Questions

Numerical Methods for CSE
Repeat Exam
FS 2018

Prof. Rima Alaifari

ETH Zürich, D-MATH

27 August 2018

1. *Periodic quadratic splines* (32pts)

Template: `1.cpp`.

Given knots $x_i \in \mathbb{R}$ and data points $y_i \in \mathbb{R}$ for $i = 0, \dots, N$ we want to find a periodic quadratic spline f which interpolates them. Assume that $0 < x_0 < x_1 < \cdots < x_N$, N is odd and $N \geq 3$.

(a) (6pts) Let

$$f(x) = a_i(x - x_i)^2 + b_i(x - x_i) + c_i \text{ for } x \in [x_i, x_{i+1}]. \tag{1}$$

Derive on paper the coefficients a_i, b_i, c_i, either explicitly, recursively or as solutions to a linear system of equations which depends only on $\{(x_j, y_j), j = 0, \dots, N\}$, so that

(α) $f(x_i) = y_i$ for every i,

(β) f has continuous first derivative on the interval (x_0, x_N),

(γ) $f'(x_0) = f'(x_N)$.

Hint: simplifying as much as possible the linear system derived in this subproblem could be very useful in the following subproblems.

Solution: Let $\Delta x_i = x_{i+1} - x_i$, $\Delta y_i = y_{i+1} - y_i$ for $i = 0, \dots, N-1$. By imposing the required conditions on each interval (x_i, x_{i+1}), we obtain that the coefficients must solve the $3N \times 3N$ linear system

$$\left(\begin{array}{c|c|c} Z & Z & I \\ \hline D^2 & D & I \\ \hline 2D & P & Z \end{array}\right) \begin{pmatrix} a_0 \\ \vdots \\ a_{N-1} \\ b_0 \\ \vdots \\ b_{N-1} \\ c_0 \\ \vdots \\ c_{N-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ y_{N-1} \\ y_1 \\ \vdots \\ y_N \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

where

- Z is the $N \times N$ zero matrix and I the identity;
- D is the diagonal matrix with elements $\Delta x_0, \dots, \Delta x_{N-1}$ on the diagonal;
- P is the $N \times N$ permutation matrix of $(1, -1, 0, \dots, 0)$, that is its row i has zeroes everywhere except for a 1 in position i and a -1 in position $(i + 1) \bmod N$.

For example, for $N = 3$ the system we need to solve is

$$\left(\begin{array}{ccc|ccc|ccc} & & & & & & 1 & & \\ & & & & & & & 1 & \\ & & & & & & & & 1 \\ \hline \Delta x_0^2 & & & \Delta x_0 & & & 1 & & \\ & \Delta x_1^2 & & & \Delta x_1 & & & 1 & \\ & & \Delta x_2^2 & & & \Delta x_2 & & & 1 \\ \hline 2\Delta x_0 & & & 1 & -1 & & & & \\ & 2\Delta x_1 & & & 1 & -1 & & & \\ & & 2\Delta x_2 & -1 & & 1 & & & \end{array}\right) \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ b_0 \\ b_1 \\ b_2 \\ c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_1 \\ y_2 \\ y_3 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

Although the formulation derived is sufficient to solve the exercise and to code up a solution for Point (d), we can further simplify the system using simple row operations. For instance subtracting row i from row $N + i$ for $i = 0, \dots, N - 1$ we obtain $c_i = y_i$ and we can reduce the system to

$$\left(\begin{array}{c|c} D^2 & D \\ \hline 2D & P \end{array}\right) \begin{pmatrix} a_0 \\ \vdots \\ a_{N-1} \\ b_0 \\ \vdots \\ b_{N-1} \end{pmatrix} = \begin{pmatrix} \Delta y_0 \\ \vdots \\ \Delta y_{N-1} \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

Then by multiplying row i by Δx_i and subtracting its double from row $N+i$ we obtain

$$\left(\begin{array}{c|c} D & I \\ \hline Z & Q \end{array}\right)\begin{pmatrix} a_0 \\ \vdots \\ a_{N-1} \\ b_0 \\ \vdots \\ b_{N-1} \end{pmatrix} = \begin{pmatrix} \Delta y_0/\Delta x_0 \\ \vdots \\ \Delta y_{N-1}/\Delta x_{N-1} \\ -2\Delta y_0/\Delta x_0 \\ \vdots \\ -2\Delta y_{N-1}/\Delta x_{N-1} \end{pmatrix}, \tag{2}$$

where now Q is the permutation matrix of $(-1,-1,0,\dots,0)$. Notice that once we solve the bottom N rows of the system, we can substitute the values of b_i and solve the first N rows for a_i. To solve the bottom N rows notice that by adding the first $N-1$ rows to to the last one with alternating signs it becomes $(0,\dots,0,0,-2)$. Therefore $b_{N-1} = \Delta y_{N-1}/\Delta x_{N-1} + 2\sum_{i=0}^{N-2}(-1)^i \Delta y_i/\Delta x_i$ and we can backsubstitute to obtain $b_i = -b_{i+1} + \Delta y_i/\Delta x_i$ for $i = 0,\dots,N-2$.

(b) (2pts) Prove/explain concisely why an f satisfying the requirements of Point (a) always exists and is unique.

Solution: The full analysis of Point (a) gives a unique way to determine the coefficients that always works under our assumptions. Even if we don't fully derive the recurrence relations but we stop at Equation (2), by expanding the determinant applying Laplace rule to the last row we can compute

$$d := \det\left(\begin{array}{c|c} D^2 & D \\ \hline Z & Q \end{array}\right) = \det(D^2)((-1)^{3N-1}\det F + (-1)^{4N-2}\det(F^T))$$

where F is a lower bidiagonal $(N-1)\times(N-1)$ matrix with only -1 on the two non-zero diagonals. Since N is odd

$$d = 2\det(D^2)\det F = 2\Delta x_0^2 \dots \Delta x_{N-1}^2 (-1)^{N-1} \neq 0.$$

(c) (2pts) What part of the statement in Point (b) fails when N is even? Why?

Solution: When N is even the last row of Q is the sum with alternating signs of all the previous rows, while all the other rows are still linearly independent, which means there exist infinite f's which satisfy the requirements of Point (a). Thus although existence is guaranteed, uniqueness is not.

(d) (10pts) Implement a `C++/Eigen` function `quadraticSpline` which given the datapoints `VectorXd x`, `VectorXd y` returns a matrix with N rows and 3 columns, which has in position $(i,0),(i,1),(i,2)$ respectively the coefficients a_i, b_i, c_i of (1).

Solution:

```
MatrixXd quadraticSpline(const VectorXd &x, const VectorXd &y) {
    int N = x.size()-1;
    VectorXd dx = x.tail(N) - x.head(N);
    VectorXd dx2 = dx.cwiseProduct(dx);
    MatrixXd A = MatrixXd::Zero(3*N, 3*N);

    // require f(x_i) = y_i
    for(int i=0; i<N; i++){
        A(i, 2*N+i) = 1;
    }

    // require f(x_{i+1}) = y_{i+1}
    for(int i=0; i<N; i++){
        A(N+i, i) = dx2(i);
        A(N+i, N+i) = dx(i);
        A(N+i, 2*N+i) = 1;
    }

    // require f' to be continuous at x_i
```

```
    for(int i=0; i<N; i++){
        A(2*N+i, i) = 2*dx(i);
        A(2*N+i, N+i) = 1;
        A(2*N+i, N + ((i+1) % N)) = -1;
    }

    VectorXd rhs(3*N);
    rhs << y.head(N), y.tail(N), VectorXd::Zero(N);

    VectorXd out = A.fullPivLu().solve(rhs);
    return Map<MatrixXd>(out.data(), N, 3);
}
```

(e) (12pts) Implement a C++/Eigen function `quadraticSplineFast` which takes the same input and returns the same output of `quadraticSpline`, but runs in only $O(N)$ time.
Hint: if you are unable to derive an explicit $O(N)$ procedure, in this exercise you can assume that the `SparseLU` solver from `Eigen` runs in $O(n)$ time when implemented correctly for an $n \times n$ matrix which has $O(1)$ non-empty diagonals.
Solution: An explicit $O(N)$ implementation:

```
MatrixXd quadraticSplineFast(const VectorXd &x, const VectorXd &y) {
    int N = x.size()-1;
    VectorXd dx = x.tail(N) - x.head(N);
    VectorXd dx2 = dx.cwiseProduct(dx);
    VectorXd dy = y.tail(N) - y.head(N);
    MatrixXd out(N,3);

    VectorXd b = dy.cwiseQuotient(dx);
    int sign = 1;

    for(int i=0; i<N-1; i++){
        sign *= -1;
        b(N-1) += sign*b(i);
    }

    for(int i=N-2; i>=0; i--){
        b(i) = -b(i+1) + 2*dy(i)/dx(i);
    }

    VectorXd a = - b.cwiseQuotient(dx) + dy.cwiseQuotient(dx2);

    out << a, b, y.head(N);
    return out;
}
```

An implementation with the `SparseLU` solver:

```
MatrixXd quadraticSplineFast_SparseLU(const VectorXd &x, const VectorXd &y) {
    int N = x.size()-1;
    VectorXd dx = x.tail(N) - x.head(N);
    VectorXd dx2 = dx.cwiseProduct(dx);
    SparseMatrix<double> A(3*N, 3*N);
    A.reserve(VectorXd::Constant(3*N, 3));

    // require f(x_i) = y_i
    for(int i=0; i<N; i++){
        A.insert(i, 2*N+i) = 1;
    }
```

```
    // require f(x_{i+1}) = y_{i+1}
    for(int i=0; i<N; i++){
        A.insert(N+i, i) = dx2(i);
        A.insert(N+i, N+i) = dx(i);
        A.insert(N+i, 2*N+i) = 1;
    }

    // require f' to be continuous at x_i
    for(int i=0; i<N; i++){
        A.insert(2*N+i, i) = 2*dx(i);
        A.insert(2*N+i, N+i) = 1;
        A.insert(2*N+i, N + ((i+1) % N)) = -1;
    }

    A.makeCompressed();

    VectorXd b(3*N);
    b << y.head(N), y.tail(N), VectorXd::Zero(N);

    SparseLU<SparseMatrix<double>> solver;
    solver.compute(A);
    VectorXd out = solver.solve(b);

    return Map<MatrixXd>(out.data(), N, 3);
}
```

2. *Data fitting* (22pts)

Template: `2.cpp`.

Let

$$f(k) := \alpha + \beta \sin\left(\frac{2\pi}{366}k\right) + \gamma \cos\left(\frac{2\pi}{366}k\right) + \delta \sin\left(\frac{\pi}{366}k\right)\cos\left(\frac{\pi}{366}k\right), \tag{3}$$

where $k \in \mathbb{N}$ and $\alpha,\ \beta,\ \gamma,\ \delta$ are some parameters.

Consider the following data on the number $H(k)$ of daylight hours in the k-th day of the year:

k	11	32	77	121	152
$H(k)$	9	10	12	14	15

We want to find the best approximation of this data in the least squares sense, using the function $f(k)$.

(a) (4pts) Derive on paper the linear least squares problem (LLSQ)

$$\mathbf{x}^* = \underset{\mathbf{x}\in\mathbb{R}^n}{\arg\min} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2\,, \tag{4}$$

by specifying $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{b} \in \mathbb{R}^m$, for $m, n \in \mathbb{N}$.

Solution: Let $\mathbf{A}_{i,:}$ denote the i-th row of matrix $\mathbf{A}$. We have

$$\mathbf{x} = \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix},$$

$$\mathbf{b} = \begin{pmatrix} 9 \\ 10 \\ 12 \\ 14 \\ 15 \end{pmatrix},$$

$$\mathbf{A}_{i,} = \begin{pmatrix} 1 & \sin\left(\frac{2\pi}{366}k_i\right) & \cos\left(\frac{2\pi}{366}k_i\right) & \sin\left(\frac{\pi}{366}k_i\right)cos\left(\frac{\pi}{366}k_i\right) \end{pmatrix}, \text{ for } i = 1, 2, \ldots, 5.$$

(b) (3pts) Is the solution to the above LLSQ unique? Explain. **Solution:**
The solution is not unique. Using $\sin(2x) = 2\sin(x)\cos(x)$, rewrite

$$\begin{aligned} f(k) &= \alpha + \beta \sin\left(\frac{2\pi}{366}k\right) + \gamma \cos\left(\frac{2\pi}{366}k\right) + \delta \sin\left(\frac{\pi}{366}k\right)\cos\left(\frac{\pi}{366}k\right) \\ &= \alpha + (\beta + \delta/2) \sin\left(\frac{2\pi}{366}k\right) + \gamma \cos\left(\frac{2\pi}{366}k\right). \end{aligned}$$

Hence, the second and fourth column in the matrix $\mathbf{A}$ are linearly dependent, which implies that $r := \text{rank}(\mathbf{A})$ is less than n and there is no unique solution.

(c) (10pts) Implement a `C++/Eigen` function `llsq_gsol` which given `MatrixXd A` and `VectorXd b` returns the *generalized* solution `VectorXd x` and the least squares error `llsq_err`.

Hint: Use SVD.

Solution:

```
double llsq_gsol(const MatrixXd& A, const VectorXd& b, VectorXd& x) {

    double llsq_err = 0;

    size_t m = A.rows();
    size_t n = A.cols();
    JacobiSVD<MatrixXd> svd(A, ComputeFullU | ComputeFullV); // SVD decomposition
```

```
    int r=0;
    double tol=1e-12;
    VectorXd sv = svd.singularValues();
    for (int i=0; i<sv.size();; i++) { // number of non-zero singular values
        if (fabs(sv(i)) > tol)
        r++;
    }

    // Compute the matrices in Moore-Penrose pseudoinverse
    MatrixXd invD_(r,r);
    for (int i=0; i<r; i++)
        invD_(i,i) = 1.0/sv(i);

    MatrixXd U = svd.matrixU();
    MatrixXd U_ = U.block(0,0,m,r);

    MatrixXd V = svd.matrixV();
    MatrixXd V_ = V.block(0,0,n,r);

    x = V_*invD_*U_.transpose()*b; // Generalized solution

    llsq_err = (U.block(0,r,m,m-r).transpose()*b).norm() // least-squares error

    return llsq_err;
}
```

(d) (5pts) Run your implementation of `llsq_solve` for the LLSQ in (a) and print the solution and the least squares error.

Solution:

```
int main() {

    size_t m = 5;
    size_t n = 4;

    VectorXd t(m), b(m);
    t << 11, 32, 77, 121, 152;
    b << 9, 10, 12, 14, 15;

    MatrixXd A(m,n);
    for (int i=0; i<m; i++) {
        A(i,0) = 1;
        A(i,1) = sin(2*M_PI*t(i)/366);
        A(i,2) = cos(2*M_PI*t(i)/366);
        A(i,3) = sin(M_PI*t(i)/366)*cos(M_PI*t(i)/366);
    }

    VectorXd x(n);
    double llsq_err = llsq_svd(A, b, x);

    std::cout << "x: " <<  x.transpose() << std::endl;
    std::cout << "llsq error: " << llsq_err << std::endl;
}
```

For the given data fitting problem,

$$\mathbf{x}^* = (11.9072,\ 0.69409,\ -2.98301,\ 0.347045)^\top$$
$$\|\mathbf{A}\mathbf{x}^* - \mathbf{b}\|_2 = 0.27035$$

3. *Unconstrained optimization* (32pts)

Template: `3.cpp`.

According to the WHO growth reference data from the year 2007, the weight of 5-year-old girls in the world can be modelled by the Gaussian distribution function

$$f(w;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(w-\mu)^2}{2\sigma^2}\right). \tag{5}$$

Given M randomized samples of weights

$$S = \{w_1, w_2, \ldots, w_M\}, \text{ for } M \in \mathbb{N}, \tag{6}$$

the objective is to estimate the mean μ and variance σ.

(a) (3pt) Formulate on paper the setting described above as an unconstrained minimization problem,

$$\mathbf{x}^* = \arg\min_{\mathbf{x}\in\mathbb{R}^n} \mathbf{F}_S(\mathbf{x}), \tag{7}$$

by specifying $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{F}_S : \mathbb{R}^n \to \mathbb{R}^m$, for $m, n \in \mathbb{N}$.

Solution:

As the weight of girl i is independent of weight of girl j, the likelihood of observing the set S is

$$P(\mu,\sigma;S) = \prod_{j=1}^{M} f(w_j;\mu,\sigma).$$

To estimate μ and σ, $P(\mu,\sigma;S)$ has to be maximized. As maximizing P is equivalent to minimizing $-\log(P)$, we get the following minimization problem:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}\in\mathbb{R}^n} \mathbf{F}_S(\mathbf{x}),$$

where

$$\begin{aligned}
\mathbf{x} &:= (\mu,\sigma)^\top,\\
F_S(\mathbf{x}) &:= -\log(P(\mu,\sigma;S))\\
&= -\log\left(\prod_{j=1}^{M} f(w_j;\mu,\sigma)\right)\\
&= -\sum_{j=1}^{M}\log(f(w_j;\mu,\sigma)).
\end{aligned}$$

(b) (4pts) Derive on paper the gradient of $\mathbf{F}_S(\mathbf{x})$ in (a).

Solution:

Notation: $f_j := f(w_j;\mu,\sigma)$.

We have

$$\begin{aligned}
\nabla_{\mathbf{x}} F_S(\mathbf{x}) &= \left(\frac{\partial F_S}{\partial x_0}, \frac{\partial F_S}{\partial x_1}\right)^\top\\
&= \left(-\sum_{j=1}^{M}\frac{1}{f_j}\frac{\partial f_j}{\partial x_0}, -\sum_{j=1}^{M}\frac{1}{f_j}\frac{\partial f_j}{\partial x_1}\right)^\top\\
&= -\sum_{j=1}^{M}\frac{1}{f_j}\nabla_{\mathbf{x}} f_j,
\end{aligned}$$

and

$$\frac{\partial f_j}{\partial x_0} = \frac{\partial f_j}{\partial \mu} = \frac{w_j - \mu}{\sigma^2} f_j,$$
$$\frac{\partial f_j}{\partial x_1} = \frac{\partial f_j}{\partial \sigma} = \left(-\frac{1}{\sigma} + \frac{(w_j - \mu)^2}{\sigma^3} \right) f_j.$$

This yields

$$\nabla_{\mathbf{x}} F_S(\mathbf{x}) = -\sum_{j=1}^{M} \left(\frac{w_j - \mu}{\sigma^2}, -\frac{1}{\sigma} + \frac{(w_j - \mu)^2}{\sigma^3} \right)^\top.$$

(c) (5pts) Implement a C++/Eigen function:

```
VectorXd evalGradF(const VectorXd& S, const VectorXd& x);
```

which computes the gradient of $\mathbf{F}_S(\mathbf{x})$ devised in (b).

Solution:

```
// routine:  evalGradF
// " Computes the Gradient of F(x;S) "
// (in) S: randomized samples of weights
// (in) x:  model parameters
// (out) gradF: Gradient of F(x;S) wrt x.
VectorXd evalGradF(const VectorXd& S, const VectorXd& x) {

    int n = x.size();
    int M = S.size();
    VectorXd gradF(n);

    gradF.setZero();
    for (int j=0; j<M; j++) {
        double w = S(j);
        gradF(0) += (w - x(0))/(x(1)*x(1));
        gradF(1) += -1./x(1) + (w - x(0))*(w - x(0))/pow(x(1),3);
    }
    gradF *= -1;

    return gradF;
}
```

(d) (4pts) Derive on paper the Hessian of $\mathbf{F}_S(\mathbf{x})$ in (a).

Solution: We have

$$\mathbf{H}_{F_S}(\mathbf{x}) \equiv \left(\frac{\partial(\nabla_{\mathbf{x}} F_S)}{\partial \mu}, \frac{\partial(\nabla_{\mathbf{x}} F_S)}{\partial \sigma} \right).$$

Using the gradient of $F_S(\mathbf{x})$ from b), we get

$$\frac{\partial(\nabla_{\mathbf{x}} F_S)}{\partial \mu} = -\sum_{j=1}^{M} \left(-\frac{1}{\sigma^2}, -\frac{2(w_j - \mu)}{\sigma^3} \right)^\top,$$
$$\frac{\partial(\nabla_{\mathbf{x}} F_S)}{\partial \sigma} = -\sum_{j=1}^{M} \left(-\frac{2(w_j - \mu)}{\sigma^3}, \frac{1}{\sigma^2} - \frac{3(w_j - \mu)^2}{\sigma^4} \right)^\top,$$

which implies

$$\mathbf{H}_{F_S}(\mathbf{x}) = -\sum_{j=1}^{M} \begin{pmatrix} -\frac{1}{\sigma^2} & -\frac{2(w_j-\mu)}{\sigma^3} \\ -\frac{2(w_j-\mu)}{\sigma^3} & \frac{1}{\sigma^2} - \frac{3(w_j-\mu)^2}{\sigma^4} \end{pmatrix}.$$

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

```
MatrixXd evalHessF(const VectorXd& S, const VectorXd& x);
```

which computes the Hessian of $\mathbf{F}_S(\mathbf{x})$ devised in (d).

Solution:

```
// routine:  evalHessF
// " Computes the Hessian of F(x;S) "
// (in) S: randomized samples of weights
// (in) x:  model parameters
// (out) hessF: Hessian of F(x;S) wrt x.
MatrixXd evalHessF(const VectorXd& S, const VectorXd& x) {

    int n = x.size();
    int M = S.size();
    MatrixXd hessF(n,n);

    hessF.setZero();
    for (int j=0; j<M; j++) {
        double w = S(j);
        hessF(0,0) += -1./(x(1)*x(1));
        hessF(1,0) += -2*(w - x(0))/pow(x(1),3);
        hessF(0,1) += -2*(w - x(0))/pow(x(1),3);
        hessF(1,1) += 1./(x(1)*x(1)) - 3*(w - x(0))*(w - x(0))/pow(x(1),4);
    }
    hessF *= -1;

    return hessF;
}
```

(f) (2pts) Write explicitly on paper an iteration of the Newton method to solve subproblem (a).

Solution: The Newton iteration for the optimization problem is

$$\mathbf{x}_{k+1} = \mathbf{x}_k - (\mathbf{H}_{F_S}(\mathbf{x}_k))^{-1}\nabla_{\mathbf{x}} F_S(\mathbf{x}_k).$$

(g) (5pts) Implement a C++/Eigen function for the Newton optimization devised in (f):
`void newtonOpt(const VectorXd& S, const double tol, const int maxItr, VectorXd& x);`
Use appropriate termination criteria.

Solution:

```
// routine:  newtonOpt
// " Solves the unconstrained minimization problem using Newton method "
// (in) S: randomized samples of weights
// (in) tol:  tolerance
// (in) maxItr:  maximum iterations
// (in/out) x:  model parameters
void newtonOpt(const VectorXd& S, const double tol, const int maxItr,
    VectorXd& x) {

    const int n = x.size();

    VectorXd dx;
    VectorXd gradF(n);
    MatrixXd hessF(n,n);

    for (int itr=0; itr<maxItr; itr++) {

        gradF = evalGradF(S, x);
        hessF = evalHessF(S, x);

        dx = hessF.fullPivLu().solve(gradF);
```

```
        std::cout << itr << "\t" << dx.norm() << std::endl;
        if (dx.norm() <= tol)
            break;

        x -= dx;
    }

}
```

(h) (3pt) Print the estimates for mean and variance obtained from your implementation, for the data set given in the template.

Solution:

```
// routine:  runNewtonOpt
// " Executes the Newton Optimization "
// (in) S: randomized samples of weights
void runNewtonOpt(const VectorXd& S) {

    double tol = 1e-8; // tolerance
    int maxItr = 100; // maximum iterations

    VectorXd x(2);
    x(0) = 20; x(1) = 3; // initial guess
    newtonOpt(S, tol, maxItr, x);
    std::cout << "Mean and variance:\n" << x.transpose() << std::endl;
}
```

We get the estimates: $\mu = 20.29$, $\sigma = 3.72302$.