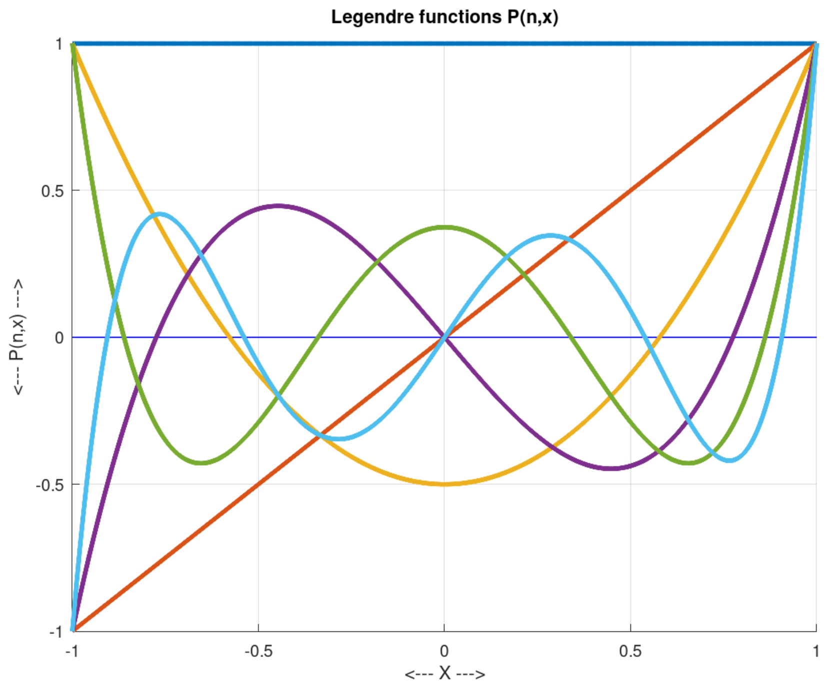
Polynomials:
The Legendre Polynomial Family

MATH2070: Numerical Methods in Scientific Computing I

Location: http://people.sc.fsu.edu/∼jburkardt/classes/math2070_2019/polynomial_legendre/polynomial_legendre.pdf



The initial members of the Legendre polynomial family.

Legendre Polynomials

- *What are the drawbacks of the monomial basis?*
- *How do we define orthogonality of a polynomial basis?*
- *How do we define, evaluate, and represent Legendre polynomials?*
- *How do we project a function onto a basis?*
- *How do we compute a Legendre least squares approximant?*
- *How do we interpolate and approximate?*

1 Problems using monomials for interpolation

Because we are limited to discrete, finite quantities, we often use polynomials for tasks such as interpolation and approximation. Thus, if the approximate solution to a problem is a polynomial, written

$$p(x) = c_0 + c_1 x + c_2 x^2 + ... + c_d x^d$$

then actually our solution is the vector $c = [c_0, c_1, ..., c_d]$ in the space of d-degree polynomials, using as our basis vectors: the monomials $\{1, x, x^2, ..., x^d\}$.

In other applications, we know that it's very useful to choose a set of basis vectors that are orthogonal and have unit norm, or, at least, which have a relatively small overlap with each other. When two basis functions have only a small difference, then projections and other tasks become difficult to do precisely. Because of the limitations of 16 digit arithmetic, using a bad basis can actually cause our calculations to become worthless.

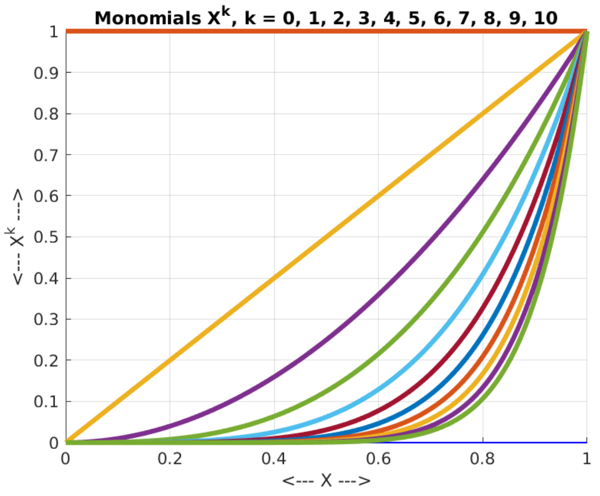
As an example, consider the problem of interpolating a function $f(x)$ over the interval $[0, 1]$ using a monomial basis. Suppose we have n equally spaced sample points, and we want to construct the interpolant of degree $d = n - 1$ by computing the coefficients c of the monomials.

We can simply write down the interpolation conditions:

$$\begin{pmatrix} 1 & x_0 & ... & x_0^d \\ 1 & x_1 & ... & x_1^d \\ ... & ... & ... & ... \\ 1 & x_d & ... & x_d^d \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ ... \\ c_d \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ ... \\ f(x_d) \end{pmatrix}$$

We naturally expect that, if we wish to get a more accurate result, we only need to increase the value of d, taking more samples, solving a bigger system, and getting an interpolant that represents a better match to the function $f(x)$. However, as d increases, the linear algebra solver may warn us that the matrix is becoming nearly singular. And indeed, our computed coefficients c become very inaccurate. This matrix is known as the **Vandermonde** matrix; it is a classic example of an ill-conditioned linear system.

This doesn't mean we can't solve this problem; it means we are going about it the wrong way. If we plot the first few monomial functions and think of them as basis vectors, we see that each additional basis vector is closer to the previous one.



The initial members of the monomial family.

To get more reliable results, we will look for an improved basis for the space of polynomials. The idea that the basis functions should not be so similar can be defined in terms of **orthogonality**.

2 Problems using monomials for approximation

Given a function $f(x)$, suppose that we wish to construct the best polynomial approximant $p(x)$ as a linear combination of $d+1$ basis monomials $\phi_i(x) = x^i$. Our error will be measured using the L2 norm of $e(p,f) = ||f(x) - p(x)||$ over the interval $[-1,+1]$. In other words, our error is not measured at a discrete set of points, but is the integral of the error at every point in the interval.

How do we determine the coefficients c in our linear combination of monomials:

$$p(x) = \sum_{i=0}^{d} c_i x^i$$

We are seeking c that minimizes the approximation error $e(p,f)$. Because square roots can be unnecessarily unpleasant, we can consider the equivalent condition of minimizing the square of the error. Let us write this in a form that makes clear the dependence on c:

$$e^2(p,f) = \int_{-1}^{+1} (\sum_{i=0}^{d} c_i x^i - f(x))^2 \, dx$$

Now the derivative with respect to the i-th component of c is

$$\frac{\partial e^2(p,f)}{\partial c_i} = 2 \int_{-1}^{+1} (\sum_{i=0}^{d} c_i x^i - f(x)) x^i \, dx$$

Setting each partial derivative to zero results in a linear system of the form $A * c = f$:

$$\int_{-1}^{+1} \begin{pmatrix} 1 & x & ... & x^d \\ x & x^2 & ... & x^{d+1} \\ ... & ... & ... & ... \\ x^d & x^{d+1} & ... & x^{2d} \end{pmatrix} dx \quad \begin{pmatrix} c_0 \\ c_1 \\ ... \\ c_d \end{pmatrix} = \int_{-1}^{+1} \begin{pmatrix} f(x) * 1 \\ f(x) * x \\ ... \\ f(x) * x^d \end{pmatrix} dx$$

The matrix on the left hand side might appear to be singular; every row seems to be a multiple of the first row. However, this is only true before we apply the integration operator to each entry, after which the matrix is actually nonsingular.

Each entry of the matrix has the form:

$$A_{i,j} = \int_{-1}^{+1} \phi_i(x) \phi_j(x) \, dx$$

Solving this system gives us the values of c, and our approximant $p(x)$. So the problem is well defined, and our natural approach requires us to compute the matrix A by evaluating $(d+1)^2$ integrals and then solving a dense $(d+1) \times (d+1)$ linear system, at a cost of order $(d+1)^3$.

When we ask for more accuracy, we need to increase d, and this means the cost rises rapidly. The other concern is that the matrix is *ill-conditioned*: each new monomial basis function is almost a linear combination of the preceding ones. This means that increasing d means that the accuracy of our linear system solution deteriorates. The concept of an ill-conditioned matrix will be explained in the follow class on Numerical Linear Algebra. For now, just be aware that:

- every square matrix A has a condition number;
- MATLAB can report the value of this condition number as `cond(A)`;
- If the condition number is 10^k, you can lose about k digits of accuracy in solving a linear system;

3 Computing the monomial approximant

So suppose we want to approximate a function $f(x)$ with a polynomial of degree d. If we use the monomial basis, then we need to compute various integrals involving powers of x and the function $f(x)$. MATLAB makes these individual calculations possible because it provides a function that estimates

```
value = integral ( @(x) x^2 * sin(x), -1.0, +1.0 );
```

Listing 1: Using MATLAB's integal() function.

With the help of `integral()`, we can set up a code to approximate any function $f(x)$ with a monomial basis up to any degree d:

```
function c = monomial_approximate ( f, d )

  A = zeros ( d+1, d+1 );
  for i = 0 : d
    for j = 0 : d
      A(i+1,j+1) = integral ( @(x) x.^i .* x.^j, -1.0, +1.0 );
    end
  end

  b = zeros ( d+1, 1 );
  for i = 0 : d
    b(i+1) = integral ( @(x) x.^i .* f(x), -1.0, +1.0 );
  end

  c = A \ b;

  return
end
```

Listing 2: monomial_approximate.m: Monomial approximation to f(x).

4 Exercise: Approximate sin(x) and humps(x)

Using `c=monomial_approximate(f,d)`, compute the monomial approximant to $\sin(x)$ of degree $d = 5$. Illustrate your result by plotting the function and its approximation together.

```
function monomial_approximate_sin ( d )
  c = monomial_approximate ( @(x)sin(x), d );
  x = linspace ( -1.0, +1.0, 101 );
  y = polyval ( c(end:-1:1), x );
  clf ( );
  hold ( 'on' );
  plot ( x, sin(x), 'g-', 'linewidth', 5  );
  plot ( x, y, 'k-', 'linewidth', 2  );
  hold ( 'off' );
```

Listing 3: monomial_approximate_sin.m: Monomial approximation to sin(x).

Now repeat this process for the function `humps(x)`. You should find that $d = 5$ does not give a very good approximation. Try doubling d with $d = 5, 10, 20, 40, 80$. At first the approximation seems to be improving, but notice what happens eventually. This suggests that the monomial basis is breaking down, because of our finite accuracy. Presumably, **all** methods break down when we go to far. But can we find another polynomial basis for approximation that would allow us to go further than the monomials do?

Our issues with accurately computing the least squares approximant to `humps(x)` suggest that we may need to find a better way to formulate our problem. In particular, we will look for a better set of basis functions for $\mathbb{P}_d$, the space of polynomials of degree d. We want basis functions that are not so closely related to each other. Our best bet is to demand that these basis functions are actually **orthogonal**. This demand will not just improve our accuracy, but actually greatly simply our calculations!

5 Orthogonality of functions

We are familiar with the idea of the orthogonality of vectors, based on the idea of a dot product or inner product. We can generalize this idea to functions $f(x)$ and $g(x)$ if we first agree on an appropriate inner product. There are four variations on this idea, which vary the ideas of *discrete* versus *continuous*, and *unit* versus *weighted*.

$$
\begin{aligned}
< f, g > &= \sum_{i=1}^{n} f(x_i)\, g(x_i) && \text{discrete, unit} \\
< f, g >_w &= \sum_{i=1}^{n} w_i f(x_i)\, g(x_i) && \text{discrete, weighted} \\
< f, g > &= \int_a^b f(x) g(x)\, dx && \text{continuous, unit} \\
< f, g >_w &= \int_a^b w(x) f(x) g(x)\, dx && \text{continuous, weighted}
\end{aligned}
$$

In the discrete cases, we need to specify the number n and location x of the vector of sample points. In the discrete weighted case, the weight vector w should contain only strictly positive values; in the continuous weighted case, the weight function $w(x)$ should be a nonnegative function.

Once we have specified which inner product we want to use, along with the associated data, we can compute inner products. In particular, $f(x)$ and $g(x)$ are orthogonal if $< f, g >= 0$. Any inner product defines a norm, and so we can write

$$||f(x)|| = \sqrt{< f, f >}$$

and we say $f(x)$ has *unit norm* or is *normalized* if $||f(x)|| = 1$.

In particular, over the interval $[a, b]$, we can define the L2 inner product and the L2 norm as

$$
\begin{aligned}
< f, f >_2 &\equiv \int_a^b f(x)^2\, dx \\
||f(x)||_2 &\equiv \sqrt{< f, f >_2} = \sqrt{\int_a^b f(x)^2\, dx}
\end{aligned}
$$

From now on, we will normally omit the "2" subscript, unless there is a need to emphasize that we are using an L2-type norm.

Since the continuous inner products involve integration, we are not always able to produce an exact result. We saw earlier that we can compute dot products and norms of polynomial functions exactly in the case of a continuous unit inner product, but if the inner product includes a weight function $w(x)$, then this may no longer be possible.

In cases involving an inner product or norm based on an integral, we may need to rely on quadrature to estimate the desired value. This could be done using Gauss quadrature. Alternatively, we can use MATLAB's `integral()` function. For instance, to compute

$< f, g >$	q = integral (@f(x)*g(x), a, b)
$< f, g >_w$	q = integral (@w(x)*f(x)*g(x), a, b)
$\|\|f\|\|$	q = sqrt (integral (@f(x).^2, a, b))
$\|\|f\|\|_w$	q = sqrt (integral (@w(x).*f(x).^2, a, b))

6 Example: The Legendre polynomials

Let us suppose that we are interested in least squares approximation in which the error is measured by the L2 norm, and that we are seeking a better set of basis polynomials. Consider the family of **Legendre polynmials**, whose typical element of degree d is designated $P_d(x)$. We will claim that these polynomials are orthogonal with respect to the L2 norm, and hence should be used for the approximation problem.

Somewhat mysteriously, the polynomials are usually defined by stating that they satisfy a three term recurrence:

$$\begin{aligned}
P_0(x) &= 1\\
P_1(x) &= x\\
P_d(x) &= \frac{(2d-1)xP_{d-1}(x)-(d-1)P_{d-2}(x)}{d}
\end{aligned}$$

In other words, it seems that if you want to know the value of $P_3(\pi)$, you have to find it by computing and combining the values of $P_0(\pi), P_1(\pi)$, and $P_2(\pi)$:

$$\begin{aligned}
P_0(\pi) &= 1\\
P_1(\pi) &= \pi\\
P_2(\pi) &= \frac{(2d-1)xP_{d-1}(x)-(d-1)P_{d-2}(x)}{d}\\
&= \frac{3\pi\pi - 1*1}{2}\\
&= \frac{3}{2}\pi^2 - 0.5;\\
P_3\pi &= \frac{(2d-1)xP_{d-1}(x)-(d-1)P_{d-2}(x)}{d}\\
&= \frac{5\pi(\frac{3}{2}\pi^2-0.5)-2\pi}{3}\\
&= 2.5\pi^3 - 1.5\pi
\end{aligned}$$

This may seem like an awkward way to define the polynomials, but it does mean that if we are asked to compute the value of a Legendre polynomial of *any* order, we only need to know the first two values and the rule for computing the next value given the previous two. This makes for a very compact code.

7 What is the value of $P_d(x)$?

Given a point x, we can compute the value $v(x)$ of a Legendre polynomial of order d, such as $P_7(x)$ by starting the recurrence and running up from $k = 0$ to 7. At each step, to compute $v_k(x)$, we only need to keep the last two values, $v_{k-1}(x)$ and $v_{k-2}(x)$:

```
function v = legendre_value ( d, x )

  v = ones ( size ( x ) );  <-- We do this to allow x to be a vector
  if ( d == 0 )
    return
  end

  vkm1 = v;
  v = x;

  for k = 2 : d

    vkm2 = vkm1;
    vkm1 = v;
    v = ( ( 2 * k - 1 ) * x .* vkm1    ...    <-- "dot" here for vectors.
        - (       k - 1 ) *       vkm2 ) ...
        / (       k       );

  end

  return
end
```

Listing 4: legendre_value.m: Evaluating the d-th Legendre polynomial.

Now we can product a plot of $P_0(x)$ through $P_5(x)$ using a code like *legendre_plot.m*, which should look like the plot at the beginning of this document.

Although the Legendre polynomials are defined for the interval $[-1, 1]$, our approximation problem might be framed over some arbitrary interval $[a, b]$. We can easily adjust the polynomials so that they are defined over $[a, b]$ and are L2-orthogonal over tthat interval, simply by applying a linear transformation to the argument.

If x is the argument in $[a, b]$, then we apply the transformation $xi \leftarrow \frac{x-a}{b-a}$, and evaluate the Legendre polynomial at xi:

```
function v = legendre_value_ab ( d, x, a, b )

  xm1p1 = ( ( 1.0 - x ) * a ;
          + ( 1.0 + x ) * b );
          /    2.0;

  v = legendre_value ( d, xm1p1 );

  return
end
```

Listing 5: legendre_value_ab.m: Evaluate Legendre polynomial defined in arbitrary interval.

8 What is the power form representation of $P_d(x)$?

The three term recurrence allows us to evaluate a Legendre polynomial, but perhaps we want to actually look at the power form representation, that is, the vector of polynomial coefficients.

The first few polynomials have the form:

$$
\begin{aligned}
P_0(x) &= 1 \\
P_1(x) &= x \\
P_2(x) &= -0.5 + 1.5x^2 \\
P_3(x) &= -1.5x + 2.5x^3 \\
P_4(x) &= 0.375 - 3.75x^2 + 4.375x^4 \\
P_5(x) &= 1.875x - 8.75x^3 + 7.875x^5
\end{aligned}
$$

It turns out that we can use the recurrence to figure out the coefficients in the power form representation. The coefficient vectors for $P_0(x)$ and $P_x(1)$ are `c0 = [1]` and `c1=[0,1]`. If we have the two previous coefficient vectors `ckm2` and `ckm1`, then here's how we compute `c`:

1. create `c` as a list of length `k+1`;
2. multiply `ckm2` by `-(k-1)/k` and add to `c`.
3. shift `ckm1` one index to the right, multiply by `(2k-1)/k`, and add to `c`;

The code for such a procedure looks like this:

```
function c = r8poly_legendre ( d )

  c = [ 1.0 ];
  if ( d == 0 )
    return
  end

  ckm1 = c;
  c = [ 0.0; 1.0 ];

  for k = 2 : d
    ckm2 = ckm1;
    ckm1 = c;
    c = zeros ( k + 1, 1 );
    c(1:k-1) =              (   - k + 1 ) * ckm2(1:k-1) / ( k );
    c(2:k+1) = c(2:k+1) + ( 2 * k - 1 ) * ckm1(1:k  ) / ( k );
  end

  return
end
```

Listing 6: legendre_coefficients.m: Coefficients of d-th Legendre polynomial.

9 Is $P_i(x)$ orthogonal to $P_j(x)$?

Let's verify that the Legendre polynomials of distinct index are orthogonal. If we have the polynomial coefficients, we could compute the integral exactly, but let's work with the recurrence formula, which only gives us the value at a point. To test the orthogonality of $P_i(x)$ and $P_j(x)$, we can use MATLAB's `integral()` function. Note that we are able to form a complicated expression for the integrand, including a "dot" operator. MATLAB knows which variable is to be integrated over because of the `@(x)` marker:

```
function [ n1, n2, q, r ] = legendre_dot_product ( i, j )

  q = integral ( @(x) legendre_value(i,x) .* legendre_value(j,x), -1.0, +1.0 );

  ni = sqrt ( integral ( @(x) (legendre_value(i,x)).^2, -1.0, +1.0 ) );
  nj = sqrt ( integral ( @(x) (legendre_value(j,x)).^2, -1.0, +1.0 ) );
  r = q / ni / nj;
  angle = acos ( r ) * 180.0 / pi;

  fprintf ( '\n' );
  fprintf ( ' ||P%d(x)|| = %g\n', i, ni );
  fprintf ( ' ||P%d(x)|| = %g\n', j, nj );
  fprintf ( ' <P%d(x),P%d(x)> = %g\n', i, j, q );
  fprintf ( ' <P%d(x),P%d(x)> / ||P%d(x)|| / ||P%d(x)|| = %g\n', i, j, i, j, r );
  fprintf ( ' Angle = %g degrees\n', angle );

  return
end
```

Listing 7: legendre_dot_product.m: The dot product of two Legendre polynomials.

While this test shows us that Legendre polynomials are orthogonal, it also shows that they are **not** normalized!

10 Evaluate a linear combination of Legendre polynomials

The coefficient vector c of a polynomial in standard form is evaluated by using the coefficients as weights applied to the monomial basis. This is how to think of the formula:

$$p(x) = c_0 + c_1 x + ... + c_d x^d$$

Now, instead, we use the Legendre basis to approximate a function $f(x)$, and we get a coefficient vector c. These coefficients are weights applied to Legendre polynomials. We are defining an expansion of the form

$$f(x) \approx p(x) = c_0 p_0(x) + c_1 p_1(x) + ... + c_d p_d(x)$$

and so if we wish to evaluate $p(x)$ we need to work a little harder.

Given c, we need a procedure that will allow us to evaluate our linear combination of Legendre polynomials. We could simply call `legendre_value()` repeatedly for each value of i, multiplying by c_i and summing. But since evaluating a Legendre polynomial means evaluating all the lower order polynomials as well, we might as well just evaluate the highest order, and sum as we go along:

```
function value = legendre_combo_value ( c, x )

  d = length ( c ) - 1;
  value = zeros ( size ( x ) );

  v = ones ( size ( x ) );    <-- v holds the Legendre polynomial p(i,x)
  value = value + c(1) * v;   <-- value sums up c(i) * p(i,x)
  if ( d == 0 )
    return
  end

  vkm1 = v;
  v = x;
  value = value + c(2) * v;

  for k = 2 : d

    vkm2 = vkm1;
    vkm1 = v;
    v = ( ( 2 * k - 1 ) * x .* vkm1    ...
        - (     k - 1 ) *      vkm2 )  ...
```

```
      /  (       k       );
    value = value + c(k+1) * v;
  end

  return
end
```

Listing 8: legendre_combo_value.m: Evaluate a Legendre expansion.

11 Exercise: Evaluate some Legendre combinations

The coefficient vector `c=[0,0,0,0,1]` corresponds to the Legendre polynomial $p_4(x)$. Use `legendre_combo_value()` to sample this function over $[-1,+1]$ and plot it. Compare your plot to the formula $p_4(x) = 0.375 - 3.75x^2 + 4.375x^4$.

The coefficient vector `c=[0, -0.2853, 0, -1.6088, 0, 1.1764, 0, -0.2959]` corresponds to a degree 7 Legendre approximation to `f(x)=sin(5x)`. Plot the approximation and compare it to $f(x)$.

12 The norm of a linear combination of Legendre polynomials

A linear combination of Legendre functions could be represented by

$$p(x) = c_0 P_0(x) + c_1 P_1(x) + ... + c_d P_d(x)$$

To evaluate the L2 norm of $p(x)$, we first evaluate $p(x)^2$, which would result in $(d+1) \times (d+1)$ terms, of the form $c_i c_j P_i(x) P_j(x)$. Then we need to integrate these terms over $[-1,+1]$. But because of orthogonality, every term in which $i \neq j$ will drop out, leaving us with:

$$||p(x)||^2 = c_0^2 ||P_0(x)||^2 + c_1^2 ||P_1(x)||^2 + ... + c_d^2 ||P_d(x)||^2$$

If the Legendre polynomials have been normalized, we would have an even simpler formula!

```
function legendre_combo_norm_test ( c )

  c = c(:);

  d = length ( c ) - 1;
  v = legendre_combo_value ( c, x );
  q = integral ( @(x) ( legendre_combo_value ( c, x ) ).^2, -1.0, +1.0 );
  vnorm = sqrt ( q );
  fprintf ( 1, '  ||v||          = %g\n', vnorm );

  pn = zeros ( d, 1 );
  for i = 0 : d
    pn(i+1) = sqrt ( integral ( @(x) ( legendre_value(i,x)).^2, -1.0, +1.0 ) );
  end
  cnorm = sqrt ( sum ( ( c .* pn ).^2 ) );
  fprintf ( 1, '  sqrt(sum(c^2)) = %g\n', cnorm );

  return
end
```

Listing 9: legendre_combo_norm.m: Norm of a Legendre linear combination.

13 Least squares approximation with Legendre polynomials

Although we chose the monomials as our basis, we can set up the problem using any basis for the space of polynomials. The first $d+1$ Legendre polynomials can be used instead. That means that our system matrix becomes

$$\int_{-1}^{+1} \begin{pmatrix} P_0P_0 & P_0P_1 & ... & P_0P_d \\ P_1P_0 & P_1P_1 & ... & P_1P_d \\ ... & ... & ... & ... \\ P_dP_0 & P_dP_1 & ... & P_dP_d \end{pmatrix} dx \quad \begin{pmatrix} c_1 \\ c_2 \\ ... \\ c_n \end{pmatrix} = \int_{-1}^{+1} \begin{pmatrix} f(x)*P_0 \\ f(x)*P_1 \\ ... \\ f(x)*P_d \end{pmatrix} dx$$

But since the Legendre polynomials are orthogonal, this matrix is actually diagonal. And that means we can write out the solution explicitly:

$$c_i = \frac{< f, P_i >}{< P_i, P_i >} = \frac{\int_{-1}^{+1} f(x)P_i\,dx}{\int_{-1}^{+1} P_iP_i\,dx}$$

Instead of having to build a large system matrix, and worry about the inaccuracies of linear system solving, we just project our function onto each Legendre polynomial, and normalize. So our main concern is ensuring that we estimate our integrals accurately. For us, that can just mean relying on MATLAB's `integrate()` function. Getting rid of the linear system is a huge advantage to the Legendre basis.

14 Computing the Legendre approximant

```
function c = legendre_approximate ( f, d )

  c = zeros ( d + 1, 1 );
  for i = 0 : d
    fp = integral ( @(x) f(x) .* legendre_value ( i, x ), -1.0, +1.0 );
    pp = integral ( @(x) ( legendre_value ( i, x ) ).^2, -1.0, +1.0 );
    c(i+1) = fp / pp;
  end

  return
end
```

Listing 10: legendre_approximate.m: Approximation with Legendre basis.

15 Exercise: Approximate the humps function

We recall that the `humps()` function was not well approximated using the monomial basis. We can try again, but this time using the Legendre polynomials as our basis for the polynomial space $\mathbb{P}_n$.

For increasing values $d = 5, 10, 20, 40$, use `c=monomial_approximate(f,d)` to request the coefficients of the least squares approximant to `humps()`, and compare the approximant and the function in a plot:

```
function legendre_approximate_humps ( d )

  c = legendre_approximate ( @(x)humps(x), d );

  x = linspace ( -1.0, +1.0, 101 );
  y = legendre_combo_value ( c, x );
  clf ( );
  hold ( 'on' );
  plot ( x, humps(x), 'g-', 'linewidth', 5  );
  plot ( x, y, 'k-', 'linewidth', 2  );
  hold ( 'off' );
```

Listing 11: legendre_approximate_humps.m: Compare humps(x) and approximant.

16 Computing Assignment #12

We showed how to use the Legendre recurrence relationship to work out the polynomial coefficients of the Legendre polynomials, that is, the power form representation.

Another polynomial family is known as the *Chebyshev polynomials*. A typical element of degree d is symbolized by $T_d(x)$ The three term recurrence is:

$$\begin{aligned} T_0(x) &= 1; \\ T_1(x) &= x; \\ T_d(x) &= 2\,x\,T_{d-1}(x) - T_{d-2}(x) \end{aligned}$$

Write a program *c=chebyshev_coefficients(d)* that accepts a degree d and returns the polynomial coefficients of $T_d(x)$. If you start from the code *legendre_coefficients.m*, you won't need to make too many changes. If you give your program the input value $d = 5$, you should expect it to return the coefficient vector `c=[0, 5, 0, -20, 0, 16 ]` because

$$T_5(x) = 5x - 20x^3 + 16x^5$$

Turn in: your file *hw12.m* by Wednesday, November 20.