}$ by a polynomial of the lowest possible degree k, for which the residual $\boldsymbol{r}_k \leq \delta$. Let $\boldsymbol{P}_k$ denote the matrix $[\boldsymbol{p_0}, \ldots, \boldsymbol{p_k}]$ consisting of the first $k+1$ columns of $\boldsymbol{P}$.
The following formula explicitly computes the residual for the approximation of f by a polynomial of degree k:

$$\boldsymbol{r}_k = \boldsymbol{f} - \boldsymbol{P}_k \boldsymbol{b}_k$$

You can also compute the errors using the recursion:

$$||\boldsymbol{r}_{k+1}||^2 = ||\boldsymbol{r}_k||^2 - b_{k+1}^2 ||\boldsymbol{p}_{k+1}||^2$$

It is now easy to write a program, which iteratively increases the approximation level until the residual is smaller than δ.

Change of Basis

We have seen various ways of basically computing the same polynomial approximation of a function. We introduce some new notation:

$$\begin{aligned}
\text{monomials: } \boldsymbol{m}(x) &= [1, x, x^2, \ldots, x^n]^T \\
\text{Lagrange: } \boldsymbol{l}(x) &= [l_0(x), l_1(x), \ldots, l_n(x)]^T \\
\text{Newton: } \boldsymbol{\pi}(x) &= [\pi_0(x), \pi_1(x), \ldots, \pi_n(x)]^T \\
\text{least squares: } \boldsymbol{p}(x) &= [p_0(x), p_1(x), \ldots, p_n(x)]^T
\end{aligned}$$

We also introduce the notation for the coefficient vectors:

$$\begin{aligned}
\text{monomials: } \boldsymbol{a} &= [a_0, a_1, \ldots, a_n]^T \\
\text{function values (Lagrange): } \boldsymbol{f} &= [f_0, f_1, \ldots, f_n]^T \\
\text{Newton: } \boldsymbol{d} &= [d_0, d_1, \ldots, d_n]^T \\
\text{least squares: } \boldsymbol{b} &= [b_0, b_1, \ldots, b_n]^T
\end{aligned}$$

So, using the notation above, our interpolation polynomial is described by a scalar product:

$$P(n) = \boldsymbol{a}^T \boldsymbol{m}(x) = \boldsymbol{f}^T \boldsymbol{l}(x) = \boldsymbol{d}^T \boldsymbol{\pi}(x) = \boldsymbol{b}^T \boldsymbol{p}(x)$$

Certain transformations are fairly easy:

- Lagrange and monomial: $\boldsymbol{l} = \boldsymbol{V^{-1}}^T \boldsymbol{m} \Leftrightarrow \boldsymbol{V}^T \boldsymbol{l} = \boldsymbol{m}$
- Lagrange and Newton: $\boldsymbol{U}\boldsymbol{l}(x) = \boldsymbol{\pi}(x)$

where $\boldsymbol{U}$ denotes the upper right matrix:

$$\boldsymbol{U}^T = \begin{pmatrix}
1 & & & & \\
1 & x_1 - x_0 & & & \\
1 & x_2 - x_0 & (x_2 - x_0)(x_2 - x_1) & & \\
\vdots & \vdots & \vdots & \ddots & \\
1 & x_n - x_0 & (x_n - x_0)(x_n - x_1) & \ldots & \prod_{j=0}^{n-1}(x_n - x_j)
\end{pmatrix}$$

For the relation between monomials and Newton polynomials we would like to have a lower triangular matrix $\boldsymbol{L}$ with $\boldsymbol{L}\boldsymbol{\pi}(x) = \boldsymbol{m}(x)$. In a first step, we define the function $H_p(x_1, \ldots, x_k)$ recursively as:

$$\begin{aligned}
H_p(x_0) &= x_0^p \\
H_1(x_0, \ldots, x_k) &= \sum_{j=0}^{k} x_j \\
H_p(x_0, x_1) &= \sum_{j=0}^{p} x_0^j x_1^{p-j} = \sum_{j=0}^{p} H_j(x_0) H_{p-j}(x_1)
\end{aligned}$$

We can than prove (done in the script), that:

$$H_p(x_0, \ldots, x_k) = \frac{H_{p+1}(x_0, \ldots, x_{k-1}) - H_{p+1}(x_1, \ldots, x_k)}{x_0 - x_k}$$

Consider the interpolation polynomial in monomials

$$P(x) = a_0 + a_1 x + \cdots + a_n x^n$$

and the system of equations containing the Vandermonde matrix:

$$\sum_{j=0}^{n} a_j x_i^j = f(x_i), \quad i = 0, \ldots, n$$

Then the divided difference scheme can be used to compute the coefficients a_j:

$$f[x_i, \ldots, x_{i+k}] = a_k + \sum_{j=k+1}^{n} a_j \cdot H_{j-k}(x_i, \ldots, x_{i+k})$$

This leads us to the relation:

$$f[x_0, \ldots, x_n] = a_n = \frac{P^{(n)}(0)}{n!}$$

Since the divided differences $f[x_0, \ldots, x_j]$ are exactly the Newton coefficients, we get $\boldsymbol{L}^T \boldsymbol{a} = \boldsymbol{d}$ where

$$\boldsymbol{L} = \begin{pmatrix}
1 & & & & \\
H_1(x_0) & 1 & & & \\
H_2(x_0) & H_1(x_0, x_1) & 1 & & \\
\vdots & \vdots & \ddots & \ddots & \\
H_n(x_0) & H_{n-1}(x_0, x_1) & \ldots & H_1(x_0, \ldots, x_{n-1}) & 1
\end{pmatrix}$$

Comparing the representations of the polynomials gives us:

$$\boldsymbol{a}^T \boldsymbol{m} = \boldsymbol{d}^T \boldsymbol{\pi} = \boldsymbol{a}^T \boldsymbol{L} \boldsymbol{\pi} \Rightarrow \boldsymbol{L}\boldsymbol{\pi} = \boldsymbol{m}$$

We also obtain the LU-decomposition of the Vandermonde matrix:

$$\boldsymbol{V}^T = \boldsymbol{L} \cdot \boldsymbol{U} \;\Rightarrow\; \boldsymbol{V} = \boldsymbol{U}^T \boldsymbol{L}^T$$

Recall the orthogonal matrix $\boldsymbol{P}$ of the orthogonal polynomials. Let $\boldsymbol{D} = \operatorname{diag}\{||p_0||, ||p_1||, \ldots, ||p_n||\}$ denote the diagonal matrix computed by $\boldsymbol{D}^2 = \boldsymbol{P}^T \boldsymbol{P}$. We obtain:

$$\boldsymbol{P}^T \boldsymbol{P} \boldsymbol{b} = \boldsymbol{P}^T \boldsymbol{f} \;\Rightarrow\; \boldsymbol{b} = \boldsymbol{D}^{-2} \boldsymbol{P}^T \boldsymbol{f}$$

Inserting this in the equation:

$$\boldsymbol{f}^T \boldsymbol{l}(x) = \boldsymbol{b}^T \boldsymbol{p}(x) \quad \Rightarrow \quad \boldsymbol{P}^T \boldsymbol{l}(x) = \boldsymbol{p}(x)$$

There must exist a lower triangular matrix $\boldsymbol{G}$ for which the following holds:

$$\boldsymbol{G}\boldsymbol{p}(x) = \boldsymbol{m}(x)$$

Inserting all our interpolation points $x = x_0, \ldots, x_n$, we get the matrix equation:

$$\boldsymbol{GP}^T = \boldsymbol{V}^T \;\Rightarrow\; \boldsymbol{V} = \boldsymbol{PG}^T = (\boldsymbol{PD}^{-1})(\boldsymbol{DG}^T)$$

Which is exactly the QR-decomposition of $\boldsymbol{V}$. So to compute $\boldsymbol{G}$, you have two possibilities:

- you decompose $\boldsymbol{V} = \boldsymbol{QR}$ and then:

$$\boldsymbol{R} = \boldsymbol{DG}^T \rightarrow \boldsymbol{G} = \boldsymbol{R}^T\boldsymbol{D}^{-1}$$

- you solve for $\boldsymbol{G}$ in $\boldsymbol{GP}^T = \boldsymbol{V}^T$, which gives you:

$$\boldsymbol{G} = \boldsymbol{V}^T\boldsymbol{PD}^{-2}$$

Replacing $\boldsymbol{m}(x) = \boldsymbol{G}\boldsymbol{p}(x)$ in $\boldsymbol{p}(x)^T\boldsymbol{b} = \boldsymbol{m}(x)^T\boldsymbol{a}$, we get:

$$\boldsymbol{p}(x)^T\boldsymbol{b} = \boldsymbol{p}(x)^T\boldsymbol{G}^T\boldsymbol{a} \;\Rightarrow\; \boldsymbol{G}^T\boldsymbol{a} = \boldsymbol{b}$$

Aitken-Neville Interpolation

The idea is to interpolate the function value $f(z)$ by computing a sequence $\{P_n(z)\}$ of interpolation polynomial values, each polynomial using more interpolation points than the last. Our hope is then that the sequence will converge.

T_{ij} ist defined as the (unique) polynomial of degree $\leq j$ which interpolates the data $\{x_{i-j}, \ldots, x_i\}$ and the corresponding function values $\{y_{i-j}, \ldots, y_i\}$. These polynomials can be computed through the following recursion:

$$T_{i0} = y_i$$
$$T_{ij} = \frac{(x_i - x)T_{i-1,j-1} + (x - x_{i-j})T_{i,j-1}}{x_i - x_{i-j}},\ j = 1, \ldots, i$$

For each i, you go through all j and then proceed with the next i.

In concrete applications, the scheme is not used to compute the polynomials, but directly their evaluations at an interpolation point z, thus the hope of getting a convergent sequence of values for $T_{kk}(z)$.

2.4 Extrapolation

Extrapolation is identical to interpolation, except that the value is outside of the range of given values. Without loss of generality, we may assume that the function we want to interpolate is always at the point 0, since if it isn't, we can simply apply a variable transformation.

Assuming the unknown value

$$\lim_{x \to 0} f(x) = a_0$$

is difficult to compute, we can evaluate some function values $f(x_i), x_i > 0$ and construct the Aitken-Neville scheme. We continue taking more and more points and hope for the sequence $T_{kk}(0)$ to converge to a_0.
This is the case if, for example, the function f has an asymptotic expansion of the form:

$$f(x) = a_0 + a_1 x + \cdots + a_k x^k + R_k(x), \quad |R_k(x)| \leq Cx^{k+1}$$

and the sequence of x_i is chosen to converge to 0 as well:

$$x_{i+1} \leq c \cdot x_i, \quad 0 < c < 1$$

Interesting is the fact that since we always evaluate the interpolation polynomials at $x = 0$, the recurrence simplifies to:

$$T_{ij} = \frac{x_i T_{i-1,j-1} - x_{i-j} T_{i,j-1}}{x_i - x_{i-j}}$$

Furthermore, we can choose the special sequence $x_i = x_0 \cdot 2^{-i}$ and the recurrence becomes:

$$T_{ij} = \frac{2^{-j} T_{i-1,j-1} - T_{i,j-1}}{2^{-j} - 1}$$

Another special case is if the odd powers of x are missing in the asymptotic expansion. We can then use the following recursion to get a faster growing degree and therefore a better approximation with the same amount of data:

$$T_{ij} = \frac{x_i^2 T_{i-1,j-1} - x_{i-j}^2 T_{i,j-1}}{x_i^2 - x_{i-j}^2}$$

Of course, the two special cases can be combined resulting in:

$$T_{ij} = \frac{4^{-j} T_{i-1,j-1} - T_{i,j-1}}{4^{-j} - 1}$$

3 Piecewise Polynomial Interpolation

3.1 Classical Cubic Splines

Since the interpolation polynomial may not always yield acceptable results, it is sometimes better to interpolate the data piecewise with lower-degree polynomials, thus avoiding a possibe oscillating interpolation polynomial.

The idea is, for given data $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$, to interpolate in each interval $[x_i, x_{i+1}]$ by a polynomial $P_i(x)$. We would then like to require identical evaluations and derivatives, giving us the equations:

$$P_i(x_i) = y_i, \qquad P_i(x_{i+1}) = y_{i+1}$$
$$P_i'(x_i) = y_i', \qquad P_i'(x_{i+1}) = y_{i+1}'$$

Since we have 4 equations, an obvious choice for the polynomial degree is 3.
We would also like to express P_i as a function of the distance from x_i relative to x_{i+1}, so we parametrise it:

$$h_i = x_{i+1} - x_i, \quad t = \frac{x - x_i}{h_i}$$
$$\Rightarrow Q_i(t) = P_i(x_i + th_i), \quad t \in [0, 1]$$

Inserting the function value and derivative conditions, we obtain the closed expression for the interval interpolation polynomial Q_i:

$$Q_i(t) = y_i(1 - 3t^2 + 2t^3) + y_{i+1}(3t^2 - 2t^3) + h_i y_i'(t - 2t^2 + t^3) + h_i y_{i+1}'(-t^2 + t^3)$$

Using a slightly more efficient technique, the polynomial Q_i can be computed using only 3 multiplications and 8 additions/subtractions.
To perform an interpolation of z, you proceed as follows:

1. find the interval, for which $x_i \leq z < x_{i+1}$ holds
2. compute t_z for that specific z
3. compute and evaluate $Q_i(t_z)$

Derivatives for the Spline Function

Since in general you cannot assume that the derivatives are known, a reasonable idea is to estimate them. For this task, there are 2 strategies:

Strategy 1: Defective spline functions For inner points we take the slope of the line through neighbouring points:

$$y'_i = \frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}} = \frac{y_{i+1} - y_{i-1}}{h_i + h_{i-1}}$$

For the two boundary points we have different possibilities:

1. Use the slope of the line through the first/last two points:

$$y'_1 = \frac{y_2 - y_1}{x_2 - x_1}, \quad y'_n = \frac{y_n - y_{n-1}}{x_n - x_{n-1}}$$

2. Natural boundary conditions:

$$Q''_1(0) = 0 \text{ and } Q''_{n-1}(1) = 0$$

After solving the equations, we get:

$$y'_1 = \frac{3}{2}\frac{y_2 - y_1}{h_1} - \frac{1}{2}y'_2 \text{ and } y'_n = \frac{3}{2}\frac{y_n - y_{n-1}}{h_{n-1}} - \frac{1}{2}y'_{n-1}$$

3. Assume the points are periodic (y_1 and y_n represent the same point) and use the slope through neighbours:

$$y'_1 = y'_n = \frac{y_2 - y_{n-1}}{h_1 + h_{n-1}}$$

Strategy 2 The idea here is to choose the first derivatives y'_i in a way that the second derivatives will be continuous:

$$P''_i(x_{i+1}) = P''_{i+1}(x_{i+1})$$

By inserting those conditions, we get the following underdefined system of $n-2$ equations for n variables:

$$\begin{pmatrix} b_1 & a_1 & b_2 & & & \\ & b_2 & a_2 & b_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & b_{n-2} & a_{n-2} & b_{n-1} \end{pmatrix} \begin{pmatrix} y'_1 \\ y'_2 \\ \vdots \\ y'_n \end{pmatrix} = \begin{pmatrix} 3(d_2 + d_1) \\ 3(d_3 + d_2) \\ \vdots \\ 3(d_{n-1} + d_{n-2}) \end{pmatrix}$$

where a, b, c are defined as follows:

$$a_i = \frac{2}{h_i} + \frac{2}{h_{i+1}}, \quad b_i = \frac{1}{h_i}, \quad \text{for } i = 1, \ldots, n-2$$
$$d_i = \frac{y_{i+1} - y_i}{h_i^2}, \quad \text{for } i = 1, \ldots, n-1$$

To determine the derivatives uniquely, we need 2 more equations. Similarly to the defective spline strategy, we have 3 possibilities:

1. Natural boundary conditions: $P''_1(x_1) = P''_{n-1}(x_n) = 0$
   We obtain the 2 missing equations:

$$\frac{2}{h_1}y'_1 + \frac{1}{h_1}y'_2 = 3d_1$$
$$\frac{1}{h_{n-1}}y'_{n-1} + \frac{2}{h_{n-1}}y'_n = 3d_{n-1}$$

   This solution represents the interpolation function g through all the points, which has the minimum oscillation. It can be shown that g solves the following minimization problem:

$$\min_g \int_{x_1}^{x_n} (g''(x))^2 dx$$

2. Not a knot condition (not discussed in class):
   $P_1(x) \equiv P_2(x)$ and $P_{n-1}(x) \equiv P_{n-2}(x)$

3. Periodic boundary conditions (not discussed in class):
   $P'_1(x_1) = P'_{n-1}(x_n)$ and $P''_1(x_1) = P''_{n-1}(x_n)$

3.2 Sherman-Morrison-Woodbury Formula

Many interpolation problems require, at one point or another, the solution to the problem:

$$(\boldsymbol{A} + \boldsymbol{U}\boldsymbol{V}^T)\boldsymbol{x} = \boldsymbol{b}$$

Sherman, Morrison and Woodbury developed the following formula:

$$(\boldsymbol{A} + \boldsymbol{U}\boldsymbol{V}^T)^{-1} = \boldsymbol{A}^{-1} - \boldsymbol{A}^{-1}\boldsymbol{U}(\boldsymbol{I} + \boldsymbol{V}^T\boldsymbol{A}^{-1}\boldsymbol{U})^{-1}\boldsymbol{V}^T\boldsymbol{A}^{-1}$$

At first glance, this formula looks particularly complicated. But for interpolation problems, A is often a tridiagonal matrix, which makes the whole computation much easier.

3.3 Spline Curves

Given n points (x_i, y_i), we would like to connect them by a curve. Every plane curve is represented by a parametric function which consists of a function for x and a function for y (both of s):

$$(x(s), y(s)), \quad s_1 \le s \le s_n$$

So we interpret the tuples as function values of the two functions of the same s:

$$(x_i, y_i) = (x(s_i), y(s_i))$$

The sequence of s_i can be chosen arbitrarily, but we must pay attention to monotonicity, since we want our two functions ($x(s)$ and $y(s)$) to go through the points in-order. One possibility is to use the distance from one point to the next. A value of s is then the total sum of distances to the first point:

$$s_1 = 0$$
$$s_{i+1} = s_i + \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$$

With the given tuples, you can interpolate $x(s)$ and $y(s)$ using any spline method. For closed curves it is wise to use periodic boundary conditions to avoid a cusp at endpoints.

3.4 Polynomial Curves

Consider a so-called polynomial curve in $\mathbb{R}^d$ with $\boldsymbol{a}_i \in \mathbb{R}^d$:

$$\boldsymbol{P}_n(t) = \sum_{i=0}^{n} \boldsymbol{a}_i t^i$$

This function represents a parametric polynomial of $d+1$ dimensions: every t gets mapped to a value in each of the d dimensions.

Even though these polynomials can be computed analogically to the case $d = 1$ discussed earlier, many computer graphics areas put less focus on interpolating the polynomial than on constructing nice curves. For this task, the use of another basis is preferred.

3.5 Bezier Curves

Bezier curves are polynomial curves represented in the basis of Bernstein polynomials:

$$\boldsymbol{C}_n(t) = B_{0,n}(t)P_0 + B_{1,n}(t)P_1 + \cdots + B_{n,n}(t)P_n, \quad t \in [0,1]$$

The basis polynomials are defined as:

$$B_{i,n}(t) = \binom{n}{i}(1-t)^{n-i}t^i$$

The points P_i are vectors and are called control points. To fit a Bezier curve through 2-dimensional points in the plane, all the control points are 2-dimensional. The curve is then parametrized by t. Note that a Bezier curve always interpolates the first and last point. The control points "pull" the curve towards them.

Let $\boldsymbol{C}_k(t; P_i, P_{i+1}, \ldots, P_{i+k})$ denote the Bezier curve of degree k defined by the control points $P_i, P_{i+1}, \ldots, P_{i+k}$, then:

$$\begin{aligned}\boldsymbol{C}_{k+1}(t; P_i, \ldots, P_{i+k+1}) &= (1-t)\cdot \boldsymbol{C}_k(t; P_i, \ldots, P_{i+k}) \\ &\quad + t \cdot \boldsymbol{C}_k(t; P_{i+1}, \ldots, P_{i+k+1})\end{aligned}$$

4 Least Squares

4.1 The Least Squares Problem

Given a function:

$$\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m, \quad n \leq m$$

$$\boldsymbol{f}\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} f_1(x_1, \ldots, x_n) \\ f_2(x_1, \ldots, x_n) \\ \vdots \\ f_m(x_1, \ldots, x_n) \end{pmatrix}$$

we would like to find a point $\boldsymbol{x} \in \mathbb{R}^n$ such that $\boldsymbol{f}(\boldsymbol{x}) \approx 0$. Expressed more precisely, we would like to achieve a minimal residual $\boldsymbol{r}$:

$$\boldsymbol{r} = ||\boldsymbol{f}(\boldsymbol{x})||_2^2 = \sum_{i=1}^{m} f_i(\boldsymbol{x})^2 = \min$$

From now on, $||x||$ will denote $||x||_2$ unless explicitly indicated.

4.2 Notations and Definitions

Gradient

Let $h : \mathbb{R}^n \to \mathbb{R}$ be a function of the form:

$$h\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = y$$

The gradient of h is defined as the column vector of derivatives:

$$\text{grad } h = \nabla h = \frac{\partial h(\boldsymbol{x})}{\partial \boldsymbol{x}} = \left(\tfrac{\partial h(\boldsymbol{x})}{\partial x_1}, \ldots, \tfrac{\partial h(\boldsymbol{x})}{\partial x_n}\right)^T = \left(h_{x_1}, \ldots, h_{x_n}\right)^T$$

The direction of ∇h represents the direction of steepest growth and $||\nabla h||$ represents the slope of h in that direction.

Jacobian Matrix

Let $\boldsymbol{g} : \mathbb{R}^n \to \mathbb{R}^m$ be a function of the form:

$$\boldsymbol{g}\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} g_1(x_1, \ldots, x_n) \\ \vdots \\ g_m(x_1, \ldots, x_n) \end{pmatrix}$$

The Jacobian Matrix of $\boldsymbol{g}$ is defined as the matrix of its first derivatives:

$$J_{\boldsymbol{g}} = (\nabla g_1, \ldots, \nabla g_m)^T = \begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \cdots & \frac{\partial g_1}{\partial x_n} \\ \frac{\partial g_2}{\partial x_1} & \cdots & \frac{\partial g_2}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial g_m}{\partial x_1} & \cdots & \frac{\partial g_m}{\partial x_n} \end{pmatrix}$$

Hessian Matrix

Let $h : \mathbb{R}^n \to \mathbb{R}$ be a function of the form:

$$h\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = y$$

The Hessian of h is the $n \times n$ matrix is the Jacobian of ∇h:

$$\text{hess } h = H_h(\boldsymbol{x}) = J_{\nabla h}(\boldsymbol{x}) = \begin{pmatrix} h_{x_1x_1} & h_{x_1x_2} & \ldots & h_{x_1x_n} \\ h_{x_2x_1} & h_{x_2x_2} & \ldots & h_{x_2x_n} \\ \vdots & \vdots & \vdots & \vdots \\ h_{x_nx_1} & h_{x_nx_2} & \ldots & h_{x_nx_n} \end{pmatrix}$$

Notice that the Hessian matrix is always symmetric if all second derivatives of h are continuous.

Taylor Approximation

The Taylor series of a function $d : \mathbb{R}^n \to \mathbb{R}$ is given by:

$$\begin{aligned} d(\boldsymbol{x}+\boldsymbol{h}) &= d(\boldsymbol{x}) + \nabla d(\boldsymbol{x})^T\boldsymbol{h} + \frac{1}{2}\boldsymbol{h}^T H_d(\boldsymbol{x})\boldsymbol{h} + O(||\boldsymbol{h}||^3) \\ &\approx d(\boldsymbol{x}) + \nabla d(\boldsymbol{x})^T\boldsymbol{h} \end{aligned}$$

For a vector function $\boldsymbol{f}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), \ldots, f_m(\boldsymbol{x}))^T$ we develop each component separately. In matrix notation, we obtain:

$$\boldsymbol{f}(\boldsymbol{x}+\boldsymbol{h}) \approx \begin{pmatrix} f_1(\boldsymbol{x}) + \nabla f_1(\boldsymbol{x})^T\boldsymbol{h} \\ \vdots \\ f_m(\boldsymbol{x}) + \nabla f_1(\boldsymbol{x})^T\boldsymbol{h} \end{pmatrix} = \boldsymbol{f}(\boldsymbol{x}) + J_f(\boldsymbol{x}) \cdot \boldsymbol{h}$$

Extracting Rows and Columns

Given a $m \times n$ matrix A, we will denote:

- the i-th column of A by $\boldsymbol{cA}_i$
- the j-th row of A by $\boldsymbol{rA}_j$

Descent Direction

A vector $\boldsymbol{v}$ is called a descent direction of a function $\Phi(\boldsymbol{x})$ at the point $\boldsymbol{x}$ if:

$$\nabla\Phi(\boldsymbol{x})^T \cdot \boldsymbol{v} < 0$$

The intuitive explanation is that a negative scalar product between the steepest upward slope $\nabla\Phi(\boldsymbol{x})$ and $\boldsymbol{v}$ is that the angle between the two vectors is $> 90°$. Thus $\boldsymbol{v}$ must point

downhill.

The algebraic eplanation is based on Taylor expansion:

$$\Phi(\boldsymbol{x}+\lambda\boldsymbol{v}) = \Phi(\boldsymbol{x}) + \lambda \cdot \underbrace{\nabla\Phi(\boldsymbol{x})^T\boldsymbol{v}}_{<0} + O(\lambda^2)$$

For sufficiently small λ, the quadratic rest is negligable. Thus the following holds:

$$\Phi(\boldsymbol{x}+\lambda\boldsymbol{v}) < \Phi(\boldsymbol{x})$$

4.3 Newton's Method

As defined earlier, our aim is to solve $\boldsymbol{f}(\boldsymbol{x}) \approx 0$ for some $\boldsymbol{f}: \mathbb{R}^n \to \mathbb{R}^m$. To be able to do so, we define a helper function:

$$\Phi(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}$$
$$\Phi(\boldsymbol{x}) = \frac{1}{2} \cdot ||\boldsymbol{f}(\boldsymbol{x})||^2 = \frac{1}{2} \cdot \sum_{i=1}^{m} f_i(\boldsymbol{x})^2$$

Solving the least squares problem for $\boldsymbol{f}$ is now equivalent to minimizing Φ. To minimize Φ, we need to fulfill the condition $\nabla\Phi = 0$. Expressing this in terms of $\boldsymbol{f}$ instead of Φ, we get:

$$\nabla\Phi = \begin{pmatrix} \frac{\partial\Phi}{\partial x_1} \\ \vdots \\ \frac{\partial\Phi}{\partial x_n} \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{m} f_i(\boldsymbol{x}) \cdot \frac{\partial f_i}{\partial x_1} \\ \vdots \\ \sum_{i=1}^{m} f_i(\boldsymbol{x}) \cdot \frac{\partial f_i}{\partial x_n} \end{pmatrix} = J_f(\boldsymbol{x})^T \boldsymbol{f}(\boldsymbol{x}) = 0$$

As we can see, $\nabla\Phi = {J_f}^T \cdot \boldsymbol{f} = 0$ is a nonlinear system of n equations in n unknowns (the components of the vector $\boldsymbol{x}$). We decide to solve this with Newton's multidimensional fixed point iteration.

Since Newton's iteration requires the Jacobian of the function we want to iterate upon (in our case $\nabla\Phi$), we observe that $J_{\nabla\Phi} = H_\Phi$.

Our (quadratic) Newton's iteration thus becomes:

1. Estimate a starting point $\boldsymbol{x}_0$
2. Update the equation using:
   $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - {H_\Phi}^{-1} \cdot \nabla\Phi = \boldsymbol{x}_k - {H_\Phi}^{-1}(\boldsymbol{x}_k) J_f(\boldsymbol{x}_k) \boldsymbol{f}(\boldsymbol{x}_k)$

Our goal is to get an iteration which only depends on $\boldsymbol{f}$. This is almost the case. The only thing left to do is to express H_Φ in terms of $\boldsymbol{f}$. In the script it has been shown that:

$$H_\Phi(\boldsymbol{x}) = {J_f}^T(\boldsymbol{x}) \cdot J_f(\boldsymbol{x}) + \sum_{i=0}^{m} f_i(\boldsymbol{x}) \cdot H_{f_i}(\boldsymbol{x})$$

So for definitive Newton's iteration for the non-linear least squares problem you first estimate an $\boldsymbol{x}_0$ and then repeat the following steps:

- Solve the linear system to get the correction $\boldsymbol{h}$:

$$\left({J_f}^T(\boldsymbol{x}_k) J_f(\boldsymbol{x}_k) + \sum_{i=0}^{m} f_i(\boldsymbol{x}_k) \cdot H_{f_i}(\boldsymbol{x}_k) \right) \cdot \boldsymbol{h} = -J_f(\boldsymbol{x}_k)^T \boldsymbol{f}(\boldsymbol{x}_k)$$

- iterate $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{h}$

This algorithm can also be interpreted differently. Instead of minimizing the function Φ by computing $\boldsymbol{x}$ for $\nabla\Phi(\boldsymbol{x}) = 0$, we can decide to approximate Φ by $Q(\boldsymbol{h})$ around every iteration point (giving us a new Q at every iteration step):

$$Q(\boldsymbol{h}) = \Phi(\boldsymbol{x}_k) + \nabla\Phi(\boldsymbol{x}_k)^T \boldsymbol{h} + \frac{1}{2} \boldsymbol{h}^T H_\Phi(\boldsymbol{x}_k) \boldsymbol{h}$$

We then minimize Q by requiring:

$$\nabla Q(\boldsymbol{h}) = 0 \quad \Leftrightarrow \quad H_\Phi(\boldsymbol{x}_k)\boldsymbol{h} + \nabla\Phi(\boldsymbol{x}_k) = 0$$

As we can see, we get the same iteration. Note that the iteration above implicitly represents a reapproximation of Q at every step without actually computing the new Q at every step.

4.4 The Gauss-Newton Method

The Gauss-Newton method is an alternative way of solving a non-linear least squares problem $\boldsymbol{f}(\boldsymbol{x}) \approx 0$. It consists of taking the linear approximation of $\boldsymbol{f}$

$$\boldsymbol{f}(\boldsymbol{x}) \approx \boldsymbol{f}(\boldsymbol{x}_k) + J_f(\boldsymbol{x}_k)\boldsymbol{h}$$

and solving a linear least squares problem $\boldsymbol{f}(\boldsymbol{x}_k) + J_f(\boldsymbol{x}_k)\boldsymbol{h} \approx 0$. The solution to the linearized problem is then used as the next iteration point for the non-linear problem. The algorithm becomes:

1. Estimate a starting point $\boldsymbol{x}_0$
2. Solve the linear least squares problem for $\boldsymbol{h}$:
$$J_f(\boldsymbol{x}_k)\boldsymbol{h} \approx -\boldsymbol{f}(\boldsymbol{x}_k)$$
$$J_f^T J_f \boldsymbol{h} = -J_f^T \boldsymbol{f}$$
3. Iterate $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{h}$

With this method, a single iteration is simpler to compute, but since convergence is only linear (as opposed to the quadratic Newton's method), more steps are required.

4.5 Method Analysis

In this section we would like to analyze the 2 methods which we have seen so far. Our goal is to prove that iteration correction $\boldsymbol{h}$ of both methods are descent directions. This would mean that the correction always decreases the value of Φ, which would be good, since the task is to minimize Φ.

Newton

For this method, $\boldsymbol{h}$ is obtained by solving the linear system of equations:

$$H_\Phi(\boldsymbol{x}_k)\boldsymbol{h} = -\nabla\Phi(\boldsymbol{x}_k)$$

If $H_\Phi(\boldsymbol{x}_k)$ is non-singular, we can transform the equation into:

$$\boldsymbol{h} = -H_\Phi(\boldsymbol{x}_k)^{-1}\nabla\Phi(\boldsymbol{x}_k)$$
$$\nabla\Phi(\boldsymbol{x}_k)^T\boldsymbol{h} = -\nabla\Phi(\boldsymbol{x}_k)^T H_\Phi(\boldsymbol{x}_k)^{-1}\nabla\Phi(\boldsymbol{x}_k)$$

The last equation is identical to the definition of a descent direction, so what we want to show is that the right side of this equation will always be negative, i.e:

$$\nabla\Phi(\boldsymbol{x}_k)^T H_\Phi(\boldsymbol{x}_k)^{-1}\nabla\Phi(\boldsymbol{x}_k) > 0$$

To prove this, it suffices to show that $H_\Phi(\boldsymbol{x}_k)$ is positive definite. First, we observe that a Hessian is always symmetric (prerequisite for positive definiteness) for continuous derivatives.

For our purposes, it also suffices to show that $H_\Phi(\boldsymbol{x}_k)$ is positive definite in the neighbourhood of the local minimum $\boldsymbol{x}^*$ to which our iteration should converge. Since $\boldsymbol{x}^*$ is a local minimum, we know:

- $\nabla\Phi(\boldsymbol{x}^*) = 0$
- $\Phi(\boldsymbol{x}^*) < \Phi(\boldsymbol{x}^* + \boldsymbol{h}), \quad \boldsymbol{h} \neq 0$

Taylor expansion of Φ aorund $\boldsymbol{x}^*$ gives us:

$$\Phi(\boldsymbol{x}^* + \boldsymbol{h}) = \Phi(\boldsymbol{x}^*) + \underbrace{\nabla\Phi(\boldsymbol{x}^*)^T\boldsymbol{h}}_{=0} + \frac{1}{2}\boldsymbol{h}^T H_\Phi(\boldsymbol{x}^*)\boldsymbol{h} + O(||h||^3)$$

Since $\boldsymbol{x}^*$ is a local minimum, $\boldsymbol{h}^T H_\Phi(\boldsymbol{x}^*)\boldsymbol{h}$ has to be ≥ 0 for every vector $\boldsymbol{h} \neq 0$ (in the neighbourhood of $\boldsymbol{x}^*$). This means that $H_\Phi(\boldsymbol{x})$ $\underbrace{\text{must be}}_{WHY??}$ positive semidefinite in the neighbourhood of $\boldsymbol{x}^*$.

So we obtained: if $H_\Phi(\boldsymbol{x}_k)$ is non-singular in a neighbourhood of $\boldsymbol{x}^*$, then Newton correction is a descent direction.

Gauss-Newton

Here the correction term is computed by solving the linear system:

$$J_f(\boldsymbol{x}_k)^T J_f(\boldsymbol{x}_k)\boldsymbol{h} = -\underbrace{J_f(\boldsymbol{x}_k)^T \boldsymbol{f}(\boldsymbol{x}_k)}_{=\nabla\Phi(\boldsymbol{x}_k)}$$

The first observation is that the matrix ${J_f}^T J_f$ is always symmetric and positive definite:

$$\boldsymbol{z}^T {J_f}^T J_f \boldsymbol{z} = (J_f\boldsymbol{z})^T(J_f\boldsymbol{z}) = ||J\boldsymbol{z}||_2^2 > 0, \quad \text{for } \boldsymbol{z} \neq 0$$

If the matrix $J_f(\boldsymbol{x}_k)^T J_f(\boldsymbol{x}_k)$ is non-singular, we can transform:

$$\begin{aligned} \boldsymbol{h} &= -(J_f^T J_f)^{-1}\nabla\Phi \\ \nabla\Phi^T\boldsymbol{h} &= -\underbrace{\nabla\Phi^T(J_f^T J_f)^{-1}\nabla\Phi}_{>0} \end{aligned}$$

This directly proves that $\boldsymbol{h}$ is a descent direction.

4.6 Levenberg-Marquart Algorithm

During an iteration with the goal of minimizing $\Phi(x)$, the best local descent direction is the negative gradient:

$$\boldsymbol{h} = -\nabla\Phi(\boldsymbol{x}_k) = -J_f(\boldsymbol{x}_k)^T\boldsymbol{f}(\boldsymbol{x}_k)$$

However, the best local minimization direction is not always the best direction for global optimization. The idea of the Levenberg-Marquart algorithm is a compromise between the negative gradient and Newton-Gauss:

$$(J_f(\boldsymbol{x}_k)^T J_f(\boldsymbol{x}_k) + \lambda I)\boldsymbol{h} = -J_f(\boldsymbol{x}_k)^T\boldsymbol{f}(\boldsymbol{x}_k)$$

In this update equation, we have to choose a λ:

- for $\lambda = 0$ we get the regular Newton-Gauss algorithm
- for $\lambda >> 0$ the correction has the direction of the negative gradient, since the Jacobian part of the left side becomes very small compared to the huge diagonal part (which doesn't change the gradient direction)

Choosing a good λ is not easy. Many startegies have been proposed, this one is from Brown and Dennis:

$$\lambda = c||\boldsymbol{f}(\boldsymbol{x}_k)||, \quad c = \begin{cases} 10 & \text{for } 10 \leq ||\boldsymbol{f}(\boldsymbol{x}_k)|| \\ 1 & \text{for } 1 \leq ||\boldsymbol{f}(\boldsymbol{x}_k)|| < 10 \\ 0.01 & \text{for } ||\boldsymbol{f}(\boldsymbol{x}_k)|| < 1 \end{cases}$$

4.7 Singular Value Decomposition

Definition of SVD

Let $A \in \mathbb{R}^{m\times n}, m \geq n$ be a matrix. Then there exist unitary matrices $U \in \mathbb{R}^{m\times m}$, $V \in \mathbb{R}^{n\times n}$ and the diagonal matrix $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_n) \in \mathbb{R}^{m\times n}$ with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$, such that:

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

The column vectors of $U = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_m]$ are called left singular vectors. The columns of $V = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n]$ are the right singular vectors.

The diagonal elements of Σ are called the singular values of A. If σ_r is the smallest singular value with $\sigma_r > 0$, then A has rank r.

Types of the SVD

Let $r = rank(A)$. Then $\sigma_{r+1}, \ldots, \sigma_n = 0$. Starting with the SVD $A = U\Sigma V^T$, we now define the matrices:

$$\begin{aligned} &U_1 = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n], \Sigma_1 = \text{diag}(\sigma_1, \ldots, \sigma_n) \\ &U_r = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_r], \Sigma_r = \text{diag}(\sigma_1, \ldots, \sigma_r), V_r = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_r] \end{aligned}$$

The relation between Σ and Σ_r is given by:

$$\Sigma = \begin{pmatrix} \Sigma_1 \\ 0 \end{pmatrix} = \begin{pmatrix} \Sigma_r & 0 \\ 0 & 0 \end{pmatrix} \in \mathbb{R}^{m\times n}$$

We differenciate between the following types of SVD:

- Full SVD
  $U \in \mathbb{R}^{m\times m}, V \in \mathbb{R}^{n\times n}, \Sigma \in \mathbb{R}^{m\times n}$
  $A = U\Sigma V^T$
  MATLAB: [U, S, V] = svd(A)

- Economic SVD
  $U_1 \in \mathbb{R}^{m\times n}, V \in \mathbb{R}^{n\times n}, \Sigma_1 \in \mathbb{R}^{n\times n}$
  $A = U_1\Sigma_1 V^T$
  MATLAB: [U, S, V] = svd(A,0)

- Reduced SVD:
  $U_r \in \mathbb{R}^{m\times r}, V_r \in \mathbb{R}^{n\times r}, \Sigma_r \in \mathbb{R}^{r\times r}$
  $A = U_r\Sigma_r {V_r}^T$

Note that the reduced SVD does not have an own MATLAB command, since it requires the rank of A. This is a problem because it is difficult to differenciate between a very small singular value and a sigular value equal to zero.

Computation of the SVD

To compute the SVD of a matrix A with $\text{rank}(A) = r$, we perform the following steps:

1. compute the Eigenvalue decomposition of the regular matrix A^TA and obtain V:

   $$A^TA = VDV^T$$

   note that all there are exactly r non-zero Eigenvalues in D and that these are all positive

2. for $D = \text{diag}(\lambda_1, \dots, \lambda_n)$ compute the singular values

   $$\sigma_i = \sqrt{(\lambda_i)}, \quad i = 1, \dots, n$$
   $$\Rightarrow \ \Sigma_n = \text{diag}(\sigma_1, \dots, \sigma_n)$$

3. compute the missing matrix U_1 as follows:

   $$U_1 = AV{\Sigma_1}^{-1}$$
   $$\Rightarrow U_1\Sigma_1V^T = AV{\Sigma_1}^{-1}\Sigma_1V^T = AVV^T = A$$

Now that the economic SVD $A = U_1\Sigma_1V^T$ has been computed you can:

- remove the rightmost columns of Σ_1 to get Σ_r
- concatenate Σ_1 with zeroes to get Σ
- remove the rightmost columns of V to get V_r
- remove the rightmost columns of U_1 to get U_r
- extend U_1 to U by adding arbitrary vectors and orthogonalizing them with a technique such as Gram-Schmidt

Properties of the SVD

If $A = U\Sigma V^T$ then the column vectors of U are the eigenvectors of AA^T and the column vectors of V are eigenvectors of A^TA. The non-zero eigenvalues of AA^T and A^TA are the same and all positive. The roots of these eigenvalues are the singular values of A.
This gives us an alternative method of computing the SVD of A, but the algorithm above is generally more popular, since computing two EVDs is expensive.

If $A = U\Sigma V^T$ then:

$$||A||_2 = \sigma_1, \quad ||A||_F = \sqrt{\sum_{i=1}^{n} \sigma_i^2}$$

Applications of the SVD

Approximation: Since any matrix multiplication can be written as a sum of rank-1 matrices, we can always express A as:

$$A = U_r\Sigma_r{V_r}^T = \sum_{i=1}^{r} \boldsymbol{u}_i{\boldsymbol{v}_i}^T\sigma_i$$

We also know that U, V are unitary (orthogonal), so the following holds:

$$\forall i : ||\boldsymbol{u}_i|| = ||\boldsymbol{v}_i|| = 1$$

Applying the definition of the matrix 2-norm to such a rank-1 matrix, we obtain:

$$||\boldsymbol{u}_i{\boldsymbol{v}_i}^T||_2^2 = \max_{||x||=1} ||\boldsymbol{u}_i \underbrace{({\boldsymbol{v}_i}^T\boldsymbol{x})}_{\in\mathbb{R}}||^2 = \underbrace{||\boldsymbol{u}_i||}_{=1} \cdot \max_{||x||=1} ({\boldsymbol{v}_i}^T\boldsymbol{x})^2 = 1$$
$$\Rightarrow ||\boldsymbol{u}_i{\boldsymbol{v}_i}^T||_2 = \sqrt{1} = 1$$

As we can see, the matrix A is computed by a weighted sum of matrices with the same norm. The weights are the singular values, of which we know:

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > 0$$

So the matrix A can be approximated by a matrix of lower rank by dropping the smallest singular values, i.e. setting them to zero.
More formally, if $\mathcal{M}$ denotes the set of $m \times n$ matrices with rank p, then the solution to $\min_{X\in\mathcal{M}} ||A - X||_F$ is given by:

$$X = A_p = \sum_{i=1}^{p} \boldsymbol{u}_i{\boldsymbol{v}_i}^T\sigma_i$$

Optimization: If $A = U\Sigma V^T$ then the problem $||A\boldsymbol{x}|| = \min$ subject to the constraint $||\boldsymbol{x}|| = 1$ has the solution:

$$\boldsymbol{x} = \boldsymbol{v}_n, \quad ||A\boldsymbol{v}_n|| = \sigma_n$$

and the problem $||A\boldsymbol{x}|| = \max$ subject to the constraint $||\boldsymbol{x}|| = 1$ has the solution:

$$\boldsymbol{x} = \boldsymbol{v}_1, \quad ||A\boldsymbol{v}_1|| = \sigma_1 \quad \Rightarrow \quad ||A||_2 = \sigma_1$$

The Pseudoinverse

If $A = U_1\Sigma_1V^T$ with $\Sigma_1 = \text{diag}(\sigma_1, \dots, \sigma_r, 0, \dots, 0) \in \mathbb{R}^{n\times n}$ then the pseudoinverse A^+ of A is computed as follows:

$$A^+ = V\Sigma^+U^T$$
$$\text{with } \Sigma^+ = (\Sigma_1^+ \ 0) \in \mathbb{R}^{n\times m}, \quad \Sigma_1^+ = \text{diag}(\frac{1}{\sigma_1}, \dots, \frac{1}{\sigma_r}, 0, \dots, 0)$$

The Penrose theorem states that $Y = A^+$ is the only solution to the matrix equations:

$$AYA = A, \quad YAY = Y$$
$$(AY)^T = AY, \quad (YA)^T = YA$$

The pseudoinverse is also useful for the general solution of the linear least squares problem $A\boldsymbol{x} \approx \boldsymbol{b}, A \in \mathbb{R}^{m\times n}$. The general solution is:

$$\boldsymbol{x} = A^+\boldsymbol{b} + (I - A^+A)\boldsymbol{w}, \quad \boldsymbol{w} \text{ arbitrary}$$

or, using the SVD notation for U_r and V_r:

$$\boldsymbol{x} = V_r\Sigma_r^{-1}{U_r}^T\boldsymbol{b} + V_{\text{rest}} \cdot \boldsymbol{c}$$
$$V_{\text{rest}} = [\boldsymbol{v}_{r+1}, \dots, \boldsymbol{v}_n]$$

where we introduced the arbitrary vector $c = V_{\text{rest}}^T\boldsymbol{w}$.
If A has a full rank, i.e. $rank(A) = n$, then the least squares solution is unique:

$$\boldsymbol{x} = A^+\boldsymbol{b}$$

Condition Numbers

The result of a numerical computation is the exact result of a slightly perturbed problem. A problem is well conditioned if the results do not differ too much when solving a perturbed problem.
Consider the problem $A\boldsymbol{x} = \boldsymbol{b}$ for $A \in \mathbb{R}^{n\times n}$ and the perturbed system $(A+\varepsilon B)\boldsymbol{x}(\varepsilon) = \boldsymbol{b}$. Since in general ε is very small, we expand $\boldsymbol{x}(\varepsilon) \approx \boldsymbol{x}(0) + \dot{\boldsymbol{x}}(0)\varepsilon$.

Solving the equations leads us to the formula:

$$\frac{||\boldsymbol{x}(\varepsilon) - \boldsymbol{x}(0)||}{||\boldsymbol{x}(0)||} \approx \underbrace{||A^{-1}|| \cdot ||A||}_{\kappa} \cdot \frac{||\varepsilon B||}{||A||}$$

If we use the 2-norm for the matrices, the following holds:

$$||A|| = \sigma_{max}(A), \quad ||A^{-1}|| = \frac{1}{\sigma_{min}(A)} \Rightarrow \kappa = \frac{\sigma_{max}}{\sigma_{min}}$$

Note that if A is singular, then $\sigma_{min} = 0$ and the condition number κ becomes infinite.
The interpretation of κ is that we can expect the numerical solution to deviate from the real solution in the last κ digits.

We now consider the linear least squares problems

$$||A\boldsymbol{x} - \boldsymbol{b}|| = \min, \quad ||(A+\varepsilon B)\boldsymbol{x}(\varepsilon) - \boldsymbol{b}|| = \min, \quad A, B \in \mathbb{R}^{m\times n}$$

If we compute the solution using the pseudoinverse A^+, we obtain the error formula:

$$\frac{||\boldsymbol{x}(\varepsilon) - \boldsymbol{x}(0)||}{||\boldsymbol{x}(0)||} \approx \kappa \left(2 + \kappa \frac{||\boldsymbol{r}||}{||A||\ ||\boldsymbol{x}||}\right) \frac{||\varepsilon B||}{||A||}$$
$$\text{with } \kappa = ||A|| \cdot ||A^+|| = \frac{\sigma_1(A)}{\sigma_r(A)}$$

As we can see, the residue $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$ is decisive for the error. For a good model ($\boldsymbol{r}$ small) we get an error linear in κ. On the other hand, if we chose a bad model, thus making $\boldsymbol{r}$ larger, the error becomes quadratic in κ.

One can show that if we solve the linear least squares problem with the normal equations $A^TA\boldsymbol{x} = A^T\boldsymbol{b}$, the error will always be quadratic in κ, independantly of our model.

Fundamental Subspaces

Associated with any matrix $A \in \mathbb{R}^{m\times n}$ are the 4 fundamental subspaces $\mathcal{R}(A), \mathcal{R}(A^T), \mathcal{N}(A), \mathcal{N}(A^T)$. The following relations hold:

$$\mathbb{R}^m = \mathcal{R}(A) \oplus \mathcal{N}(A^T)$$
$$\mathbb{R}^n = \mathcal{R}(A^T) \oplus \mathcal{N}(A)$$

With the help of the pseudoinverse we can describe the projectors P on these subspaces:

$$P_{\mathcal{R}(A)} = AA^+, \quad P_{\mathcal{R}(A^T)} = A^+A$$
$$P_{\mathcal{N}(A^T)} = I - AA^+, \quad P_{\mathcal{N}(\mathcal{A})} = I - A^+A$$

Once again applying our notation
$U = [U_r|U_{\text{rest}}], V = [V_r|V_{\text{rest}}]$, we obatain:

$$P_{\mathcal{R}(A)} = U_r{U_r}^T, \quad P_{\mathcal{R}(A^T)} = V_r{V_r}^T$$
$$P_{\mathcal{N}(A^T)} = U_{\text{rest}}U_{\text{rest}}^T, \quad P_{\mathcal{N}(\mathcal{A})} = V_{\text{rest}}{V_{\text{rest}}}^T$$