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

November 29, 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.

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.

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

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

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.

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

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

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

derive

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

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

(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 $\mathbf{A}[\alpha_j]_{j=0}^{n-1} = \mathbf{b}$.

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

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

(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 $\{\widehat{c}_1, \ldots, \widehat{c}_n\}$.

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 $[a, b]$.

(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.

Exercise 5.4. *Lagrange vs. Newton interpolation.*

Fix $n \in \mathbb{N}$ and let $x_0, \ldots, x_n \in \mathbb{R}$ be distinct nodes. Denote by $L_0, \ldots, L_n$ and $N_0, \ldots, N_n$ the Lagrange and Newton polynomials for these nodes. Moreover, let $y_0, \ldots, y_n \in \mathbb{R}$. 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 $a_0, \ldots, a_n \in \mathbb{R}$ 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 $a_0, \ldots, a_n$ for given $y_0, \ldots, y_n$. What is the complexity for large n?

(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.

(c) Implement the constructor

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

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

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

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

(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 $i \in \{0, \dots, n\}$. 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?

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