Polynomial class manual

J. Delgado J.M. Peña

September 7, 2015

0.1 Installation

Extract the contents of the file `Polynomial.zip` into a directory of your choice, e.g. Polynomial-path. This newly created directory should now be added to the MATLAB path. This can be done temporarily (for one MATLAB session) by executing

```
>> addpath (genpath('Polynomial-path/Polynomial_1.0'))
```

on the MATLAB command line. The package can also be added to the MATLAB path permanently via `File/Set Path...` and `Add with Subfolders...`. The `Polynomial` package is now ready for use.

0.2 Introduction

This section describes the structure of the software library. It consists of a Matlab m-file, Polynomial.m, containing the datum definition of the class, the declaration of its member functions and operator-overloadings, and of several m-files containing the member functions and the nonmember functions used by those. An object of the Polynomial class possesses a piece of private data, `BernsCoeff`, the constructor function `Polynomial`, the public member functions `disp`, `char`, `getDegree`, `getCoeff`, `double`, `degreeElevation`, `degreeReduction`, `Eval`, `diff`, `integral`, `integrate`, `plot` and `subsref`, and the overloading of the operators $+$ (function `plus`), $-$ (function `minus`), $*$ (function `mtimes`), / (function `mrdivide`) and $\wedge$ (function `mpower`). In addition, the software library also contains the usual functions `Vs`, `Casteljau`, `CompVs`, `Horner`, `TwoSum`, `TwoProduct`, `DivRem`, `Split`, `powerDC`, `bd`, `interpolation` and `monomial2Bernstein`, implemented each of them in its corresponding m-file, which are used, directly or indirectly, by member functions of the class.

In our software library we provide five different ways of constructing a polynomial in Bernstein form. In contrast to other object oriented languages like C++, Matlab does not allow to overload the constructor of a class with different input/output arguments. So we have to put the five different ways of constructing a polynomial in Bernstein form into a unique constructor. We have done it by using the possiblity of accepting any number of input arguments of a function in Matlab with `varargin` Matlab variable. The first of the five ways, `Polynomial()`, is the default constructor, which builds up the zero degree polynomial equal to zero for any value of the domain parameter. The copy constructor, invoked by using `Polynomial(poly)` where `poly` is already an existing object of the polynomial class, returns a copy of the `poly` object. In the third way of constructing a polynomial, the user of the library provides a vector coeff $= (c_0, \dots, c_n)$ with the coefficients of the polynomial respect to the Bernstein basis straightforwardly, $p(t) = \sum_{i=0}^{n} c_i b_i^n(t)$, by using `Polynomial(coeff,'c')`, where `'c'` means control polygon. In Section 3 of the paper we present the mathematical foundations for the two nontrivial ways of constructing polynomials in the Bernstein form, that is, from the interpolation conditions (`Polynomial(t,p)`) and from the monomial representation (`Polynomial(c,'m')` or `Polynomial(c)`). The design and implementation of the constructor of the class and the nonmember functions `interpolation`, `bd` and monomial2Bernstein are presented. Section 4 of the paper includes a compensated version of the VS algorithm together with a dynamic error estimate. Finally, the design and implementation of an adaptative evaluation algorithm Eval, based in the evaluation algorithms Vs, Casteljau and CompVs, is shown in Section 5 of the paper.

For completeness and usefulness we have included in the software library some other functions. The public member function getDegree(obj) returns the degree of the object obj of the Polynomial class. Taking into account that the class data BernsCoeff is private, the public member functions getCoeff(obj) and double(obj) return a vector with the coefficients of the object obj respect to the Bernstein basis. We have implemented a private member function char(obj), which creates a formatted display of the Polynomial object obj as a linear combination of the basis of Bernstein polynomials of the corresponding degree. This formatted display is used by the member function disp(obj), which determines how Matlab displays a Polynomial object obj on the command line. The member function subsref(obj) enables you to specify a value for the independent variable as a subscript, access the BernsCoeff property with dot notation, and call methods with dot notation. In the public member function plot we overload the usual function in order to plot objects of the Polynomial class. A polynomial of a certain degree n can always be represented exactly as a polynomial of a higher degree. Degree elevation is necessary for the addition and substraction of polynomials of different degrees in Bernstein form. The usual method to carry out this degree elevation for polynomials is shown in [1] and [4], for example. Nevertheless, recently in [3] Sánchez-Reyes has shown the advantages of performing the degree elevation by using convolution (conv function in Matlab). So we have implemented the public member function degreeElevation(obj,k), which performs an elevation of k degrees of the Polynomial object obj by using conv. The differentation and integration of polynomials in Bernstein form are simple operations, which only involve linear combinations of the Bernstein coefficients of the corresponding polynomial. The formulas for computing $p'(x)$, $\int p(x)\,dx$ and $\int_0^1 p(x)\,dx$ can be seen in [2], and the public member functions diff(obj), integrate(obj) and integral(obj) implement them, respectively. The public member functions plus(obj1,obj2) and minus(obj1,obj2) implement the usual addition and subtraction of polynomials, overloading the corresponding operators. First, when necessary, the degree of one of the two polynomials is elevated by using degreeElevation(obj,k) with an adequate k. The public member functions mtimes(obj1,obj2) and mpower(obj,k) implement the operators $*$ and $\wedge$. In contrast to the usual form, implemented in [4], here we also use, like in the case of function degreeElevation, Matlab convolution function conv, which is more efficient as pointed out in [3].

In the following two sections we show how to use the software package. However, all the following explanations an commands can be followed in an interactive way through invoking script manual (file manual.m).

0.3 Constructing objects of the class

The Polynomial class provides a mold to construct polynomials represented in the basis formed by the Bernstein basis of degree n given by $b_i^n(x) = \binom{n}{i} x^i (1 -$

$x)^{n-i}$. There are five different ways of constructing an object of this class:

1. The default constructor, by using `Polynomial()`, creates the zero degree polynomial whose value is zero at any point

```
>> poly1 = Polynomial()
poly1 =
0
```

2. If $p(x) = \sum_{i=0}^{n} a_i x^i$ then, by using `Polynomial([a_0,a_1,...,a_n])` or `Polynomial([a_0,a_1,...,a_n],'m')`, we contruct this polynomial represented in the basis formed by the Bernstein polynomials of degree n. So, for the polynomial $p(t) = 1 + 2x + 3x^2$ it would be

```
>> poly2 = Polynomial([1,2,3])
poly2 =
bin(2,0)*(1-x)^2 + 2*bin(2,1)*x*(1-x) + 6*bin(2,2)*x^2
```

   or

```
>> poly2 = Polynomial([1,2,3],'m')
poly2 =
bin(2,0)*(1-x)^2 + 2*bin(2,1)*x*(1-x) + 6*bin(2,2)*x^2
```

3. Given the interpolation conditions $p(x_i) = q_i$, $i = 0, 1, \ldots, n$, we can construct the interpolating polynomial represented in the corresponding Bernstein basis by using `Polynomial([x_0,x_1,...,x_n],[q_0,q_1,...,q_n])`. For example,

```
>> poly3 = Polynomial([0.25,0.5,0.75],[1,-2,3])
poly3 =
12*bin(2,0)*(1-x)^2 - 18*bin(2,1)*x*(1-x) + 16*bin(2,2)*x^2
```

   creates the polynomial of degree 2 represented in the Bernstein basis (b_0^2, b_1^2, b_2^2) satisfying the conditions $p(0.25) = 1$, $p(0.5) = -2$ and $p(0.75) = 3$.

4. If the coefficients c_i of the polynomial with respect to the basis formed by the Bernstein polynomial are known $(p(t) = \sum_{i=0}^{n} c_i b_i^n(x))$, the the polynomial can be constructed by using `Polynomial([c_0,c_1,...,c_n],'c')`. For example,

```
>> poly4 = Polynomial(0:1:3,'c')
poly4 =
bin(3,1)*x(1-x)^2 + 2*bin(3,2)*x^2(1-x) + 3*bin(3,3)*x^3
```

5. Finally, the copy constructor:

```
>> poly5 = poly2
poly5 =
bin(2,0)*(1-x)^2 + 2*bin(2,1)*x*(1-x) + 6*bin(2,2)*x^2
```

or

```
>> poly5 = Polynomial(poly2)
poly5 =
bin(2,0)*(1-x)^2 + 2*bin(2,1)*x*(1-x) + 6*bin(2,2)*x^2
```

0.4 Using the methods of the class

1. `getDegree` returns the apparent degree of the polynomial

   ```
   >> poly5.getDegree()
   ans =
       2
   >> poly4.getDegree()
   ans =
       3
   ```

2. `getCoeff` and `double` return the coefficients of the polynomial respect to the Bernstein basis

   ```
   >> poly5.getCoeff()
   ans =
       1     2     6
   >> poly4.double()
   ans =
       0     1     2     3
   ```

3. A polynomial represented in a Bernstein basis can also be represented in a Bernstein basis of greated degree. `degreeElevation` performs this degree elevation. For example the code

   ```
   >> poly5 = poly5.degreeElevation(2)

   poly5 =

   bin(4,0)*(1-x)^4 + 1.5*bin(4,1)*x(1-x)^3
    + 2.5*bin(4,2)*x^2(1-x)^2 + 4*bin(4,3)*x^3(1-x) + 6*bin(4,4)*x^4
   ```

   returns the degree 2 polynomial `poly5` represented in the Bernstein basis of degree 4 (=2+2).

4. Although a polynomial is represented in a Bernstain basis of a certain n degree, the true degree of the polynomial can be lower. The method `degreeReduction` checks the true degree of a polynomial and returns the same polynomial represented in the Bernstein basis of the lowest possible degree. For example,

```
>> poly5 = poly5.degreeReduction()

poly5 =

bin(2,0)*(1-x)^2 + 2*bin(2,1)*x*(1-x) + 6*bin(2,2)*x^2
```

5. The usual arithmetic operations can be performed with the usual operators $+$ (sum), $-$ (substraction), $*$ (product), $/$ (division) and $\wedge$ (power). Let us see some examples:

```
>> poly2+poly4

ans =

bin(3,0)*(1-x)^3 + 2.6667*bin(3,1)*x(1-x)^2
 + 5.3333*bin(3,2)*x^2(1-x) + 9*bin(3,3)*x^3
>> poly2-poly4

ans =

bin(3,0)*(1-x)^3 + 0.66667*bin(3,1)*x(1-x)^2
 + 1.3333*bin(3,2)*x^2(1-x) + 3*bin(3,3)*x^3
>> poly2*poly4

ans =

0.6*bin(5,1)*x(1-x)^4 + 1.8*bin(5,2)*x^2(1-x)^3
 + 4.5*bin(5,3)*x^3(1-x)^2 + 9.6*bin(5,4)*x^4(1-x)
 + 18*bin(5,5)*x^5
>> [q,r] = poly2/poly4

q =

0.66667*bin(1,0)*(1-x) + 1.6667*bin(1,1)*x

r =

1
>> poly6=poly4*q+r

poly6 =

bin(4,0)*(1-x)^4 + 1.5*bin(4,1)*x(1-x)^3
 + 2.5*bin(4,2)*x^2(1-x)^2 + 4*bin(4,3)*x^3(1-x)
 + 6*bin(4,4)*x^4
>> poly6.degreeReduction()

ans =

bin(2,0)*(1-x)^2 + 2*bin(2,1)*x*(1-x) + 6*bin(2,2)*x^2
```

```
>> poly2

poly2 =

bin(2,0)*(1-x)^2 + 2*bin(2,1)*x*(1-x) + 6*bin(2,2)*x^2
>> poly2^2

ans =

bin(4,0)*(1-x)^4 + 2*bin(4,1)*x(1-x)^3
 + 4.6667*bin(4,2)*x^2(1-x)^2 + 12*bin(4,3)*x^3(1-x)
 + 36*bin(4,4)*x^4
```

6. The method `Eval` evaluates a polynomial. It can be invoked in two different ways: `Eval(x)` and `Eval(x,prec)`. The first one evaluates the polynomial at the points in x with a default pretended percision of $10e-12$, whereas the second one performs the same evaluation with the pretended precision *prec*. Let us see some examples:

```
>> format short e
>> poly4.Eval([0:0.25:1])

ans =

           0   7.5000e-01   1.5000e+00   2.2500e+00   3.0000e+00
         NaN   9.3675e-16   7.7716e-16   6.1062e-16   4.4409e-16
           0   1.0000e+00   1.0000e+00   1.0000e+00   1.0000e+00

>> poly4.Eval([0:0.25:1],5*1e-16)

ans =


           0   7.5000e-01   1.5000e+00   2.2500e+00   3.0000e+00
         NaN   2.2204e-16   2.2204e-16   2.2204e-16   4.4409e-16
           0   1.0000e+00   1.0000e+00   1.0000e+00   1.0000e+00
```

As we can observe, the method returns for the 1×4 vector of points a 3×4 matrix, where the first row are consists of the evaluations of the polynomial at the corresponding points in x, the second row provides, when possible, upper bounds of the relative error for the evaluation, and the third row consists of flags (1 means that relative error is lower than *prec*, 0 means that either relative error is not less than *prec* or it is not known about). If x is a column vector we obtain the traspose matrix:

```
>> poly4.Eval([0;0.25;0.5;0.75;1])

ans =

           0          NaN             0
```

```
   7.5000e-01   9.3675e-16   1.0000e+00
   1.5000e+00   7.7716e-16   1.0000e+00
   2.2500e+00   6.1062e-16   1.0000e+00
   3.0000e+00   4.4409e-16   1.0000e+00
```

The same evaluation can be performed with `obj(x)` or `obj(x,prec)`:

```
>> poly4(0:0.25:1)

ans =

             0   7.5000e-01   1.5000e+00   2.2500e+00   3.0000e+00
           NaN   9.3675e-16   7.7716e-16   6.1062e-16   4.4409e-16
             0   1.0000e+00   1.0000e+00   1.0000e+00   1.0000e+00

>> poly4(0:0.25:1,5*1e-16)

ans =

             0   7.5000e-01   1.5000e+00   2.2500e+00   3.0000e+00
           NaN   2.2204e-16   2.2204e-16   2.2204e-16   4.4409e-16
             0   1.0000e+00   1.0000e+00   1.0000e+00   1.0000e+00
```

7. `diff` returns the derivate of a polynomial. For example,

```
>> poly7 = poly4.diff()

poly7 =

3*bin(2,0)*(1-x)^2 + 3*bin(2,1)*x*(1-x) + 3*bin(2,2)*x^2
```

8. `integrate` returns the indefinite integral of a polynomial.

```
>> poly7.integrate()

ans =

bin(3,1)*x(1-x)^2 + 2*bin(3,2)*x^2(1-x) + 3*bin(3,3)*x^3
```

9. `integral` returns the definite integral of a polynomial in the interval $[0,1]$:

```
>> poly7.integral()

ans =

     3
```

10. The method `plot` display a graphic of the polynomial in $[0,1]$. This method can be invoked in two ways: `plot()` or `plot(step)`. The first one, makes the graphic taking a mesh of the interval $[0,1]$ with step 0.01, whereas the second one takes a mesh with step `step`. Let us see a couple of examples:

```
>> poly5.plot()
```

obtaining the graphic shown in Figure 1

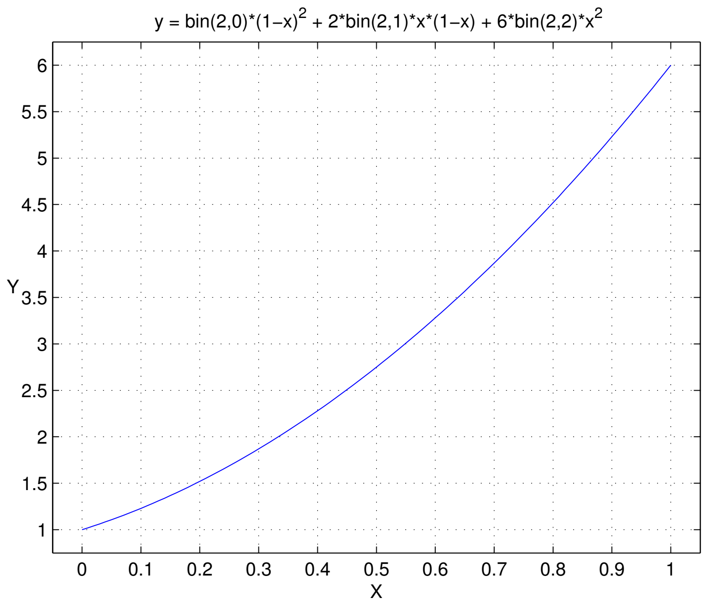

Figure 1: Example 1

and

```
>> poly5.plot(0.25)
```

obtaining the graphic shown in Figure 2

0.5 Test scripts

In order to test the usefulness of the software package we have included three scripts: `test1`, `test2` and `test3` in the files `test1.m`, `test2.m` and `test3.m`, respectively.

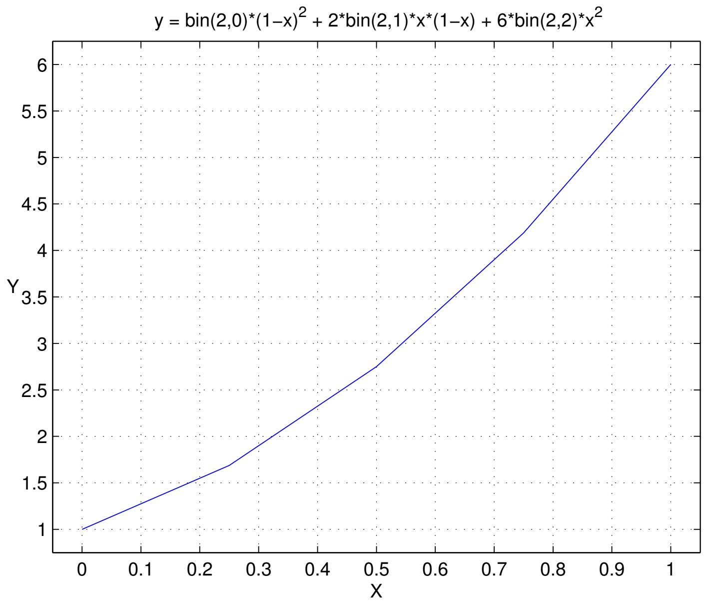

Figure 2: Example 2

Bibliography

[1] Farin, G. *Curves and Surfaces for Computer Aided Geometric Design*, fifth ed. Academic Press, Inc., San Diego, CA, 2002.

[2] Farouki, R. T., and Rajan, V. T. Algorithms for polynomials in bernstein form. *Computer Aided Geometric Design 5* (1988), 1–26.

[3] Sánchez-Reyes, J. Algebraic manipulation in the bernstein form made simple via convolutions. *Computer-Aided Design 35* (2003), 959–967.

[4] Tsai, Y. F., and Farouki, R. T. Algorithm 812: Bpoly: An object-oriented library of numerical algorithms for polynomials in bernstein form. *ACM Transactions on Mathematical Software 27* (2001), 267–296.