Numerical Methods for CSE
Final Exam HS2018

Prof. Rima Alaifari

ETH Zürich, D-MATH

January 29, 2019

Exercise 1. *Curve fitting* (26 pts) [*Template: 1.cpp*]

(a) (2 pts) Let $n \geq 3$ and $\mathbf{x} := (x_1, \ldots, x_n)^\top, \mathbf{y} := (y_1, \ldots, y_n)^\top \in \mathbb{R}^n$, where the entries of $\mathbf{x}$ are distinct. We want to determine the coefficients $\mathbf{c} := (c_0, c_1, c_2)^\top \in \mathbb{R}^3$ of a parabola

$$p_{\mathbf{c}}(x) := c_2 x^2 + c_1 x + c_0$$

such that

$$E(\mathbf{c}) := \sum_{i=1}^{n} |p_{\mathbf{c}}(x_i) - y_i|^2$$

is minimized (see Figure 1).

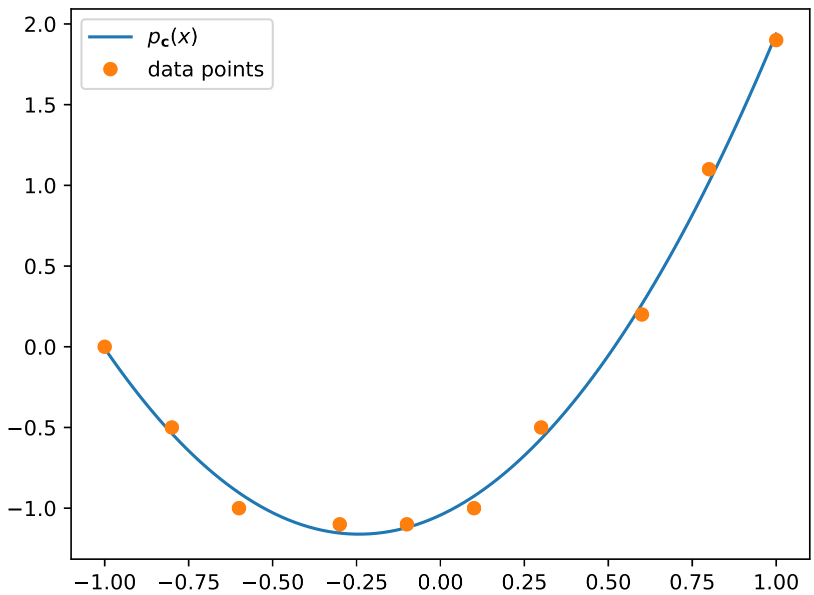

Figure 1: The parabola $p_{\mathbf{c}}(x)$ fitted to the data points $\{(x_i, y_i)\}_{i=1}^n$.

Formulate this as a linear *least squares problem*: For given $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ as above, find a matrix $\mathbf{A} \in \mathbb{R}^{n \times 3}$ and a vector $\mathbf{b} \in \mathbb{R}^n$ such that for all $\mathbf{c} \in \mathbb{R}^3$, we have

$$E(\mathbf{c}) = \|\mathbf{A}\mathbf{c} - \mathbf{b}\|_2^2. \tag{1}$$

Solution:

$$\mathbf{A} = \begin{pmatrix} 1 & x_1 & x_1^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 \end{pmatrix}, \quad \mathbf{b} = \mathbf{y}$$

(b) (4 pts) Implement a `C++/Eigen` function

```
Eigen::MatrixXd GetA(const Eigen::VectorXd &x);
```

that for given $\mathbf{x}$ as above returns the matrix $\mathbf{A}$ in (1).

Solution:

```
Eigen::MatrixXd GetA(const Eigen::VectorXd &x) {
    int n = x.size();
    Eigen::MatrixXd A(n, 3);

    for (int i = 0; i < n; ++i) {
        A(i, 0) = 1.0;
        A(i, 1) = x(i);
        A(i, 2) = x(i) * x(i);
    }

    return A;
}
```

fitting.cpp

(c) (4 pts) Implement a C++/Eigen function

```
Eigen::VectorXd LeastSquares(const Eigen::MatrixXd &A, const Eigen::VectorXd &b);
```

that for given $\mathbf{A}$ and $\mathbf{b}$ returns the least squares solution $\mathbf{c}$ of (1). You may use any Eigen solver of your choice.

Solution:

```
Eigen::VectorXd LeastSquares(const Eigen::MatrixXd &A, const Eigen::VectorXd &b) {
    return (A.transpose() * A).ldlt().solve(A.transpose() * b);
}
```

fitting.cpp

(d) (8 pts) Let $k \in \mathbb{N}$ and $\mathbf{B} \in \mathbb{R}^{m \times n}$, where $m, n \geq k$. Implement a C++/Eigen function

```
Eigen::MatrixXd BestApprox(const Eigen::MatrixXd &B, int k);
```

that returns the $m \times n$ matrix that is the best rank-k approximation of $\mathbf{B}$, using the *singular value decomposition* of $\mathbf{B}$.

Solution:

```
Eigen::MatrixXd BestApprox(const Eigen::MatrixXd &B, int k) {
    Eigen::JacobiSVD<Eigen::MatrixXd> svd(B, Eigen::ComputeThinU | Eigen::ComputeThinV);

    Eigen::VectorXd s = svd.singularValues();
    s.tail(s.size() - k) = Eigen::VectorXd::Zero(s.size() - k);
    Eigen::MatrixXd S = s.asDiagonal();

    return svd.matrixU() * S * svd.matrixV().transpose();
}
```

fitting.cpp

(e) (8 pts) Let $n \in \mathbb{N}$ and consider data points $(x_1, y_1), \ldots, (x_n, y_n) \in (0, \infty) \times (0, \infty)$. Let $\mathbf{B}_1 \in \mathbb{R}^{2 \times n}$ denote the best rank-1 approximation of

$$\mathbf{B} := \begin{pmatrix} x_1 & \cdots & x_n \\ y_1 & \cdots & y_n \end{pmatrix}. \tag{2}$$

Implement a C++/Eigen function

```
double FitLineThroughOrigin(const Eigen::MatrixXd &B);
```

that takes $\mathbf{B}$ as in (2) and computes $\mathbf{B}_1$ to extract the first principal component of $\mathbf{B}$. The line through the origin along this principal component will then fit the data points in $\mathbf{B}$. The function `FitLineThroughOrigin` shall return the slope of this line (see Figure 2).

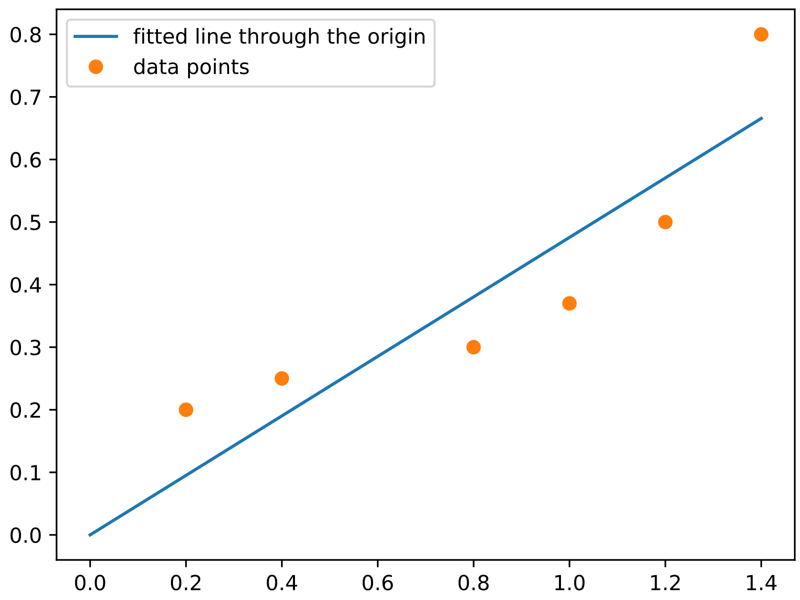

Figure 2: Line through the origin with slope computed by `FitLineThroughOrigin`.

Solution:

```
double FitLineThroughOrigin(const Eigen::MatrixXd &B) {
    Eigen::MatrixXd BB = BestApprox(B, 1);
    double a = BB.row(1).sum() / BB.row(0).sum();
    return a;
}
```

fitting.cpp

Exercise 2. *Gauss-Chebyshev quadrature* (24 pts) [*Template: 2.cpp*]

The Chebyshev polynomial of the first kind $T_n(x)$ is a polynomial of exact degree n, defined by the relation

$$T_n(x) := \cos(n \arccos(x)), \quad \text{where } x \in [-1, 1]. \tag{3}$$

(a) (4 pts) The weighted inner product of two functions $f(x)$ and $g(x)$, with respect to a given continuous and non-negative weight function $w(x)$, can be defined as:

$$\langle f, g \rangle_w := \int_a^b w(x) f(x) g(x) \, \mathrm{d}x. \tag{4}$$

f and g are said to be orthogonal with respect to the weight function $w(x)$ if

$$\langle f, g \rangle_w = 0. \tag{5}$$

For the following interval and weight function:

$$[a, b] = [-1, 1], \qquad w(x) = \frac{1}{\sqrt{1 - x^2}},$$

show that the Chebyshev polynomials of the first kind satisfy

$$\langle T_i, T_j \rangle_w = 0 \quad \text{if } i \neq j. \tag{6}$$

Hint: Substitute $x = \cos\theta$ to simplify the computation of $\langle T_i, T_j \rangle_w$. The following trigonometric identity could also be useful:

$$2 \cos x \cos y = \cos(x + y) + \cos(x - y).$$

Solution: By setting $x = \cos\theta$ and using the relations $T_n(x) = \cos n\theta$ and $\mathrm{d}x = -\sin\theta \mathrm{d}\theta$, we obtain

$$\begin{aligned}
\langle T_i, T_j \rangle_w &= \int_{-1}^{1} \frac{T_i(x) T_j(x)}{\sqrt{1 - x^2}} \, \mathrm{d}x \\
&= \int_{\pi}^{0} -\frac{\cos i\theta \cos j\theta}{\sqrt{1 - \cos^2\theta}} \sin\theta \, \mathrm{d}\theta \\
&= \int_0^{\pi} \cos i\theta \cos j\theta \, \mathrm{d}\theta.
\end{aligned}$$

For $i \neq j$,

$$\begin{aligned}
\int_0^{\pi} \cos i\theta \cos j\theta \, \mathrm{d}\theta &= \frac{1}{2} \int_0^{\pi} [\cos(i + j)\theta + \cos(i - j)\theta] \, \mathrm{d}\theta \\
&= \frac{1}{2} \left[\frac{\sin(i + j)\theta}{i + j} + \frac{\sin(i - j)\theta}{i - j} \right]_0^{\pi} \\
&= 0.
\end{aligned}$$

Hence,

$$\langle T_i, T_j \rangle_w = 0 \qquad (i \neq j).$$

(b) (5 pts) Suppose that we now wish to calculate a definite integral of $f(x)$ with a general weight function $w(x)$, namely

$$I = \int_{-1}^{1} w(x) f(x)\, \mathrm{d}x \,. \tag{7}$$

I is to be approximated by an n-point quadrature formula of the form

$$Q_{n,w}(f) \simeq \sum_{k=0}^{n-1} A_k f(x_k), \tag{8}$$

where A_k are the quadrature weights and x_k are the quadrature nodes in $[-1, 1]$.

Determine an *integral* expression for the quadrature weights A_k in terms of the quadrature nodes x_k and the general weight function $w(x)$, so that $Q_{n,w}(f)$ is guaranteed to have order $\geq n$.

Hint: In Eq.(7), substitute $f(x)$ with its polynomial Lagrange interpolant that is formed by interpolation through the quadrature nodes x_k.

Solution: If p_{n-1} is the polynomial of degree $n-1$ which interpolates $f(x)$ in any n distinct points $x_0, \ldots, x_{n-1}$, then

$$p_{n-1} = \sum_{k=0}^{n-1} f(x_k) L_k(x),$$

where L_k is the Lagrange polynomial

$$L_k(t) := \prod_{\substack{j=0 \\ j\neq k}}^{n-1} \frac{t - t_j}{t_k - t_j}, \qquad k = 0, \ldots, n-1. \tag{9}$$

Therefore,

$$\begin{aligned} I &= \int_{-1}^{1} w(x)\, p_{n-1}(x)\, \mathrm{d}x \\ &= \sum_{k=0}^{n-1} f(x_k) \int_{-1}^{1} w(x) L_k(x)\, \mathrm{d}x \\ &= \sum_{k=0}^{n-1} A_k f(x_k), \end{aligned}$$

provided that the coefficients A_k are chosen to be

$$A_k = \int_{-1}^{1} w(x) L_k(x)\, \mathrm{d}x, \quad k = 0, \ldots, n-1 \tag{10}$$

(c) (3 pts) Show that

$$\int_{-1}^{1} w(x) T_n(x) q(x)\, \mathrm{d}x = 0 \tag{11}$$

for any polynomial function $q(x)$ of degree $n-1$ or less, if

$$w(x) = \frac{1}{\sqrt{1 - x^2}}.$$

Hint: Use the orthogonality of Chebyshev polynomials as proved in (6).

Solution: Owing to (6), we can use the polynomials $\{T_0, T_1, \dots T_{n-1}\}$ as a basis to construct any polynomial of degree $n-1$. Since $q(x)$ is a polynomial function of degree $n-1$ or less, we can express $q(x)$ by its Chebyshev expansion as

$$q(x) = \sum_{k=0}^{n-1} \alpha_k T_k(x).$$

This implies that

$$\begin{aligned} \langle T_n, q \rangle_w &= \sum_{k=0}^{n-1} \alpha_k \langle T_n, T_k \rangle_w \\ &= 0. \end{aligned}$$

(d) (5 pts) If the quadrature nodes x_k, ($k = 0, \dots, n-1$), are the known n zeros of the Chebyshev polynomial $T_n(x)$, and the quadrature weights A_k are those obtained in sub-problem (b), then $Q_{n,w}(f)$ corresponds to the Gauss-Chebyshev quadrature rule.

Derive the following statement: The Gauss-Chebyshev quadrature rule is of order $2n$.

Hint: Consider $f(x)$ to be a polynomial of degree $2n-1$ and then perform long polynomial division of $f(x)$ by $T_n(x)$, that is, write $f(x)$ as:

$$f(x) = T_n(x)q(x) + r(x),$$

and use the result derived in (11).

Solution: Since $T_n(x)$ is a polynomial exactly of degree n, any polynomial $f(x)$ of degree $2n-1$ can be written (using long division by $T_n(x)$) in the form

$$f(x) = T_n(x)q(x) + r(x),$$

where $q(x)$ and $r(x)$ are polynomials each of degree at most $n-1$. Then

$$\int_{-1}^{1} w(x)f(x)\,\mathrm{d}x = \int_{-1}^{1} T_n(x)q(x)w(x)\,\mathrm{d}x + \int_{-1}^{1} r(x)w(x)\,\mathrm{d}x. \tag{12}$$

Due to Eq. (11), the first integral on the right-hand side of (12) vanishes. Thus

$$\begin{aligned} \int_{-1}^{1} w(x)f(x)\,\mathrm{d}x &= \int_{-1}^{1} r(x)w(x)\,\mathrm{d}x \\ &= \sum_{k=0}^{n-1} A_k\, r(x_k) \end{aligned}$$

since the coefficients have been chosen, according to (10), to give an exact result for polynomials of degree less than n.

Hence, $Q_n(f)$ is exact for any polynomial $f(x)$ of degree $2n-1$, implying an order of $2n$.

(e) (5 pts) For the Gauss-Chebyshev quadrature rule, it can be shown that the integral expression for the quadrature weights, as obtained in sub-problem (b), simplifies to

$$A_k = \frac{\pi}{n}, \quad \forall\, k = 0, \dots, n-1. \tag{13}$$

Implement a `C++/Eigen` function

```
double Gauss_Chebyshev(const std::function<double(double)> &f, int n);
```

that performs the n-point Gauss-Chebyshev quadrature to approximate the integral

$$I = \int_{-1}^{1} \frac{f(x)}{\sqrt{1-x^2}} \, \mathrm{d}x \; . \tag{14}$$

Solution:

```
double GaussChebyshev(const std::function<double(double)> &f, int n) {
    double q = 0.0;
    double A = M_PI/n;

    std::vector<double> x(n);
    for (int i = 0; i < n; ++i) {
        x[i] = std::cos( (i + 0.5) * M_PI / n);
        q += A * f(x[i]);
    }

    return q;
}
```

gauss_chebyshev.cpp

(f) (2 pts) Use the previous implementation to compute the quadrature error $E_{n,k}(f) := |I_k - Q_{n,w}(f)|$ for $n = 5,\ k = \{1, 2\}$, where

$$I_1 = \int_{-1}^{1} \frac{x^8}{\sqrt{1-x^2}} \, \mathrm{d}x \; ; \quad I_2 = \int_{-1}^{1} \frac{x^{10}}{\sqrt{1-x^2}} \, \mathrm{d}x \; .$$

Justify your results.

Note: The template '2.cpp' already contains the implementation for computing these errors.

Solution: We obtain the following output

```
For f(x)=x^8, Q_n(f): 0.859029 ;   Error: 3.33067e-16
For f(x)=x^10, Q_n(f):  0.76699 ;   Error: 0.00613592
```

This shows that the quadrature rule is exact for $f(x) = x^8$, however it is not exact for $f(x) = x^{10}$. This is consistent with the fact that a 5-point Gauss Chebyshev quadrature rule will be of order 10, that is, it will exactly integrate all polynomial functions $f \in \mathcal{P}_{2n-1} = \mathcal{P}_9$.

Exercise 3. *Initial value problem* (26 pts) [*Template: 3.cpp*]

Consider the second order IVP

$$\ddot{y}(t) = 2y(t)\left(1 + y^2(t)\right), \quad y(0) = 0,\ \dot{y}(0) = 1, \tag{15}$$

where $t \in (-\frac{\pi}{2}, \frac{\pi}{2})$. We want to solve it using the *implicit Euler* scheme.

(a) (3 pts) Write down (on paper) a function $\mathbf{f} : \mathbb{R}^2 \to \mathbb{R}^2$ such that

$$\dot{\mathbf{z}}(t) = \mathbf{f}\left(\mathbf{z}(t)\right), \quad \mathbf{z}(0) = (0, 1)^\top, \tag{16}$$

is equivalent to (15), where $\mathbf{z} : (-\frac{\pi}{2}, \frac{\pi}{2}) \to \mathbb{R}^2$ and $t \in (-\frac{\pi}{2}, \frac{\pi}{2})$. Moreover, implement a `C++/Eigen` function

```
Eigen::Vector2d f(const Eigen::Vector2d &x);
```

that takes $\mathbf{x} \in \mathbb{R}^2$ and returns the value $\mathbf{f}(\mathbf{x}) \in \mathbb{R}^2$.

Solution: Choose $\mathbf{f}(\mathbf{x}) = (x_2, 2x_1(1 + x_1^2))^\top$. The corresponding code is:

```
Eigen::Vector2d f(const Eigen::Vector2d &x) {
    return Eigen::Vector2d(x(1), 2.0 * x(0) * (1.0 + x(0) * x(0)));
}
```

newton.cpp

(b) (3 pts) The *implicit Euler* scheme with $N \in \mathbb{N}$ time steps of size $h > 0$ applied to (16) reads

$$\mathbf{z}^{(k+1)} = \mathbf{z}^{(k)} + h\mathbf{f}\left(\mathbf{z}^{(k+1)}\right), \quad \mathbf{z}^{(0)} = (0, 1)^\top, \tag{17}$$

for all $k \in \{0, \dots, N-1\}$. To find $\mathbf{z}^{(k+1)}$ for given $\mathbf{z}^{(k)}$, we have to solve a *non-linear* equation

$$\mathbf{F}(\mathbf{x}) = 0, \tag{18}$$

where $\mathbf{x} \in \mathbb{R}^2$ and

$$\mathbf{F}(\mathbf{x}) = \mathbf{z}^{(k)} + h\mathbf{f}\left(\mathbf{x}\right) - \mathbf{x}. \tag{19}$$

Write down (on paper) the *Jacobian* $\mathrm{D}\,\mathbf{F}(\mathbf{x})$. Moreover, implement a `C++/Eigen` function

```
Eigen::Matrix2d DF(const Eigen::Vector2d &x, double h);
```

that takes a point $\mathbf{x} \in \mathbb{R}^2$ and the step size $h > 0$ and returns the value $\mathrm{D}\,\mathbf{F}(\mathbf{x})$.

Solution: We have

$$\mathrm{D}\,\mathbf{F}(\mathbf{x}) = \begin{pmatrix} -1 & h \\ h(6x_1^2 + 2) & -1 \end{pmatrix}$$

and the code is given by:

```
Eigen::Matrix2d DF(const Eigen::Vector2d &x, double h) {
    double d = h * (6.0 * x(0) * x(0) + 2.0);
    Eigen::Matrix2d M;
    M << -1.0, h, d, -1.0;
    return M;
}
```

newton.cpp

(c) (6 pts) Implement a `C++/Eigen` function

```
Eigen::Vector2d Newton(Eigen::Vector2d x, double h, int n, double tol);
```

that returns the result of *Newton's* method applied to (18). The starting value is $\mathbf{x} \in \mathbb{R}^2$ and $h > 0$ is the step size in (19). Stop after $n \in \mathbb{N}$ iterations or when the *residual based* stopping criterion with tolerance `tol` is met.

Solution:

```cpp
Eigen::Vector2d Newton(Eigen::Vector2d x, double h, int n, double tol) {
    const Eigen::Vector2d zk = x;
    for (int i = 0; i < n; ++i) {
        if (F(x, zk, h).squaredNorm() <= tol * tol) return x;
        x += -DF(x, h).fullPivLu().solve(F(x, zk, h));
    }
    std::cout << "Warning: Maximal number of iterations reached." << std::endl;
    return x;
}
```

newton.cpp

(d) (8 pts) Implement a `C++/Eigen` function

```
Eigen::Vector2d QuasiNewton(Eigen::Vector2d x, double h, int n, double tol);
```

that performs the same task as `Newton`, but using the *Quasi-Newton* method via the *Sherman-Morrison-Woodbury* formula.

Solution:

```cpp
Eigen::Vector2d QuasiNewton(Eigen::Vector2d x, double h, int n, double tol) {
    const Eigen::Vector2d zk = x;
    Eigen::Matrix2d J_inv = DF(x, h).inverse();
    for (int i = 0; i < n; ++i) {
        Eigen::Vector2d dx = -J_inv * F(x, zk, h);
        x += dx;

        if (F(x, zk, h).squaredNorm() <= tol * tol) return x;

        Eigen::Vector2d T = J_inv * F(x, zk, h);
        J_inv += (-T * dx.transpose() / (dx.squaredNorm() + dx.transpose() * T)) *
    J_inv;
    }
    std::cout << "Warning: Maximal number of iterations reached." << std::endl;
    return x;
}
```

newton.cpp

(e) (6 pts) Implement a `C++/Eigen` function

```
Eigen::Vector2d ImplicitEuler(Eigen::Vector2d z0, double h, int N);
```

that performs the *implicit Euler* scheme (17) to solve the IVP (16) with initial value `z0`, applying $N \in \mathbb{N}$ time steps of size $h > 0$. Equation (18) should be solved by `Newton` or `QuasiNewton` with $n = 10$ iterations and tolerance `tol = 1.0e-8`. The return value of `ImplicitEuler` is the approximate solution at time $N \cdot h$.

Solution:

```cpp
Eigen::Vector2d ImplicitEuler(Eigen::Vector2d z, double h, int N) {
    for (int k = 0; k < N; ++k) {
        z = Newton(z, h, 10, 1.0e-8); //z = QuasiNewton(z, h, 10, 1.0e-8);
    }
    return z;
}
```

newton.cpp