Computational Methods in Economics

Lecture 6a - Function Approximation (Part 1)

In [1]:
```
# Author: Alex Schmitt (schmitt@ifo.de)

import datetime
print('Last update: ' + str(datetime.datetime.today()))
```

```
Last update: 2017-12-14 10:59:02.579111
```

Preliminaries

Import Modules

In [2]:
```
import numpy as np
import scipy.optimize

import matplotlib.pyplot as plt
%matplotlib inline
import seaborn

# import sys
from importlib import reload
```

This Lecture

Introduction

Suppose you are interested in a function f, but only have "limited information" about it. That is, you only know its function values at a finite number m points $x_1, x_2, \ldots, x_m$:

$$y_1 = f(x_1), y_2 = f(x_2), \ldots, y_m = f(x_m)$$

The goal of function approximation or "curve fitting" is to use this limited information to find a function $\hat{f}$ such that

$$f(x) \approx \hat{f}(x)$$

The function $\hat{f}$ (also called *approximant*) should be computationally tractable and easy to evaluate. In this course, we will confine ourselves to functions $\hat{f}$ that are linear combinations of a set of $n+1$ linearly independent *basis functions* $B_0, \ldots, B_n$:

$$\hat{f}(x) = \sum_{j=0}^{n} a_j B_j(x)$$

The number n is referred to as the *degree of approximation*. Approximating a function then means to find appropriate values for the $n+1$ *basis coefficients* $a_0, \ldots, a_n$ given the "data" $\{y_i, x_i\}_{i=1}^{m}$.

Some comments

- With respect to terminology, I use "curve fitting" and "function approximation" as synonyms: any problem where we impose a parametric, functional relationship $\hat{f}$ between a *dependent* variable on the left hand side and a (vector of) *explanatory* variables on the right hand side. These variables are "data" (not necessarily from the real world!) in the sense that given those, we want to find the parameters of the function that give a "good fit".
- The first distinction is whether we assume the functional relationship is *linear or nonlinear in the parameters*. Most methods used in empirical or numerical work rely on linear methods, in particular on the use of polynomials.

To simplify the exposition, I will first introduce the most important concepts for univariate functions. What follows below also holds for multivariate functions, in which case each of the "points" $x_1, x_2, \ldots, x_m$ is actally a vector. For clarity, I will use bold notation in this case: $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$, where

$$\mathbf{x}_i = (x_{1i}, x_{2i}, \ldots, x_{ki}).$$

Then, we approximate a function $f(\mathbf{x})$ by:

$$f(\mathbf{x}) \approx \hat{f}(\mathbf{x})$$

using again information on the function values at a finite number of points:

$$y_i = f(\mathbf{x}_i), \;\; i = 1, \ldots, m.$$

Example: Linear regression (e.g. via *Ordinary Least Squares*, OLS)

An application of curve fitting (usually in a multivariate setting) that you all know is linear regression. Let y denote a dependent variable and $\mathbf{x} = (x_1, \ldots, x_k)$ a vector of k regressors. Hence, $(y_i, \mathbf{x}_i) = (y_i, x_{1i}, \ldots, x_{ki})$ corresponds to observation i.

Then, for

$$B_j(\mathbf{x}) = x_j,\ j = 1, \ldots, k,$$

and adding an error term, the problem above becomes

$$y_i = a_0 + \sum_{j=1}^{k} a_j B_j(\mathbf{x}_i) + \epsilon_i = a_0 + a_1 x_{1i} + \ldots + a_k x_{ki} + \epsilon_i,\ i = 1, \ldots, m$$

This is the simplest case of the well-known linear regression model. In this case with k regressors, we need to find $k + 1$ coefficients. See below for details on how to implement a regression algorithm. Note that this model can be easily extended to specifications which are nonlinear in the regressors (e.g. x_{1i}^2 or $x_{1i}x_{2i}$), but still linear *in the parameters*.

Preview

Broadly speaking, approximating a function requires the modeller to make three decisions:

1. **choose number and spacing of grid points**; for example, equally spaced points, Chebyshev nodes
2. **choose basis functions**; for example, monomials, orthogonal polynomials, splines
3. **choose whether to find the basis coefficients through interpolation or regression (least squares)**

For ease of exposition, we will go through this list in reverse order, starting with the third point.

Interpolation vs. Regression

Interpolation

A function $\hat{f}(x)$ *interpolates* to the data $\{x_i, y_i\}_{i=1}^m$ if

$$\hat{f}(x_i) = y_i = f(x_i),\quad i = 1, \ldots, m$$

where $x_1, x_2, \ldots, x_m$ are called *interpolation nodes* or *interpolation grid*.

In other words, the functions f and $\hat{f}$ have the same function values at the m data points. Note that this results in a system of m equations, the *interpolating/interpolation conditions*. With

$$\hat{f}(x) = \sum_{j=0}^{n} a_j B_j(x),$$

the conditions read

$$\begin{aligned} y_1 &= \sum_{j=0}^{n} a_j B_j(x_1) \\ y_2 &= \sum_{j=0}^{n} a_j B_j(x_2) \\ &\vdots \\ y_m &= \sum_{j=0}^{n} a_j B_j(x_m) \end{aligned}$$

Hence, we have $n + 1$ unknowns - the coefficients $a_0, \ldots, a_n$ - in m equations. In order to get a unique solutions, approximating a function via interpolation requires that

$$m = n + 1.$$

In other words, we need as many data points as coefficients.

We can also write this system of linear equations using matrices:

$$\begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} B_0(x_1) & B_1(x_1) & \cdots & B_n(x_1) \\ \vdots & \vdots & \vdots & \\ B_0(x_m) & B_1(x_m) & \cdots & B_n(x_m) \end{bmatrix} \begin{bmatrix} a_0 \\ \vdots \\ a_n \end{bmatrix} = \Phi \begin{bmatrix} a_0 \\ \vdots \\ a_n \end{bmatrix}$$

Φ is called the *interpolation matrix*, with $\Phi_{ij} = B_j(x_i)$. It is a square matrix, since $m = n + 1$. An interpolation "scheme" is well-defined if the interpolation nodes and basis functions are chosen such that Φ is nonsingular. As outlined below, we should also make sure that it is not ill-conditioned.

Least Squares Regression

The idea of approximating a function f via regression is that instead of requiring the approximant $\hat{f}$ to have the same function values at a given set of points, we minimize the *approximation error*, i.e. the sum of the squares of the differences between f and $\hat{f}$ at these points:

$$\min R = \sum_{i=1}^{m} \left[\hat{f}(x_i) - y_i \right]^2.$$

Intuitively, the smaller the residual R, the closer the approximant $\hat{f}(x)$ fits the data. With

$$\hat{f}(x) = \sum_{j=0}^{n} a_j B_j(x),$$

we can write the problem as

$$\min_{a_0, \ldots, a_n} \sum_{i=1}^{m} \left[\sum_{j=0}^{n} a_j B_j(x_i) - y_i \right]^2.$$

Be careful not to get confused by the two sums: the outer sum is over all points, the inner sum over all basis functions evaluated at a given point!

Note that in the special case that $m = n + 1$, the *least squares fit* given by the minimizing coefficients $a_0, \ldots, a_n$ is the same as the coefficients given by interpolation: the residual is zero, its smallest possible value. Least squares regression works also with fewer coefficients:

$$m \geq n + 1.$$

In other words, we need at least as many data points as degrees of approximation.

Choice of Basis Functions: Polynomials

One of the most frequently used families for basis functions are *polynomials*. Recall that a polynomial $p(x)$ of degree (order) n is defined as

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n,$$

where $a_0, \ldots, a_n$ are constants.

Approximating a function with a polynomial is an example of a "spectral method", which uses basis functions that are nonzero over the entire domain of the function that is approximated (except at a *finite* number of points).

Note that this particular way of writing a polynomial is called its *power series form*. As we will see later, there are other ways of writing a polynomial. The power functions $1, x, x^2, \ldots$ are also referred to as *monomials*. We can use them as a basis for polynomial interpolation, with $B_j(x) = x^j$, which is also called a *monomial basis*. The interpolation matrix defined above then reads:

$$\Phi = \begin{bmatrix} 1 & x_1 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_m & \cdots & x_m^n \end{bmatrix}$$

Recall from a previous lecture that this is also referred to as the *Vandermonde* matrix.

As an aside, why are polynomials a common choice as a basis? The answer is the *Weierstrass theorem*: If $f \in C[b, c]$, then for all $\epsilon > 0$, there exists a polynomial $p(x)$ such that

$$\forall x \in [b, c], |f(x) - p(x)| \leq \epsilon$$

In other words, we can approximate any continuous function with a desired approximation error ϵ over the interval $[b, c]$ with a polynomial of a high enough degree. Of course, the Weierstrass theorem is not of that much practical use since it doesn't tell us what the necessary degree of the approximating polynomial is.

Example: Polynomial Interpolation

The following example will illustrate *polynomial interpolation* using monomials as basis functions. Assume we want to approximate the function $f(x) = \exp(x)$ using a polynomial of degree 2. Hence, in the case of interpolation, we need $m = 3$ interpolation nodes.

In [3]:
```
## compare classroom notes
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
```

We can verify this result numerically. Start by defining the function:

In [4]:
```
def fun(x, alpha = 1):
    """
    Exponential function
    """
    y = np.exp(alpha * x)

    return y
```

We can use Numpy's **linspace** function to construct the grid:

In [5]:
```
g_min, g_max = 0, 2
```

```
x_grid = np.linspace(g_min, g_max, 3)
y_grid = fun(x_grid)
```

In order to interpolate the function, we can use the Numpy function **np.polynomial.polynomial.polyvander** which constructs the interpolation matrix for a monomial basis. Then, we solve a system of linear equations to obtain the interpolation coefficients:

In [6]:
```
A = np.polynomial.polynomial.polyvander(x_grid, 2)
print(A)
```

```
[[ 1.  0.  0.]
 [ 1.  1.  1.]
 [ 1.  2.  4.]]
```

In [7]:
```
print( np.linalg.solve(A, y_grid) )
```

```
[ 1.          0.24203561  1.47624622]
```

Unsurprisingly, there is a function in Numpy which combines these steps: **np.polynomial.polynomial.polyfit**

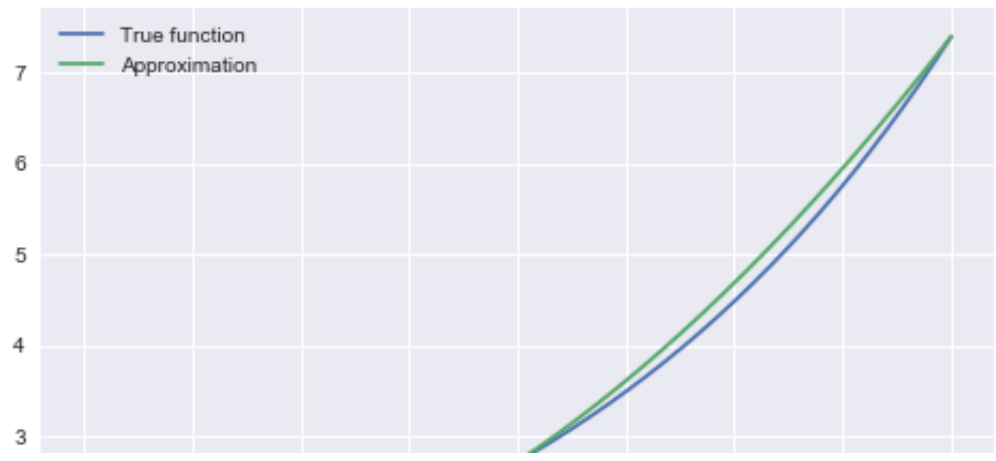
In [8]:
```
a_mono = np.polynomial.polynomial.polyfit(x_grid, y_grid, 2)
print(a_mono)
```

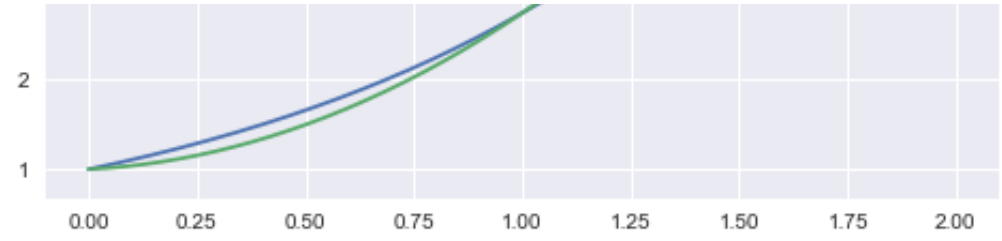
```
[ 1.          0.24203561  1.47624622]
```

We can illustrate this approximation. First, we define a dense array **x** consisting of equally spaced points in the interval. We then approximate the function values using the **np.polynomial.polynomial.polyval**, with **x** and **a_mono** as the arguments.

In [9]:
```
x = np.linspace(g_min, g_max, 1001)
y = fun(x)
y_mono = np.polynomial.polynomial.polyval(x, a_mono)

fig, ax = plt.subplots()
ax.plot(x, y, label = 'True function')
ax.plot(x, y_mono, label = 'Approximation')
ax.legend()
```

Out[9]:
```
<matplotlib.legend.Legend at 0xb5434b4320>
```





Before moving on, it should be noted when the interpolation nodes $\{x_i\}_{i=1}^m$ are distinct, there is a *unique* polynomial $p_n(x)$ of degree n that interpolates to the data $\{y_i, x_i\}_{i=1}^m$. This is intuitive, given that we solve a system of linear equations with a matrix - the Vandermonde matrix - whose column vectors are linearly independent. This property also implies that if f is itself a polynomial of degree n, approximating it with a polynomial of degree n will result in the approximant being the function itself, at every point in the domain: $\hat{f}(x) = f(x)$.

Chebyshev Basis Functions

Recall that in the context of solving systems of linear equations, we saw that the Vandermonde matrix is ill-conditioned for higher orders of n. Here's the example again:

In [10]:
```
for m in [5, 15]:
    ## define matrix
    x = np.linspace(1,5,m)
    A = np.polynomial.polynomial.polyvander(x, m-1)
    ##
    b = A @ np.ones(m)
    ## solve SLE
    x = np.linalg.solve(A, b)

    print("For n = {}, x = {}".format(m, x))
    print("For n = {}, the condition number is {}".format(m, np.lin
alg.cond(A)))
    print("----------------------------------------------------------
------------")
```

```
For n = 5, x = [ 1.  1.  1.  1.  1.]
For n = 5, the condition number is 26169.68797063433
------------------------------------------------------------------
---
For n = 15, x = [ 0.94442222  1.25794398  0.51237487  1.43634172
0.92681916  0.73381653
  1.34080974  0.77274174  1.0993391   0.96979552  1.00646588  0.9
9904084
  1.00009405  0.99999451  1.00000014]
For n = 15, the condition number is 1.9392318794936404e+18
------------------------------------------------------------------

---
```

This implies that computing basis coefficients can be hampered by serious rounding errors, in particular for a large number of interpolation nodes and hence a high degree of approximation.

Therefore, it is not recommended to use monomials as basis functions for function approximation. Instead, we usually rely on basis functions that are constructed using *orthogonal* polynomials, in particular Chebyshev polynomials.

Chebyshev Polynomials

Let n be the degree of approximation. Then for $j = 0, \ldots, n$, the **Chebyshev polynomials** $T_j(x)$ are defined as

$$T_j(x) = \cos(j\cos^{-1}(x)),$$

for $x \in [-1, 1]$. Note that this definition only works for the interval $[-1, 1]$ because this is the domain of the function $\cos^{-1} = \arccos$ (the "arccosine").

Their first derivatives are given by:

$$T_j'(x) = -\sin(j\cos^{-1}(x))\frac{-j}{\sqrt{1-x^2}} = \sin(j\cos^{-1}(x))\frac{j}{\sqrt{1-x^2}}.$$

An alternative way to define the Chebyshev polynomials is the following *recurrence relationship*:

$$T_{j+1}(x) = 2xT_j(x) - T_{j-1},$$

for $j \geq 1$, with $T_0(x) = 1$ and $T_1(x) = x$. Writing down the first few Chebyshev polynomials, we can see that the polynomial $T_n(x)$ has degree n and if n is even (odd) then $T_n(x)$ involves only even (odd) powers of x. For example,

$$T_4(x) = 8x^4 - 8x^2 + 1$$
$$T_5(x) = 16x^5 - 20x^3 + 5x$$

Note that when working with the Chebyshev polynomials numerically, we use the recursive formula, which is computationally more efficient than the direct definition, in particular for higher values of n and m. We will see an implementation in Python below.

A (univariate) polynomial $p_n(x)$ that is *represented by a Chebyshev polynomial basis* rather than a power series (monomial) basis is also referred to as a **Chebyshev series**:

$$p_n(x) = \sum_{j=0}^{n} a_j T_j(x)$$

This is a polynomial of degree n. For example, for $n = 2$, we can write it as a power

series:

$$p_n(x) = a_0T_0(x) + a_1T_1(x) + a_2T_2(x) = a_0 + a_1x + a_2(2x^2 - 1)$$
$$= (a_0 - a_2) + a_1x + (2a_2)x^2$$

The family of Chebyshev polynomials $\{T_j(x)\}$ is (mutually) *orthogonal*, which implies that

$$\int_{-1}^{1} w(x)T_i(x)T_j(x) = 0 \quad \text{for } i \neq j$$

with the *weighting function* w given by $w(x) = (1 - x^2)^{-1/2}$. Why is this a useful property in the context of function approximation? Without going into too much technical detail, we can get some intuition by plotting the Chebyshev polynomials and comparing them to the monomials, here between 0 and 1 (see below).

In [11]:
```
## graphs here
```

Note that the graphs for the Chebyshev polynomials are somewhat distinct, while the ones for the monomials are very similar to each other. Intuitively, when we can barely see a difference between two functions graphically, it is likely that the computer will have difficulty discerning the difference numerically, which contributes to the Vandermonde matrix being ill-conditioned.

In contrast, the family of Chebyshev polynomials provides a good "coverage" of functions defined on $[-1, 1]$. Hence, the shape of a polynomial is usually better characterized by the coefficients when it is written in a Chebyshev series than by the corresponding coefficients when written as a power series.

We can also illustrate this by comparing the conditioning numbers for the Vandermonde and the Chebyshev interpolation matrix for a grid between -1 and 1. The latter can be computed by the Scipy function **np.polynomial.chebyshev.chebvander**. Note that this matrix is sometimes called a *Pseudo-Vandermonde* matrix.

We can see that the Vandermonde matrix has a distinctly higher conditioning number than the Pseudo-Vandermonde matrix. As we will discuss in the next section, however, we can still decrease this number considerably.

In [12]:
```
m = 15
x = np.linspace(-1,1,m)
V = np.polynomial.polynomial.polyvander(x, m-1)
print( np.linalg.cond(V) )
```

```
1104808.52936
```

In [13]:
```
x = np.linspace(-1,1,m)
V = np.polynomial.chebyshev.chebvander(x, m-1)
print( np.linalg.cond(V) )
```

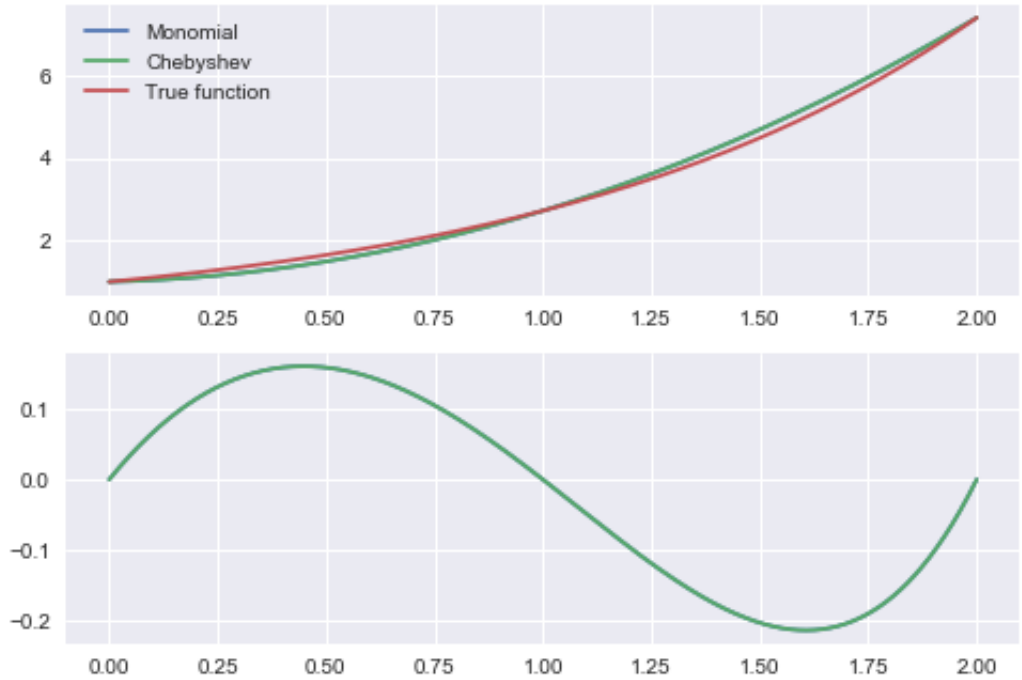
```
225.73172355
```

Finally, we can run our simple example above, this time using a Chebyshev rather than a monomial basis. In this case, for a low degree of approximation, both choices give equivalent results.

In [14]:
```
m = 3
g_min, g_max = 0, 2

x = np.linspace(g_min, g_max, 1001)
y = fun(x)

a_cheb = np.polynomial.chebyshev.chebfit(x_grid, y_grid, m-1)
y_cheb = np.polynomial.chebyshev.chebval(x, a_cheb)
```

In [15]:
```
fig, ax = plt.subplots(2, 1)
ax[0].plot(x, y_mono, label = 'Monomial')
ax[0].plot(x, y_cheb, label = 'Chebyshev')
ax[0].plot(x, y, label = 'True function')
ax[0].legend()
ax[1].plot(x, y - y_mono)
ax[1].plot(x, y - y_cheb)
```

Out[15]: [<matplotlib.lines.Line2D at 0xb54addd940>]



Constructing a Grid

Many approximation problems outside of Econometrics (where the data points/observations are given by the "real world") allow the modeler to choose the interpolation grid $x_1, x_2, \ldots, x_m$. In the simple example above, we used three equally spaced points between 0 and 2 (i.e. 0,1,2). The two features that characterize such a

grid are the number of grid points and their spacing. Both have an impact on the quality of the approximation. For the number of grid points, this is intuitive: the more data you have, the more information about the function is available. However, note that at least in the case of interpolation, the number of basis functions and coefficients increases with the number of grid points, and hence more points mean a larger system of linear equations to solve.

Why the spacing of grid points matters is less obvious.

Error in Polynomial Interpolation

Let $p_n(x)$ be a polynomial of degree n interpolating to the data $\{x_i, y_i\}_{i=1}^{m}$, with $m = n + 1$. Define $\omega_{n+1}(x) = (x - x_1)(x - x_2) \cdot \ldots \cdot (x - x_{n+1})$ and suppose that $x, x_i \in [b, c]$.

It can be shown that

$$\max_{x \in [b,c]} \left| f(x) - p_n(x) \right| \leq \max_{x \in [b,c]} \left| \omega_{n+1}(x) \right| \frac{\max_{z \in [b,c]} \left| f^{(n+1)}(z) \right|}{(n+1)!}$$

where $f^{(n+1)}$ is the $(n + 1)$st derivative of f.

The left-hand side of this expression is the *interpolation error* when approximating f with a polynomial. Hence, the right-hand side provides an *upper bound* on this error. Note that if $x = x_i$, we have $\omega_{n+1}(x) = 0$ and hence the interpolation is zero, which is intuitive. The same is true when the function f is a polynomial of degree n, since then $f^{(n+1)} = 0$.

We are interested in keeping the interpolation error as small as possible. Note that among the two terms on the right hand side, the term

$$\frac{\max_{z \in [b,c]} \left| f^{(n+1)}(z) \right|}{(n+1)!}$$

is determined by the function to approximate, and hence we cannot do anything to minimize it. However, the term

$$\max_{x \in [b,c]} \left| \omega_{n+1}(x) \right| = \max_{x \in [b,c]} \left| (x - x_1)(x - x_2) \cdot \ldots \cdot (x - x_{n+1}) \right|$$

does not depend on the function, but instead depends on our choice of the grid points $x_1, x_2, \ldots, x_n$. It turns out that the so-called *Chebyshev nodes* are a good choice to keep this term small.

Chebyshev Nodes

Let m be the number of grid points, indexed by i. The Chebyshev nodes between -1 and 1 are given by

$$z_i = \cos\left(-\frac{(2i-1)\pi}{2m}\right), \quad i = 1, \ldots, m$$

The following function implements this expression in Python.

In [16]:
```
def chebnodes(m):
    """
    Computes m Chebyshev nodes between -1 and 1.
    """
    i = np.array(list(range(1, m+1)))
    return -np.cos(0.5 * np.pi * (2 * i - 1) / m)
```

It can be shown that the Chebyshev nodes are the roots of the Chebyshev polynomial $T_m(x)$. For example, let $m = 2$. Recall that $T_2(x) = 2x^2 - 1$, which has roots at $\sqrt{0.5}$ and $-\sqrt{0.5}$. Computing the Chebyshev notes for $m = 2$ gives:

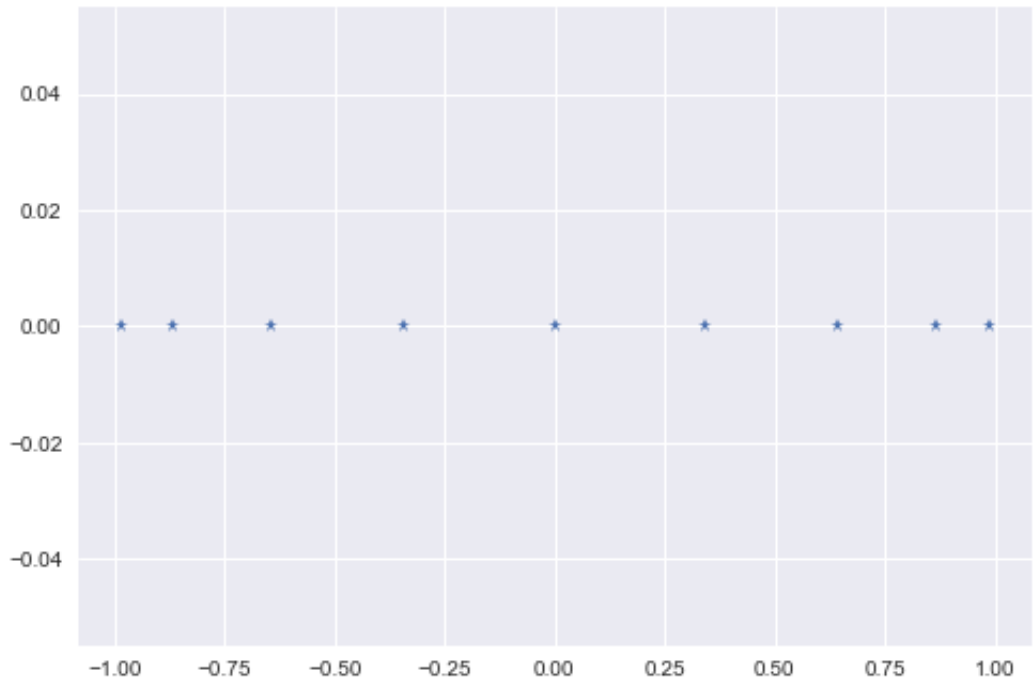
In [17]:
```
chebnodes(2)
```

Out[17]:
```
array([-0.70710678,  0.70710678])
```

Plotting the Chebyshev nodes between -1 and 1 shows that they are not equally spaced, but more "bunched up" towards the end points of this interval. Note also that these end points are *not* included in the Chebyshev nodes.

In [18]:
```
fig, ax = plt.subplots()
ax.plot(chebnodes(9), np.zeros(9), '*')
```

Out[18]:
```
[<matplotlib.lines.Line2D at 0xb54acaefd0>]
```

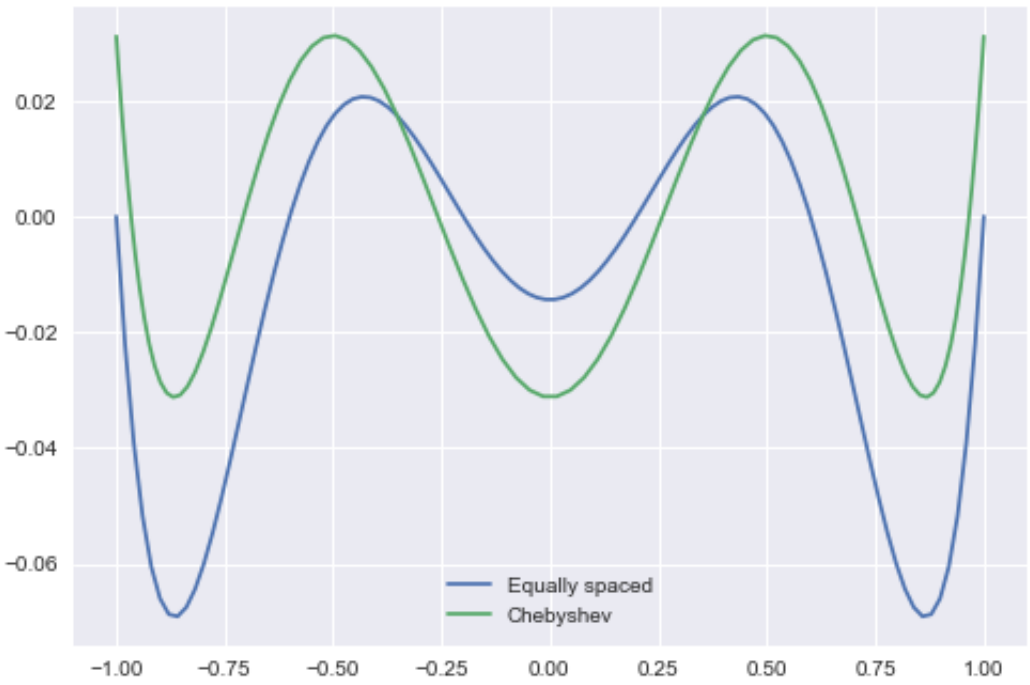


The following piece of code illustrates the difference between Chebyshev nodes and equally spaced grid points when evaluating the function $\omega_{n+1}(x)$. For five grid points, the maximum value of $\left|\omega_{n+1}(x)\right|$ (which is what matters for the error bound) is cut by

more than half when using Chebyshev polynomials. It is easy to check that this difference is even higher when increasing the number of grid points.

In [19]:
```
def eval_omega(x, nodes):
    return np.prod([(x - node) for node in nodes], axis = 0)

## evaluate omega at equally spaced grid points and plot
n = 5
nodes = np.linspace(-1, 1, n+1)
x = np.linspace(-1, 1, 100)

fig, ax = plt.subplots()
ax.plot(x, eval_omega(x, nodes), label = 'Equally spaced')

## evaluate omega at Chebyshev nodes and plot
nodes = chebnodes(n+1)
x = chebnodes(100)

ax.plot(x, eval_omega(x, nodes), label = 'Chebyshev')
ax.legend()
```

Out[19]:
```
<matplotlib.legend.Legend at 0xb54bf2a9b0>
```



This result can also be formalized: for a continuously differentiable function f, the interpolation error when approximating it with a polynomial of order n on $n+1$ Chebyshev nodes on the interval $[b, c]$ is bounded by:

$$\max_{x \in [b,c]} \left| f(x) - p_n(x) \right| \leq \frac{6}{n}(c-b)[\log(n)/\pi + 1] \max_{z \in [b,c]} \left| f'(z) \right|$$

Importantly, this error goes to zero as n rises. Hence, we can achieve a desired degree of accuracy (i.e., make the interpolation error as small as possible) by increasing the degree of approximation and hence the number of Chebyshev nodes.

Note that while we have stated this property in terms of interpolation, using Chebyshev nodes also puts a bound on the approximation error when using least-

squares regression.

One could suspect that a crucial disadvantage of using Chebyshev nodes is that they are only defined between -1 and 1. However, with $\{z_i\}$ denoting the m Chebyshev nodes on $[-1, 1]$, we can easily define the corresponding Chebyshev nodes on $[b, c]$ as:

$$y_i = b + \frac{(z_i + 1)(c - b)}{2} \quad \text{for } i = 1, \ldots, m$$

This is implemented in the function **chebgrid** below. The reverse operation is executed by the function **chebconvert**, which is going to be useful below.

In [20]:
```
def chebgrid(b, c, m):
    """
    Computes num Chebyshev nodes on the interval [b, c].
    """
    z = chebnodes(m)
    return (c - b) * 0.5 * (z + 1) + b

def chebconvert(x, b, c):
    """
    Transforms nodes between [b, c] to the interval [-1,1].
    """
    return 2. * (x - b) / (c - b) - 1
```

Two more remarks on Chebyshev nodes:

- It may not only be possible to evaluate the function f at the Chebyshev nodes and hence obtain the data $y_i = f(z_i)$; if this is the case, one should still choose the grid points such that they are denser close to the end points of the interpolation interval.
- *Extrapolation*, i.e. approximating a function *outside* of $[b, c]$ using the approximant $\hat{f}$ found by interpolation at the Chebyshev nodes can be disastrous (even more so than when using equally spaced points). In general, extrapolation is never a good idea in the context of function approximation.

Combining Chebyshev Nodes with a Chebyshev Basis

We can summarize the main takeaways from the last two sections:

- Choose Chebyshev nodes (when possible) as the approximation grid, in order to minimize the approximation error
- Use a Chebyshev basis (i.e., Chebyshev polynomials as the basis functions)

in order to make the interpolation matrix well-conditioned

With respect to the last point, it turns out that we can do even better when evaluating a Chebyshev basis at the Chebyshev nodes. Formally, we look at

$$\Phi_c = \begin{bmatrix} T_0(z_1) & T_1(z_1) & \cdots & T_n(z_1) \\ \vdots & \vdots & \vdots & \\ T_0(z_m) & T_1(z_m) & \cdots & T_n(z_m) \end{bmatrix}$$

where $m = n + 1$ and z_i are the Chebyshev nodes defined above. Also recall that for $j = 0, \ldots, n$, the Chebyshev polynomials $T_j(x)$ are defined as:

$$T_{j+1}(x) = 2xT_j(x) - T_{j-1},$$

for $j \geq 1$, with $T_0(x) = 1$ and $T_1(x) = x$. In the following, I will refer to Φ_c as the "Chebyshev matrix".

The Python function **chebmatrix** implements the Chebyshev matrix, i.e. evaluates the Chebyshev polynomials up to degree n at m Chebyshev nodes, with $m \geq n + 1$. For this, the user need to provide the arguments **deg** (the degree of approximation) and **m**. The function returns an **m**-by-**deg+1** Numpy array.

In addition, **chebmatrix** also evaluates the Chebyshev polynomials at a scalar or array **x**. In this case, the function still returns an **m**-by-**deg+1**, where **m** is the length of the input **x**.

In [21]:
```
def chebmatrix(deg, m = None, x = None):
    """
    Computes the m-by-(deg+1) matrix with Chebyshev basis functions
of degree deg for m Chebyshev nodes.
    """
    ## check if a second argument is provided
    assert (m != None or np.sum(x) != None), "Please provide the nu
mber of grid points or an input vector/scalar x!"

    ## check if x values are provided
    if x is None: # default: Chebyshev nodes between -1 and 1 (for
interpolation/regression)
        z = chebnodes(m)
    elif isinstance(x, (list, tuple, np.ndarray)):    # arbitrary v
ector (for approximation)
        z, m  = x, len(x)
    else:    # arbitrary scalar (for approximation)
        z, m = x, 1

    ## define numpy array and fill second column
    T = np.ones((m, deg + 1))
    T[:,1] = z

    ## loop over columns in T; each column corresponds to the Cheby
shev basis functions for deg col_idx
    for col_idx in range(1, deg):
```

```
        T[:,col_idx+1] = 2 * z * T[:,col_idx] - T[:,col_idx - 1]
    return T
```

In [22]:
```
## Examples for different ways to use chebmatrix function
print(chebmatrix(4, 2))
print(chebmatrix(4, x = 0.1))
print(chebmatrix(4, x = np.array([0.1, 0.2])))
# print(chebmatrix(4)) -> only one argument, throws an error!
```

```
[[  1.00000000e+00  -7.07106781e-01   2.22044605e-16   7.07106781
e-01
   -1.00000000e+00]
 [  1.00000000e+00   7.07106781e-01  -2.22044605e-16  -7.07106781
e-01
   -1.00000000e+00]]
[[ 1.      0.1    -0.98   -0.296   0.9208]]
[[ 1.      0.1    -0.98   -0.296   0.9208]
 [ 1.      0.2    -0.92   -0.568   0.6928]]
```

As seen above, Numpy's **polynomial.chebyshev.chebvander** gives the same matrix. Using **chebmatrix** provides a short-cut, in particular for the default Chebyshev matrix case.

In [23]:
```
## Use Numpy's polynomial.chebyshev package
print( np.polynomial.chebyshev.chebvander(chebnodes(2), 4) )
print( np.polynomial.chebyshev.chebvander(np.array([0.1, 0.2]), 4)
)
```

```
[[  1.00000000e+00  -7.07106781e-01   2.22044605e-16   7.07106781
e-01
   -1.00000000e+00]
 [  1.00000000e+00   7.07106781e-01  -2.22044605e-16  -7.07106781
e-01
   -1.00000000e+00]]
[[ 1.      0.1    -0.98   -0.296   0.9208]
 [ 1.      0.2    -0.92   -0.568   0.6928]]
```

To see the advantage of using the Chebyshev matrix, compare its condition number to a matrix with a Chebyshev basis evaluated at an equally spaced grid (as above).

In [24]:
```
m = 15
x = np.linspace(-1,1,m)
Phi = np.polynomial.chebyshev.chebvander(x, m-1)
print( np.linalg.cond(Phi) )
```

```
225.73172355
```

In [25]:
```
Phi_c = chebmatrix(m-1, m)
print( np.linalg.cond(Phi_c) )
```

```
1.41421356237
```

In the latter case, we get a condition number of $\sqrt{2}$. Importantly, this condition number

is not only very near the minimum of 1, but it is also independent from the degree of approximation and number of Chebyshev nodes. This implies that combining a Chebyshev basis with Chebyshev nodes gives a very well-conditioned interpolation matrix, and hence the interpolation equation $\Phi_c a = y$ can be solved accurately and efficiently.

More generally, note that the Chebyshev matrix with $m \geq n + 1$ is *orthogonal*: if the Chebyshev polynomials are evaluated at m Chebyshev nodes, the following holds:

$$\sum_{i=1}^{m} T_j(z_i) T_k(z_i) = \begin{cases} 0 \text{ if } j \neq k \\ m \text{ if } j = k = 0 \\ \frac{m}{2} \text{ if } j = k \neq 0 \end{cases}$$

We can confirm this numerically:

In [26]:
```
m = 9
T = chebmatrix(5, m)
print(np.sum(T[:,2] * T[:,3]), np.sum(T[:,5] * T[:,4]), np.su
m(T[:,0] * T[:,2]))
m = 12
T = chebmatrix(5, m)
print(np.sum(T[:,0] * T[:,0]), np.sum(T[:,5] * T[:,5]), np.su
m(T[:,4] * T[:,4]))
```

```
-9.99200722163e-16 2.94209101526e-15 -8.881784197e-16
12.0 6.0 6.0
```

This implies that the $(n + 1) \times (n + 1)$ matrix $\Phi_c^T \Phi_c$ is a diagonal matrix, with the diagonal elements being equal to either $m/2$ or m. Again, we can verify this numerically:

In [27]:
```
m = 6
n = 2
Phi_c = chebmatrix(n, m)
print( Phi c.T @ Phi c)
```

```
[[  6.00000000e+00  -4.44089210e-16  -3.33066907e-16]
 [ -4.44089210e-16   3.00000000e+00  -7.07007168e-16]
 [ -3.33066907e-16  -7.07007168e-16   3.00000000e+00]]
```

Why is this useful? Recall that when approximating a function f using least-squares regression, we face the problem:

$$\min_{a_0, \ldots, a_n} \sum_{i=1}^{m} \left[\sum_{j=0}^{n} a_j B_j(x_i) - y_i \right]^2.$$

For ease of exposition, denote

$$\hat{y}_i = \sum_{j=0}^{n} a_j B_j(x_i).$$

Taking first-order conditions gives:

$$2\sum_{i=1}^{m} [\hat{y}_i - y_i] B_j(x_i) = 0, \; j = 0, \ldots, n$$

Rearranging this expression, we get the following system of equation for $j = 0, \ldots, n$:

$$\sum_{i=1}^{m} y_i B_j(x_i) = \sum_{i=1}^{m} B_j(x_i) \left[\sum_{k=0}^{n} a_k B_k(x_i) \right] = \sum_{i=1}^{m} \sum_{k=0}^{n} a_k B_j(x_i) B_k(x_i)$$

$$= \sum_{k=0}^{n} a_k \sum_{i=1}^{m} B_j(x_i) B_k(x_i)$$

Now, suppose we use Chebyshev basis functions, evaluated at the Chebyshev polynomials:

$$\sum_{i=1}^{m} y_i T_j(z_i) = \sum_{k=0}^{n} a_k \sum_{i=1}^{m} T_j(z_i) T_k(z_i)$$

Recall from above that

$$\sum_{i=1}^{m} T_j(z_i) T_k(z_i) = \begin{cases} 0 & \text{if } j \neq k \\ m & \text{if } j = k = 0 \\ \frac{m}{2} & \text{if } j = k \neq 0 \end{cases}$$

Hence, in the sum over k on the right hand side, all elements are zero except for $k = j$:

$$\sum_{i=1}^{m} y_i T_j(z_i) = a_j \sum_{i=1}^{m} T_j(z_i) T_j(z_i)$$

This easy to solve for a_j:

$$a_j = \frac{\sum_{i=1}^{m} y_i T_j(z_i)}{\sum_{i=1}^{m} T_j(z_i) T_j(z_i)} = \begin{cases} \frac{\sum_{i=1}^{m} y_i T_j(z_i)}{m} & \text{if } j = k = 0 \\ 2\frac{\sum_{i=1}^{m} y_i T_j(z_i)}{m} & \text{if } j = k \neq 0 \end{cases}, \quad j = 0, \ldots, n$$

In other words, when using a Chebyshev basis evaluated at the Chebyshev nodes, we can solve the minimization problem underlying least-squares regression in a straightforward fashion.

We can express the same idea using matrix notation. In this case, the regression problem can be written as (verify this!):

$$\min_{a} \left| \left| \ \Phi a - y\ \right| \right|^2 = (\Phi a - y)^T (\Phi a - y)$$

and recall that the general Φ has been defined as:

$$\Phi = \left[\begin{array}{cccc} B_0(x_1) & B_1(x_1) & \cdots & B_n(x_1) \\ \vdots & \vdots & \vdots \\ B_0(x_m) & B_1(x_m) & \cdots & B_n(x_m) & \end{array} \right]$$

It can be shown (and you should remember from econometrics) that solving the minimization problem gives the following system of linear equations:

$$(\Phi^T \Phi) a = \Phi^T y$$

When using the Chebyshev matrix Φ^T, $\Phi^T \Phi$ is diagonal, and hence we know that we can solve this system very easily. This last expression is equivalent to the expression for a_j above.

Summary: A Chebyshev Approximation Algorithm

Based on the insights above, we can state the following algorithm for approximating a function f for $x \in [a, b]$ using a polynomial of degree n and $m \ge n + 1$ grid points:

(i) Compute the $m \ge n + 1$ Chebyshev nodes on $[-1, 1]$:

$$z_i = \cos\left(- \frac{(2i - 1)\pi}{2m}\right), \quad i = 1, ..., m$$

(ii) Translate the nodes to the $[a, b]$ interval:

$$x_i = a + \frac{(z_i + 1) (b-a)}{2}, \quad i = 1,...,m$$

(iii) Evaluate f at the (translated) nodes:

$$y_i = f(x_i), \quad i = 1,...,m$$

(iv) Compute the Chebyshev coefficients a_i, $i = 0, ..., n$:

- when $m = n + 1$, solve the system of linear equations

$$\begin{split} \left[\begin{array}{c} y_{1} \\ \vdots \\ y_{m} \end{array} \right] = \left[\begin{array}{cccc} T_0(z_1) & T_1(z_1) & \cdots & T_n(z_1) \\ \vdots & \vdots & \vdots \\ T_0(z_m) & T_1(z_m) & \cdots & T_n(z_m) & \end{array} \right] \left[\begin{array}{c} a_{0} \\ \vdots \\ a_{n} \end{array} \right] = \Phi \left[\begin{array}{c} a_{0} \\ \vdots \\ a_{n} \end{array} \right] \end{split}$$

- when $m > n + 1$, solve the system of linear equations

$$(\Phi^T \Phi) a = \Phi^T y$$

or, equivalently, compute

$$\begin{equation} a_j = \frac{\sum_{i = 1}^m y_i T_j(z_i)}{\sum_{i = 1}^m T_j(z_i) T_j(z_i)} = {\begin{cases}\frac{\sum_{i = 1}^m y_i T_j(z_i)}{m} \quad \text{if}\ j=k=0\\ 2\frac{\sum_{i = 1}^m y_i T_j(z_i)}{m} \quad \text{if}\ j=k\neq 0\end{cases}}, \quad j = 0, ..., n \end{equation}$$

(v) Find the approximant $\hat{f}(x)$ by evaluating

$$\begin{equation} \hat{f}(x) = \sum_{j = 0}^{n} a_j T_j\left(2 \frac{x - a}{b - a} - 1 \right) \end{equation}$$

Note that in the last expression, the term in the parenthesis comes from translating a value x in the interval $[a, b]$ to a value in the interval $[-1, 1]$, as done by the function **chebconvert** defined above.

The following function **chebapprox** implements this algorithm: it uses **chebmatrix** to compute the Chebyshev basis functions (by default at the Chebyshev nodes; it also allows using a more general grid, but we are not going to consider this case) and then solves the resulting system of linear equations using either interpolation (if the approximation degree corresponds to the length of the data vector **y**) or regression.

In [28]:
```
def chebapprox(y, deg, v = None):
    """
    Function to compute the Chebyshev coefficients using interpolat
ion or regression
    """
    m = len(y)
    if v == None:
        T = chebmatrix(deg, m)
    else:
        z = convert(v[0], v[1], v[2])
        T = chebmatrix(deg, x = z)


    if deg == m-1: # interpolation (default)
        coef = np.linalg.solve(T,y)
    else:
        coef = np.ones(deg + 1)
        for idx_deg in range(deg + 1):
            coef[idx_deg] = sum(y * T[:,idx_deg]) / sum(T[:,idx_de
g]**2)

    return coef
```

Again, there is a Numpy function, **np.polynomial.chebyshev.chebfit**, that does the same (but not with the default Chebyshev nodes). We will see its use below.

Before moving to examples for using this algorithms, let's consider Boyd's Moral Principle (via J. Fernandez-Villaverde):

1. When in doubt, use Chebyshev polynomials (unless the solution is spatially periodic, in which case an ordinary Fourier series is better).
2. Unless you are sure that another set of basis functions is better, use

Chebyshev polynomials.
3. Unless you are really, really sure that another set of basis functions is better, use Chebyshev polynomials.

Examples: Univariate Function Approximation in Python

Example 1: Function Approximation for known functions

Below I approximate the function $y = \exp(-\alpha x)$ between 0 and 2 (compare the Matlab example by Miranda and Fackler for a different interval).

In [29]:
```
def fun(x, alpha = 2):
    """
    Exponential function
    """
    return np.exp(-alpha * x)
```

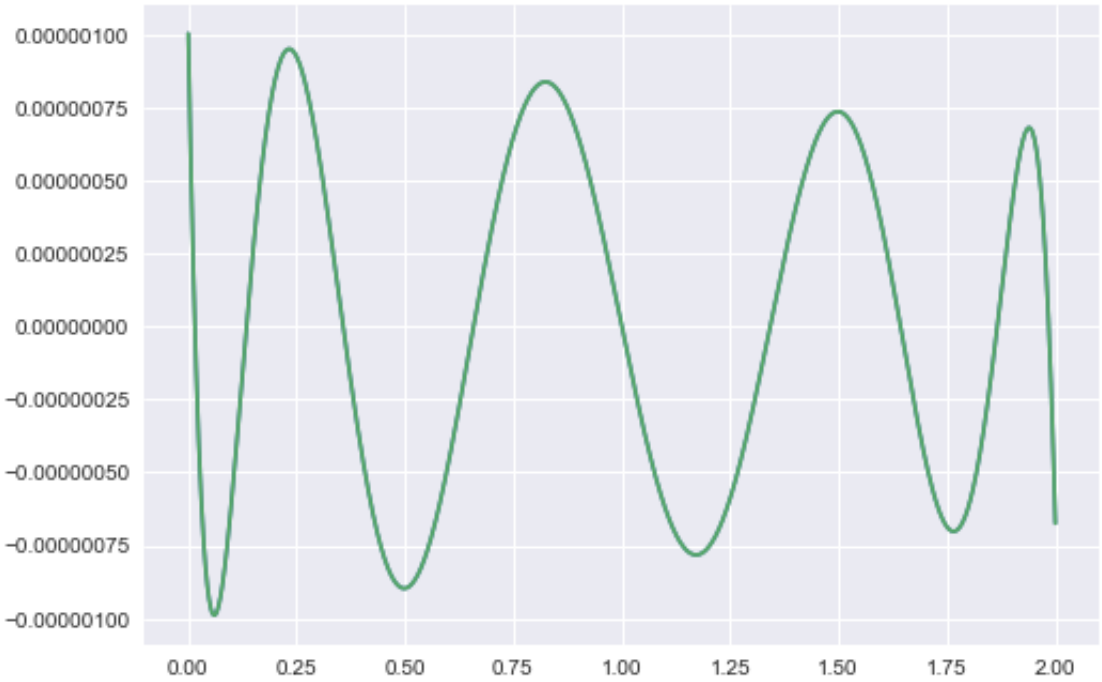
In [30]:
```
## min and max for approximation grid
g_min, g_max = 0, 2
## number of grid points
m = 9
## Chebyshev grid and function values
p_nodes = chebgrid(g_min, g_max, m)
y = fun(p_nodes)

a1 = chebapprox(y, m-1)
a2 = np.polynomial.chebyshev.chebfit(chebconvert(p_nodes, g_min,
g_max), y, m-1)

print(a1)
print(a2)
```

```
[  3.08508323e-01  -4.30538578e-01   1.86478067e-01  -5.75824453
e-02
   1.37307308e-02  -2.65952214e-03   4.33119221e-04  -6.07958356
e-05
   7.41574370e-06]
[  3.08508323e-01  -4.30538578e-01   1.86478067e-01  -5.75824453
e-02
   1.37307308e-02  -2.65952214e-03   4.33119221e-04  -6.07958356
e-05
   7.41574370e-06]
```

In [31]:
```
## interpolate over wide grid and plot
x = np.linspace(g_min, g_max, 1001)
y_approx2 = np.polynomial.chebyshev.chebval(chebconvert(x, g_min,
g_max), a2)
```

```
y_approx1 = chebmatrix(m-1, x = chebconvert(x, g_min, g_max)) @ a1

## plot approximation errors
fig, ax = plt.subplots()
ax.plot(x, fun(x) - y_approx2)
ax.plot(x, fun(x) - y_approx1)
```

Out[31]: [<matplotlib.lines.Line2D at 0xb54c816e80>]



Example 2: Approximating the Derivative

Suppose you have a function f approximated by a Chebyshev series:

$$f(x) \approx p_n(x) = \sum_{j = 0}^n a_j T_j(x)$$

Then, the derivative of f can be approximated by the derivative of p_n, which makes use of the derivatives of the Chebyshev basis functions:

$$f'(x) \approx p'_n(x) = \sum_{j = 0}^n a_j T'_j(x)$$

As an example, consider $n = 3$ and recall the Chebyshev basis functions from above:

$$p_3(x) = \sum_{j = 0}^3 a_j T_j(x) = a_0 + a_1 x + a_2 (2x^2 - 1) + a_3 (4x^3 - 3x)$$

Taking the first derivative w.r.t. x gives:

$$\frac{\partial p_3(x)}{\partial x} = a_1 + 4a_2x + 12a_3 x^2 - 3a_3 = (a_1 + 3a_3) + (4 a_2) x + (6a_3) (2x^2 - 1)$$

Using the last rearrangement, we can see that this can be expressed as a Chebyshev series with $n = 2$:

$$\frac{\partial p_3(x)}{\partial x} = (a_1 + 3a_3) T_0(x) + (4 a_2) T_1(x) + (6a_3) T_2(x)$$

In other words, for a Chebyshev series with parameters (a_0, a_1, a_2, a_3), the first derivative is a Chebyshev series with coefficients (a_1 + 3a_3, 4 a_2, 6a_3). An analogous argument can be made for any Chebyshev series of degree n: its first derivative is a Chebyshev series of degree n -1. The Numpy function **chebder** takes the n+1 coefficients (a_0, ..., a_n) as an input and computes the n coefficients to be used for the approximation of the first derivative:

In [32]:
```
print(np.polynomial.chebyshev.chebder([1,2,3,4]))
```

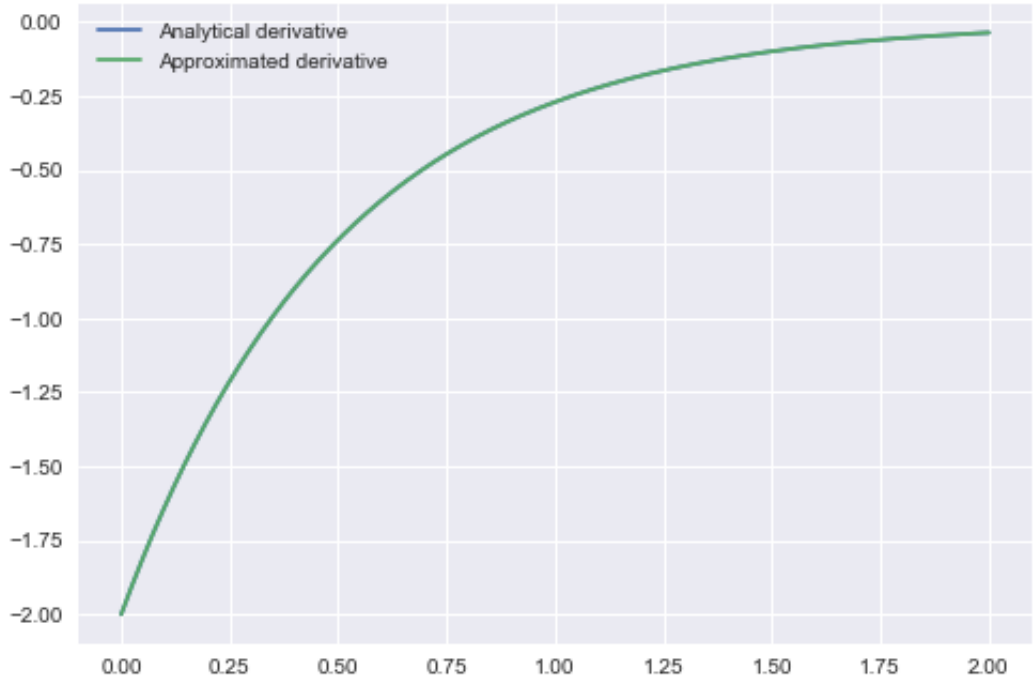
```
[ 14.  12.  24.]
```

In [33]:
```
def fun(x, alpha = 2):
    """
    Exponential function
    """
    y = np.exp(-alpha * x)

    return y

def fun_d(x, alpha = 2):
    """
    Derivatie of the exponential function
    """
    y = -alpha * np.exp(-alpha * x)

    return y
```

In [34]:
```
## min and max for approximation grid
g_min, g_max = 0, 2

## number of grid points
m = 9

## Chebyshev grid and function values
x_nodes = chebgrid(g_min, g_max, m)
y = fun(x_nodes)

## interpolation step
a = chebapprox(y, m - 1)

## Coefficients to approximate derivative
ad = np.polynomial.chebyshev.chebder(a)
```

In [35]:
```
## interpolate over wide grid
x = np.linspace(g_min, g_max, 1001)
yd_approx = np.polynomial.chebyshev.chebval(chebconvert(x, g_min,
g_max), ad)

## plot approximation errors
fig, ax = plt.subplots()
ax.plot(x, fun_d(x), label = 'Analytical derivative')
ax.plot(x, yd_approx, label = 'Approximated derivative')
ax.legend()
```

Out[35]:
```
<matplotlib.legend.Legend at 0xb54c8c6ef0>
```


Analytical derivative
Approximated derivative
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
−1.75
−2.00
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00