# NumCSE exercise sheet 5
# Splines and quadrature

alexander.dabrowski@sam.math.ethz.ch
soumil.gurjar@sam.math.ethz.ch
oliver.rietmann@sam.math.ethz.ch

December 3, 2018

**Exercise 5.1**. *Cubic spline.*

Recall that the cubic spline $s$ interpolating a given data set $(t_0, y_0), \dots, (t_n, y_n)$ is a $C^2$ function on $[t_0, t_n]$ which is a polynomial of third degree on every subinterval $[t_j, t_{j+1}]$ for $j = 0, \dots, n-1$, and such that $s(t_j) = y_j$ for every $j = 0, \dots, n$. To ensure uniqueness we impose the additional boundary conditions $s''(t_0) = s''(t_n) = 0$.
Recall that since we can represent a polynomial of degree $d$ as a vector of length $d+1$ which contains the polynomial's coefficients, a cubic spline on a data set of length $n+1$ can be represented as a $4 \times n$ matrix, where the column $j$ specifies the coefficients of the interpolating polynomial on the interval $[t_j, t_j + 1]$.

1. Implement a C++ function `cubicSpline` which takes as input vectors $T = (t_0, \dots, t_n)$ and $Y = (y_0, \dots, y_n)$, and returns the matrix representing the cubic spline which interpolates such a dataset.

   Hint: implement the formulae from the tablet notes to calculate the second derivatives of the splines in the points $t_j$, then use them to build the matrix associated to the spline.

   **Solution:**

```
MatrixXd cubicSpline(const VectorXd &T, const VectorXd &Y) {
    // returns the matrix representing the spline interpolating the data
    // with abscissae T and ordinatae Y. Each column represents the coefficients
    // of the cubic polynomial on a subinterval.
    // Assumes T is sorted, has no repeated elements and T.size() == Y.size().

    int n = T.size() - 1; // T and Y have length n+1

    VectorXd h = T.tail(n) - T.head(n); // vector of lengths of subintervals

    // build the matrix of the linear system associated to the second derivatives
    MatrixXd A = MatrixXd::Zero(n-1, n-1);
    A.diagonal()   = (T.segment(2,n-1) - T.segment(0,n-1))/3;
    A.diagonal(1)  = h.segment(1,n-2)/6;
    A.diagonal(-1) = h.segment(1,n-2)/6;

    // build the vector of the finite differences of the data Y
    VectorXd slope = (Y.tail(n) - Y.head(n)).cwiseQuotient(h);

    // right hand side vector for the system with matrix A
    VectorXd r = slope.tail(n-1) - slope.head(n-1);
```

```

    // solve the system and fill vector of second derivatives
    VectorXd sigma(n+1);
    sigma.segment(1,n-1) = A.partialPivLu().solve(r);
    sigma(0) = 0; // "simple" boundary conditions
    sigma(n) = 0; // "simple" boundary conditions

    // build the spline matrix with polynomials' coefficients
    MatrixXd spline(4, n);
    spline.row(0) = Y.head(n);
    spline.row(1) = slope - h.cwiseProduct(2*sigma.head(n) + sigma.tail(n))/6;
    spline.row(2) = sigma.head(n)/2;
    spline.row(3) = (sigma.tail(n) - sigma.head(n)).cwiseQuotient(6*h);

    return spline;
}
```

cubic_spline.cpp

2. Implement a C++ function which given a cubic spline, its interpolation nodes and a vector of evaluation points, returns the value the spline takes on the evaluation points.

**Solution:**

```
VectorXd evalCubicSpline(const MatrixXd &S, const VectorXd &T, const VectorXd &
    evalT) {
    // Returns the values of the spline S calculated in the points evalT.
    // Assumes T is sorted, with no repetitions.

    int n = evalT.size();
    VectorXd out(n);

    for (int i=0; i < n; i++) {
        for (int j=0; j < T.size()-1; j++) {
            if (evalT(i) < T(j+1) || j==T.size()-2) {
                double x = evalT(i) - T(j);
                out(i) = S(0,j) + x*(S(1,j) + x*(S(2,j) + x*S(3,j)));
                break;
            }
        }
    }

    return out;
}
```

cubic_spline.cpp

3. Run some tests of your spline evaluation function (see template).

**Exercise 5.2**. *Gauss-Legendre quadrature rule.*

An $n$-point quadrature formula on $[a, b]$ provides an approximation of the value of an integral through a *weighted sum* of point values of the integrand:

$$\int_a^b f(x)\,\mathrm{d}t \;\approx\; Q_n(f) := \sum_{j=1}^{n} w_j^n\, f(c_j^n), \tag{1}$$

where $w_j^n$ are called quadrature weights $\in \mathbb{R}$ and $c_j^n$ quadrature nodes $\in [a, b]$.

The order of a quadrature rule $Q_n : C^0([a, b]) \to \mathbb{R}$ is defined as the maximal degree+1 of polynomials for which the quadrature rule is guaranteed to be exact. It can also be shown that the maximal order of an $n$-point quadrature rule is $2n$. So the natural question to ask is if such a family $Q_n$ of $n$-point quadrature formulas exist where $Q_n$ is of order $2n$. If yes, how do we find the nodes corresponding to it?

Let us assume that there exists a family of $n$-point quadrature formulas on $[-1, 1]$ of order $2n$, i.e.

$$Q_n(f) := \sum_{j=1}^{n} w_j^n f(c_j^n) \approx \int_{-1}^{1} f(t)\,\mathrm{d}t\,, \quad w_j \in \mathbb{R}\,,\ n \in \mathbb{N}\,, \tag{2}$$

and the above approximation is exact for polynomials $\in \mathcal{P}_{2n-1}$.

Define the $n$-degree polynomial

$$\bar{P}_n(t) := (t - c_1^n) \cdot \dots \cdot (t - c_n^n)\,, \quad t \in \mathbb{R}\,.$$

If we are able to obtain $\bar{P}_n(t)$, we can compute its roots numerically to obtain the nodes for the quadrature formula.

(a) For every $q \in \mathcal{P}_{n-1}$, verify that $\bar{P}_n(t) \perp q$ in $L^2([-1, 1])$ i.e.

$$\int_{-1}^{1} q(t)\bar{P}_n(t)\,\mathrm{d}t = 0. \tag{3}$$

**Solution:**

$$\forall\, q \in \mathcal{P}_{n-1} : \quad q \cdot \bar{P}_n \in \mathcal{P}_{2n-1}$$

$$\Longrightarrow \underbrace{\int_{-1}^{1} q(t) \cdot \bar{P}_n(t)\,\mathrm{d}t}_{\langle q, \bar{P}_n \rangle_{L^2([-1,1])}} \underbrace{=}_{\text{exact QF on } \mathcal{P}_{2n-1}} \sum_{j=1}^{n} w_j^n q(c_j^n) \underbrace{\bar{P}_n(c_j^n)}_{=0,\ \forall j=(1,\dots,n)} = 0.$$

Thus, we have proved $\bar{P}_n \perp \mathcal{P}_{n-1}$ in $L^2([-1, 1])$.

(b) Switching to a monomial representation of $\bar{P}_n$

$$\bar{P}_n = t^n + \alpha_{n-1}t^{n-1} + \dots + \alpha_1 t + \alpha_0\,,$$

derive

$$\sum_{j=0}^{n-1} \alpha_j \int_{-1}^{1} t^\ell t^j\,\mathrm{d}t = -\int_{-1}^{1} t^\ell t^n\,\mathrm{d}t \qquad \forall\, \ell = 0\,\dots, n-1. \tag{4}$$

*Hint:* Use (3) with the monomials $1, t, \dots, t^{n-1}$ and with $\bar{P}_n$ in its monomial representation.

**Solution:** We know that:

$$\int_{-1}^{1} q(t)\bar{P}_n(t)\,\mathrm{d}t = 0 \quad \forall q \in \mathcal{P}_{n-1}.$$

This yields $n$ conditions:

$$\begin{aligned}
&\int_{-1}^{1} \bar{P}_n t^\ell \,\mathrm{d}t = 0 \qquad \forall\, \ell = 0\,\ldots, n-1 \\
\Leftrightarrow &\int_{-1}^{1} t^\ell \underbrace{\left(t^n + \sum_{j=0}^{n-1} \alpha_j t^j\right)}_{\bar{P}_n} \mathrm{d}t = 0 \quad \forall \ell = 0,\ldots,n-1 \\
\Longrightarrow &\sum_{j=0}^{n-1} \alpha_j \int_{-1}^{1} t^\ell t^j \,\mathrm{d}t = -\int_{-1}^{1} t^\ell t^n \,\mathrm{d}t.
\end{aligned}$$

(c) Find expressions for $\mathbf{A}$ and $\mathbf{b}$ such that the coefficients of the monomial expansion can be obtained by solving a linear system of equation $\mathbf{A}[\alpha_j]_{j=0}^{n-1} = \mathbf{b}$.

**Solution:** (4) can be rewritten as: $\mathbf{A}[\alpha_j]_{j=0}^{n-1} = \mathbf{b}$, where

$$\mathbf{A}_{j,\ell} = \int_{-1}^{1} t^\ell t^j \,\mathrm{d}t = \langle t^\ell, t^j \rangle_{L^2([-1,1])}.$$

and

$$\mathbf{b}_\ell = -\int_{-1}^{1} t^\ell t^n \,\mathrm{d}t = \langle t^\ell, t^n \rangle_{L^2([-1,1])}.$$

(d) Show that $[\alpha_j]_{j=0}^{n-1}$ exists and is unique.

*Hint:* verify that $\mathbf{A}$ is symmetric positive definite.

**Solution:** We can see that $\mathbf{A}$ is symmetric. Moreover,

$$\begin{aligned}
\mathbf{x}^\top \mathbf{A}\mathbf{x} &= \sum_{\ell=0}^{n-1} x_\ell \left( \sum_{j=0}^{n-1} \int_{-1}^{1} t^j t^\ell \,\mathrm{d}t\; x_j \right) \\
&= \int_{-1}^{1} \left( \sum_{\ell=0}^{n-1} x_\ell t^\ell \right) \left( \sum_{j=0}^{n-1} x_j t^j \right) \mathrm{d}t \\
&= \int_{-1}^{1} \left( \sum_{j=0}^{n-1} x_j t^j \right)^2 \mathrm{d}t > 0 \quad \text{if } x \neq 0.
\end{aligned}$$

Thus, $\mathbf{A}$ is symmetric positive definite $\Longrightarrow$ $[\alpha_j]_{j=0}^{n-1}$ exists and is unique.

(e) Use a 5-point Gauss quadrature rule to compare the exact solution and the quadrature approximation of

$$\int_{-3}^{3} e^t \,\mathrm{d}t.$$

The polynomial obtained in (d) and the Legendre-polynomial $P_n$ differ by a constant factor. Thus, the Gauss quadrature nodes $(\widehat{c}_j)_{j=1}^{5}$ are also the zeros of the 5-th Legendre polynomial

$P_5$. Here, we provide the zeros of $P_5$ for simplicity, but they should ideally be obtained by a numerical method for obtaining roots (e.g Newton-Raphson method). Thus,

$$(\widehat{c}_j)_{j=1}^5 = [-0.9061798459, -0.5384693101, 0, 0.5384693101, 0.9061798459]$$

Recall from Theorem 6.3.1 (found in Week 9 Tablet notes - pg. 9) that the corresponding quadrature weights $\widehat{w}_j$ are given by:

$$\widehat{w}_j = \int_{-1}^{1} L_{j-1}(t)\,\mathrm{d}t, \quad j = 1, \ldots, n, \tag{5}$$

where $L_j, j = 0, \ldots, n-1$, is the $j$-th Lagrange polynomial associated with the ordered node set $\{\widehat{c}_1, \ldots, \widehat{c}_n\}$.

**Solution:** The $j$-th Lagrange polynomial can be obtained by:

$$L_j(t) = \prod_{k=0, k\neq j}^{n-1} \frac{t - t_k}{t_j - t_k}.$$

After obtaining the Lagrange polynomials for $j = 0, \ldots, n-1$ using the quadrature nodes $(\widehat{c}_j)_{j=1}^5$, we can use (5) to obtain the quadrature weights. They are found to be:

$$(\widehat{w}_j)_{j=1}^5 = [0.2369268851, 0.4786286705, 0.5688888889, 0.4786286705, 0.2369268851].$$

Note that we wish to use the quadrature formula on the interval $[-3, 3]$. However, our nodes and weights have been computed for the reference interval $[-1, 1]$. Thus, we need to perform an affine transformation

$$\Phi(\tau) = \frac{1}{2}(1 - \tau)a + \frac{1}{2}(1 + \tau)b\ .$$

This allows us to use the general quadrature formula with the transformed nodes and weights, i.e.

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

with

$$c_j = \Phi(\widehat{c}_j) = \tfrac{1}{2}(1 - \widehat{c}_j)a + \tfrac{1}{2}(1 + \widehat{c}_j)b\ , \qquad w_j = \frac{|[a, b]|}{|[-1, 1]|}\widehat{w}_j = \tfrac{1}{2}(b - a)\widehat{w}_j\ .$$

The solution obtained using the quadrature approximation $(\sum_{j=1}^{n} w_j e^{(c_j)}) = 20.0355777184$.

On the other hand, the exact solution is

$$\int_{-3}^{3} e^t\,\mathrm{d}t = e^3 - e^{-3} = 20.0357498548.$$

**Exercise 5.3**. *Gauss quadrature and composite Simpson rule.*

Consider a non-empty interval $[a, b] \subseteq \mathbb{R}$ and a function $f : [a, b] \to \mathbb{R}$.

(a) Write a `C++` function

```
double GaussLegendre5(const std::function<double(double)> &f, double a, double b);
```

that applies a Gauss-Legendre quadrature of order 5 to $f$ on $[a, b]$. The corresponding nodes and weights for a function on $[-1, 1]$ are given by

```
const std::vector<double> c = { -0.90617984593,
                                -0.53846931010,
                                 0.0,
                                 0.53846931010,
                                 0.90617984593 };

const std::vector<double> w = { 0.23692688505,
                                0.47862867049,
                                0.56888888888,
                                0.47862867049,
                                0.23692688505 };
```

*Hint:* Apply a substitution in the integral to scale these nodes into $[a, b]$.

**Solution:**

```
double GaussLegendre5(const std::function<double(double)> &f, double a, double b) {
    const std::vector<double> c = { -0.90617984593, -0.53846931010, 0.0, 0.53846931010, 0.90617984593 };
    const std::vector<double> w = { 0.23692688505, 0.47862867049, 0.56888888888, 0.47862867049, 0.23692688505 };

    int m = c.size();
    std::vector<double> x(m);
    double d = b - a;
    for (int i = 0; i < m; ++i) {
        x[i] = d * (c[i] + 1.0) / 2.0 + a;
    }

    double q = .0;
    for (int i = 0; i < x.size(); ++i) {
        q += w[i] * f(x[i]);
    }

    return q * d / 2;
}
```

gauss_simpson.cpp

(b) Write a `C++` function

```
double CompositeSimpson(const std::function<double(double)> &f,
                        const std::vector<double> &x);
```

that computes a composite Simpson quadrature of $f$ for the given nodes $x_0, \ldots, x_m \in [a, b]$, where $m \in \mathbb{N}$. Your composite Simpson rule should only use $2m + 1$ evaluations of $f$.

**Solution:**

```
double CompositeSimpson(const std::function<double(double)> &f, const std::vector<
    double> &x) {
    int n = x.size();
    int m = n - 1;

    double q = 1.0 / 6.0 * (x[1] - x[0]) * f(x[0]) / 6.0;
    for (int j = 1; j < m; ++j) {
        q += 1.0 / 6.0 * (x[j + 1] - x[j - 1]) * f(x[j]);
    }
    for (int j = 1; j <= m; ++j) {
        q += 2.0 / 3.0 * (x[j] - x[j - 1]) * f((x[j] + x[j - 1]) / 2);
    }
    q += 1.0 / 6.0 * (x[m] - x[m - 1]) * f(x[m]);

    return q;
}
```

gauss_simpson.cpp

**Exercise 5.4**. *Lagrange vs. Newton interpolation.*

Fix $n \in \mathbb{N}$ and let $x_0, \dots, x_n \in \mathbb{R}$ be distinct nodes. Denote by $L_0, \dots, L_n$ and $N_0, \dots, N_n$ the Lagrange and Newton polynomials for these nodes. Moreover, let $y_0, \dots, y_n \in \mathbb{R}$. We implement the corresponding interpolants by completing the following structs:

```
struct Newton {
    Newton(const Eigen::VectorXd &x) : _x(x), _a(x.size()) { }
    void Interpolate(const Eigen::VectorXd &y);
    double operator()(double x) const;

private:
    Eigen::VectorXd _x; // nodes
    Eigen::VectorXd _a; // coefficients
};
```

interpolation.cpp

```
struct Lagrange {
    Lagrange(const Eigen::VectorXd &x);
    void Interpolate(const Eigen::VectorXd &y) { _y = y; }
    double operator()(double x) const;

private:
    Eigen::VectorXd _x; // nodes
    Eigen::VectorXd _l; // weights
    Eigen::VectorXd _y; // coefficients
};
```

interpolation.cpp

(a) Consider the Newton interpolant

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

where $x \in \mathbb{R}$. Then the coefficients $a_0, \dots, a_n \in \mathbb{R}$ solve the linear system of equations

$$\begin{pmatrix} 1 & & & & 0 \\ 1 & (x_1 - x_0) & & & \\ 1 & (x_2 - x_0) & (x_2 - x_0)(x_2 - x_1) & & \\ \vdots & \vdots & & \ddots & \\ 1 & (x_n - x_0) & \cdots & & \prod_{i=0}^{n-1}(x_n - x_i) \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ y_n \end{pmatrix}.$$

Implement the member function

```
void Newton::Interpolate(const Eigen::VectorXd &y);
```

computing the coefficients $a_0, \dots, a_n$ for given $y_0, \dots, y_n$. What is the complexity for large $n$?

**Solution:** The code below implements the so-called *divided difference* scheme. The complexity is $O(n^2)$. However, any other way of solving the linear system is also fine.

```
void Newton::Interpolate(const Eigen::VectorXd &y) {
    _a = y;
    int n = _a.size();
    for (int j = 0; j < n - 1; ++j) {
        for (int i = n - 1; i > j; --i) {
```

```
            _a(i) = (_a(i) - _a(i - 1)) / (_x(i) - _x(i - 1 - j));
        }
    }
}
```

interpolation.cpp

(b) Use the *Horner* scheme to implement the operator

```
double Newton::operator()(double x) const;
```

that computes for $x \in \mathbb{R}$ the value of $p(x)$ using only $n$ multiplications.

*Hint:* For $n = 2$ this is achieved by rewriting

$$\begin{aligned} a_2 N_2(x) + a_1 N_1(x) + a_0 N_0(x) &= a_2(x - x_1)(x - x_0) + a_1(x - x_0) + a_0 \\ &= \big(a_2(x - x_1) + a_1\big)(x - x_0) + a_0. \end{aligned}$$

Generalize this idea to arbitrary $n$.

**Solution:**

```
double Newton::operator()(double x) const {
    int n = _a.size();
    double y = _a(n - 1);
    for (int i = n - 2; i >= 0; --i) {
        y = y * (x - _x(i)) + _a(i);
    }
    return y;
}
```

interpolation.cpp

(c) Implement the constructor

```
Lagrange::Lagrange(const Eigen::VectorXd &x);
```

which computes for given nodes $x_0, \ldots, x_n$ the weights

$$\lambda_i := \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{1}{x_i - x_j},$$

where $i \in \{0, \ldots, n\}$.

**Solution:**

```
Lagrange::Lagrange(const Eigen::VectorXd &x) : _x(x), _l(x.size()), _y(x.size()) {
    int n = _x.size();
    for (int j = 0; j < n; ++j) {
        double dw = 1.0;
        for (int i = 0; i < n; ++i) {
            if (i != j) dw *= _x(j) - _x(i);
        }
        _l(j) = 1.0 / dw;
    }
}
```

interpolation.cpp

(d) Define $\omega(x) := \prod_{j=0}^{n}(x - x_j)$ where $x \in \mathbb{R}$ and recall from Exercise 4.3 (b) that[1]

$$L_i(x) = \omega(x)\,\frac{\lambda_i}{x - x_i}$$

for all $i \in \{0, \ldots, n\}$. Use this to implement the operator

`double Lagrange::operator()(double x) const;`

that computes the value of the Lagrange interpolant

$$q(x) := \sum_{i=0}^{n} y_i L_i(x).$$

What is the complexity for large $n$?

**Solution:** The complexity is $O(n)$ (see code below).

```
double Lagrange::operator()(double x) const {
    int n = _x.size();
    Eigen::VectorXd L(n);
    double wx = 1.0;
    for (int i = 0; i < n; ++i) {
        wx *= x - _x(i);
    }
    for (int i = 0; i < n; ++i) {
        L(i) = wx * _l(i) / (x - _x(i));
    }
    return _y.dot(L);
}
```

interpolation.cpp

[1] We encounter a division by zero if $x = x_i$. You may ignore this issue.