NumCSE

Autumn Semester 2017

Prof. Rima Alaifari

Exercise sheet 7
Polynomial Interpolation

P. Bansal

Problem 7.1: Lagrange interpolant and Newton basis

In this problem, we look at some properties of Lagrange polynomials and **compute by hand** the Lagrange interpolant for a small data set. Alternatively, we compute the interpolating polynomial using the Newton basis.

Let $t_j \in \mathbb{R}$, for $j = 0, \ldots, n$, represent distinct nodes i.e. $t_i \neq t_j$ if $i \neq j$. Let L_i denote the i-th Lagrange polynomial for these given nodes. Hence, the Lagrange interpolant p through the data points $(t_i, y_i)_{i=0}^n$ has the representation

$$p(t) = \sum_{i=0}^{n} y_i L_i(t) \quad \text{with} \quad L_i(t) := \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{t - t_j}{t_i - t_j} \,. \tag{7.1}$$

Let $\omega(t) := \prod_{j=0}^n (t - t_j)$ and $\lambda_i := \dfrac{1}{\omega'(t_i)}$.

(a) Prove the following:

(i) $\displaystyle\sum_{i=0}^{n} L_i(t) = 1 \quad \forall t \in \mathbb{R}$

Hint: Set $y_i = 1$ in (7.1). Use the uniqueness property of Lagrange interpolants.

SOLUTION:

Consider the interpolation points $(t_i, 1)_{i=0}^n$. Clearly, an interpolating polynomial for these points is $p \equiv 1$. By uniquess of Lagrange interpolants, we have $1 = p(x) = \sum_{i=0}^n L_i(x)$.

(ii) $L_i(t) = \omega(t) \dfrac{\lambda_i}{t - t_i}$

SOLUTION:

Computing the derivative using the product rule:

$$\omega'(t) = \sum_{i=0}^{n} \prod_{\substack{j=0 \\ j\neq i}}^{n} (t - t_j)$$

Since $(t - t_j) = 0$ when $t = t_j$, it follows that $\omega'(t_i) = \prod_{\substack{j=0 \\ j\neq i}}^n (t_i - t_j)$. Therefore:

$$\begin{aligned} L_i(t) &= \frac{\prod_{\substack{j=0 \\ j\neq i}}^n (t - t_j)}{\prod_{\substack{j=0 \\ j\neq i}}^n (t_i - t_j)} \\ &= \frac{1}{(t - t_i)} \frac{\prod_{j=0}^n (t - t_j)}{\omega'(t_i)} \\ &= \omega(t) \frac{\lambda_i}{t - t_i} \end{aligned}$$

(iii) $p(t) = \left(\sum_{i=0}^{n} \frac{\lambda_i y_i}{t - t_i} \right) \left(\sum_{i=0}^{n} \frac{\lambda_i}{t - t_i} \right)^{-1}$ (**Barycentric interpolation formula**)

SOLUTION:

Substitute 7.(a).1) and 7.a).ii) in (7.1), then simplify to obtain the result.

(b) The following data is given:

i	t_i	y_i
0	-1	2
1	0	-4
2	1	6

(i) Compute the Lagrange interpolant $p(t)$ for the given data.

SOLUTION:

$$L_0(t) = \frac{t^2 - t)}{2} \quad (7.2a)$$
$$L_1(t) = 1 - t^2 \quad (7.2b)$$
$$L_2(t) = \frac{t^2 + t)}{2} \quad (7.2c)$$
$$p(t) = t^2 - t + 4(t^2 - 1) + 3(t^2 + t) \quad (7.2d)$$
$$= 8t^2 + 2t - 4 \quad (7.2e)$$

(ii) Use the Newton basis approach to compute the interpolating polynomial $\tilde{p}(t)$ for the given data.

SOLUTION:

$\tilde{p}(t) := \sum_{i=0}^{2} a_i N_i(t)$

$$N_0(t) = 1 \quad (7.3a)$$
$$N_1(t) = (t - t_0) \quad (7.3b)$$
$$N_2(t) = (t - t_0)(t - t_1) \quad (7.3c)$$
$$(7.3d)$$

$$a_0 = 2 \quad (7.4a)$$
$$a_1 = -6 \quad (7.4b)$$
$$a_2 = 8 \quad (7.4c)$$
$$\tilde{p}(t) = 2 - 6(t + 1) + 8t(t + 1) \quad (7.4d)$$

$$= 8t^2 + 2t - 4 \tag{7.4e}$$

(iii) Is $\tilde{p}(t)$ different from $p(t)$? If yes, why?

SOLUTION:

$\tilde{p}(t) = p(t)$ because interpolating polynomial is unique!

(iv) From an implementation viewpoint, what are the advantages of using the Newton basis compared to the Lagrange polynomials?

SOLUTION:

The Lagrange interpolant might become numerically unstable if some of the nodes t_i are close to each other, as it involves computing $t_i - t_j$ for $i \neq j$. In the Newton basis, we only compute $t - t_j$, so it is numerically stable.

Problem 7.2: Evaluating the derivatives of interpolating polynomials

Data interpolation is important for obtaining representations of constitutive relationships $t \mapsto f(t)$. Numerical methods like Newton's method often require information about the derivative f' as well. Hence, efficient algorithms to evaluate the derivatives of interpolants are needed. In this problem, we implement generalizations of the Horner scheme and the "update friendly" Aitken-Neville algorithm for computing derivatives of the interpolating polynomial.

Polynomial with monomial representation:

(a) Implement an efficient C++ template function:

```
template <typename CoeffVec>
Vector2d dpEvalHorner(const CoeffVec& c, double x);
```

which uses the Horner scheme to compute $(p(x), p'(x))$ and returns them in a 2d vector. Here p is a polynomial with coefficients in c and $p'(x)$ is the derivative of $p(x)$.

SOLUTION:

C++11-code 7.1: Solution of (a)

```
template <typename CoeffVec>
Eigen::Vector2d dpEvalHorner (const CoeffVec& c, const double x) {
  Eigen::Vector2d val;
  double px, dpx;
  int s = c.size();

  px = c[0];
  for (int i = 1; i < s; ++i) { px = x*px+c[i]; }

  dpx = (s-1)*c[0];
  for (int i = 1; i < s-1; ++i) { dpx = x*dpx+(s-i-1)*c[i]; }

  val(0) = px;
  val(1) = dpx;

  return val;
}
```

(b) For testing, write a naive C++ implementation which computes $p(x)$ and $p'(x)$ by simply summing the monomials constituting the polynomial:

```
template <typename CoeffVec>
Vector2d dpEvalNaive(const CoeffVec& c, double x);
```

SOLUTION:

C++11-code 7.2: Solution of (b)

```
template <typename CoeffVec>
```

```cpp
Eigen::Vector2d dpEvalNaive(const CoeffVec& c, const double x) {
  Eigen::Vector2d val;
  double px,dpx;
  int n=c.size();

  px = c[0]*std::pow(x, n-1);
  for (int i = 1; i < n; ++i) { px = px + c[i]*std::pow(x, n-i-1); }

  dpx = (n-1)*c[0]*std::pow(x, n-2);
  for (int i = 1; i < n-1; ++i) {dpx = dpx + (n-i-1)*c[i]*std::pow(x, n-i-2); }

  val(0) = px;
  val(1) = dpx;

  return val;
}
```

(c) What is the asymptotic complexity of `dpEvalHorner` and `dpEvalNaive`? Compare the runtimes of the two functions for polynomials of degree up to $2^{20}-1$.

SOLUTION:

In both cases, the algorithm requires $\approx n$ multiplications and additions, and the asymptotic complexity is therefore $\mathcal{O}(n)$. The naive implementation also calls the `pow()` function, which may be costly. Please refer to Fig. 1.

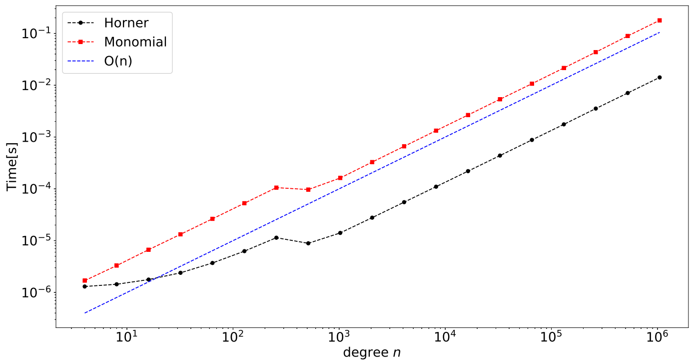

Fig. 1

Extending the Aitken-Neville algorithm to compute the derivative of the polynomial interpolant:

(d) Implement an efficient C++ function:

```cpp
double derivIpolEvalAN(const VectorXd & t,
                       const VectorXd & y,
                       const double x);
```

which returns the derivative $p'(x)$ of the polynomial $p \in \mathcal{P}_n$ interpolating the data points (t_i, y_i), $i = 0, \ldots, n$, for pairwise distinct nodes $t_i \in \mathbb{R}$ and measured data values $y_i \in \mathbb{R}$.

To test your implementation, compare the result from `derivIpolEvalAN` for the data given in Sub-problem 7.b) with the derivative of the corresponding interpolating polynomial.

Hint: Differentiate the underlying recursion formula of the Aitken-Neville algorithm.

SOLUTION:

Differentiating the recursion formula we obtain

$$
\begin{aligned}
p_{k,k}(x) &\equiv y_k, \qquad k = 0,\dots,n,\\
p'_{k,k}(x) &\equiv 0, \qquad k = 0,\dots,n,\\
p_{k,l}(x) &= \frac{(x-t_k)p_{k+1,l}(x) - (x-t_l)p_{k,l-1}(x)}{t_l - t_k},\\
p'_{k,l}(x) &= \frac{p_{k+1,l}(x) + (x-t_k)p'_{k+1,l}(x) - p_{k,l-1}(x) - (x-t_l)p'_{k,l-1}(x)}{t_l - t_k}.
\end{aligned}
$$

C++11-code 7.3: Example code using Aitken-Neville scheme

```cpp
double derivIpolEvalAN(const VectorXd & t,
                       const VectorXd & y,
                       const double x) {

    assert(t.size() == y.size());

    VectorXd p(y);
    VectorXd dP = VectorXd::Zero(y.size());

    for(int i = 1; i < y.size(); ++i) {
        for(int k = i-1; k >= 0; --k) {

            dP(k) = (p(k+1) + (x-t(k))*dP(k+1) - p(k) - (x-t(i))*dP(k))
                          / (t(i) - t(k));

            p(k) = ((x-t(k))*p(k+1) - (x-t(i))*p(k)) / (t(i) - t(k));
        }
    }

    return dP(0);
}
```
