Numerical Algorithms

Fabian Bosshard

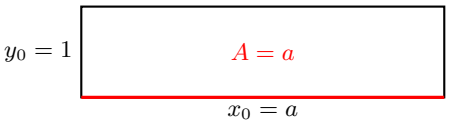
February 6, 2026

Contents

1 Introduction

1.1 Root Finding

Goal: Find $\sqrt{a}$ for $a > 0$.



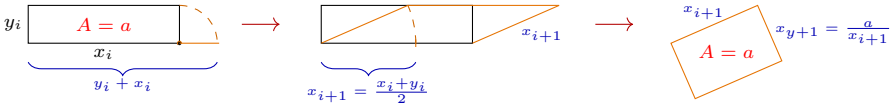
Idea: successively transform conserving area until perfect square.

Straightedge and compass approach:

$$x_{i+1} = \frac{x_i + y_i}{2}$$

Since $A = x_i \cdot y_i = a \implies y_i = \frac{a}{x_i}$ (can always express like this):

$$x_{i+1} = \frac{x_i + \frac{a}{x_i}}{2} = \frac{x_i^2 + a}{2x_i} = x_i - \frac{x_i^2 - a}{2x_i}$$



special case of **Newton's Method**, i.e. finding $x \in \mathbb{R}$ such that $f(x) = 0$:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Remark: $|x - x_i|$ decreases quadratically in i (can be shown using Taylor expansion).

BUT careful: can also fail badly if

- x_0 is too far from correct solution x
- $f'(x_i) \approx 0$ for some x_i
- f' unbounded close to the root x

2 Iterative Methods

2.1 Jacobi Method

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} \right), \qquad i = 1, \ldots, n$$

or in matrix notation:

$$\boldsymbol{x}^{(k+1)} = \underline{\boldsymbol{D}}^{-1} \big(\boldsymbol{b} - \underline{\boldsymbol{L}} \boldsymbol{x}^{(k)} - \underline{\boldsymbol{U}} \boldsymbol{x}^{(k)} \big),$$

where

$$\underline{\boldsymbol{A}} = \underline{\boldsymbol{L}} + \underline{\boldsymbol{D}} + \underline{\boldsymbol{U}} = \begin{bmatrix} 0 & & \\ & \ddots & \\ & & 0 \end{bmatrix} + \begin{bmatrix} a_{11} & & 0 \\ & \ddots & \\ 0 & & a_{nn} \end{bmatrix} + \begin{bmatrix} 0 & & \\ & \ddots & \\ & & 0 \end{bmatrix}$$

Can be improved if always the most recent values of $\boldsymbol{x}$ are used:

2.2 Gauss–Seidel Method

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right)$$

or

$$\boldsymbol{x}^{(k+1)} = \underline{\boldsymbol{D}}^{-1} \big(\boldsymbol{b} - \underline{\boldsymbol{L}} \boldsymbol{x}^{(k+1)} - \underline{\boldsymbol{U}} \boldsymbol{x}^{(k)} \big) \quad \Longleftrightarrow \quad \boldsymbol{x}^{(k+1)} = (\underline{\boldsymbol{D}} + \underline{\boldsymbol{L}})^{-1} \big(\boldsymbol{b} - \underline{\boldsymbol{U}} \boldsymbol{x}^{(k)} \big)$$

Advantages:

- Averaged faster (often)
- Easier to implement, because $x_i^{(k)}$ can simply be overwritten by $x_i^{(k+1)}$

Diagonal dominance:

$$|a_{ii}| \geq \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}|, \qquad i = 1, \ldots, n$$

Both methods are guaranteed to converge to the correct solution if $\underline{\boldsymbol{A}}$ is diagonally dominant.

Cost: For each iteration, the cost is $\mathcal{O}(n^2)$ and only $\mathcal{O}(n)$ if $\underline{\boldsymbol{A}}$ is sparse.

3 Polynomial Interpolation

A polynomial $p : \mathbb{R} \to \mathbb{R}$ of degree at most n is most commonly represented in the monomial basis $\{1, x, x^2, \dots, x^n\}$ as

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

where $a_0, \dots, a_n \in \mathbb{R}$. It can be evaluated in $\mathcal{O}(n)$ operations using Horner's scheme.

Algorithm 1 Horner's scheme

Require: coefficients $a_0, \dots, a_n \in \mathbb{R}$, evaluation point $x \in \mathbb{R}$
Ensure: value $p(x) = \sum_{i=0}^{n} a_i x^i$

```
p ← a_n
for i = n − 1, ..., 0 do
    p ← p · x + a_i
return p
```

3.1 Polynomial Curves

In curve design, we consider polynomial curves $\boldsymbol{P} : \mathbb{R} \to \mathbb{R}^d$ (in applications typically $d \in \{2, 3\}$, but the theory works for arbitrary d). Using the monomial basis, a polynomial curve of degree at most n can be written as

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

where $\boldsymbol{A}_0, \dots, \boldsymbol{A}_n \in \mathbb{R}^d$. It can be evaluated with Horner's scheme by replacing scalar operations with vector operations.

Algorithm 2 Horner's scheme

Require: coefficients $\boldsymbol{A}_0, \dots, \boldsymbol{A}_n \in \mathbb{R}^d$, evaluation point $t \in \mathbb{R}$
Ensure: value $\boldsymbol{P}(t) = \sum_{i=0}^{n} \boldsymbol{A}_i t^i$

```
P ← A_n
for i = n − 1, ..., 0 do
    P ← P · t + A_i
return P
```

Often we are interested in finding an interpolating polynomial curve. Given points $\boldsymbol{P}_0, \dots, \boldsymbol{P}_n \in \mathbb{R}^d$ and distinct parameter values $t_0, \dots, t_n \in \mathbb{R}$, we seek a polynomial curve $\boldsymbol{P}$ of degree at most n such that

$$\boldsymbol{P}(t_i) = \boldsymbol{P}_i$$

where $i = 0, \dots, n$.

Writing $\underline{\boldsymbol{A}} = [\boldsymbol{A}_0, \dots, \boldsymbol{A}_n]^\top \in \mathbb{R}^{(n+1)\times d}$ and $\underline{\boldsymbol{P}} = [\boldsymbol{P}_0, \dots, \boldsymbol{P}_n]^\top \in \mathbb{R}^{(n+1)\times d}$, these conditions yield the linear system

$$\underline{\boldsymbol{V}}\,\underline{\boldsymbol{A}} = \underline{\boldsymbol{P}} \tag{3.1}$$

where $\underline{\boldsymbol{V}} \in \mathbb{R}^{(n+1)\times(n+1)}$ is the Vandermonde matrix

$$\underline{\boldsymbol{V}} = [t_i^j]_{i,j=0}^{n} = \begin{bmatrix} 1 & t_0 & t_0^2 & \cdots & t_0^n \\ 1 & t_1 & t_1^2 & \cdots & t_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & t_n & t_n^2 & \cdots & t_n^n \end{bmatrix}$$

In principle, one can solve a system like (3.1) in $\mathcal{O}(n^3)$, but the special structure of $\underline{\boldsymbol{V}}$ can be used to design an $\mathcal{O}(n^2)$ algorithm. We do not go into the details here, since interpolation can also be done more directly without ever forming the monomial coefficients.

3.2 Neville's Algorithm

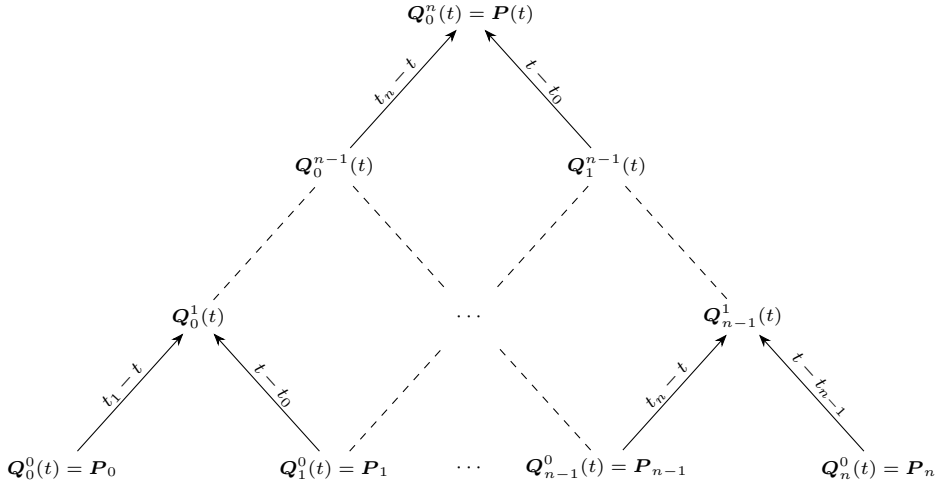
Neville evaluates the interpolating polynomial curve $\boldsymbol{P}(t)$ at a given $t \in \mathbb{R}$ *on-the-fly*. It is based on the following recursive formulation. Define the constant curves

$$\boldsymbol{Q}_i^0(t) \equiv \boldsymbol{P}_i$$

for $i = 0, \ldots, n$. Then define recursively for $j = 1, \ldots, n$,

$$\boldsymbol{Q}_i^j(t) = \frac{t_{i+j} - t}{t_{i+j} - t_i} \boldsymbol{Q}_i^{j-1}(t) + \frac{t - t_i}{t_{i+j} - t_i} \boldsymbol{Q}_{i+1}^{j-1}(t)$$

for $i = 0, \ldots, n - j$. Lower degree curves are linearly blended to form higher degree curves, using affine combinations. The recursion can be visualized as follows:



Note that the common denominators of the affine weights are omitted for clarity. They are always obtained by summing the two weights, i.e. $t_{i+j} - t_i = (t_{i+j} - t) + (t - t_i)$.

$\boldsymbol{Q}_0^n(t)$ is the interpolating polynomial curve evaluated at t. The method costs $\mathcal{O}(n^2)$ arithmetic operations.

Algorithm 3 Neville

Require: data $(t_i, \boldsymbol{P}_i)$ for $i = 0, \ldots, n$ with distinct t_i, evaluation parameter $t \in \mathbb{R}$
Ensure: value $\boldsymbol{P}(t)$ of the interpolating polynomial curve

```
for i = 0, ..., n do
    Q_i ← P_i                                   ▷ Q_i stores Q_i^j in-place
for j = 1, ..., n do
    for i = 0, ..., n − j do
        Q_i ← ((t_{i+j} − t)/(t_{i+j} − t_i)) Q_i + ((t − t_i)/(t_{i+j} − t_i)) Q_{i+1}
return Q_0
```

3.3 Lagrange Form

From Neville's algorithm, we can derive an explicit representation of the interpolating polynomial, namely the Lagrange form

$$\boldsymbol{P}(t) = \sum_{i=0}^{n} \boldsymbol{P}_i \, L_i^n(t) \tag{3.2}$$

with Lagrange basis functions

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

where $i = 0, \ldots, n$. The interpolation points $\boldsymbol{P}_0, \ldots, \boldsymbol{P}_n$ thus act as coefficients with respect to these Lagrange basis functions of degree n for parameter values $t_0, \ldots, t_n$.

Like the monomials, the L_i^n form a basis of Π_n. They satisfy two important properties:

- Lagrange property: $L_i^n(t_j) = 0$ for $j \neq i$ and $L_i^n(t_i) = 1$
- Partition of unity: $\sum_{i=0}^n L_i^n(t) \equiv 1$

The first property implies immediately that $\boldsymbol{P}(t_i) = \boldsymbol{P}_i$.

3.4 Barycentric Form

Define $L(t) = \prod_{i=0}^n (t - t_i)$ and

$$w_i = \prod_{\substack{j=0 \\ j \neq i}}^n \frac{1}{t_i - t_j}$$

where $i = 0, \ldots, n$. Then the interpolant (3.2) can be written as

$$\boldsymbol{P}(t) = L(t) \sum_{i=0}^n \frac{w_i}{t - t_i} \boldsymbol{P}_i$$

and since $L(t) \sum_{i=0}^n \frac{w_i}{t-t_i} = \sum_{i=0}^n L_i^n(t) \equiv 1$, as

$$\boldsymbol{P}(t) = \frac{\sum_{i=0}^n \frac{w_i}{t-t_i} \boldsymbol{P}_i}{\sum_{i=0}^n \frac{w_i}{t-t_i}} \tag{3.3}$$

This barycentric form is typically very stable in floating point arithmetic and yields an $\mathcal{O}(n)$ evaluation (Algorithm 4) once the weights w_i are known. To compute them, it takes $\mathcal{O}(n^2)$ operations. Since they depend only on the nodes t_i, they can be precomputed and reused for different data $\boldsymbol{P}_i$.

Algorithm 4 Barycentric evaluation of the interpolant

Require: nodes $t_0, \ldots, t_n$ (distinct), data $\boldsymbol{P}_0, \ldots, \boldsymbol{P}_n \in \mathbb{R}^d$, weights $w_0, \ldots, w_n$, $t \in \mathbb{R}$
Ensure: value $\boldsymbol{P}(t)$

```
N ← 0_d, D ← 0
for i = 0, ..., n do
    if t = t_i then
        return P_i
    else
        W ← w_i / (t − t_i)
        N ← N + W P_i
        D ← D + W
return N/D
```

3.5 Newton Form and Divided Differences

Another representation to write the interpolant (3.2) is the Newton form

$$\boldsymbol{P}(t) = \sum_{i=0}^n \boldsymbol{B}_i N_i(t) \tag{3.4}$$

with Newton basis functions

$$N_i(t) = \prod_{j=0}^{i-1} (t - t_j)$$

where $i = 0, \ldots, n$. The Newton basis functions also form a basis of Π_n. The coefficients $\boldsymbol{B}_i = \boldsymbol{P}[t_0, \ldots, t_i] \in \mathbb{R}^d$ are the i^{th} order *divided differences* of the interpolation points $\boldsymbol{P}_i$ with respect to the parameter values t_i and result from the recursive definition

$$\begin{aligned} \boldsymbol{P}[t_i] &= \boldsymbol{P}_i && i = 0, \ldots, n \\ \boldsymbol{P}[t_i, \ldots, t_{i+j}] &= \frac{\boldsymbol{P}[t_{i+1}, \ldots, t_{i+j}] - \boldsymbol{P}[t_i, \ldots, t_{i+j-1}]}{t_{i+j} - t_i} && i = 0, \ldots, n-j, \quad j = 1, \ldots, n \end{aligned}$$

As for the Barycentric Form, the coefficients $\boldsymbol{B}_i$ can be computed in $\mathcal{O}(n^2)$ operations as a preprocessing step, and the evaluation then requires $\mathcal{O}(n)$ operations with a modified Horner scheme (Algorithm 5).

Algorithm 5 modified Horner scheme to evaluate Newton form

Require: coefficients $\boldsymbol{B}_0, \ldots, \boldsymbol{B}_n \in \mathbb{R}^d$, nodes $t_0, \ldots, t_{n-1}$, evaluation parameter $t \in \mathbb{R}$
Ensure: value $\boldsymbol{P}(t) = \sum_{i=0}^{n} \boldsymbol{B}_i N_i(t)$

```
P ← B_n
for i = n − 1, ..., 0 do
    P ← P · (t − t_i) + B_i
return P
```

4 Bézier Curves and Splines

Polynomial interpolation is often unsuitable for curve design:

- monomial/Newton coefficients do not correspond intuitively to shape,
- the interpolant can be unstable for (nearly) equidistant parameters,
- changing one data point may affect the curve globally.

These issues motivate representing polynomial curves in the *Bernstein basis*.

4.1 Bézier Curves

A Bézier curve $\boldsymbol{F} : [a, b] \to \mathbb{R}^d$ of degree n is defined by control points $\boldsymbol{C}_0, \ldots, \boldsymbol{C}_n \in \mathbb{R}^d$. It can be evaluated at $t \in [a, b]$ by repeatedly computing convex combinations (de Casteljau):

$$\boldsymbol{C}_i^0(t) = \boldsymbol{C}_i$$

where $i = 0, \ldots, n$ and for $j = 1, \ldots, n$,

$$\boldsymbol{C}_i^j(t) = \frac{b-t}{b-a}\,\boldsymbol{C}_i^{j-1}(t) + \frac{t-a}{b-a}\,\boldsymbol{C}_{i+1}^{j-1}(t)$$

where $i = 0, \ldots, n-j$ and then $\boldsymbol{F}(t) = \boldsymbol{C}_0^n(t)$.

Algorithm 6 de Casteljau

Require: control points $\boldsymbol{C}_0, \ldots, \boldsymbol{C}_n \in \mathbb{R}^d$, interval $[a, b]$, parameter $t \in [a, b]$
Ensure: value $\boldsymbol{F}(t)$
1: $\alpha \leftarrow \frac{b-t}{b-a}$, $\beta \leftarrow 1 - \alpha$
2: **for** $i = 0, \ldots, n$ **do**
3: $\quad \boldsymbol{D}_i \leftarrow \boldsymbol{C}_i$
4: **for** $j = 1, \ldots, n$ **do**
5: $\quad$ **for** $i = 0, \ldots, n-j$ **do**
6: $\quad\quad \boldsymbol{D}_i \leftarrow \alpha\,\boldsymbol{D}_i + \beta\,\boldsymbol{D}_{i+1}$
7: **return** $\boldsymbol{D}_0$

Bernstein form. With the affine reparameterization $s(t) = \frac{t-a}{b-a} \in [0, 1]$, the same curve can be written explicitly as

$$\boldsymbol{F}(t) = \sum_{i=0}^{n} \boldsymbol{C}_i\, B_i^n(s(t)) \tag{4.1}$$

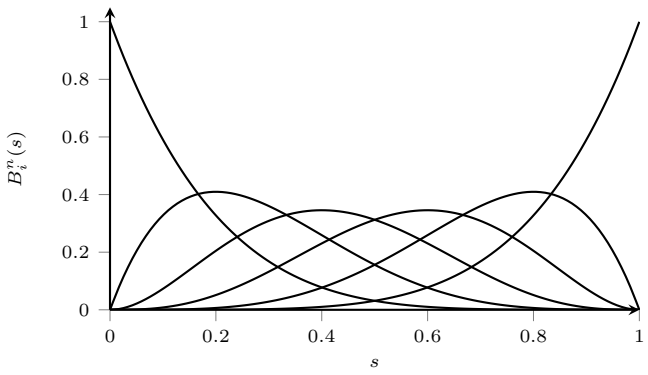
where $t \in [a, b]$ and the Bernstein polynomials are

$$B_i^n(s) = \binom{n}{i} s^i (1-s)^{n-i} \tag{4.2}$$

where $i = 0, \ldots, n$ and

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

is the binomial coefficient.



Fact 4.1. Besides forming a basis for the space of polynomials of degree at most n, the Bernstein polynomials satisfy the following properties:

- Partition of unity: $\sum_{i=0}^{n} B_i^n(s) \equiv 1$
- Positivity: $B_i^n(s) \geq 0$ for $s \in [0,1]$
- Endpoints: $B_i^n(0) = \delta_{i,0}$ and $B_i^n(1) = \delta_{i,n}$ (where $\delta_{i,j}$ is the Kronecker delta)
- Symmetry: $B_i^n(s) = B_{n-i}^n(1-s)$
- Recursion formula: $B_i^n(s) = sB_{i-1}^{n-1}(s) + (1-s)B_i^{n-1}(s)$ with $B_i^n = 0$ for $i < 0$ or $i > n$
- Derivative: $\big(B_i^n\big)'(s) = n\big(B_{i-1}^{n-1}(s) - B_i^{n-1}(s)\big)$
- Locality: B_i^n has a local maximum at $s = i/n$

These properties almost immediately "translate" to corresponding properties of Bézier curves:

- Affine invariance: $\phi\big(\sum_{i=0}^{n} \boldsymbol{C}_i B_i^n(s)\big) = \sum_{i=0}^{n} \phi(\boldsymbol{C}_i) B_i^n(s)$ for any affine mapping $\phi : \mathbb{R}^d \to \mathbb{R}^d$
- Convex hull: $\boldsymbol{F}(t) \in \operatorname{conv}\{\boldsymbol{C}_0, \ldots, \boldsymbol{C}_n\}$
- Endpoint interpolation: $\boldsymbol{F}(a) = \boldsymbol{C}_0$ and $\boldsymbol{F}(b) = \boldsymbol{C}_n$
- Derivative: $\boldsymbol{F}'(t) = \frac{n}{b-a} \sum_{i=0}^{n-1} (\boldsymbol{C}_{i+1} - \boldsymbol{C}_i) B_i^{n-1}(s)$
- Endpoint tangent: $\boldsymbol{F}'(a) = \frac{n}{b-a}(\boldsymbol{C}_1 - \boldsymbol{C}_0)$ and $\boldsymbol{F}'(b) = \frac{n}{b-a}(\boldsymbol{C}_n - \boldsymbol{C}_{n-1})$
- Symmetry: $\sum_{i=0}^{n} \boldsymbol{C}_i B_i^n(s) = \sum_{i=0}^{n} \boldsymbol{C}_{n-i} B_i^n(1-s)$
- Local control: $\boldsymbol{C}_i$ influences $\boldsymbol{F}$ the most at $t = a + i(b-a)/n$

In $\mathbb{R}^2$, Bézier curves satisfy the *variation diminishing property*: any line intersects the curve $\boldsymbol{F}$ no more often than it intersects its control polygon. Therefore, $\boldsymbol{F}$ cannot exhibit unexpected oscillations. ◁

Remark 4.1. Recall the binomial theorem: $(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i}$ ◀

Another property is

$$\begin{aligned}
\sum_{i=0}^{n} \frac{i}{n} B_i^n(s) &= \sum_{i=0}^{n} \frac{i}{n} \frac{n!}{i!(n-i)!} s^i (1-s)^{n-i} \\
&= s \sum_{i=1}^{n} \frac{(n-1)!}{(i-1)!(n-i)!} s^{i-1} (1-s)^{n-i} \\
&= s \sum_{i=1}^{n} \frac{(n-1)!}{(i-1)!((n-1)-(i-1))!} s^{i-1} (1-s)^{(n-1)-(i-1)} \\
&= s \sum_{j=0}^{n-1} \binom{n-1}{j} s^j (1-s)^{(n-1)-j} \\
&= s \sum_{j=0}^{n-1} B_j^{n-1}(s) \\
&= s
\end{aligned}$$

Theorem 4.2 (Degree elevation of Bézier curves)**.** Given a Bézier curve

$$\boldsymbol{F}(t) = \sum_{i=0}^{n} \boldsymbol{C}_i \, B_i^n(s) \tag{4.3}$$

of degree n with control points $\boldsymbol{C}_0, \ldots, \boldsymbol{C}_n \in \mathbb{R}^d$. Then the same curve admits a representation of degree $n+1$,

$$\boldsymbol{F}(t) = \sum_{i=0}^{n+1} \boldsymbol{C}_i^* \, B_i^{n+1}(s) \tag{4.4}$$

with control points $\boldsymbol{C}_0^* = \boldsymbol{C}_0$, $\boldsymbol{C}_i^* = \frac{i}{n+1}\boldsymbol{C}_{i-1} + \frac{n+1-i}{n+1}\boldsymbol{C}_i$ for $i = 1, \ldots, n$, $\boldsymbol{C}_{n+1}^* = \boldsymbol{C}_n$. ◁

Proof. Using (4.2) and the binomial identities

$$\binom{n}{i} = \frac{n+1-i}{n+1}\binom{n+1}{i} \qquad \binom{n}{i} = \frac{i+1}{n+1}\binom{n+1}{i+1}$$

we obtain the two relations

$$(1-s)B_i^n(s) = \frac{n+1-i}{n+1}\,B_i^{n+1}(s) \qquad sB_i^n(s) = \frac{i+1}{n+1}\,B_{i+1}^{n+1}(s)$$

Hence

$$B_i^n(s) = (1-s)B_i^n(s) + sB_i^n(s) = \frac{n+1-i}{n+1}B_i^{n+1}(s) + \frac{i+1}{n+1}B_{i+1}^{n+1}(s)$$

Insert into (4.3):

$$\boldsymbol{F}(t) = \sum_{i=0}^{n} \frac{n+1-i}{n+1}\boldsymbol{C}_i\,B_i^{n+1}(s) + \sum_{i=0}^{n} \frac{i+1}{n+1}\boldsymbol{C}_i\,B_{i+1}^{n+1}(s)$$

Reindex the second sum by $j = i+1$:

$$\boldsymbol{F}(t) = \sum_{i=0}^{n} \frac{n+1-i}{n+1}\boldsymbol{C}_i\,B_i^{n+1}(s) + \sum_{j=1}^{n+1} \frac{j}{n+1}\boldsymbol{C}_{j-1}\,B_j^{n+1}(s)$$

Now collect the coefficients of $B_k^{n+1}(s)$. For $k=0$ and $k=n+1$ we obtain $\boldsymbol{C}_0^* = \boldsymbol{C}_0$ and $\boldsymbol{C}_{n+1}^* = \boldsymbol{C}_n$. For $k = 1, \ldots, n$, both sums contribute and yield

$$\boldsymbol{C}_k^* = \frac{k}{n+1}\boldsymbol{C}_{k-1} + \frac{n+1-k}{n+1}\boldsymbol{C}_k$$

which proves the claim. □

4.2 Bézier Splines

High-degree Bézier curves are rarely used for designing complex curves for 2 reasons: $\mathcal{O}(n^2)$ evaluation with Algorithm 6 becomes slower and local control is not very effective for large n, so the shape becomes difficult to control. Instead, one typically joins several low-degree Bézier segments into a spline.

Let $\boldsymbol{F}_0 : [t_0, t_1] \to \mathbb{R}^d$ and $\boldsymbol{F}_1 : [t_1, t_2] \to \mathbb{R}^d$ be Bézier curves of the same degree n with control points $\boldsymbol{C}_{0,0}, \ldots, \boldsymbol{C}_{0,n}$ and $\boldsymbol{C}_{1,0}, \ldots, \boldsymbol{C}_{1,n}$, respectively.

They are called C^k-continuous at t_1 if $\boldsymbol{F}_0(t_1) = \boldsymbol{F}_1(t_1)$, $\boldsymbol{F}_0'(t_1) = \boldsymbol{F}_1'(t_1)$, …, $\boldsymbol{F}_0^{(k)}(t_1) = \boldsymbol{F}_1^{(k)}(t_1)$.

There is also the weaker notion of *geometric continuity*. The two curves $\boldsymbol{F}_0$ and $\boldsymbol{F}_1$ are called G^1-continuous at t_1 if $\boldsymbol{F}_0(t_1) = \boldsymbol{F}_1(t_1)$ and $\boldsymbol{F}_0'(t_1) \parallel \boldsymbol{F}_1'(t_1)$, i.e. the tangent directions agree, but their magnitudes may differ. Clearly, C^1-continuity implies G^1-continuity, but not conversely.

Example 4.2 (Curve design). In graphic editors, curves are often represented as piecewise cubic Bézier curves. At a join point P,

- C^{-1}: the segments do not share the same point P (gap)
- C^0: the segments share the same point P (corner allowed)
- G^1: the two Bézier handles at P are collinear (smooth join)
- C^1: the handles are collinear and of equal length (matched derivatives)

Thus, G^1-continuity corresponds to a visually smooth join, while C^1-continuity additionally enforces equal parametrization speed. ◀

4.2.1 C^0-continuity

By the endpoint interpolation property from Fact 4.1 we have

$$\boldsymbol{F}_0(t_1) = \boldsymbol{C}_{0,n}$$
$$\boldsymbol{F}_1(t_1) = \boldsymbol{C}_{1,0}$$

so the 2 curves join continuously at t_1 iff $\boldsymbol{C}_{0,n} = \boldsymbol{C}_{1,0}$.

4.2.2 C^1-continuity

Similarly, by the endpoint tangent property from Fact 4.1, we have

$$\boldsymbol{F}_0'(t_1) = \frac{n}{t_1 - t_0}(\boldsymbol{C}_{0,n} - \boldsymbol{C}_{0,n-1})$$
$$\boldsymbol{F}_1'(t_1) = \frac{n}{t_2 - t_1}(\boldsymbol{C}_{1,1} - \boldsymbol{C}_{1,0})$$

so the 2 curves join with C^1-continuity at t_1 iff $\boldsymbol{C}_{0,n-1}$, $\boldsymbol{C}_{0,n} = \boldsymbol{C}_{1,0}$, $\boldsymbol{C}_{1,1}$ lie on a common line and are an affine image of the parameter values t_0, t_1, t_2, i.e.,

$$\boldsymbol{C}_{0,n} = \boldsymbol{C}_{1,0} = (1-\lambda)\boldsymbol{C}_{0,n-1} + \lambda\,\boldsymbol{C}_{1,1}$$

with $\lambda = \frac{t_1 - t_0}{t_2 - t_0}$ or, equivalently, $t_1 = (1-\lambda)t_0 + \lambda t_2$.

These 2 observations can be used to construct the simplest kind of Bézier splines, see Example 4.3.

Example 4.3 (Cubic C^1-continuous Bézier splines)**.** Given interpolation points $\boldsymbol{P}_0, \ldots, \boldsymbol{P}_m \in \mathbb{R}^d$ at parameters $t_0 < \cdots < t_m$ and tangents $\boldsymbol{T}_0, \ldots, \boldsymbol{T}_m \in \mathbb{R}^d$, define m cubic Bézier pieces $\boldsymbol{F}_i : [t_i, t_{i+1}] \to \mathbb{R}^d$ by control points

$$\boldsymbol{C}_{i,0} = \boldsymbol{P}_i$$
$$\boldsymbol{C}_{i,1} = \boldsymbol{P}_i + \frac{t_{i+1} - t_i}{3}\,\boldsymbol{T}_i$$
$$\boldsymbol{C}_{i,2} = \boldsymbol{P}_{i+1} - \frac{t_{i+1} - t_i}{3}\,\boldsymbol{T}_{i+1}$$
$$\boldsymbol{C}_{i,3} = \boldsymbol{P}_{i+1}$$

where $i = 0, \ldots, m-1$. Then $\boldsymbol{F}$ interpolates both positions and tangents, and is C^1 at the knots. This is called a *cubic Hermite spline*.

Automatic tangent choices. If tangents are not given, they can be generated from the interpolation points. For the interior tangents $i = 1, \ldots, m-1$, 2 common choices are:

- **FMILL:** ($\curvearrowright$ milling devices) $\boldsymbol{T}_i = \frac{\boldsymbol{P}_{i+1} - \boldsymbol{P}_{i-1}}{t_{i+1} - t_{i-1}}$ (finite difference)
- **Bessel:** $\boldsymbol{T}_i = \boldsymbol{Q}_i'(t_i)$, $\boldsymbol{Q}_i$ is the quadratic interpolant to $\boldsymbol{P}_{i-1}$, $\boldsymbol{P}_i$, $\boldsymbol{P}_{i+1}$ at t_{i-1}, t_i, t_{i+1}

Endpoint tangents can be chosen via:

- **Overhauser:** $\boldsymbol{T}_0 = \boldsymbol{Q}_1'(t_0)$, $\boldsymbol{T}_m = \boldsymbol{Q}_{m-1}'(t_m)$
- **Natural end:** $\boldsymbol{F}_0''(t_0) = \boldsymbol{F}_{m-1}''(t_m) = \boldsymbol{0} \Rightarrow \boldsymbol{C}_{0,1} = \frac{\boldsymbol{C}_{0,0} + \boldsymbol{C}_{0,2}}{2}$, $\boldsymbol{C}_{m-1,2} = \frac{\boldsymbol{C}_{m-1,1} + \boldsymbol{C}_{m-1,3}}{2}$

For closed curves ($\boldsymbol{P}_0 = \boldsymbol{P}_m$), endpoints are treated like interior points with the two choices above. ◀

4.2.3 C^2 continuity

The two curves $\boldsymbol{F}_0$ and $\boldsymbol{F}_1$ join with C^2-continuity at t_1, if, in addition to the conditions from Section 4.2.2, there exists a point $\boldsymbol{D} \in \mathbb{R}^d$ such that

$$\boldsymbol{C}_{0,n-1} = (1-\lambda)\boldsymbol{C}_{0,n-2} + \lambda\boldsymbol{D}$$
$$\boldsymbol{C}_{1,1} = (1-\lambda)\boldsymbol{D} + \lambda\boldsymbol{C}_{1,2}$$

with the same $\lambda = \frac{t_1 - t_0}{t_2 - t_0}$. This is called the *A-frame condition* and guarantees that $\boldsymbol{C}_{0,n-2}$, $\boldsymbol{C}_{0,n-1}$, $\boldsymbol{D}$, as well as $\boldsymbol{D}$, $\boldsymbol{C}_{1,1}$, $\boldsymbol{C}_{1,2}$ lie on a common line and are affine images of the parameter values t_0, t_1, t_2.

Example 4.4 (Cubic C^2-continuous Bézier splines)**.** Let $\boldsymbol{P}_0, \ldots, \boldsymbol{P}_m \in \mathbb{R}^d$ be given at parameters $t_0 < \cdots < t_m$. Define cubic Bézier pieces $\boldsymbol{F}_i : [t_i, t_{i+1}] \to \mathbb{R}^d$ with $\boldsymbol{C}_{i,0} = \boldsymbol{P}_i$, $\boldsymbol{C}_{i,3} = \boldsymbol{P}_{i+1}$. Choose the two interior control points $\boldsymbol{C}_{0,1}$ and $\boldsymbol{C}_{0,2}$ (or symmetrically $\boldsymbol{C}_{m-1,1}$ and $\boldsymbol{C}_{m-1,2}$) arbitrarily. Then imposing C^1 continuity and the A-frame condition at all interior knots uniquely determines all remaining control points. Hence a cubic C^2-continuous Bézier spline interpolating the data has exactly two degrees of freedom. ◀

4.3 B-Splines

The difficulty of controlling the shape of a Cubic C^2-continuous Bézier splines motivates B-splines, which are piecewise polynomial curves with built-in maximal smoothness and intuitive local control.

4.3.1 Definition and evaluation

A B-spline curve $\boldsymbol{F}$ of degree n with $m+1$ pieces is defined by control points $\boldsymbol{D}_0, \dots, \boldsymbol{D}_{n+m} \in \mathbb{R}^d$ and knots

$$\underbrace{t_0 \le \cdots \le t_{n-1}}_{n \text{ knots}} \le \underbrace{t_n \le \cdots \le t_{n+m+1}}_{m+2 \text{ knots}} \le \underbrace{t_{n+m+2} \le \cdots \le t_{2n+m+1}}_{n \text{ knots}}$$

i.e. a non-decreasing *knot vector* of length $2n+m+2$. The curve domain is between the $m+2$ "internal" knots, i.e. $\boldsymbol{F} : [t_n, t_{n+m+1}] \to \mathbb{R}^d$. To evaluate $\boldsymbol{F}(t)$ for $t \in [t_n, t_{n+m+1}]$, first find k such that $t_{n+k} \le t \le t_{n+k+1}$. Then repeatedly compute convex combinations:

$$\boldsymbol{D}_i^0(t) = \boldsymbol{D}_i$$

where $i = k, \dots, k+n$, and for $j = 1, \dots, n$,

$$\boldsymbol{D}_i^j(t) = \frac{t_{n+i+1} - t}{t_{n+i+1} - t_{i+j}} \boldsymbol{D}_i^{j-1}(t) + \frac{t - t_{i+j}}{t_{n+i+1} - t_{i+j}} \boldsymbol{D}_{i+1}^{j-1}(t)$$

where $i = k, \dots, k+n-j$ and then finally set $\boldsymbol{F}(t) = \boldsymbol{D}_k^n(t)$. This is the de Boor algorithm; it is the B-spline analogue of Neville and de Casteljau.

Like Neville's algorithm and de Casteljau's algorithm, de Boor's algorithm costs $\mathcal{O}(n^2)$, but since here n is typically small (often $2 \le n \le 5$), this is not an issue.

Algorithm 7 de Boor

Require: control points $\boldsymbol{D}_0, \dots, \boldsymbol{D}_{n+m} \in \mathbb{R}^d$, knots $t_0, \dots, t_{2n+m+1}$, parameter $t \in [t_n, t_{n+m+1}]$
Ensure: value $\boldsymbol{F}(t)$

```
k ← 0
while t_{n+k+1} < t do
    k ← k + 1
for i = 0, ..., n do
    E_i ← D_{i+k}
for j = 1, ..., n do
    for i = 0, ..., n − j do
        E_i ← ((t_{n+i+k+1} − t) / (t_{n+i+k+1} − t_{i+k+j})) E_i + ((t − t_{i+k+j}) / (t_{n+i+k+1} − t_{i+k+j})) E_{i+1}
return E_0
```

The curve $\boldsymbol{F}$ is polynomial of degree n on each interval $[t_{n+k}, t_{n+k+1}]$ and C^{n-1} at interior knots if knots are simple (i.e. not repeated).

4.3.2 B-spline basis functions

A B-spline curve can be written explicitly as

$$\boldsymbol{F}(t) = \sum_{i=0}^{n+m} \boldsymbol{D}_i \, N_i^n(t) \tag{4.5}$$

where $t \in [t_n, t_{n+m+1}]$. N_i^n, $i = 0, \dots, n+m$, are the *B-spline basis functions* of degree n with respect to the knots $t_0, \dots, t_{2n+m+1}$.

They are defined recursively as follows. Define the piecewise constant functions

$$N_i^0(t) = \chi_{[t_i, t_{i+1})}(t) := \begin{cases} 1 & t \in [t_i, t_{i+1}) \\ 0 & \text{otherwise} \end{cases}$$

for $i = 0, \dots, 2n+m$. Then define recursively for $j = 1, \dots, n$

$$N_i^j(t) = \frac{t - t_i}{t_{i+j} - t_i} N_i^{j-1}(t) + \frac{t_{i+j+1} - t}{t_{i+j+1} - t_{i+1}} N_{i+1}^{j-1}(t)$$

for $i = 0, \dots, 2n+m-j$ with the convention that terms with zero denominators are treated as zero.

Fact 4.3. The functions N_i^j are piecewise polynomials of degree j and C^{j-1}-continuous. They satisfy

- Partition of unity: $\sum_{i=0}^{2n+m-j} N_i^j(t) = 1$ for $t \in [t_j, t_{2n-j+m+1}]$
- Positivity: $N_i^j(t) > 0$ for $t \in (t_i, t_{i+j+1})$

- Local support: $N_i^j(t) = 0$ for $t \notin [t_i, t_{i+j+1}]$
- Derivative: $\big(N_i^j\big)'(t) = \frac{j}{t_{i+j}-t_i} N_i^{j-1}(t) - \frac{j}{t_{i+j+1}-t_{i+1}} N_{i+1}^{j-1}(t)$

These translate to B-spline curve properties. For $t \in [t_{n+k}, t_{n+k+1}]$

- Convex hull: $\boldsymbol{F}(t) \in \operatorname{conv}\{\boldsymbol{D}_k, \dots, \boldsymbol{D}_{k+n}\}$
- Local control: $\boldsymbol{F}(t)$ depends only on $\boldsymbol{D}_k, \dots, \boldsymbol{D}_{k+n}$
- Derivative: $\boldsymbol{F}'(t) = \sum_{i=1}^{n+m} \boldsymbol{D}'_i N_i^{n-1}(t)$ with $\boldsymbol{D}'_i = \frac{n}{t_{i+n}-t_i}\big(\boldsymbol{D}_i - \boldsymbol{D}_{i-1}\big)$
- Affine invariance: $\phi\big(\sum_{i=0}^{n+m} \boldsymbol{D}_i N_i^n(t)\big) = \sum_{i=0}^{n+m} \phi(\boldsymbol{D}_i) N_i^n(t)$ for any affine $\phi : \mathbb{R}^d \to \mathbb{R}^d$

As for Bézier curves, B-spline curves also satisfy the variation diminishing property in $\mathbb{R}^2$. ◁

4.3.3 Interpolation

Given points $\boldsymbol{P}_0, \dots, \boldsymbol{P}_{m+1} \in \mathbb{R}^d$, a standard approach is to interpolate them at the knots $t_n, \dots, t_{n+m+1}$, that is

$$\boldsymbol{F}(t_{n+i}) = \boldsymbol{P}_i$$

for $i = 0, \dots, m+1$. Using (4.5) this yields the linear system

$$\underline{\boldsymbol{M}}\boldsymbol{D} = \boldsymbol{P}$$

with unknown control points $\boldsymbol{D} = [\boldsymbol{D}_0, \dots, \boldsymbol{D}_{n+m}]^\top$, interpolation points $\boldsymbol{P} = [\boldsymbol{P}_0, \dots, \boldsymbol{P}_{m+1}]^\top$, and

$$\underline{\boldsymbol{M}} = \begin{bmatrix} N_0^n(t_n) & N_1^n(t_n) & \cdots & N_{n+m}^n(t_n) \\ N_0^n(t_{n+1}) & N_1^n(t_{n+1}) & \cdots & N_{n+m}^n(t_{n+1}) \\ \vdots & \vdots & \ddots & \vdots \\ N_0^n(t_{n+m+1}) & N_1^n(t_{n+m+1}) & \cdots & N_{n+m}^n(t_{n+m+1}) \end{bmatrix} \in \mathbb{R}^{(m+2)\times(n+m+1)}$$

This system is underdetermined with $n-1$ degrees of freedom. Hence, additional conditions are needed to get a unique solution. Moreover, due to local support, $\underline{\boldsymbol{M}}$ is a band matrix with non-zero entries only near the diagonal.

Example 4.5 (Cubic B-spline interpolation)**.** For cubic interpolation, $n = 3$, the matrix is tridiagonal up to two additional boundary rows. The two free parameters correspond to the two degrees of freedom in Example 4.4. But they can be fixed more conveniently via end conditions! ☺

Common choices are

- **Natural:** $\boldsymbol{F}''(t_3) = \boldsymbol{F}''(t_{m+4}) = \boldsymbol{0}$
- **Clamped:** $\boldsymbol{F}'(t_3) = \boldsymbol{T}_0$ and $\boldsymbol{F}'(t_{m+4}) = \boldsymbol{T}_{m+1}$
- **Periodic:** $\boldsymbol{F}'(t_3) = \boldsymbol{F}'(t_{m+4})$ and $\boldsymbol{F}''(t_3) = \boldsymbol{F}''(t_{m+4})$
- **Not-a-knot:** $\lim\limits_{t \nearrow t_4} \boldsymbol{F}'''(t) = \lim\limits_{t \searrow t_4} \boldsymbol{F}'''(t)$ and $\lim\limits_{t \nearrow t_{m+3}} \boldsymbol{F}'''(t) = \lim\limits_{t \searrow t_{m+3}} \boldsymbol{F}'''(t)$

These conditions can be expressed in terms of basis functions and control points and added as the first and last row, turning the system into a square invertible system that uniquely determines the control points.

In the functional setting, $d = 1$, the natural interpolating cubic B-spline minimizes the bending energy

$$\|f''\|_2^2 = \int_{t_3}^{t_{m+4}} \big(f''(t)\big)^2 \,\mathrm{d}t$$

among all interpolating C^2 functions. ◀

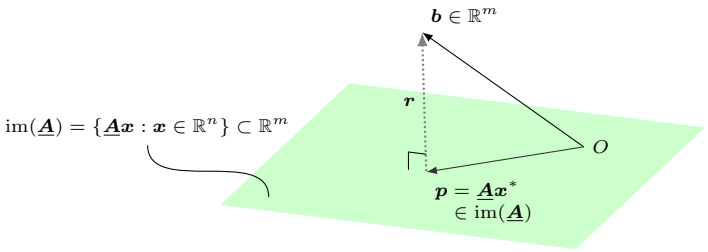
5 Linear Least Squares

Let $\underline{\boldsymbol{A}} \in \mathbb{R}^{m \times n}$, $\boldsymbol{b} \in \mathbb{R}^m$ and consider the system of equations

$$\underline{\boldsymbol{A}}\boldsymbol{x} = \boldsymbol{b} \tag{5.1}$$

with $m > n$ (overdetermined system). If $\boldsymbol{b}$ does not happen to lie in the hyperplane spanned by the columns of $\underline{\boldsymbol{A}}$, i.e. $\boldsymbol{b} \notin \mathrm{im}(\underline{\boldsymbol{A}})$, then there is no solution, i.e. it is inconsitent.

For the optimal solution, we require that $\boldsymbol{p} = \underline{\boldsymbol{A}}\boldsymbol{x}^* \in \mathrm{im}(\underline{\boldsymbol{A}})$ is as close as possible to $\boldsymbol{b}$, which means that the residual $\boldsymbol{r} = \boldsymbol{b} - \underline{\boldsymbol{A}}\boldsymbol{x}^*$ is perpendicular to $\mathrm{im}(\underline{\boldsymbol{A}})$:



Note that $\boldsymbol{p} \in \mathrm{im}(\underline{\boldsymbol{A}})$ iff $\boldsymbol{p} = \underline{\boldsymbol{A}}\boldsymbol{x}^*$ for some $\boldsymbol{x}^* \in \mathbb{R}^n$ and that

$$\begin{aligned}
\boldsymbol{b} - \boldsymbol{p} \text{ perpendicular to } \mathrm{im}(\underline{\boldsymbol{A}}) &\iff (\boldsymbol{b} - \underline{\boldsymbol{A}}\boldsymbol{x}^*) \perp \{\underline{\boldsymbol{A}}\boldsymbol{x} : \boldsymbol{x} \in \mathbb{R}^n\} \\
&\iff (\underline{\boldsymbol{A}}\boldsymbol{x})^\top (\boldsymbol{b} - \underline{\boldsymbol{A}}\boldsymbol{x}^*) = 0 \quad \forall \boldsymbol{x} \in \mathbb{R}^n \\
&\iff \boldsymbol{x}^\top \underline{\boldsymbol{A}}^\top (\boldsymbol{b} - \underline{\boldsymbol{A}}\boldsymbol{x}^*) = 0 \quad \forall \boldsymbol{x} \in \mathbb{R}^n \\
&\iff \underline{\boldsymbol{A}}^\top (\boldsymbol{b} - \underline{\boldsymbol{A}}\boldsymbol{x}^*) = \boldsymbol{0}_n \\
&\iff \underline{\boldsymbol{A}}^\top \boldsymbol{b} - \underline{\boldsymbol{A}}^\top \underline{\boldsymbol{A}}\boldsymbol{x}^* = \boldsymbol{0}_n
\end{aligned}$$

implying that the least-squares solution satisfies the normal equations

$$\boxed{\underline{\boldsymbol{A}}^\top \underline{\boldsymbol{A}}\boldsymbol{x}^* = \underline{\boldsymbol{A}}^\top \boldsymbol{b}} \tag{5.2}$$

The residual $\boldsymbol{r} = \boldsymbol{b} - \underline{\boldsymbol{A}}\boldsymbol{x}^*$ is usually measured in 3 ways:

- 2-norm: $\|\boldsymbol{r}\|_2 = \sqrt{\sum_{i=1}^m r_i^2}$
- squared error: $\mathrm{SE} = \|\boldsymbol{r}\|_2^2 = \sum_{i=1}^m r_i^2$
- root mean squared error: $\mathrm{RMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^m r_i^2} = \frac{\|\boldsymbol{r}\|_2}{\sqrt{m}}$

5.1 Data Fitting (Functional Models)

One important application of least squares is data fitting, where we want to fit a model to given data points.

Suppose we are given m data points $(t_1, y_1), \ldots, (t_m, y_m)$, typically coming from measurements or simulations and may contain measurement noise or numerical errors. We want to choose a model $y = f(t)$ that depends linearly on parameters $c_1, \ldots, c_n$, i.e.,

$$f(t) = \sum_{j=1}^n c_j \varphi_j(t)$$

for some given basis functions $\varphi_1, \ldots, \varphi_n : \mathbb{R} \to \mathbb{R}$. The data fitting workflow can be split into four steps:

1. **Model choice:** Pick a model $y = f(t)$ with parameters $\boldsymbol{c} = (c_1, \ldots, c_n)^\top$ that is reasonable from a modeling point of view.
2. **Force the model to match the data.** Plug the data into the model $y_i = f(t_i) = \sum_{j=1}^n c_j \varphi_j(t_i)$, $i = 1, \ldots, m$, and write this as a linear system

$$\underline{\boldsymbol{A}}\boldsymbol{c} = \begin{bmatrix} \varphi_1(t_1) & \ldots & \varphi_n(t_1) \\ \vdots & & \vdots \\ \varphi_1(t_m) & \ldots & \varphi_n(t_m) \end{bmatrix} \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \boldsymbol{b}$$

   where $\underline{\boldsymbol{A}} \in \mathbb{R}^{m \times n}$, $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$.

3. **Solve the normal equations:** Find the vector $\boldsymbol{c}^*$ that solves $\underline{\boldsymbol{A}}^\top \underline{\boldsymbol{A}} \boldsymbol{c}^* = \underline{\boldsymbol{A}}^\top \boldsymbol{b}$ or use a numerically more stable method (see Section 5.3).
4. **Report an error measure:** Compute the residual $\boldsymbol{r} = \boldsymbol{b} - \underline{\boldsymbol{A}}\boldsymbol{c}^*$ and one of the standard error measures.

5.2 Curve Fitting (Parametric Models)

The previous setting fits *functions* $y = f(t)$. In many geometric applications, however, we want to fit a *parametric curve* in $\mathbb{R}^d$ to points $\boldsymbol{P}_0, \dots, \boldsymbol{P}_m \in \mathbb{R}^d$.

1. **Model choice:** Choose basis functions $B_0, \dots, B_n : \mathbb{R} \to \mathbb{R}$ (for example monomials, Bernstein polynomials, or B-spline basis functions) and consider the parametric curve
$$\boldsymbol{F}(t) = \sum_{i=0}^{n} \boldsymbol{C}_i B_i(t)$$
where $\boldsymbol{C}_i \in \mathbb{R}^d$. Pick some parameter values $t_0, \dots, t_m$.
2. **Force the model to match the data:** Consider the $m+1$ equations $\boldsymbol{P}_k = \boldsymbol{F}(t_k)$. Define the matrices
$$\underline{\boldsymbol{B}} = \begin{bmatrix} B_0(t_0) & \dots & B_n(t_0) \\ \vdots & & \vdots \\ B_0(t_m) & \dots & B_n(t_m) \end{bmatrix} \in \mathbb{R}^{(m+1)\times(n+1)}$$
and $\underline{\boldsymbol{C}} = [\boldsymbol{C}_0, \dots, \boldsymbol{C}_n]^\top \in \mathbb{R}^{(n+1)\times d}$, $\underline{\boldsymbol{P}} = [\boldsymbol{P}_0, \dots, \boldsymbol{P}_m]^\top \in \mathbb{R}^{(m+1)\times d}$. Then the fitting conditions $\boldsymbol{P}_k \approx \boldsymbol{F}(t_k)$ can be written compactly as $\underline{\boldsymbol{B}}\underline{\boldsymbol{C}} \approx \underline{\boldsymbol{P}}$.
3. **Solve the normal equations:** We now have a matrix-valued least squares problem
$$\min_{\underline{\boldsymbol{C}}} \|\underline{\boldsymbol{P}} - \underline{\boldsymbol{B}}\underline{\boldsymbol{C}}\|_F^2$$
where $\|\cdot\|_F$ is the Frobenius norm
$$\|\underline{\boldsymbol{R}}\|_F^2 = \sum_{i,j} R_{i,j}^2 \tag{5.3}$$
The normal equations become $\underline{\boldsymbol{B}}^\top \underline{\boldsymbol{B}}\, \underline{\boldsymbol{C}}^* = \underline{\boldsymbol{B}}^\top \underline{\boldsymbol{P}}$ and this decouples into d standard least squares problems, one for each component of the curve.
4. **Report an error measure:** compute the residual matrix $\underline{\boldsymbol{R}} = \underline{\boldsymbol{P}} - \underline{\boldsymbol{B}}\underline{\boldsymbol{C}}^* \in \mathbb{R}^{(m+1)\times d}$ and one of the standard error measures, where the 2-norm is now replaced by the Frobenius norm (5.3).

Initial parameterization. We still have to choose parameter values $t_0, \dots, t_m$. A common choice is
$$t_0 = 0, \qquad t_i = t_{i-1} + \|\boldsymbol{P}_i - \boldsymbol{P}_{i-1}\|^\alpha, \quad i = 1, \dots, m,$$
for some exponent $\alpha \geq 0$. Typical choices are:

- $\alpha = 0$: uniform parameterization
- $\alpha = \frac{1}{2}$: centripetal parameterization
- $\alpha = 1$: chord length parameterization

In practice, centripetal and chord length parameterizations often yield better fits than uniform parameterization.

Parameter correction. Even with a good parameterization, the residual vectors

$$\boldsymbol{e}_i = \boldsymbol{P}_i - \boldsymbol{F}(t_i)$$

usually do not measure the minimal distance between the data point $\boldsymbol{P}_i$ and the curve $\boldsymbol{F}$, since the error vectors $\boldsymbol{e}_i$ are *not* necessarily orthogonal to the curve. Our goal is now to modify the parameters such that the $\boldsymbol{e}_i$ become orthogonal to the curve $\boldsymbol{F}$.

A simple improvement is to locally linearize $\boldsymbol{F}$ at t_i by a first order Taylor expansion

$$\boldsymbol{F}(t) \approx \boldsymbol{F}(t_i) + (t - t_i)\boldsymbol{F}'(t_i) = \boldsymbol{G}_i(t).$$

We then choose a corrected parameter t_i^* such that the vector from $\boldsymbol{P}_i$ to $\boldsymbol{G}_i(t_i^*)$ is orthogonal to the tangent $\boldsymbol{F}'(t_i)$. This leads to the update

$$t_i^* = t_i + \frac{(\boldsymbol{P}_i - \boldsymbol{F}(t_i))^\top \boldsymbol{F}'(t_i)}{\|\boldsymbol{F}'(t_i)\|^2}$$

Replacing t_i by t_i^* (and re-solving the least squares problem) usually reduces the distances $\|\boldsymbol{P}_i - \boldsymbol{F}(t_i)\|$ and improves the overall fit. In practice, one often performs several iterations of curve fitting plus parameter correction.

5.3 QR Factorization

The normal equations $\underline{\boldsymbol{A}}^\top \underline{\boldsymbol{A}}\boldsymbol{x}^* = \underline{\boldsymbol{A}}^\top \boldsymbol{b}$ can be numerically problematic, since the condition number of $\underline{\boldsymbol{A}}^\top \underline{\boldsymbol{A}}$ is approximately the square of that of $\underline{\boldsymbol{A}}$. A more sophisticated method is to compute the least-squares solution directly from $\underline{\boldsymbol{A}}$ without forming $\underline{\boldsymbol{A}}^\top \underline{\boldsymbol{A}}$. This can be done via the *QR factorization* of $\underline{\boldsymbol{A}}$.

5.3.1 Classical Gram-Schmidt

Let $\underline{\boldsymbol{A}} = (\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n) \in \mathbb{R}^{m\times n}$ with linearly independent columns. The Gram–Schmidt process constructs orthonormal vectors $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n \in \mathbb{R}^m$ spanning the same subspace:

$$\boldsymbol{y}_j = \boldsymbol{a}_j - \sum_{k=1}^{j-1} (\boldsymbol{a}_j^\top \boldsymbol{q}_k)\, \boldsymbol{q}_k, \quad \boldsymbol{q}_j = \frac{\boldsymbol{y}_j}{\|\boldsymbol{y}_j\|}$$

If we set $r_{k,j} = \boldsymbol{a}_j^\top \boldsymbol{q}_k$, $k < j$ and $r_{j,j} = \|\boldsymbol{y}_j\|$, then we can write

$$\underline{\boldsymbol{A}} = \underline{\boldsymbol{Q}}\,\underline{\boldsymbol{R}}$$

where $\underline{\boldsymbol{Q}} = (\boldsymbol{q}_1, \ldots, \boldsymbol{q}_n) \in \mathbb{R}^{m\times n}$ has orthonormal columns and $\underline{\boldsymbol{R}} \in \mathbb{R}^{n\times n}$ is upper triangular. This is the *reduced* QR factorization and can be computed using Algorithm 8.

Algorithm 8 Classical Gram-Schmidt orthogonalization

Require: matrix $\underline{\boldsymbol{A}} \in \mathbb{R}^{m\times n}$, where $\boldsymbol{A}_j$, $j = 1, \ldots, n$ are linearly independent
Ensure: reduced QR factorization $\underline{\boldsymbol{A}} = \underline{\boldsymbol{Q}}\,\underline{\boldsymbol{R}}$ with $\underline{\boldsymbol{Q}} \in \mathbb{R}^{m\times n}$, $\underline{\boldsymbol{R}} \in \mathbb{R}^{n\times n}$

```
for j = 1, ..., n do
    y ← A_j
    for i = 1, ..., j − 1 do
        r_{i,j} ← q_i^⊤ A_j
        y ← y − r_{i,j} q_i                ▷ subtract projection of A_j onto q_i
    r_{j,j} ← ‖y‖_2
    q_j ← y / r_{j,j}                      ▷ normalize to unit length
```

When the method is successful, it is customary to fill out the matrix of orthogonal unit vectors $\underline{\boldsymbol{Q}}$ to a complete basis of $\mathbb{R}^m$, to achieve the *full* QR factorization. This can be done, for example, by adding $m-n$ extra vectors to the $\boldsymbol{A}_j$, so that the m vectors span $\mathbb{R}^m$, and carrying out the Gram-Schmidt method. In terms of the basis of $\mathbb{R}^m$ formed by $q_1, \ldots, q_m$, the original vectors can be expressed as

$$\underline{\boldsymbol{A}} = \underline{\boldsymbol{Q}}\,\underline{\boldsymbol{R}}$$

where $\underline{\boldsymbol{Q}} \in \mathbb{R}^{m\times m}$ and $\underline{\boldsymbol{R}} \in \mathbb{R}^{m\times n}$ with zeros below the $n \times n$ upper triangular block.

Definition 5.1. A square matrix $\underline{\boldsymbol{Q}}$ is orthogonal if $\underline{\boldsymbol{Q}}^\top = \underline{\boldsymbol{Q}}^{-1}$ ◀

Fact 5.1. An orthogonal matrix $\underline{\boldsymbol{Q}} \in \mathbb{R}^{m\times m}$ satisfies

- $\underline{\boldsymbol{Q}}^\top$ also orthogonal
- preserves the Euclidian norm of a vector:

$$\|\underline{\boldsymbol{Q}}\boldsymbol{x}\|_2 = \sqrt{(\underline{\boldsymbol{Q}}\boldsymbol{x})^\top(\underline{\boldsymbol{Q}}\boldsymbol{x})} = \sqrt{\boldsymbol{x}^\top\underline{\boldsymbol{Q}}^\top\underline{\boldsymbol{Q}}\boldsymbol{x}} = \sqrt{\boldsymbol{x}^\top\boldsymbol{x}} = \|\boldsymbol{x}\|_2 \tag{5.4}$$

  for all $\boldsymbol{x} \in \mathbb{R}^m$
- product of two orthogonal matrices is orthogonal
- condition number $\kappa_2(\underline{\boldsymbol{Q}}) = 1$
- $\kappa_2(\underline{\boldsymbol{A}}) = \kappa_2(\underline{\boldsymbol{Q}}\underline{\boldsymbol{A}}) = \kappa_2(\underline{\boldsymbol{A}}\underline{\boldsymbol{Q}})$ ◁

5.3.2 Solving least squares problems

For an overdetermined system $\underline{\boldsymbol{A}}\boldsymbol{x} = \boldsymbol{b}$,

$$\|\underbrace{\underline{\boldsymbol{A}}\boldsymbol{x} - \boldsymbol{b}}_{=:\boldsymbol{r}}\|_2 = \|\underline{\boldsymbol{Q}}\underline{\boldsymbol{R}}\boldsymbol{x} - \boldsymbol{b}\|_2 = \|\underline{\boldsymbol{Q}}(\underline{\boldsymbol{R}}\boldsymbol{x} - \underline{\boldsymbol{Q}}^\top\boldsymbol{b})\|_2 \overset{(5.4)}{=} \|\underbrace{\underline{\boldsymbol{R}}\boldsymbol{x} - \underline{\boldsymbol{Q}}^\top\boldsymbol{b}}_{=:\boldsymbol{e}}\|_2$$

since $\underline{\boldsymbol{Q}}$ is orthogonal. Minimizing the norm of $\boldsymbol{r} \in \mathbb{R}^m$ is therefore equivalent to minimizing the norm of $\boldsymbol{e} \in \mathbb{R}^m$. Writing out $\boldsymbol{e}$ as

$$\boldsymbol{e} = \begin{bmatrix} e_1 \\ \vdots \\ e_n \\ \hdashline e_{n+1} \\ \vdots \\ e_m \end{bmatrix} = \left[\begin{array}{ccc} r_{1,1} & \dots & r_{1,n} \\ & \ddots & \vdots \\ 0 & & r_{n,n} \\ \hdashline 0 & \dots & 0 \\ \vdots & & \vdots \\ 0 & \dots & 0 \end{array}\right] \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} - \begin{bmatrix} d_1 \\ \vdots \\ d_n \\ \hdashline d_{n+1} \\ \vdots \\ d_m \end{bmatrix} = \underline{\boldsymbol{R}}\boldsymbol{x} - \boldsymbol{d}$$

where $\boldsymbol{d} = \underline{\boldsymbol{Q}}^\top\boldsymbol{b}$, we notice that the last $m - n$ components of $\boldsymbol{e}$ are always $-d_{n+1}, \dots, -d_m$, regardless of $\boldsymbol{x}$. So, all we can do to minimize the norm of $\boldsymbol{e}$ is to make the first n components as small as possible. But we can actually make them zero by solving

$$\hat{\underline{\boldsymbol{R}}}\boldsymbol{x}^* = \hat{\boldsymbol{d}} \tag{5.5}$$

where $\hat{\boldsymbol{d}} = [d_1, \dots, d_n]^\top$ and $\hat{\underline{\boldsymbol{R}}} \in \mathbb{R}^{n\times n}$ is the upper triangular part of $\underline{\boldsymbol{R}}$. Hence, $\boldsymbol{x}^*$ of (5.5) is the least-squares solution of $\underline{\boldsymbol{A}}\boldsymbol{x} = \boldsymbol{b}$, and the squared error of the residual is $\|\boldsymbol{r}\|_2^2 = \|\boldsymbol{e}\|_2^2 = d_{n+1}^2 + \dots + d_m^2$.

5.3.3 Modified Gram–Schmidt and Householder

An improvement in stability can be achieved by replacing $\boldsymbol{A}_j$ in line 4 of Algorithm 8 by the current $\boldsymbol{y}$ vector, which is mathematically equivalent since

$$r_{i,j} = \boldsymbol{q}_i^\top\boldsymbol{y} = \boldsymbol{q}_i^\top\left(\boldsymbol{A}_j - \sum_{k=1}^{i-1} r_{k,j}\boldsymbol{q}_k\right) = \boldsymbol{q}_i^\top\boldsymbol{A}_j - \sum_{k=1}^{i-1} r_{k,j}\underbrace{\boldsymbol{q}_i^\top\boldsymbol{q}_k}_{=0} = \boldsymbol{q}_i^\top\boldsymbol{A}_j$$

For even better stability and efficiency, QR is often computed with Householder reflections.

5.4 Weighted Least Squares

While the least squares solution $\boldsymbol{x}^*$) minimizes the 2-norm of the residual $\boldsymbol{r} = \boldsymbol{b} - \underline{\boldsymbol{A}}\boldsymbol{x}$, that is,

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} \|\boldsymbol{r}\|_2 = \arg\min_{\boldsymbol{x}} \|\boldsymbol{r}\|_2^2 = \arg\min_{\boldsymbol{x}} \sum_{i=1}^{m} |r_i|^2$$

where $r_i = b_i - (\underline{\boldsymbol{A}}\boldsymbol{x})_i$, some of the residual errors r_i can be rather big, while others are very small. But sometimes we want to emphasize some data points more than others. If we want, for example, to guarantee some error tolerance, then it might be better to reduce the *maximum* error and search for

$$\boldsymbol{x}^\star = \arg\min_{\boldsymbol{x}} \|\boldsymbol{r}\|_\infty = \arg\min_{\boldsymbol{x}} \max_{i=1,\dots,m} |r_i| \tag{5.6}$$

where $r_i = b_i - (\underline{\boldsymbol{A}}\boldsymbol{x})_i$. This can be done by solving a *linear programming* problem, but also via *weighted* least squares. The main idea is to give more importance to the data with big residuals when minimizing the norm of the residual and consider a weighted norm instead,

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} \sum_{i=1}^{m} w_i |b_i - (\underline{\boldsymbol{A}}\boldsymbol{x})_i|^2 \tag{5.7}$$

given weights $w_1, \ldots, w_m > 0$. The sum can be written in matrix form as

$$\sum_{i=1}^{m} w_i |b_i - (\underline{\boldsymbol{A}}\boldsymbol{x})_i|^2 = \left\| \underline{\boldsymbol{W}}^{1/2} (\boldsymbol{b} - \underline{\boldsymbol{A}}\boldsymbol{x}) \right\|_2^2$$

where $\underline{\boldsymbol{W}} = \operatorname{diag}(w_1, \ldots, w_m)$. The corresponding normal equations are

$$\underline{\boldsymbol{A}}^\top \underline{\boldsymbol{W}} \underline{\boldsymbol{A}}\, \boldsymbol{x}^* = \underline{\boldsymbol{A}}^\top \underline{\boldsymbol{W}} \boldsymbol{b}$$

If we want to find a solution of (5.6), it is not enough to just solve the weighted least squares problem once (5.7) with some fixed weights w_i Instead, we have to iteratively re-weigh the data by updating the weights according to

$$w_i^* = w_i |b_i - (\underline{\boldsymbol{A}}\boldsymbol{x}^*)_i| \tag{5.8}$$

for $i = 1, \ldots, m$, thus increasing the importance of the data with big residuals and decreasing the weight for data with small residuals. After a few iterations of solving (5.7) and correcting the weights as in (5.8), we get the optimal solution $\boldsymbol{x}^\star$ of (5.6). This solution is characterized by the fact that the corresponding residual $\boldsymbol{r}^\star = \boldsymbol{b} - \underline{\boldsymbol{A}}\boldsymbol{x}^\star$ has at least $n+1$ alternating extremal components.

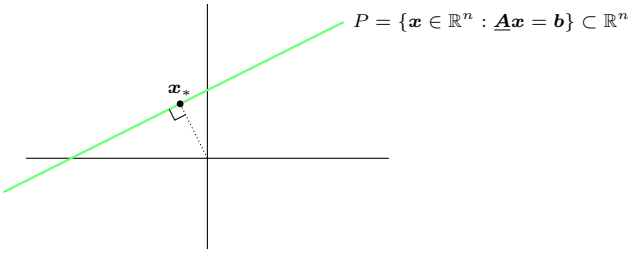
5.5 Least-Norm Solutions

If $\underline{\boldsymbol{A}} \in \mathbb{R}^{m \times n}$ with $m < n$, then the system $\underline{\boldsymbol{A}}\boldsymbol{x} = \boldsymbol{b}$ is *underdetermined* and typically has infinitely many solutions.

Among all solutions, we are often interested in the one with smallest 2-norm:

$$\boldsymbol{x}_* = \arg\min \big\{ \|\boldsymbol{x}\|_2 : \underline{\boldsymbol{A}}\boldsymbol{x} = \boldsymbol{b} \big\}$$

Since the set $P = \{\boldsymbol{x} \in \mathbb{R}^n : \underline{\boldsymbol{A}}\boldsymbol{x} = \boldsymbol{b}\}$ of all solutions can be seen geometrically as an affine subspace (a hyperplane) in $\mathbb{R}^n$, this particular least-norm solution $\boldsymbol{x}_*$ is the point in P that is closest to the origin, which is equivalent to saying that $\boldsymbol{x}_*$ is orthogonal to $\boldsymbol{y} - \boldsymbol{x}_*$ for any $\boldsymbol{y} \in P$:



Assuming $\underline{\boldsymbol{A}}$ has full row rank, then $\boldsymbol{x}_*$ is given by

$$\boldsymbol{x}_* = \underline{\boldsymbol{A}}^\top (\underline{\boldsymbol{A}}\underline{\boldsymbol{A}}^\top)^{-1} \boldsymbol{b}$$

and it is not hard to verify that this choice satisfies indeed the aforementioned orthogonality condition,

$$\begin{aligned}
(\boldsymbol{y} - \boldsymbol{x}_*)^\top \boldsymbol{x}_* &= (\boldsymbol{y} - \boldsymbol{x}_*)^\top \underline{\boldsymbol{A}}^\top (\underline{\boldsymbol{A}}\underline{\boldsymbol{A}}^\top)^{-1} \boldsymbol{b} \\
&= (\underline{\boldsymbol{A}}(\boldsymbol{y} - \boldsymbol{x}_*))^\top (\underline{\boldsymbol{A}}\underline{\boldsymbol{A}}^\top)^{-1} \boldsymbol{b} \\
&= (\underline{\boldsymbol{A}}\boldsymbol{y} - \underline{\boldsymbol{A}}\boldsymbol{x}_*)^\top (\underline{\boldsymbol{A}}\underline{\boldsymbol{A}}^\top)^{-1} \boldsymbol{b} \\
&= 0
\end{aligned}$$

for all $\boldsymbol{y} \in P$, since $\boldsymbol{x}_*, \boldsymbol{y} \in P$ imply that $\underline{\boldsymbol{A}}\boldsymbol{x}_* = \underline{\boldsymbol{A}}\boldsymbol{y} = \boldsymbol{b}$. One can view $\boldsymbol{x}_*$ as the orthogonal projection of the origin onto the hyperplane P.

5.6 Pseudoinverse

Both the least-squares solution $\boldsymbol{x}^*$ and least-norm solution $\boldsymbol{x}_*$ can be expressed nicely using the *pseudoinverse* $\underline{\boldsymbol{A}}^\dagger \in \mathbb{R}^{n\times m}$ of $\underline{\boldsymbol{A}} \in \mathbb{R}^{m\times n}$:

- If $m > n$ and $\underline{\boldsymbol{A}}$ has full column rank n, then

  $$\underline{\boldsymbol{A}}^\dagger = (\underline{\boldsymbol{A}}^\top \underline{\boldsymbol{A}})^{-1} \underline{\boldsymbol{A}}^\top$$

  and $\boldsymbol{x}^* = \underline{\boldsymbol{A}}^\dagger \boldsymbol{b}$ is the least-squares solution of $\underline{\boldsymbol{A}}\boldsymbol{x} \approx \boldsymbol{b}$.

- If $m < n$ and $\underline{\boldsymbol{A}}$ has full row rank m, then

  $$\underline{\boldsymbol{A}}^\dagger = \underline{\boldsymbol{A}}^\top (\underline{\boldsymbol{A}}\underline{\boldsymbol{A}}^\top)^{-1}$$

  and $\boldsymbol{x}_* = \underline{\boldsymbol{A}}^\dagger \boldsymbol{b}$ is the least-norm solution of $\underline{\boldsymbol{A}}\boldsymbol{x} = \boldsymbol{b}$.

In both cases, if we consider the singular value decomposition

$$\underline{\boldsymbol{A}} = \underline{\boldsymbol{U}}\underline{\boldsymbol{\Sigma}}\underline{\boldsymbol{V}}^\top$$

of $\underline{\boldsymbol{A}}$ and define the pseudoinverse of $\underline{\boldsymbol{\Sigma}} \in \mathbb{R}^{m\times n}$ to be the matrix $\underline{\boldsymbol{\Sigma}}^\dagger \in \mathbb{R}^{n\times m}$ with the reciprocals $1/\sigma_i$, $i = 1, \ldots, r$, on the diagonal, where $r = \min(m, n)$ is the rank of $\underline{\boldsymbol{A}}$ and $\underline{\boldsymbol{\Sigma}}$, then

$$\underline{\boldsymbol{A}}^\dagger = \underline{\boldsymbol{V}}\underline{\boldsymbol{\Sigma}}^\dagger \underline{\boldsymbol{U}}^\top$$

This definition of the pseudoinverse $\underline{\boldsymbol{A}}^\dagger$ works even if $\underline{\boldsymbol{A}}$ is not a full rank matrix, and in that case $\boldsymbol{x} = \underline{\boldsymbol{A}}^\dagger \boldsymbol{b}$ is among all vectors that minimize $\|\boldsymbol{b} - \underline{\boldsymbol{A}}\boldsymbol{x}\|_2$ the one with least norm.

6 Nonlinear Least Squares

Suppose we are given m functions $r_i : \mathbb{R}^n \to \mathbb{R}$, $i = 1, \dots, m$ with $m > n$ and would like to simultaneously solve the m equations

$$\begin{bmatrix} r_1(\boldsymbol{x}) \\ \vdots \\ r_m(\boldsymbol{x}) \end{bmatrix} = \boldsymbol{0}_m \tag{6.1}$$

in the n unknowns $x_1, \dots, x_n$. As in the case of linear least squares, this is not possible in general, but we can minimize instead the energy $E : \mathbb{R}^n \to \mathbb{R}$, which is given as the sum of squares

$$E(\boldsymbol{x}) = \sum_{i=1}^{m} r_i(\boldsymbol{x})^2 = \boldsymbol{r}(\boldsymbol{x})^\top \boldsymbol{r}(\boldsymbol{x}) \geq 0 \tag{6.2}$$

where $\boldsymbol{r} = [r_1, \dots, r_m]^\top : \mathbb{R}^n \to \mathbb{R}^m$. If there exists a point $\boldsymbol{x}$ with $\boldsymbol{r}(\boldsymbol{x}) = \boldsymbol{0}_m$, then $E(\boldsymbol{x}) = 0$ and the minimum is zero; otherwise, any minimizer $\boldsymbol{x}^*$ of (6.2) is called a *nonlinear least squares solution* of the system (6.1).

To minimize E, we set the gradient $\nabla E : \mathbb{R}^n \to \mathbb{R}^n$ to zero,

$$\nabla E(\boldsymbol{x}) = 2\, \underline{\boldsymbol{J}}_{\boldsymbol{r}}(\boldsymbol{x})^\top \boldsymbol{r}(\boldsymbol{x}) = \boldsymbol{0}_n$$

where

$$\underline{\boldsymbol{J}}_{\boldsymbol{r}}(\boldsymbol{x}) = \frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{r}(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial r_1(\boldsymbol{x})}{\partial x_1} & \cdots & \frac{\partial r_1(\boldsymbol{x})}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial r_m(\boldsymbol{x})}{\partial x_1} & \cdots & \frac{\partial r_m(\boldsymbol{x})}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

denotes the Jacobian of $\boldsymbol{r}$. It is convenient to define

$$\boldsymbol{F}(\boldsymbol{x}) = \frac{1}{2} \nabla E(\boldsymbol{x}) = \underline{\boldsymbol{J}}_{\boldsymbol{r}}(\boldsymbol{x})^\top \boldsymbol{r}(\boldsymbol{x}) \in \mathbb{R}^n \tag{6.3}$$

since nonlinear least squares can then be seen as the problem of finding a zero of $\boldsymbol{F}$, which can be done with the multivariate Newton method. Let $\underline{\boldsymbol{J}}_{\boldsymbol{F}}(\boldsymbol{x}) = \frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{F}(\boldsymbol{x}) \in \mathbb{R}^{n \times n}$ denote the Jacobian of $\boldsymbol{F}$. Given a current iterate $\boldsymbol{x}^{(k)}$, a Newton step $\boldsymbol{s}^{(k)}$ is obtained by

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \left(\underline{\boldsymbol{J}}_{\boldsymbol{F}}(\boldsymbol{x}^{(k)}) \right)^{-1} \boldsymbol{F}(\boldsymbol{x}^{(k)}) \tag{6.4}$$

where, using (6.3) and applying the product rule, we have

$$\begin{aligned}
\underline{\boldsymbol{J}}_{\boldsymbol{F}}(\boldsymbol{x}) &= \frac{\partial}{\partial \boldsymbol{x}} \left(\left(\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{r}(\boldsymbol{x}) \right)^\top \boldsymbol{r}(\boldsymbol{x}) \right) \\
&= \left(\frac{\partial}{\partial \boldsymbol{x}} \left(\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{r}(\boldsymbol{x}) \right)^\top \right) \boldsymbol{r}(\boldsymbol{x}) + \left(\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{r}(\boldsymbol{x}) \right)^\top \left(\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{r}(\boldsymbol{x}) \right) \\
&= \underbrace{\left(\frac{\partial^2}{\partial \boldsymbol{x}^2} \boldsymbol{r}(\boldsymbol{x}) \right)^\top}_{\in \mathbb{R}^{m \times (n \times n)}} \boldsymbol{r}(\boldsymbol{x}) + \left(\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{r}(\boldsymbol{x}) \right)^\top \left(\frac{\partial}{\partial \boldsymbol{x}} \boldsymbol{r}(\boldsymbol{x}) \right) \\
&= \underbrace{[\underline{\boldsymbol{H}}_{r_1}(\boldsymbol{x}), \dots, \underline{\boldsymbol{H}}_{r_m}(\boldsymbol{x})]}_{\in \mathbb{R}^{(n \times n) \times m}} \begin{bmatrix} r_1(\boldsymbol{x}) \\ \vdots \\ r_m(\boldsymbol{x}) \end{bmatrix} + \underline{\boldsymbol{J}}_{\boldsymbol{r}}(\boldsymbol{x})^\top \underline{\boldsymbol{J}}_{\boldsymbol{r}}(\boldsymbol{x}) \\
&= \sum_{i=1}^{m} r_i(\boldsymbol{x}) \underline{\boldsymbol{H}}_{r_i}(\boldsymbol{x}) + \underline{\boldsymbol{J}}_{\boldsymbol{r}}(\boldsymbol{x})^\top \underline{\boldsymbol{J}}_{\boldsymbol{r}}(\boldsymbol{x})
\end{aligned}$$

where $\underline{\boldsymbol{H}}_{r_i}(\boldsymbol{x}) \in \mathbb{R}^{n \times n}$ is the Hessian of r_i. Computing $\underline{\boldsymbol{J}}_{\boldsymbol{F}}$ exactly involves the calculation of mn^2 second derivatives of the residuals r_i, which can be expensive. The Gauss-Newton method simplifies Newton's method by replacing $\underline{\boldsymbol{J}}_{\boldsymbol{F}}$ with the first order approximation

$$\underline{\boldsymbol{J}}_{\boldsymbol{F}}(\boldsymbol{x}) \approx \underline{\boldsymbol{J}}_{\boldsymbol{r}}(\boldsymbol{x})^\top \underline{\boldsymbol{J}}_{\boldsymbol{r}}(\boldsymbol{x}) \tag{6.5}$$

Replacing $\underline{\boldsymbol{J}}_{\boldsymbol{F}}$ in (6.4) by the approximation (6.5), we obtain a linear least squares problem which can be solved with the same techniques as in the linear case (normal equations, QR factorization, etc.).

Remark 6.1. If each residual is linear in $\boldsymbol{x}$, say $r_i(\boldsymbol{x}) = b_i - \boldsymbol{a}_i^\top \boldsymbol{x}$, then $\boldsymbol{r}(\boldsymbol{x}) = \boldsymbol{b} - \underline{\boldsymbol{A}}\boldsymbol{x}$ with $\underline{\boldsymbol{A}}$ having rows $\boldsymbol{a}_i^\top$, and $\underline{\boldsymbol{J}}_{\boldsymbol{r}}(\boldsymbol{x}) = -\underline{\boldsymbol{A}}$ is constant. In this case, the Gauss–Newton step reduces to the normal equations $\underline{\boldsymbol{A}}^\top \underline{\boldsymbol{A}} \boldsymbol{x}^* = \underline{\boldsymbol{A}}^\top \boldsymbol{b}$, and the method converges in a single iteration to the linear least-squares solution. ◀

Algorithm 9 Gauss-Newton method

Require: residual function $\boldsymbol{r} : \mathbb{R}^n \to \mathbb{R}^m$, $\boldsymbol{x}^{(0)}$, $k_{\max}$, $\varepsilon_{\text{tol}} > 0$
Ensure: approximation $\boldsymbol{x}^*$ that (locally) minimizes $E(\boldsymbol{x}) = \|\boldsymbol{r}(\boldsymbol{x})\|_2^2$

```
for k = 0, ..., k_max − 1 do
    r^(k) ← r(x^(k))
    J^(k) ← J_r(x^(k))
    solve J^(k)ᵀ J^(k) s^(k) = J^(k)ᵀ r^(k) for s^(k)
    x^(k+1) ← x^(k) − s^(k)
    if ‖s^(k)‖_2 < ε_tol then
        break
```

Depending on the initial guess $\boldsymbol{x}^{(0)}$, Algorithm 9 may converge very slowly. This can be improved, either by adding a "regularization term", which is known as the *Levenberg-Marquardt* method, or by using the exact derivative $\underline{\boldsymbol{J}}_{\boldsymbol{F}}$.