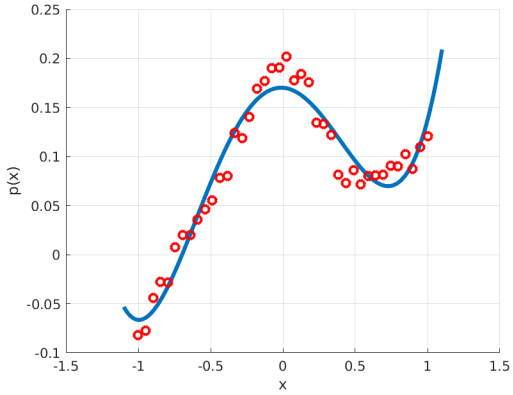
Polynomials
Mathematical Programming with Python

https://people.sc.fsu.edu/~jburkardt/classes/mpp_2023/polynomials/polynomials.pdf

---



A set of data seems to behave almost like a fifth degree polynomial. Can the formula suggest an explanation for these results?

Polynomials

- *Polynomials are a mathematician's favorite functions;*
- *They are easy to represent, analyze, and manipulate;*
- *Many functions can be approximated by polynomials to high accuracy;*
- *Finding the roots of a polynomial can be a difficult task;*
- *There are many special families of polynomials for particular problems such as interpolation and quadrature*

1 Polynomials

Mathematically, a polynomial of degree n is a function $p(x)$, which can be represented by a formula involving $n+1$ coefficients c, of the form:

$$p(x) = c_0 + c_1 x + ... + c_{n-1} x^{n-1} + c_n x^n$$

We assume c_n is not zero. We will generally assume that the coefficients c and the dependent variable x are real numbers.

For convenience, we will sometimes write degree(p) to denote the degree of a polynomial. To be clear, degree$(8 - 5x + 7x^2 + 2x^3) = 3$

Given the $n+1$ coefficients c of a polynomial $p(x)$, we know that the derivative $p'(x)$ can be represented by

$$p'(x) = c_1 + 2c_2 x + ... + (n-1)c_{n-1} x^{n-2} + nc_n x^{n-1}$$

Computationally, we would create a coefficient array `d[]` of n elements, and, for $0 \leq i \leq n-1$, set $d[i] = (i+1) * c[i+1]$

We can also define $\int p(x)$, an antiderivative of $p(x)$, writing

$$\int p(x) = C + c_0 x + \frac{c_1}{2}x^2 + ... + \frac{c_{n-1}}{n}x^n + \frac{c_n}{n+1}x^{n+1}$$

where C is an arbitrary constant which we could set to zero for convenient. Computationally, we create a coefficient array `e[]` of $n+2$ elements, and, for $1 \leq i \leq n+1$, set $e[i] = \frac{c[i-1]}{i}$, setting e[0] = 0.

A polynomial of degree n always has n roots r_i, and hence has the sometimes convenient alternate representation:

$$p(x) = \prod_{i=0}^{i<n}(x - r_i)$$

Even if the coefficients c are real, some or all of the roots may be complex numbers.

Given the roots r, it is fairly easy to compute the coefficient vector c. On the other hand, from the coefficients c, it can be a difficult task to approximate the set of roots r.

For a sufficiently differentiable function $f(x)$, we can construct a Taylor polynomial of degree n that approximates the function in a neighborhood of a point x_0 as:

$$f(x) = \sum_{i=0}^{n} \frac{f^{(n)}(x_0)}{i!}(x - x_0)^i + r(x)$$

where $r(x)$ represents a remainder or error term of the form $\frac{f^{n+1}(\xi)}{(n+1)!}(x - x_0)^{n+1}$ for some $x_0 \leq \xi \leq x$. Assuming we know how to differentiate $f(x)$, the Taylor series gives us a method of constructing a sequence of polynomials that presumably approximate the function over a given interval.

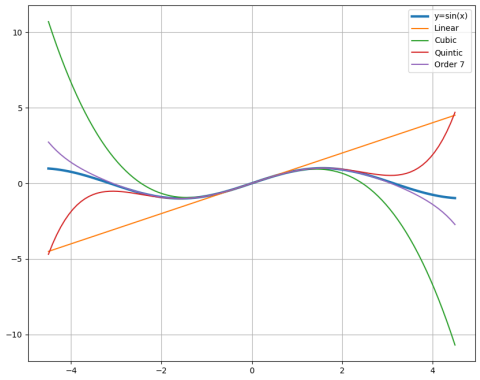
Other methods for finding a polynomial that is close to some function or set of data include *Lagrange interpolation* and *Legendre approximation*. In general, we are trying to understand a complicated function by discovering the appropriate polynomial to which we can apply many analytical tools.

2 Taylor Polynomial Demonstration

The function $\sin(x)$ has the power series expansion

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

We can compute and plot the Taylor polynomials, centered at $x_0 = 0$, and see how they compare to the original function. All values of $\sin(x)$ stay between -1 and +1, and the function has an infinite number of maxima and minima. Nonconstant polynomials, on the other hand, have only finitely many local extreme points, and must eventually go to $\pm\infty$. This means that only locally, near $x = x_0$, can we expect our polynomials to mimic the behavior of $\sin(x)$.



Local approximation of $\sin(x)$ by Taylor polynomials.

The plot suggests that the approximation power is actually very good near x_0. However, if we had plotted even a slightly greater x interval, we would see that the polynomials quickly diverge to infinity, indicating that this kind of polynomial representation is only locally useful.

3 The Python Polynomial Class

While it is possible for the devoted programmer to implement the functions necessary to work with polynomials, the `numpy` library includes a `Polynomial` package, which implements all these manipulations as function calls.

Typically, one gets access to the package by the following statements:

```
import numpy as np
import numpy.polynomial.polynomial as poly  # Needed to access various methods
```

Following these declarations, the library functions are called with the `poly.` prefix.

To illustrate, we will begin by defining three polynomials, `p()`, `q`, and ttr. We do this by specifying an array containing the coefficients. The Polynomial package convention is that, for a polynomial of degree n, the zero-th array entry is the constant term, and the n-th entry is the coefficient of x^n.

Our polynomials are

$$p(x) = x^2 + 4$$
$$q(x) = -6 + 11x - 6x^2 + x^3$$
$$r(x) = 2x - 4$$

Here are things we can do:

```
#  Define polynomials by coefficient vectors:
  import numpy.polynomial.polynomial as poly
  p = poly.Polynomial( [ 4, 0, 1 ] )
  q = poly.Polynomial( [ -6, 11, -6, 1 ] )
  r = poly.Polynomial( [ -4, 2 ] )
```

```
#  Manipulate coefficient vectors:
  pc = p.coef
```

```
  print ( pc )
  if ( pc[0] == 0 ):
    print ( 'Polynomial is divisible by x!' )
```

```
#  Request the degree of the polynomial:
  d = p.degree
```

Note that the degree is not necessarily the same as the length (minus 1) of the coefficient vector!

```
#  Add, subtract, multiply, divide, exponentiate, compose polynomials:
  ppq = p + q
  pmq = p - q
  ptq = p * q
  psq = p**2
  pofq = p(q)
```

```
#  Synthetic division: q = f * r + s:
  f = q // r  # note the double slash!
  g = q % r   # polynomial remainder
  q = f * r + g
```

In particular, to see what is going on with the division and remainder functions, consider performing "synthetic division" to divide polynomial `q` by `r`. We are seeking a factor polynomial `f` and remainder polynomial `g` so that $q(x) = f(x) * r(x) + g(x)$. The $f(x)$ factor polynomial is computed by `q//r` and the remainder by `q%r`. You can confirm the correctness of this calculation by evaluating $f(x) * r(x) + g(x)$ and comparing it to $q(x)$. If you think you have found a root of q, this is one way to verify.

Once you have defined a polynomial, you can use standard function notation to evaluate it at a point x or at an array of points:

```
#  Evaluate polynomials at one or many points:
  import numpy.polynomial.polynomial as poly
  p = poly.Polynomial ( [ 4, 0, 1 ] )
  print ( 'p(4) = ', p(4) )
  x = np.linspace ( -2, +2, 101 )
  plt.plot ( x, p(x) )
  plt.show ( )
```

Instead of specifying the coefficients, we can define a polynomial by giving its roots. (It's not too hard to write your own function that computes the coefficients based on the roots.). In this case, Python will normalize the coefficints so that the highest degree coefficient is 1.

```
#  Compute polynomial (and coefficients) from roots:
  import numpy.polynomial.polynomial as poly
  roots = np.array ( [ 1.0, 2.0, 3.0 ] )
  s = poly.Polynomial.fromroots( roots )
```

A significantly more difficult task is to determine the roots of a polynomial. In this case, generally, numerical methods must be used, which produce a good approximation to these values. Also, even if the coefficients are real, some roots may be complex.

```
#  Roots of a polynomial given coefficients
  import numpy.polynomial.polynomial as poly
  p = poly.Polynomial ( [ 4.0, 0.0, 1.0 ] )
  roots = p.roots
  q = poly.Polynomial ( [ -6.0, 11.0, -6.0, 1.0 ] )
  roots = q.roots
  x = np.linspace ( 0.0, 4.0, 101 )
  plt.plot ( x, q(x), 'b-' )
  plt,plot ( q.roots, np.zeros ( len(roots), 'ro' )
```

The derivative and antiderivative polynomials can be easily computed as well. The Python functions need a copy of the polynomial coefficient vector.

```
#  Derivative and antiderivative:
  import numpy.polynomial.polynomial as poly
  c = np.array ( [ 4.0, 0.0, 1.0 ] )
  p = poly.Polynomial ( c )
  pd = poly.polyder ( c )
  pa = poly.polyint ( c )
```

4 Example applying Newton's Method to a Polynomial

```
  import numpy as np
  import numpy.polynomial.polynomial as poly

  c = [ -6.0, 11.0, -6.0, 1.0 ]
  q = poly.Polynomial ( c )

  cp = poly.polyder ( c )
  qp = poly.Polynomial ( cp )

  x = 3.5
  n = 0

  while ( 1.0e-8 < abs ( q ( x ) ) ):
    n = n + 1
    dx = - q ( x ) / qp ( x )
    x = x + dx

  print ( '  After', n, 'iterations, q(', x, ') = ', q ( x ) )
```

5 Lagrange Basis Polynomials

Suppose we have *ndata* pairs of data values (x_j, y_j). (We assume that the x_j values are distinct!) We are interested in discovering a simple functional relationship, of the form $y = f(x)$, for which this data is evidence. We might assume that this functional form is a polynomial. By using the fact that a polynomial of degree n can have n zeros, it is possible to convince yourself that there is a unique polynomial $p(x)$ of degree $n = ndata - 1$ so that $p(x_j) = y_j$ for all n data values. How could we determine this polynomial?

If we think about all the polynomials of degree n evaluated at the points x_j, we have a kind of linear vector space. When you find yourself in a vector space, it's natural to look for a simple basis for that space. Here, we can use basis vectors $\ell_i(x)$ which are n degree polynomials such that

$$\ell_i(x_j) = \begin{cases} 1 \text{ if } i = j \\ 0 \text{ if } i \neq j \end{cases}$$

This means we can write a formula as follows

$$\ell_i(x) = \frac{\prod_{j=0, j\neq i}^{j=n} (x - x_j)}{\prod_{j=0, j\neq i}^{j=n} (x_i - x_j)}$$

The top of the fraction verifies that $\ell_i(x)$ is zero in all the right places, and the bottom guarantees that $\ell_i(x_i) = 1$. Because both top and bottom products always skip one of the potential $ndata = n + 1$ factors,

the resultant polynomial does, indeed, have degree n. The basis functions $\ell_i(x)$ that we have constructed are known as the *Lagrange interpolating polynomials* for the given set of data.

This means we could evaluate, for instance, $\ell_i(x)$ as follows:

```
x = 1.234
li = 1.0
for j in range ( 0, ndata ):
  if ( i != j )
    li = li * ( x - xdata[j] ) / ( xdata[i] - xdata[j] )
```

6 Lagrange Interpolation

You should see that we can now construct a polynomial that passes exactly through our data values, namely

$$p(x) = \sum_{i=0}^{i=n} y(x_i)\, \ell_i(x)$$

Now suppose that we have `ndata` data values in arrays `xdata,ydata`, and we want to plot the polynomial that passes through this data.

```
xplot = np.linspace ( 0.0, 5.0, nplot )
pplot = np.zeros ( nplot )

for i in range ( 0, ndata ):
  li = np.ones ( nplot )
  for j in range ( 0, ndata ):
    if ( i != j ):
      li = li * ( xplot - xdata[j] ) / ( xdata[i] - xdata[j] )
  pplot = pplot + ydata[i] * li

plt.plot ( xplot, yplot, 'b-' )
plt.plot ( xdata, ydata, 'ro' )  # Confirm the polynomial goes through the data
```

7 Approximation by a Linear Polynomial

When we have a small number of accurate data points, it may seem natural to try to represent this information by finding a polynomial that exactly matches the data, and varies smoothly in between the data points. However, when we have large amounts of data, or we expect that the data is somewhat inaccurate, another approach is to *approximate* the data, that is, to find a polynomial $p(x)$ which is close to the data in some sense. Often, going exactly through the data requires a high degree, highly oscillatory polynomial, whereas, if we will accept an approximation, a lower degree, more regular-looking polynomial will do the job.

A common measurement of closeness is to compute the ℓ^2 norm of the error, that is, the square root of the sum of the squared differences between the data and the polynomial:

$$e = \sqrt{\sum_{i=0}^{i<ndata} (p(xdata_i) - ydata_i)^2}$$

To solve this problem, we represent the approximating polynomial in terms of its coefficients c, and then seek to minimize the error e by finding values of c so that $\frac{\partial e}{\partial c_i} = 0$ for each component i.

If we will be satisfied with a linear approximation, then we can work out the solution to this problem by hand, resulting in:

$$\bar{x} = \frac{1}{ndata} \sum_{i=0}^{i<ndata} x_i$$

$$\bar{y} = \frac{1}{ndata} \sum_{i=0}^{i<ndata} y_i$$

$$c_1 = \frac{(x - \bar{x})^T (y - \bar{y})}{(x - \bar{x})^T (x - \bar{x})}$$

$$c_0 = \bar{y} - c_1 \bar{x}$$

$$p(x) = c_0 + c_1 x$$

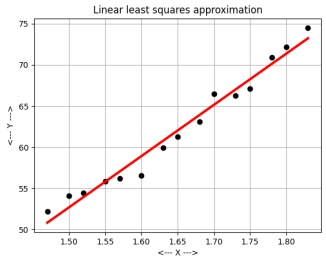
For example, given the data:

```
x = [  1.47,  1.50,  1.52,  1.55,  1.57,  1.60,  1.63,  1.65,  1.68,  1.70,
       1.73,  1.75,  1.78,  1.80,  1.83 ]
y = [ 52.21, 54.12, 54.48, 55.84, 56.20, 56.57, 59.93, 61.29, 63.11, 66.47,
      66.28, 67.10, 70.92, 72.19, 74.46 ]
```

the above procedure produces the approximating polynomial:

$$p(x) = -39.062 + 61.272 * x$$

The plot suggests that this result is correct:



A linear approximation to 15 data values.

8 Approximation by a General Polynomial

If all data was roughly linear, we could stop here. But simply plotting our data points may suggest that a linear model is inadequate, whereas a somewhat higher degree polynomial might capture much more of the behavior suggested by our data. While the least squares procedure can easily be applied to such problems, we will need to be careful in order to get an accurate result. Using the standard polynomial basis $\{1, x, x^2, x^3, ...\}$ builds in a potential for error, because the higher degree basis polynomials are actually too similar in behavior. As the user requests approximating polynomials of higher and higher degree, the computed solution may actually not be the best possible, and eventually the calculation may even fail because the internal linear system becomes nearly singular.

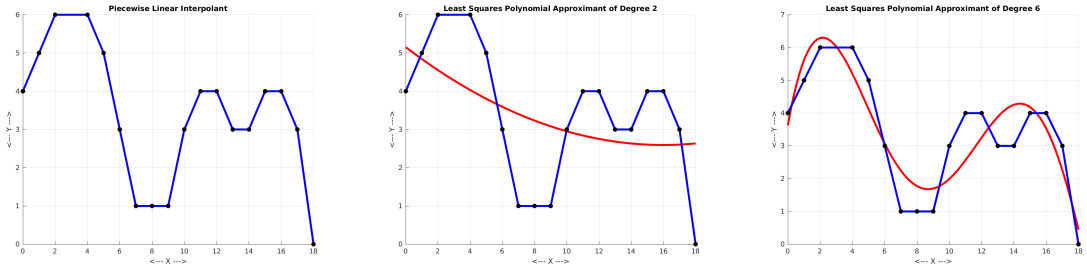
For this reason, users are advised to rely on prewritten least squares solvers, which internally might use a much better basis, such as the Legendre polynomials. The Python `polynomial` library includes exactly such a procedure, called `fit`. Here is how it would be used to solve our simple linear problem, returning the approximating polynomial coefficients in an array `c`:

```
degree = 1
c = poly.polyfit ( xdata, ydata, degree )
```

Consider the following set of data:

```
xdata = [ 1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18 ]
ydata = [ 5, 6, 6, 5, 3, 1, 1, 1,  3,  4,  4,  3,  3,  4,  4,  3,  0 ]
```

It is a simple task to use `polyfit()` to approximate this data with degree = 2 and degree = 6:



Seventeen data values apprroximated by degree 2 and degree 6 polynomials.

By increasing the degree, we have clearly improved the degree of approximation. However, we may hesitate to go much further, since we will be computing polynomials that have to oscillate wildly to improve the approximation. We would be better stopping here, or trying to gather more data that will help us form a better picture of what is going on.

9 The L^2 norm of a polynomial

The norm of a function $f(x)$ is usually defined over a finite interval $[a, b]$, and is typically defined by some integral formula. The most common is the L^2 norm, which is defined by

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

Function norms are useful when making estimates of integrals, errors, or convergence rates.

A polynomial $p(x)$ is a function too, so we can take its norm over a finite interval. The Python `polynomial` class makes it possible to compute this result exactly. For example:

```
  import numpy as np
  import numpy.polynomial.polynomial as poly

  a = 0.0
  b = 5.0
#
#  Define q
#
  c = [ -6.0, 11.0, -6.0, 1.0 ]
  q = poly.Polynomial ( c )
#
#  Define q^2
#
  q2 = q * q
#
#  Compute the antiderivative q2int of q^2
#
```

```
  cq2 = q2.coef
  cq2int = poly.polyint ( cq2 )
  q2int = poly.Polynomial ( cq2int )
#
#  norm = sqrt ( integral ( q^2 ) )
#
  norm = np.sqrt ( q2int ( b ) - q2int ( a ) )
```

To check whether this result is reasonable, let's use the trapezoidal rule to estimate the same integral:

```
  x = np.linspace ( a, b, 101 )
  q2x = q(x) ** 2
  norm = np.sqrt ( np.trapz ( x, q2x ) )
```

10 The maximum of a polynomial

Over a finite interval $[a, b]$, a polynomial $p(x)$ must take on a maximum value. Can we compute the argument x^* and maximum value $p(x^*)$?

Calculus tells us to look for extreme values where the derivative $p'(x) = 0$. Our calculus teacher reminds us that, over a finite interval $[a, b]$, we must also consider the possibility that the maximum value can occur at one of the endpoints. So our strategy for finding the maximum might be:

1. From $p(x)$, define $p'(x)$;
2. Compute the roots of $p'(x)$;
3. Evaluate $p(x)$ at every root of $p'(x)$ that is in $[a, b]$;
4. Evaluate $p(x)$ at a and b;
5. Return the maximum value observed.

The exercises ask you to find the maximum of $q(x) = -6 + 11x - 6x^2 + x^3$ over various intervals.

11 The Continuous Approximation Problem

As we have seen, the Python function `polyfit()` can solve the problem of approximating a (discrete) set of data values `xdata,ydata`. But it is an entirely different, and much more difficult problem, to approximate at given function $f(x)$ over some interval $[a, b]$. We want to ask for the "best" approximation, and so we need first to agree on a measure of error. It is most common to use the L^2 norm for this purpose. Let us further suppose that our approximation will be made by specifying some polynomial $p(x)$ of degree n. So we our problem is specified as determining $p^*(x)$, such that $||f - p^*||_2$ minimizes the L^2 error over all possible degree n polynomials.

If we use the standard power-sum form for a polynomial, then we can define our polynomial in terms of a coefficient vector c, and set up our optimization problem by differenting the error with respect to each coefficient c_i and solving the result linear system. The only problem is that this linear system will be badly conditioned; that is, especially if we increase the polynomial degree h, the computed solution will become increasingly inaccurate because of our finite arithmetic system.

Instead of the power-sum polynomials, we can instead write our unknown polynomial as a sum of $n + 1$ Legendre polynomials, that is:

$$p(x) = c_0 L_0(x) + c_1 L_1(x) + ... + c_n L_n(x)$$

Now instead of minimizing the error, let's minimize the square of the error:

$$e(x) = \int_a^b (f(x) - \sum_{i=0}^{i=n} c_i L_i(x))^2 \, dx$$

so the i-th minimization equation will be

$$\frac{\partial e(x)}{\partial c_j} = 0 = 2 * \int_a^b (f(x) - \sum_{i=0}^{i=n} c_i L_i(x)) L_j(x) \, dx$$

but, as we will see, $\int_a^b L_i(x) \, L_j(x) \, dx = 0$ if $i \neq j$, implying

$$\frac{\partial e(x)}{\partial c_j} = 2 * \int_a^b (f(x) - c_j L_j(x)) L_j(x) \, dx$$

or, in other words,

$$c_j = \frac{\int_a^b f(x) L_j(x) \, dx}{\int_a^b L_j^2(x) \, dx}$$

In other words, there is now no linear system to solve for the coefficients, which we can instead compute directly.

It remains to be seen how we can produce these amazingly useful polynomials $L_i(x)$.

12 Legendre Polynomials

The Legendre polynomials are defined for the interval $[-1, +1]$ To begin with, let's list the first few Legendre polynomials

$$\begin{aligned}
L_0(x) &= 1 \\
L_1(x) &= x \\
L_2(x) &= (3x^2 - 1)/2 \\
L_3(x) &= (5x^3 - 3x)/2 \\
L_4(x) &= (35x^4 - 30x^2 + 3)/8 \\
L_5(x) &= (63x^5 - 70x^3 + 15x)/8 \\
L_6(x) &= (231x^6 - 315x^4 + 105x^2 - 5)/16
\end{aligned}$$

Here are a few properties of Legendre polynomial $L_i(x)$ that are worth checking:

- $-1 \leq L_i(x) \leq +1$ for $-1 \leq x \leq +1$
- $L_i(-1) = \pm 1$, $L_i(1) = +1$
- $\int_{-1}^{+1} L_i(x) \, dx = 0$ for $0 < i$
- $||L_i(x)||_2^2 = \int_{-1}^{+1} L_i(x) L_i(x) \, dx = \frac{2}{2i+1}$
- $< L_i, L_j >= \int_{-1}^{+1} L_i(x) L_j(x) \, dx = 0$ for $i \neq j$

The Legendre polynomials are pairwise orthogonal over the interval $[-1, +1]$ with respect to the L^2 norm. Let's verify that for L_4 and L_6

```
l4 = poly.Polynomial ( [3, 0, -30, 0, 35] ) / 8.0
l6 = poly.Polynomial ( [-5, 0, 105, 0, -315, 0, 231] ) / 16.0
l46 = l4 * l6
l46_int_coef = poly.polyint ( l46.coef )
l46_int = poly.Polynomial ( l46_int_coef )
dot46 = l46_int ( +1.0 ) - l46_int ( -1.0 )
(result: dot46 = 0 )
```

13 Three term recurrence representation

The Legendre polynomials satisfy a three term recurrence relationship. This means two things:

- Given formulas for $L_i(x)$ and $L_{i-1}(x)$, you have a formula for $L_{i+1}(x)$;
- Given *values* for $L_i(x)$ and $L_{i-1}(x)$ at some x, you have the value $L_{i+1}(x)$;

For the Legendre polynomials, the two starting values, and the three term recurrence relation for going further are:

1. $L_0(x) = 1$
2. $L_1(x) = x$
3. $L_{i+1}(x) = \frac{(2i+1)\,x\,L_i(x) - i\,L_{i-1}(x)}{i+1}$

For instance, we can immediately determine the formula for $L_2(x)$:

$$\begin{aligned} L_2(x) &= \frac{(2*1+1)\,x\,L_1(x) - 1\,L_0(x)}{1+1} \\ &= \frac{3}{2}(x) - \frac{1}{2}1 \\ &= \frac{3}{2}x - \frac{1}{2} \end{aligned}$$

and of course, knowing $L_2(x)$, we can define $L_3(x)$ and continue as far as we want. Using the Python polynomial functions, we can automate this process.

Consider the following procedure for defining $L_5(x)$:

```
x = poly.Polynomial ( [ 0, 1 ] )  # we have to define "x" as a polynomial!
li = poly.Polynomial ( [ 1 ] )
lip1 = poly.Polynomial ( [ 0, 1 ] )
for i in range ( 1, 5 ):
  lim1 = li
  li = lip1
  lip1 = ( ( 2 * i + 1 ) * x * li - i * lim1 ) / ( i + 1 )
```

It is also true that if we simply have the values of the Legendre polyomials $L_0(x)$ and $L_1(x)$ at one or more points x, we can compute the values of the higher polyomials as well. Consider this procedure for plotting the Legendre function $L_5(x)$:

```
n = 101
x = np.linspace ( -1.0, +1.0, n )  # Now x is just a list of values
li = np.ones ( n )
lip1 = x.copy ( )  # Remember, "lip1=x" doesn't do what we want!
for i in range ( 1, 5 ):
  lim1 = li.copy()
  li = lip1.copy()
  lip1 = ( ( 2 * i + 1 ) * x * li - i * lim1 ) / ( i + 1 )
plt.plot ( x, lip1, 'b-' )
```

14 Computing a Legendre Polynomial Approximation

We have already seen that the `polyfit()` function will produce a polynomial $p(x)$ that approximates a discrete set of data. But now we are interested in approximating a function $f(x)$, in a way that minimizes the L^2 norm of the error. So far, we only know how to work with Legendre polynomials on the interval $[-1,+1]$, so let's try to approximate the function $f(x) = \frac{1}{1+5x^2}$ over this interval, using a 3rd degree polynomial. That is, we are seeking the 4 values c such that

$$f(x) \approx c_0 L_0(x) + c_1 L_1(x) + c_2 L_1(x) + +c_3 L_1(x)$$

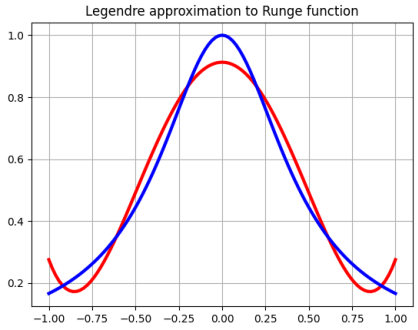
From our previous work, we know the formula for the coefficients, and we can estimate the corresponding integrals using `trapz()`: immediately:

```
npoly = 3
c = np.zeros ( npoly + 1 )
nx = 101
x = np.linspace ( -1.0, +1.0, nx )
for i in range ( 0, npoly + 1 ):
  if ( i == 0 )
    li = np.ones ( nx )
  elif ( i == 1 )
    lim1 = li.copy()
    li = x.copy()
  else
    lim2 = lim1.copy()
    lim1 = li.copy()
    li = ( ( 2 * i - 1 ) * x * lim1 - ( i - 1 ) * lim2 ) / i
  y = f(x) * li
  lii = 2 / ( 2 * i + 1 )
  c[i] = trapz ( x, y ) / lii
```

Once we have the coefficients `c`, we can make a plot that compares our approximation $p(x)$ to the function $f(x)$:

```
npoly = 3
nx = 101
x = np.linspace ( -1.0, +1.0, nx )
y = np.zeros ( nx )
for i in range ( 0, npoly + 1 ):
  if ( i == 0 )
    li = np.ones ( nx )
  elif ( i == 1 )
    lim1 = li.copy()
    li = x.copy()
  else
    lim2 = lim1.copy()
    lim1 = li.copy()
    li = ( ( 2 * i - 1 ) * x * lim1 - ( i - 1 ) * lim2 ) / i
  y = y + c[i] * li

plt.plot ( x, y, 'r-' )
plt.plot ( x, f(x), 'b-' )
```



Degree 5 Legendre polynomial approximation of $\frac{1}{1+5x^2}$.

Our plot strongly suggests that we have made an excellent approximation to our target function over the interval $[-1, +1]$, although it is also clear that the approximation will become terrible if we move outside this interval.

The three-term recurrent relationship may seem a bit awkward, certainly compared to our simple power series polynomial basis functions $\{1, x, x^2, ...\}$. However, this approach allows us to automatically construct an orthogonal basis for polynomials, which makes our computations reliable, and especially simplifies the task of projection.

15 Exercises

1. Using the Python `polynomial` function `polyder()`, create a function which accepts as input a value x_0, a function $f(x)$, and a set of derivative values $c = [f(x_0), f'(x_0), f''(x_0), ..., f^n(x_0)]$. Using this information, create a plot that compares $f(x)$ and its Taylor polynomial of degree n.
2. Instead of using the Python `polynomial` function `.fromroots()`, write your own function which accepts a vector of roots `r`, and returns the coefficient vector `c` of the corresponding polynomial $p(x)$. Demonstrate your code for the case where `r=[-2,0,2,2,5]`.
3. Show that, for a given set of data, the Lagrange basis polynomials must sum to 1. Don't use properties of polynomials; use properties of fractions. You can make a general case. For simplicity, you could consider a situation in which we have three data values. Show, in a convincing argument, by writing out the definitions, that $\ell_0(x) + \ell_1(x) + \ell_2(x) = 1$
4. If you are careful, it is possible to use Python's `polynomial` class to define and evaluate a set of Lagrange basis polynomials. Try to set up a general code that accepts `xdata,ydata`, and creates a plot of the polynomial that interpolates this data. For each index i, you can define $\ell_i(x)$ in two steps: define a temporary $p(x)$ using the `.fromroots()` function using all `xdata` except the i-th value; then define $\ell_i(x) = \frac{p(x)}{p(xdata_i)}$. Now you just need a loop that sums up the values $ydata_i\, \ell_i(x)$ in order to make your plot.
5. Write a function which accepts a polynomial $p(x)$ and an interval $[a, b]$, and returns the argument x^* in $[a, b]$ at which $p(x)$ attains its maximum value. Consider the polynomial $q(x) = -6+11x-6x^2+x^3$. Where is its maximum value in the interval [1,3]. In [2,4]. In [0,1]?
6. The L^1 norm of a function $f(x)$ over the interval $[a, b]$, written $||f||_1$, is defined as the integral of $|f(x)|$ over the interval. Write a function which accepts a polynomial $p(x)$ and an interval $[a, b]$, and returns $||p||_1$. Note that $|p(x)|$ is **not** a polynomial, and so computing this integral exactly is not so simple. Can you still do it exactly? Can you estimate it approximately?
7. The L^∞ norm of a function $f(x)$ over the interval $[a, b]$, written $||f||_\infty$, is defined as the maximum value of $|f(x)|$ over the interval. Write a function which accepts a polynomial $p(x)$ and an interval

$[a, b]$, and returns $||p||_\infty$.

8. It was claimed that, for any Legendre function $L_i(x)$ with $0 < i$, we have that $\int_{-1}^{+1} L_i(x)\,dx = 0$. Verify this claim for $i = 0, 1, 3, 10$.
9. It was claimed that, for any Legendre function $L_i(x)$, we have that $||L_i(x)||_2^2 = \frac{2}{2i+2}$. Verify this claim for $i = 0, 1, 3, 10$.
10. We have seen that the Legendre polynomials $L_n(x)$ are orthogonal over the interval $[-1, +1]$ with respect to the L^2 norm. That is, if $i \neq j$, then $\int_{-1}^{+1} L_i(x)L_j(x)\,dx = 0$. Compare the situation that occurs if we use the power basis $P_n(x) = x^n$. Compute a table of the inner products $< P_i(x), P_j(x) >$ for i and j from 0 to 10.
11. The Chebyshev polynomials $T_n(x)$ are also defined over the interval $[-1, +1]$ by a three term recurrence relation:

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_{n+1}(x) &= 2xT_n - T_{n-1} \end{aligned}$$

Using this recurrence formula by hand, what is the formula for $T_4(x)$? Using the Python polynomial functions, write a code that computes the polynomial coefficients for a Chebyshev polynomial of degree n. Use it to print out a table of these polynomials for n from 0 to 10.