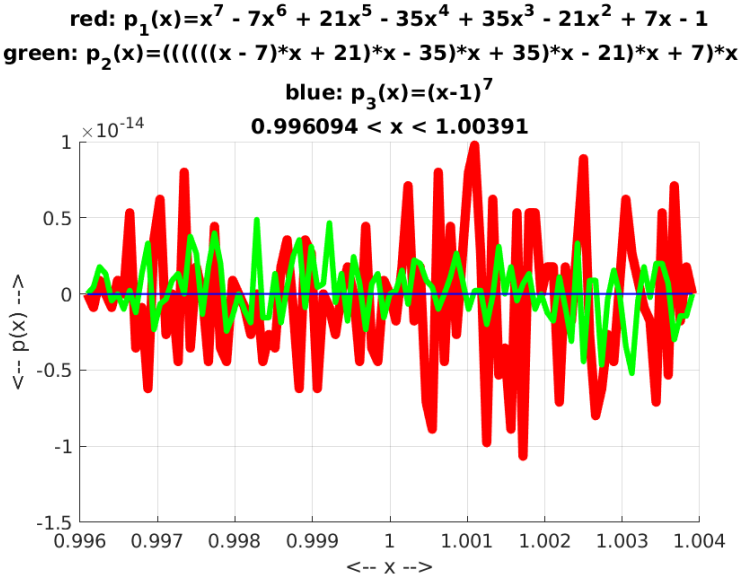
Polynomials:
Representation, Evaluation, Operations

MATH2070: Numerical Methods in Scientific Computing I

Location: http://people.sc.fsu.edu/~jburkardt/classes/math2070_2019/polynomial_operations/polynomial_operations.pdf

---



Evaluating three representations of the same polynomial.

Polynomial Computations

- *What choices are there for representing a polynomial?*
- *What is an efficient evaluation procedure?*
- *How do we add, multiply, divide, differentiate, and integrate?*

A (real) polynomial of degree d can be defined in **power form** as

$$p(x) = c_0 + c_1 x + c_2 x^2 + ... + c_d x^d \quad \text{Mathematical power form (0-based)}$$

where the quantities c_i are real numbers known as the coefficients of the polynomial. Here, the subscript of the coefficient c_i matches the corresponding power x^i. This is the most natural way to describe a polynomial.

We assume that c_d is not 0, otherwise the polynomial actually has a degree lower than d.

1 Power form (1-based)

MATLAB does not allow the use of 0 as an array index. The simplest way to deal with this issue is to use a "1-based" power form, in which c_{i+1} multiplies x^i:

$$p(x) = c_1 + c_2 x + c_3 x^2 + ... + c_{d+1} x^d \quad \text{power form (1-based)}$$

Because it is still pretty easy to think about polynomials this way, this is the form that we will prefer for most of the following discussion.

Thus, the degree-3 polynomial

$$p(x) = 1.1 + 3.3x^2 - 4.44x^3$$

could be stored as a MATLAB array of 4 entries:

```
c = [ 1.1, 0.0, 3.3, 4.44 ]
```

2 Print a polynomial

If we want to print out a polynomial, we could just print the vector of coefficients `c`. It is much more intelligible to try to reproduce the full mathematical form, if possible. Here is a very basic example that suggest how we might do this:

```
function poly_print ( c )
  d = length ( c ) - 1;
  for i = 0 : d
    fprintf ( '%g x^%d\n', c(i+1), i );
  end
  return
end
```

Listing 1: Basic poly_print.m code.

While we could work harder to make nicer output, here's what happens if we call the function to print out $3x^5 - 2x^3 + x^2 - 5x - 17$:

```
  c = [ -17, -5, 1, -2, 0, 3 ];
  poly_print ( c );

-17 x^0
-5 x^1
1 x^2
-2 x^3
0 x^4
3 x^5
```

Listing 2: Using poly_print().

For somewhat nicer output, we could try the fancier function `r8poly_print(c)`.

```
  c = [ -17, -5, 1, -2, 0, 3 ];
  r8poly_print ( c );

  p(x) =         3.000000 * x^5
         -       2.000000 * x^4
         +       1.000000 * x^2
         -       5.000000 * x
         -      17.000000
```

Listing 3: Using the fancier r8poly_print().

which prints the polynomial in the more traditional form in which the highest power is shown first. It also omits powers of x with zero coefficient.

3 Add two polynomials

Adding two polynomials just means combining coefficients of the same power. The polynomials may have different degrees; the degree of the sum should be the higher of the two degrees. We can simplify the process by copying the coefficients of the higher degree polynomial, and then adding the coefficients of the lower degree polynomial:

```
function c3 = poly_add ( c1, c2 )
  d1 = length ( c1 ) - 1;
  d2 = length ( c2 ) - 1;
  d3 = max ( d1, d2 );
  c3 = zeros ( d3+1, 1 );

  if ( d2 < d1 )
    c3 = c1;
    for i = 0 : d2
      c3(i+1) = c3(i+1) + c2(i+1);
    end
  else
    c = c2;
    for i = 0 : d1
      c3(i+1) = c3(i+1) + c1(i+1);
    end
  end

  return
end
```

Listing 4: Polynomial addition.

Why couldn't we simply say the following?

```
    for i = 0 : d3
      c3(i+1) = c1(i+1) + c2(i+1);
    end
```

Listing 5: Explain why this is a bad idea.

4 Multiply two polynomials

Mathematically, if $p_3 = p_1 * p_2$, then the coefficients of p_3 can be computed by

$$c_{3_k} = \sum_{i+j=k} c_{1_i} \, c_{2_j} \quad 0 \leq k \leq d_1 + d_2$$

So computationally, multiplying two polynomials means computing every possible product $c1(i) * c2(j)$ and adding that to $c3(i+j)$. But we have to remember to increment our coefficient indices by 1. The degree of the product will be the sum of the degrees of the factors:

```
function c3 = poly_multiply ( c1, c2 )
  d1 = length ( c1 ) - 1;
  d2 = length ( c2 ) - 1;
  d3 = d1 + d2;
  c3 = zeros ( d3+1, 1 );

  for i = 0 : d1
    for j = 0 : d2
      c3(i+j+1) = c3(i+j+1) + c1(i+1) * c2(j+1);
    end
  end
```

```
12
13    return
14  end
```

Listing 6: Polynomial multiplication.

Use this function to compute $(2x-3)(5x^2+4x+6)$.

5 Quotient and remainder of two polynomials

Polynomial division of p_1 by p_2 yields quotient polynomial q and remainder polynomial r so that:

$$p_1 = q * p_2 + r$$

We are all familiar with how to divide one polymonial into another, but the details are a little difficult to work out. Here is how to do it:

```
1  function [ c3, c4 ] = poly_divide ( c1, c2 )
2    d1 = length ( c1 ) - 1;
3    d2 = length ( c2 ) - 1;
4
5    d3 = d1 - d2;
6    d4 = d2 - 1;
7
8    c3 = zeros ( d3 + 1, 1 );
9
10   for i = d3 : -1 : 0
11     c3(i+1) = c1(i+d2+1) / c2(d2+1);
12     c1(i+1:i+d2) = c1(i+1:i+d2) - c3(i+1) * c2(1:d2);
13   end
14
15   c4(1:d4+1) = c1(1:d4+1);
16
17   return
18 end
```

Listing 7: Quotient and remainder of polynomial division.

We can check this by dividing:

$$p_1 = x^4 + 3x^3 + 2x^2 + 5x - 2$$

by

$$p_2 = x^2 + x - 3$$

for which

$$q = x^2 + 2x + 3$$

with remainder

$$r = 8x + 7$$

and we can verify this by then recomputing

$$p1 = q * p2 + r :$$

as in the following computation:

```
1    c1 = [ -2,5,2,3,1];
2    c2 = [-3,1,1];
3    [q,r] = poly_divide ( c1, c2 );
4    qp = poly_multiply ( q, c2 );
5    qpr = poly_add ( qp, r );    <-- q*p+r should equal c1
```

Listing 8: Divide and then verify.

6 Squares and powers of a polynomial

Now that we have `poly_multiply()`, let's use it to create a function for the k-th power of a polynomial. Notice how we start the coefficient vector `c2` as a single number 1, which is the correct coefficient vector for p_1^0. Then we multiply this polynomial k times by p_1, to arrive at the final form of c_2.

```
function c2 = poly_power ( c1, k )
  c2 = 1.0;
  for i = 1 : k
    c2 = poly_multiply ( c1, c2 )
  end
  return
end
```

Listing 9: Powers of a polynomial.

7 Derivative and antiderivative of a polynomial

The coefficients of the derivative of a polynomial can easily be computed. We multiply each coefficient by the exponent of its power of x, and then shift the coefficients down one position.

```
function c2 = poly_deriv ( c1 )

  d1 = length ( c1 ) - 1;
  d2 = d1 - 1;

  for i = 0 : d2
    c2(i+1) = ( i + 1 ) * c1(i+2);
  end

  return
end
```

Listing 10: The derivative of a polynomial.

Technically, a polynomial has many antiderivatives. We'll just return the antiderivative whose constant term is 0. We need to shift each coefficient to one higher position and divide by the appropriate power.

```
function c2 = poly_antideriv ( c1 )

  d1 = length ( c1 ) - 1;
  d2 = d1 + 1;
  c2(1:d2+1) = 0.0;

  c2(1) = 0.0;
  for i = 1 : d2
    c2(i+1) = c1(i) / i;
  end

  return
end
```

Listing 11: An antiderivative of a polynomial.

8 L2 norm of a polynomial

The L2 norm of a function $f(x)$ over the finite interval $[a, b]$ is defined as:

$$||f||_2 = \sqrt{\int_a^b f(x)^2 dx}$$

If we are working with a polynomial, we can do such integrals exactly. Using functions we have already described, we can build a new function that evaluates L2 norms:

```
function value = poly_norm ( c, a, b )

  c2 = poly_square ( c );
  c2a = poly_anti_value ( c2, a );
  c2b = poly_anti_value ( c2, b );
  value = sqrt ( c2b - c2a );

  return
end
```

Listing 12: Compute the L2 norm of a polynomial.

Actually, we didn't explain how to write `poly_square()`, which returns the coefficients of the square of a polynomial, but how hard can that be?

9 Evaluate a power form polynomial

We wish to evaluate a power form polynomial $p(x)$ for a given value of x. If the polynomial is of low degree, we can simply write out the expression and be done:

$$\text{value} = 3x^4 - 7x^2 - 5x + 17$$

but if we want a procedure to handle arbitrary polynomials, we write:

```
function value = poly_value_termwise ( c, x )
  d = length ( c ) - 1;
  value = 0.0;
  for i = 0 : d
    value = value + c(i+1) * x^(i);
  end
  return
end
```

Listing 13: Evaluate a polynomial term by term.

while a nested multiplication version, sometimes called *Horner's method*, would be:

```
function value = poly_value_nested ( c, x )
  d = length ( c ) - 1;
  value = c(d+1);
  for i = d - 1 : -1 : 0
    value = value .* x + c(i+1);    <-- The period means x can be a vector
  end
  return
end
```

Listing 14: Nested multiplication for polynomial evaluation.

10 Exercise: Plot a Chebyshev polynomial

The seventh Chebyshev polynomial $T_7(x)$ is defined by

$$T_7(x) = 64x^7 - 112x^5 + 56x^3 - 7x$$

Plot this polynomial over the interval $-1 \leq x \leq 1$. Use a combination of `x=linspace()` and `y=poly_value_nested()` to set up your data.

11 Polynomials in factor form

If we know all the roots of a polynomial of degree d, then we can write it in **factored form**:

$$p(x) = \prod_{i=1}^{d} (x - r_i) \quad \text{Factor form polynomial}$$

where the quantities r_i are the roots of the polynomial. For convenience, we will assume all the roots are real.

12 Evaluate a factor form polynomial

A factored polynomial could be evaluated by:

```
function value = factor_value ( r, x )
  d = length ( r );
  value = 1.0;
  for i = 1 : d
    value = value * ( x - r(i) );
  end
  return
end
```

Listing 15: Evaluate factor form polynomial.

13 Convert from factor form to power form

We might prefer to see a factor form polynomial rewritten in power form. To do this, we just use our multiplication function repeatedly:

```
function c = factor_to_poly ( r )
  n = length ( r );
  c = 1.0;
  for i = 1 : n
    c = poly_multiply ( c, [ -r(i), 1.0 ] );  <-- Multiply by (x-r(i))
  end
  return
end
```

Listing 16: Convert factor form to power form.

14 Convert from power form to factor form

If we have the polynomial in power form, getting factor form is the same as finding all its roots. This is not an easy process in general! However, MATLAB provides a function `r=roots(c)` which will do the job. Unfortunately. `roots()` uses the MATLAB polynomial representation, which lists coefficients in the reverse order to the one I have been using. Bearing that in mind, here is our conversion:

```
function r = poly_to_factor ( c )
  r = roots ( c(end:-1:1) );
  return
end
```

Listing 17: Convert power form to factor form.

Thus, we can do the following experiment:

```
r = [ 1, 2, 3, 4];
c = factor_to_poly ( r );
r2 = poly_to_factor ( c );
```

Listing 18: Test factor to power to factor.

Does the output value of `r2` match the original values in `r`?

15 MATLAB's polynomial representation

MATLAB has a number of built-in functions that work with polynomials. However, MATLAB orders the coefficients in the opposite order from that which we are using. In other words, for MATLAB, a polynomial of degree d with coefficients c is to be thought of as

$$p(x) = c_1 x^d + c_2 x^{d-1} + c_3 x^{d-2} + ... + c_d x + cd + 1 \quad \text{MATLAB power form}$$

Thus, each term has the form $c_i\, x^{d+1-i}$, so that the coefficient index and power of x always sum to $d+1$.

I find this convention awkward to work with, which is why I have been showing you my preferred representation. However, MATLAB has a number of useful functions for analyzing a polynomial, and to use them, we have to give and receive polynomial data in MATLAB's representation.

Luckily, this is easy to do. If we have a coefficient vector `c` for a polynomial of degree d, then before we pass it to a MATLAB function like `polyval()`, we need to reverse the order of the elements. One way to do that would be to make a reversed copy of c:

```
    c_back = zeros ( d, 1 );
    for i = 1 : d + 1
      c_back(i) = c(d+2-i);   <-- Not so easy to explain or remember!
    end
    v = polyval ( c_back, x );
or
    c_back = c(d+1:-1:1);
    v = polyval ( c_back, x );
or
    c_back = c(end:-1:1);     <-- Note that "end" means the last index.
    v = polyval ( c_back, x );

but then we could just use the original array with reverse indexing:

  v = polyval ( c(end:-1:1), x );
```

Listing 19: Ways to reverse your coefficient vector.

But we can do that implicitly, simply be a reverse in expects polynomial coefficients to be numbered in the reverse order from our convention. If you have coefficients numbered so that c_1 is the constant term, than you can give MATLAB a reverse copy by passing `c(end:-1:1)`.

16 Evaluate using MATLAB's `polyval()`

MATLAB provides the function `value = polyval(c,x)`, for which x can be a vector of values. An example of this usage is available in *evaluation_compare.m*.

Suppose we want to evaluate the Chebyshev polynomial $T_7(x)$ at a set of points `x`. If we are already committed to using the power form, then we call `polyval()` this way:

```
c1 = [ 0, -7, 0, 56, 0, -112, 0, 64 ]; % power form
y1 = polyval ( c1(end:-1:1), x );
```

but if we use the MATLAB representation instead, we simply call:

```
c2 = [ 64, 0, -112, 0, 56, 0, -7, 0 ]; % MATLAB power form
y2 = polyval ( c2, x );
```

17 Exercise: Compare evaluation procedures

De Boor, Exercise 2.1-1, page 37: Evaluate the cubic polynomial $p(x) = (x - 99\pi)(x - 100\pi)(x - 101\pi)$ *at* $x = 314.15$. *Then use nested multiplication to obtain* $p(x)$ *in power form, and evaluate the power form at* $x = 314.15$ *and compare!*

```
function evaluation_compare ( x )

%% evaluation_compare compares methods of evaluating a polynomial.

  p_factor = [ 99.0*pi, 100.0*pi, 101.0*pi ];
  p_power = [ -999900*pi^3, 29999*pi^2, -300.0*pi, 1.0 ];

  value = poly_power_term_value ( p_power, x );
  fprintf ( 1, '  Power form, term-by-term evaluation = %g\n', value );

  value = poly_power_nest_value ( p_power, x );
  fprintf ( 1, '  Power form, nested evaluation = %g\n', value );

  value = poly_factor_product_value ( p_factor, x );
  fprintf ( 1, '  Factor form, product evaluation = %g\n', value );

  value = polyval ( p_power(end:-1:1), x );
  fprintf ( 1, '  Power form, polyval evaluation = %g\n', value );

  return
end
```

Listing 20: Compare polynomial evaluation methods.

Surprisingly, this exercise doesn't seem to exhibit much difference in the results. However, De Boor's example was set up during the days when single precision (32 bit) calculations were most common. We can ask MATLAB to carry out such reduced-precision computations using the `single()` function. If we repeat the comparison using single precision arithmetic, the issue becomes much clearer.

The moral is now twofold: double precision certainly helps, but the power representation of a polynomial is clearly much more subject to error than the factor form, if it is available.

This same issue can be seen in the graphic at the beginning of this lab, although the errors in the power form are only really visible at a "microscropic" scale, of the order of 10^{-14}.

18 Newton polynomial representation

The **Newton polynomial representation** will make more sense when we look at divided differences.

The Newton represention involves three quantities:

- `d`, the degree;
- `c(0:d)`, the coefficients;

- `z(1:d)`, the centers;

An abstract example of a degree $d = 3$ polynomial in Newton form would be

$$\begin{aligned} p(x) = & c(0) \\ & + c(1) * (x - z(1)) \\ & + c(2) * (x - z(1)) * (x - z(2)) \\ & + c(3) * (x - z(1)) * (x - z(2)) * (x - z(3)) \end{aligned}$$

A Newton polynomial can be written compactly as:

$$p(x) = \sum_{i=0}^{d} c_i * \prod_{j=1}^{i} (x - z_j)$$

Note that, when we are using MATLAB, we have to shift the indices of the coefficients up by 1, because MATLAB does not allow a 0 index. We can evaluate a Newton polynomial in nested form:

```
function value = newton_value ( c, z, x )
  d = length ( c ) - 1;
  value = c(d+1) * ones ( size ( x ) );
  for i = d : -1 : 1
    value = value .* ( x - z(i) ) + c(i);
  end
  return
end
```

Listing 21: Evaluate a Newton polynomial.

19 Exercise: Newton polynomial evaluation

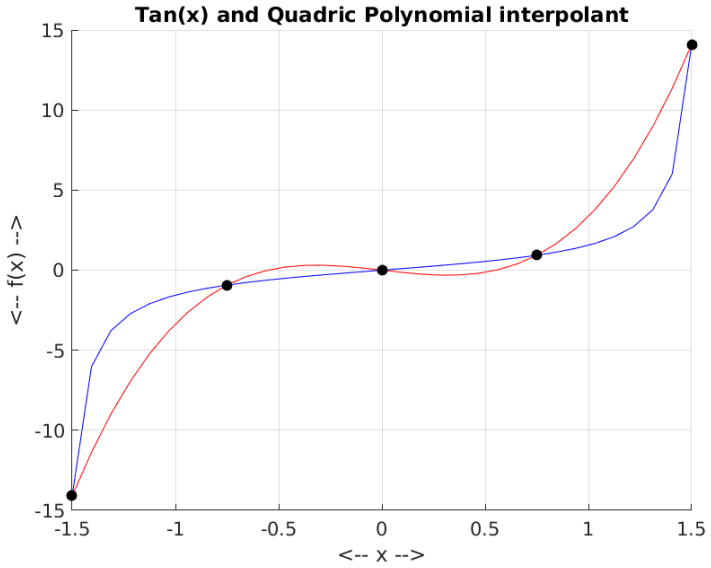
The Wikipedia page for *Newton polynomial* describes a polynomial interpolant to the tangent function, with data:

- `n = 4`, the degree;
- `c(1:n+1)`=[-14.1014, 17.5597, -10.8784, 4.83484, 0.0];
- `z(1:n)`=[-1.5, -0.75, 0,0.75, 1.5];

We can tabulate this polynomial over $[-1.5, 1.5]$, and plot the function and polyomial interpolant:

```
c = [ -14.1014, 17.5597, -10.8784, 4.83484, 0.0 ];
z = [ -1.5, -0.75, 0, 0.75, 1.5 ];
tz = tan ( z );
nx = 33;
x = linspace ( -1.5, +1.5, nx );
px = newton_value ( c, z, x );
tx = tan ( x );
for i = 1 : nx
  fprintf ( 1, ' %f %f %f\n', x(i), px(i), tx(i) );
end
hold ( 'on' );
plot ( x, px, 'r-', x, tx, 'b-' )
plot ( z, tz, 'k.', 'markersize', 20 );
hold ( 'off' );
```

Listing 22: Evaluate the Wikipedia Newton polynomial.



A Newton polynomial that interpolates $\tan(x)$ *at 5 points.*

20 Sparse form representation

The **sparse form representation** of a polynomial stores three quantities:

- `m`, the number of coefficients;
- `c(1:m)`, the coefficients;
- `e(1:m)`, the exponents;

The polynomial can be reconstructed by:

$$p(x) = \sum_{i=1}^{m} c_i x^{e_i}$$

This representation can be useful if polynomials are being considered which have high degree, but few nonzero coefficients. This representation is also easy to extend to polynomials in multiple variables $x_1, x_2, ..., x_k$; we simply make each exponent a k-dimensional object.

However, this simple and efficient representation means more work when we try to determine the degree, or add or multiply two sparse form polynomials.

Project: Write MATLAB functions to work with polynomials in the sparse representation:

- return degree of a polynomial;
- sort terms by exponent, and add terms with the same exponent;
- add or multiply two polynomials;
- divide one polynomial by another;
- convert to and from power form and factor form;
- compute derivative and antiderivative;

21 No Computing Assignment for this Lab!