NumCSE

Autumn Semester 2017

Prof. Rima Alaifari

Exercise sheet 9
Function Approximation

P. Bansal

Problem 9.1: Adaptive polynomial interpolation

For approximating a given function by a polynomial interpolant upto a desired tolerance, position of the interpolation nodes is crucial. Here, we look at an **a posteriori** adaptive strategy that employs a **greedy algorithm** to build the set of interpolation nodes based on intermediate interpolants.

Templates: `adapPolyIpol.cpp`, `newtonIpol.hpp`

The greedy algorithm for adaptive polynomial interpolation is described below:

> Given a function $f : [a, b] \mapsto \mathbb{R}$, start with an initial node set $\mathcal{T}_0 := \{\frac{1}{2}(b + a)\}$. Based on a fixed finite set $\mathcal{S} \subset [a, b]$ of **sampling points,** augment the set of nodes as
>
> $$\mathcal{T}_{n+1} = \mathcal{T}_n \cup \left\{ \underset{t \in \mathcal{S}}{\operatorname{argmax}} \, |f(t) - \mathsf{I}_{\mathcal{T}_n}(t)| \right\}, \tag{9.1}$$
>
> where $\mathsf{I}_{\mathcal{T}_n}$ is the polynomial interpolation operator for the node set $\mathcal{T}_n$, until
>
> $$\max_{t \in \mathcal{S}} |f(t) - \mathsf{I}_{\mathcal{T}_n}(t)| \leq \mathtt{tol} \cdot \max_{t \in \mathcal{S}} |f(t)| \ . \tag{9.2}$$

First, we need a function which computes the interpolating polynomial for a given set of nodes.

(a) Implement a C++ function

```
VectorXd divDiff(const VectorXd& t, const VectorXd& y);
```

which computes the coefficients of the polynomial interpolant using divided differences, refer tablet notes. Here, `t` and `y` are the interpolation nodes and the corresponding function values.

SOLUTION:

C++11-code 9.1: Divided differences

```
VectorXd divDiff(const VectorXd& t, const VectorXd& y) {
    const unsigned n = y.size()-1 ;
    VectorXd coeffs(y);

    for (int k=0; k<n; k++)
        for (int j=k; j<n; j++)
            coeffs(j+1) = (coeffs(j+1) - coeffs(k))/ (t(j+1) - t(k));

    return coeffs;
}
```

(b) Implement a C++ function

```
template <class Function>
VectorXd adapPolyIpol(const Function& f, double a, double b,
                      double tol, int N,
                      Eigen::VectorXd& adaptive_nodes);
```

that implements the algorithm described above. The arguments are: the function handle `f`, the interval bounds `a`, `b`, the relative tolerance `tol`, the number `N` of *equidistant* sampling points to compute the error:

$$\mathcal{S} := \left\{ a + (b-a)\frac{j}{N},\ j = 0, \ldots, N \right\}.$$

and the final set of interpolation nodes returned in `adaptive_nodes`. For each intermediate set $\mathcal{T}_n$, `adapPolyIpol` should compute the error:

$$\epsilon_n := \max_{t \in \mathcal{S}} |f(t) - \mathsf{T}_{\mathcal{T}_n}(t)| \tag{9.3}$$

and return these error values in a vector.

Remark: Use a suitable lambda function for the type `Function` and use the function `intPolyEval` defined in `newtonIpol.hpp` to evaluate the polynomial interpolant.

SOLUTION:

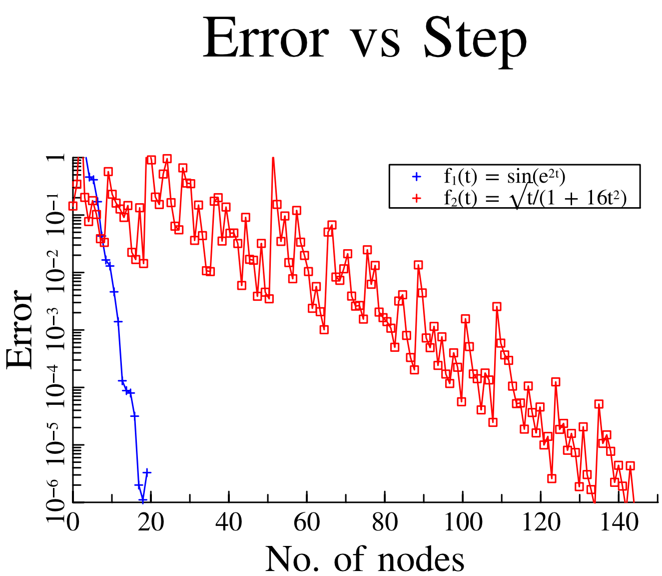
C++11-code 9.2: Adaptive polynomial interpolation

```cpp
template <class Function>
Eigen::VectorXd adapPolyIpol(const Function& f,
                             const double a, const double b,
                             const double tol, const unsigned N,
                             Eigen::VectorXd& adaptive_nodes) {

    // Generate sampling points and evaluate f there
    Eigen::VectorXd sampling_points = Eigen::VectorXd::LinSpaced(N, a, b);
    Eigen::VectorXd fvals_at_sampling_points =
        sampling_points.unaryExpr(f);

    // Approximate max|f(x)|
    double fmax = fvals_at_sampling_points.cwiseAbs().maxCoeff();

    // Adaptive mesh (initial node)
    std::vector<double> t { (a+b)/2. }; // Set of interpolation nodes
    std::vector<double> y { static_cast<double>(f((a+b)/2.)) }; // Values
        at nodes
    std::vector<double> errors; // Error at nodes

    for (int i = 0; i < N; ++i) {
        // *** Step 1:  interpolate with current nodes
        // need to convert std::vector to
        // Eigen::VectorXd to use the function interpoyval
        Eigen::Map<Eigen::VectorXd> te(t.data(), t.size());
        Eigen::Map<Eigen::VectorXd> ye(y.data(), y.size());
        Eigen::VectorXd intpolyvals_at_sampling_points;
        intPolyEval(te, ye, sampling_points,
            intpolyvals_at_sampling_points);

        // *** Step 2:  find node where error is the largest
        Eigen::VectorXd err = (fvals_at_sampling_points -
            intpolyvals_at_sampling_points).cwiseAbs();
        double max = 0; int idx = 0;
```

```cpp
        max = err.maxCoeff(&idx); // see Eigen "Visitor"

        // Step 3:  check termination criteria, return results
        if (max < tol * fmax) {
            adaptive_nodes = te;
            Eigen::Map<Eigen::VectorXd> errVsStep(errors.data(),
                errors.size());
            return errVsStep;
        }

        // Step 4:  add this node to our set of nodes and save error
        errors.push_back(max);
        t.push_back(sampling_points(idx));
        y.push_back(fvals_at_sampling_points(idx));
    }
    std::cerr << "Desired accuracy could not be reached." << std::endl;

    adaptive_nodes = sampling_points; // return all sampling points
}
```

(c) For $f_1(t) := \sin(e^{2t})$ and $f_2(t) = \frac{\sqrt{t}}{1+16t^2}$ plot ϵ_n versus n, the number of interpolation nodes, in the interval $[a,b] = [0,1]$ using `N=1000` sampling points and `tol = 1e-6`.

SOLUTION:



Problem 9.2: Chebyshev polynomials and their properties

In this problem, we will examine Chebyshev polynomials and a few of their many properties.
Templates: `bestApproxCheb.cpp`

Let $T_n \in \mathcal{P}_n$ be the n-th Chebyshev polynomial and $\xi_0^{(n)}, \dots, \xi_{n-1}^{(n)}$ be the n zeros of T_n, where

$$\xi_j^{(n)} = \cos\left(\frac{2j+1}{2n}\pi\right), \quad j = 0, \dots, n-1. \tag{9.4}$$

Define a family of discrete L^2 semi-inner products (i.e. not conjugate symmetric):

$$(f,g)_n := \sum_{j=0}^{n-1} f(\xi_j^{(n)}) g(\xi_j^{(n)}), \quad f,g \in C^0([-1,1]) \tag{9.5}$$

Also define a special weighted L^2 semi-inner product:

$$(f,g)_w := \int_{-1}^{1} \frac{1}{\sqrt{1-t^2}} f(t)g(t)\,dt \quad f,g \in C^0([-1,1]) \tag{9.6}$$

(a) Show that the Chebyshev polynomials are orthogonal w.r.t. the semi-inner product defined in Eq. (9.6), i.e. $(T_k, T_l)_w = 0$ for every $k \neq l$.

Hint: Use the trigonometric identity $2\cos(x)\cos(y) = \cos(x+y) + \cos(x-y)$.

SOLUTION:

For $k, l = 0, \dots, n$ with $k \neq l$, by using the substitution $s = \arccos t$ ($ds = -\frac{1}{\sqrt{1-t^2}}\,dt$) and simple trigonometric identities we readily compute:

$$\begin{aligned}
(T_k, T_l)_w &= \int_{-1}^{1} \frac{1}{\sqrt{1-t^2}} \cos(k \arccos t)\cos(l \arccos t)\,dt \\
&= \int_0^{\pi} \cos(ks)\cos(ls)\,ds \\
&= \frac{1}{2}\int_0^{\pi} \cos((k+l)s) + \cos((k-l)s)\,ds \\
&= \frac{1}{2}\left\{[\sin((k+l)s)/(k+l)]_0^{\pi} + [\sin((k-l)s)/(k-l)]_0^{\pi}\right\} \\
&= 0
\end{aligned}$$

since $k + l \neq 0$ and $k - l \neq 0$.

Consider the following statement:

Theorem 9.7.

The family of polynomials $\{T_0, \dots, T_n\}$ is an orthogonal basis of $\mathcal{P}_n$ with respect to the inner product $(\cdot,\cdot)_{n+1}$ defined in Eq. (9.5).

(b) Prove Thm. 9.7. **Hint:** Use the relationship of trigonometric functions and the complex exponential.

SOLUTION:

The 3-term recursion for Chebyshev polynomials, see Thm 6.1.77 in the lecture notes, establishes that the Chebyshev polynomials indeed form a basis. Next, we need to check the orthogonality condition. For $k, l = 0, \dots, n+1$ and $k \neq l$, by Eq. (9.4) we have:

$$
\begin{aligned}
(T_k, T_l)_{n+1} &= \sum_{j=0}^{n} T_k\left(\xi_j^{(n+1)}\right) T_l\left(\xi_j^{(n+1)}\right) && (9.8)\\
&= \sum_{j=0}^{n} \cos\left(k\frac{2j+1}{2(n+1)}\pi\right)\cos\left(l\frac{2j+1}{2(n+1)}\pi\right) && (9.9)\\
&= \frac{1}{2}\sum_{j=0}^{n}\left[\cos\left((k+l)\frac{2j+1}{2(n+1)}\pi\right)+\cos\left((k-l)\frac{2j+1}{2(n+1)}\pi\right)\right] && (9.10)\\
& && (9.11)
\end{aligned}
$$

It would suffice to show that:

$$
\sum_{j=0}^{n} \cos\left(m\frac{2j+1}{2(n+1)}\pi\right) = 0, \qquad \forall m \in \mathbb{Z}\backslash\{0\} \tag{9.12}
$$

Rewrite as:

$$
\sum_{j=0}^{n} \cos\left(m\frac{2j+1}{2(n+1)}\pi\right) = \mathrm{Re}\left(\sum_{j=0}^{n} e^{im\frac{2j+1}{2(n+1)}\pi}\right) = \mathrm{Re}\left(e^{im\frac{1}{2(n+1)}\pi}\sum_{j=0}^{n} e^{im\frac{j}{n+1}\pi}\right)
$$

Finally, use the formula for summing a geometric series:

$$
\begin{aligned}
e^{im\frac{1}{2(n+1)}\pi}\sum_{j=0}^{n} e^{im\frac{j}{n+1}\pi} &= e^{im\frac{1}{2(n+1)}\pi}\frac{1-e^{im\frac{n+1}{n+1}\pi}}{1-e^{im\frac{1}{n+1}\pi}}\\
= \frac{1-e^{im\pi}}{e^{-im\frac{1}{2(n+1)}\pi}-e^{im\frac{1}{2(n+1)}\pi}} &= -\frac{1-e^{im\pi}}{2}\frac{1}{i\sin\left(m\frac{1}{2(n+1)}\pi\right)}\\
\in i\cdot\mathbb{R} \quad &\Longrightarrow \mathrm{Re}\left(e^{im\frac{1}{2(n+1)}\pi}\sum_{j=0}^{n} e^{im\frac{j}{n+1}\pi}\right) = 0.
\end{aligned}
$$

(c) Implement a C++ code to numerically test the assertion of Thm. 9.7.

Use the following code for evaluating Chebyshev polynomials based on their recursive definition:

C++11-code 9.3: Evaluate Chebyshev polynomials

```
vector<double> chebPolyEval(const int &n,const double &x)
{
    vector<double> V={1,x};
    for (int k=1; k<n; k++)
        V.push_back(2*x*V[k]-V[k-1]);
```

```
    return V;
}
```

SOLUTION:

From the previous subproblem, we already know that $\{T_0, \dots, T_n\}$ is a basis for $\mathcal{P}_n$. So, we just check the orthogonality, see Code 9.3.

C++11-code 9.4: Solution of (b)

```cpp
// Check the orthogonality of Chebyshev polynomials
    n=10;
    vector<double> V;
    Eigen::MatrixXd scal(n+1,n+1);
    for (int j=0; j<n+1; j++) {
        V=chebPolyEval(n,cos(M_PI*(2*j+1)/2/(n+1)));
        for (int k=0; k<n+1; k++) scal(j,k)=V[k];
    }

    double maxOrthErr = 1e-40;
    for (int k=0; k<n+1; k++)
        for (int l=k+1; l<n+1; l++)
            maxOrthErr = max( maxOrthErr, scal.col(k).dot(scal.col(l)) );
    cout<< "Maximum orthogonality error: " << maxOrthErr <<endl;
```

(d) Given a function $f \in C^0([-1,1])$, find the best approximation $q_n \in \mathcal{P}_n$ of f in the discrete L^2-norm:

$$q_n = \underset{p \in \mathcal{P}_n}{\operatorname{argmin}} |f - p|_{n+1} ,$$

where $|v|_{n+1} = \sqrt{(v,v)_{n+1}}$ for any v. Represent q_n as an expansion in Chebyshev polynomials:

$$q_n = \sum_{j=0}^{n} \alpha_j T_j , \tag{9.13}$$

for suitable coefficients $\alpha_j \in \mathbb{R}$. The task boils down to determining the coefficients α_j.

SOLUTION:

In view of the Thm. 9.7, the family $\{T_0, \dots, T_n\}$ is an orthogonal basis of $\mathcal{P}_n$ with respect to the inner product $(\ ,\ )_{n+1}$. By Eq. (9.8) and Eq. (9.12) we have:

$$\lambda_k^2 := |T_k|_{n+1}^2 = (T_k, T_k)_{n+1} = \begin{cases} \frac{1}{2}\sum_{j=0}^{n}(\cos(0)+\cos(0)) = n+1 & \text{if } k=0 \\ \frac{1}{2}\sum_{j=0}^{n}\cos(0) = (n+1)/2 & \text{otherwise} \end{cases} \tag{9.14}$$

The family $\{T_k/\lambda_k : k = 0, \dots, n\}$ is an orthonormal basis of $\mathcal{P}_n$ with respect to the inner product $(\ ,\ )_{n+1}$. Hence:

$$q_n = \sum_{j=0}^{n} (f, T_j/\lambda_j)_{n+1} \frac{T_j}{\lambda_j} = \sum_{j=0}^{n} \alpha_j T_j, \quad \alpha_j = \frac{1}{n+1} \begin{cases} (f, T_j)_{n+1} & \text{if } k=0 \\ 2(f, T_j)_{n+1} & \text{otherwise} \end{cases}$$

(e) Implement a C++ function that returns the vector of coefficients $(\alpha_j)_j$ in Eq. (9.13) given a function f:

```
template <typename Function>
void bestApproxCheb(const Function &f, Eigen::VectorXd &alpha)
```

Note that the degree of the polynomial is indirectly passed with the length of the output `alpha`. The input `f` is a lambda-function, example:

```
auto f = [] (double & x) {return 1/(pow(5*x,2)+1);};
```

SOLUTION:

C++11-code 9.5: Solution of (e)

```
template <typename Function>
void bestApproxCheb(const Function &f, Eigen::VectorXd &alpha) {
    int n=alpha.size()-1;
    Eigen::VectorXd fn(n+1);
    for (int k=0; k<n+1; k++) {
        double temp=cos(M_PI*(2*k+1)/2/(n+1));
        fn(k)=f(temp);
    }

    vector<double> V;
    Eigen::MatrixXd scal(n+1,n+1);
    for (int j=0; j<n+1; j++) {
        V=chebPolyEval(n,cos(M_PI*(2*j+1)/2/(n+1)));
        for (int k=0; k<n+1; k++) scal(j,k)=V[k];
    }

    for (int k=0; k<n+1; k++) {
        alpha(k)=0;
        for (int j=0; j<n+1; j++) {
            alpha(k)+=2*fn(j)*scal(j,k)/(n+1);
        }
    }
    alpha(0)=alpha(0)/2;
}
```

(f) Test `bestApproxCheb` for the function $f(x) = \frac{1}{(5x)^2+1}$ and $n = 20$. Approximate the supremum norm of the approximation error by sampling on an equidistant grid with 10^6 points.

SOLUTION:

C++11-code 9.6: Solution of (f)

```
// Test the implementation
    auto f = [] (double & x) {return 1/(pow(5*x,2)+1);};
    n=20;
    Eigen::VectorXd alpha(n+1);
    bestApproxCheb(f, alpha);

    // Compute the error
    int nPts = 1e+6;
    Eigen::VectorXd X = Eigen::VectorXd::LinSpaced(nPts,-1,1);
    auto qn = [&alpha,&n] (double & x) {
        double val=0;
        vector<double> V=chebPolyEval(n,x);
        for (int k=0; k<n+1; k++) val+=alpha(k)*V[k];
        return val;
    };

    Eigen::VectorXd polyExact(nPts), polyApprox(nPts);
    for (int i=0; i<nPts; i++) {
        polyExact(i) = f(X(i));
        polyApprox(i) = qn(X(i));
    }

    double err_max=1e-20;
    for (int i=0; i<nPts; i++)
        err_max=std::max(err_max,abs(polyExact(i)-polyApprox(i)));
    cout<<"Error: "<< err_max <<endl;
```

The output is shown in Sub-problem (f).