# Lecture 7

## Numerical Functional Equation Methods

Randall Romero Aguilar, PhD

This draft: October 10, 2018

Universidad de Costa Rica

SP6534 - Economía Computacional

# Table of contents

# Introduction

In many computational economics applications, we need to replace an analytically intractable function $f : \Re^n \mapsto \Re$ with a numerically tractable approximation $\hat{f}$.

- In some applications, $f$ can be evaluated at any point of its domain, but with difficulty, and we wish to replace it with an approximation $\hat{f}$ that is easier to work with.
- In other applications, $f$ is defined implicitly via a functional equation, but the equation lacks closed-form solution and we wish to compute an approximate solution $\hat{f}$.

- We first study interpolation, a general strategy for forming a tractable approximation to a function that can be evaluated at any point of its domain.
- Methods for solving functional equations are based on interpolation principles and are studied subsequently.

# Interpolation

- Consider a real-valued function $f$ defined on an interval of the real line that can be evaluated at any point of its domain.
- Generally, we will approximate $f$ using a function $\hat{f}$ that is a finite linear combination of $n$ known basis functions $\phi_1, \phi_2, \ldots, \phi_n$ of our choosing:

$$f(x) \approx \hat{f}(x) \equiv \sum_{j=1}^{n} c_j \phi_j(x).$$

- We will fix the $n$ basis coefficients $c_1, c_2, \ldots, c_n$ by requiring $\hat{f}$ to interpolate, that is, agree with $f$, at $n$ interpolation nodes $x_1, x_2, \ldots, x_n$ of our choosing.

- The most readily recognizable basis is the monomial basis

$$\phi_0(x) = 1$$
$$\phi_1(x) = x$$
$$\phi_2(x) = x^2$$
$$\vdots$$
$$\phi_n(x) = x^n,$$

which may be used to construct polynomial approximations:

$$f(x) \approx \hat{f}(x) \equiv c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n.$$

- As we will shortly see, however, other function bases may be used to approximate functions.
- And there are different ways to choose the interpolation nodes.

- Regardless of how the $n$ basis functions and nodes are chosen, computing the basis coefficients reduces to solving a linear equation:

$$\sum_{j=1}^{n} c_j \phi_j(x_i) = f(x_i), \qquad i = 1, 2, \ldots, n.$$

- The interpolation equation can be written in the matrix format

$$\Phi c = y$$

where, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$,

$$\Phi_{ij} = \phi_j(x_i) \quad \text{and} \quad y_i = f(x_i)$$

and $c$ is the $n \times 1$ vector of basis coefficients to be determined.

- In theory, an interpolation scheme is well-defined if the basis functions and interpolation nodes are chosen such that the interpolation matrix $\Phi$ is nonsingular.
- In practice, however, the interpolation matrix must meet the more stringent requirement that it not be ill-conditioned.
- Otherwise, it will not be possible to compute the basis coefficients accurately.

Ideally, an interpolation scheme should satisfy various conditions.

- It should be theoretically possible to achieve an arbitrarily accurate approximation by increasing the number of basis functions and interpolation nodes.
- It should be possible to solve the interpolation equation quickly and accurately.
- It should be relatively inexpensive to evaluate, differentiate, integrate or otherwise work with the approximation.

- Interpolation schemes differ only in how the basis functions $\phi_j$ and interpolation nodes $x_i$ are chosen.
- We develop interpolation schemes based on two classes of basis functions:
    - Orthogonal polynomials
    - Piecewise polynomial splines

# Polynomial Interpolation

# Weierstrass Theorem

- The Weierstrass Theorem asserts that any continuous real-valued function can be approximated to an arbitrary degree of accuracy over a bounded interval by a polynomial.
- Specifically, if $f$ is continuous on $[a, b]$ and $\epsilon > 0$, then there exists a polynomial $p$ such that

$$\max_{x \in [a,b]} |f(x) - p(x)| < \epsilon.$$

- The Weierstrass theorem motivates the use of polynomials to approximate continuous functions.
- The theorem, however, is not very practical.
- It gives no guidance on how to find a polynomial that provides a desired level of accuracy.
- It does not even tell us what degree polynomial is required.

# Naive Polynomial Interpolation

- One way to construct an $n^{th}$-degree polynomial approximation $\hat{f}$ to a function $f$ over a bounded interval $[a, b]$ is as follows.
- Write the approximation

$$\hat{f}(x) \equiv \sum_{j=0}^{n} c_j x^j$$

in terms of the monomial basis functions $1, x, x^2, \ldots, x^n$.
- Fix the $n + 1$ unknown basis coefficients $c_0, c_1, \ldots, c_n$ by requiring $\hat{f}$ to agree with $f$ at the $n + 1$ equally-spaced interpolation nodes $x_i = a + ih$, where $h = (b - a)/n$.

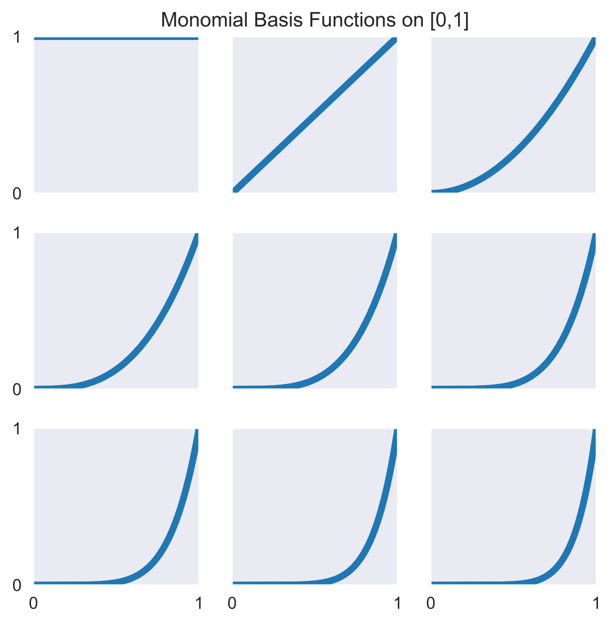



**Figure 1:** Monomial Basis Functions on $[0, 1]$

- This polynomial interpolation scheme, however, suffers from two serious, but distinct problems.
- First, the interpolation matrix is a Vandermonde matrix, which becomes increasingly ill-conditioned as the degree of the interpolating polynomial rises.
- Second, there are functions for which the approximation error explodes as the degree of the interpolating polynomial rises.
- The classic example is Runge's function:

$$f(x) = \frac{1}{1+25x^2}, \qquad -1 \leq x \leq 1.$$

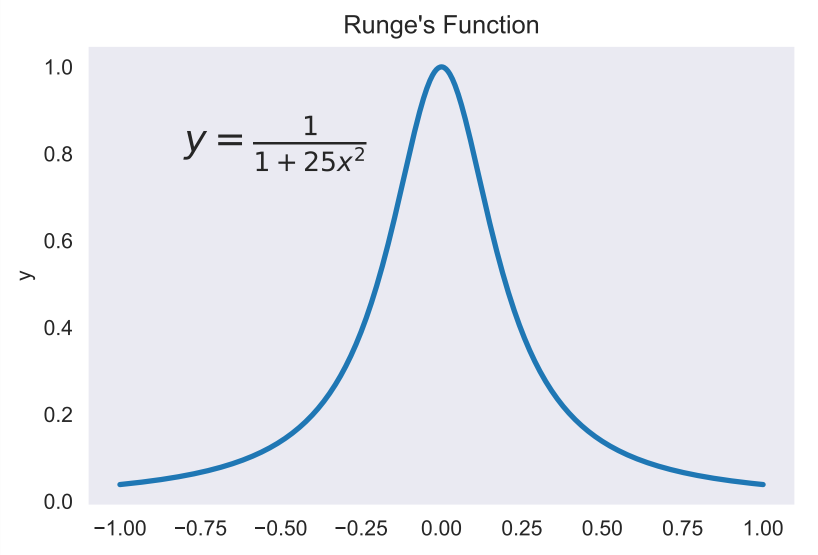



**Figure 2:** Runge's Function

# Chebychev Polynomial Interpolation

- Theory asserts that the best way to approximate a continuous function with a polynomial over a bounded interval $[a, b]$ is to interpolate it at the so-called Chebychev nodes:

$$x_i = \frac{a+b}{2} + \frac{b-a}{2} \cos\left(\frac{n-i+0.5}{n}\pi\right), \qquad i = 1, 2, \ldots, n.$$

- The Chebychev nodes are not evenly spaced and do not include the endpoints of the approximation interval.
- They are more closely spaced near the endpoints of the approximation interval and less so near the center.

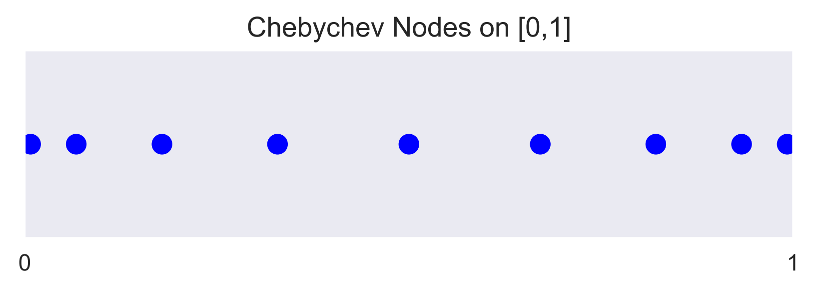



Figure 3: Chebychev Nodes on $[0, 1]$

If $f$ is continuous ...

- Rivlin's Theorem asserts that Chebychev-node polynomial interpolation is nearly optimal, that is, it affords an approximation error that is very close to the lowest error attainable with another polynomial of the same degree.
- Jackson's Theorem asserts that Chebychev-node polynomial interpolation is consistent, that is, the approximation error vanishes as the degree of the polynomial increases.

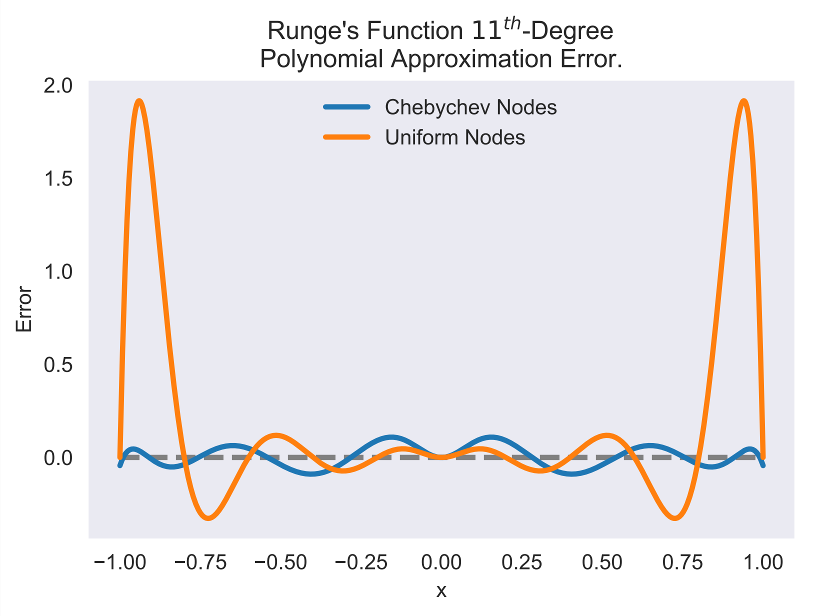



**Figure 4:** Runge's Function Polynomial Approximation Error

- When the function being approximated is smooth, Chebychev node polynomial interpolants typically exhibit errors that oscillate fairly evenly throughout the interval of approximation.
- This feature is called the Chebychev equi-oscillation property.
- Consider the Chebychev interpolant to $\exp(-x)$ on $[-1, 1]$.
- The Chebychev interpolant avoids the instability near the interval endpoints exhibited by a uniform node polynomial interpolant because the Chebychev nodes are more concentrated near the endpoints.

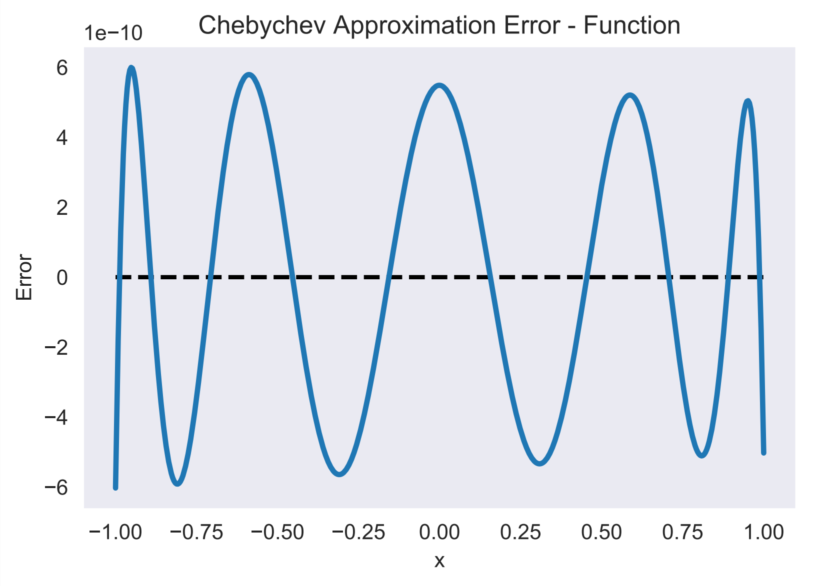



**Figure 5:** Chebychev Polynomial Interpolant Approximation Error for $e^{-x}$

- Interpolating at the Chebychev nodes offers many advantages.
- However, merely interpolating at the Chebychev nodes does not eliminate ill-conditioning.
- Ill-conditioning stems from the choice of basis functions, not the choice of interpolation nodes.
- Fortunately, there is alternative to the monomial basis that is ideal for expressing Chebychev-node polynomial interpolants.

- The optimal basis for expressing Chebychev-node polynomial interpolants is called the Chebychev polynomial basis.
- The Chebychev polynomials are defined for $z \in [-1, 1]$ as

$$\begin{aligned} T_0(z) &= 1 \\ T_1(z) &= z \\ T_2(z) &= 2z^2 - 1 \\ T_3(z) &= 4z^3 - 3z \\ &\vdots \\ T_j(z) &= 2zT_{j-1}(z) - T_{j-2}(z). \end{aligned}$$

- They can be defined for arbitrary intervals $[a, b]$ via the transformation $z = 2\frac{x-a}{b-a} - 1$ for $x \in [a, b]$.

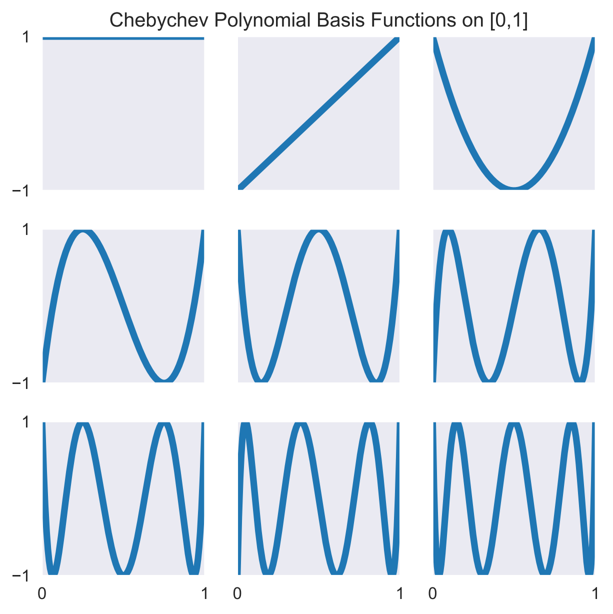



Figure 6: Chebychev Polynomial Basis Functions on $[0, 1]$

- Combining the Chebychev basis polynomials and Chebychev interpolation nodes yields an extremely well-conditioned interpolation equation.
- The Chebychev interpolation matrix is orthogonal, that is, $\Phi'\Phi$ is diagonal.
- Its condition number is $\sqrt{2}$, regardless of the degree of interpolation, which is near the absolute minimum of 1.
- This implies that basis coefficients can be computed accurately, regardless of the number of basis functions.

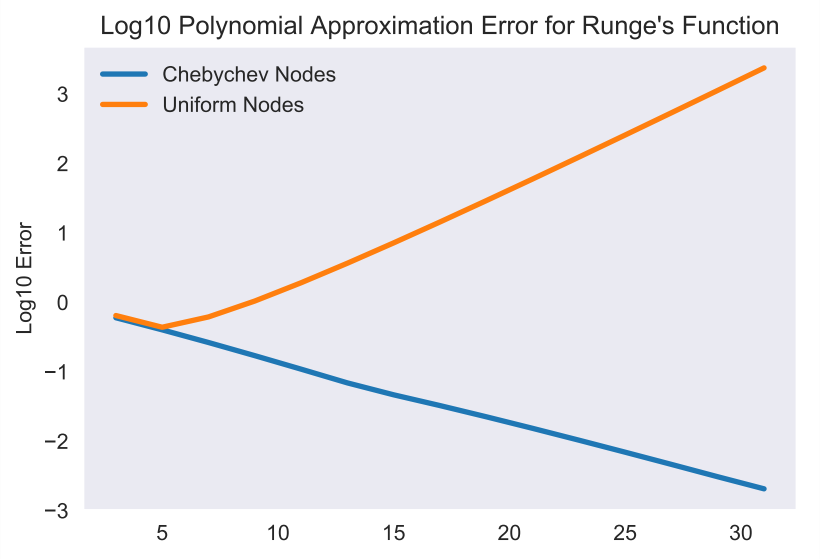



Figure 7: Interpolation Matrix Condition Number

# Spline Interpolation

# Introduction

- Piecewise polynomial splines, or simply splines for short, are a rich, flexible class of functions that may be used instead of high degree polynomials to approximate a real-valued function over a bounded interval.
- Generally, an order $k$ spline consists of a series of $k^{th}$ degree polynomial segments spliced together so as to preserve continuity of derivatives of order $k-1$ or less.

- Two classes of splines are often employed in practice.
- A first-order or linear spline is a series of line segments spliced together to form a continuous function.
- A third-order or cubic spline is a series of cubic polynomials segments spliced together to form a twice continuously differentiable function.

# Linear Splines

- Linear splines use line segments to connect points on the graph of the function to be approximated.
- They are particularly easy to construct and work with in practice, which explains their widespread popularity.

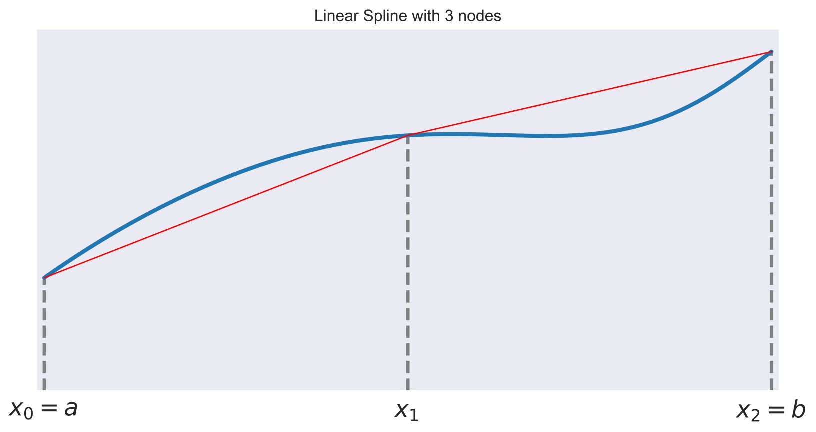



Figure 8: Linear Spline Interpolation, 2 Intervals

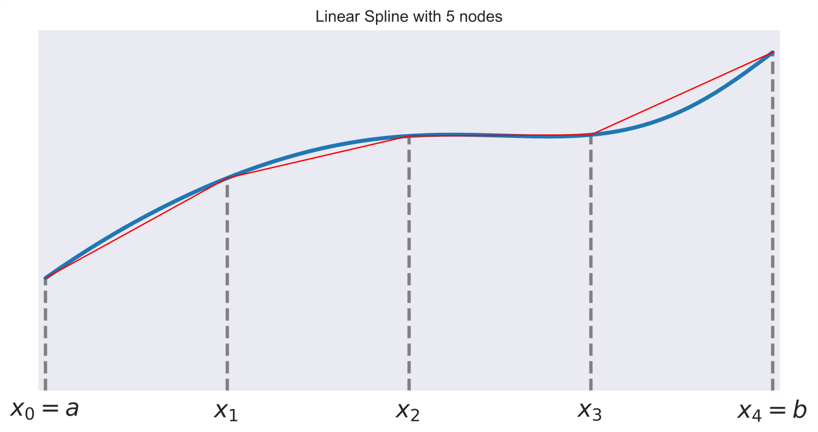



**Figure 9:** Linear Spline Interpolation, 4 Intervals

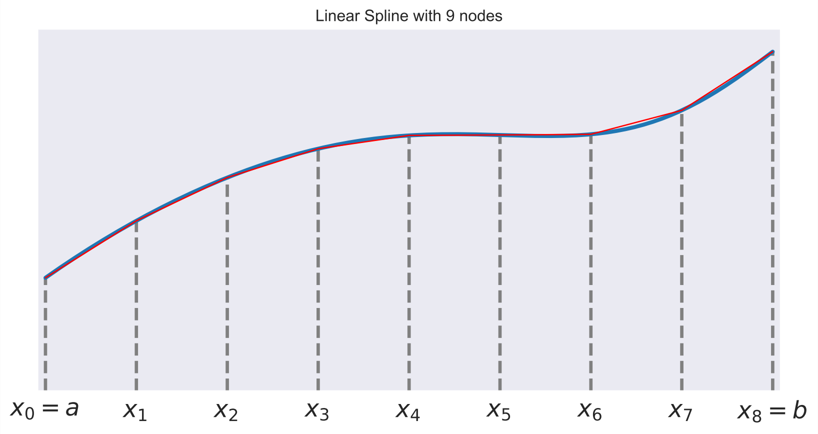



**Figure 10:** Linear Spline Interpolation, 8 Intervals

A linear spline with $n+1$ evenly-spaced interpolation nodes $x_0, x_1, \ldots, x_n$ on the interval $[a, b]$ may be written as a linear combination of the $n+1$ basis functions

$$\phi_j(x) = \begin{cases} 1 - \frac{|x-x_j|}{h} & |x - x_j| \leq h \\ \\ 0 & \text{otherwise.} \end{cases}$$

where $h = (b-a)/n$ is the distance between the nodes.

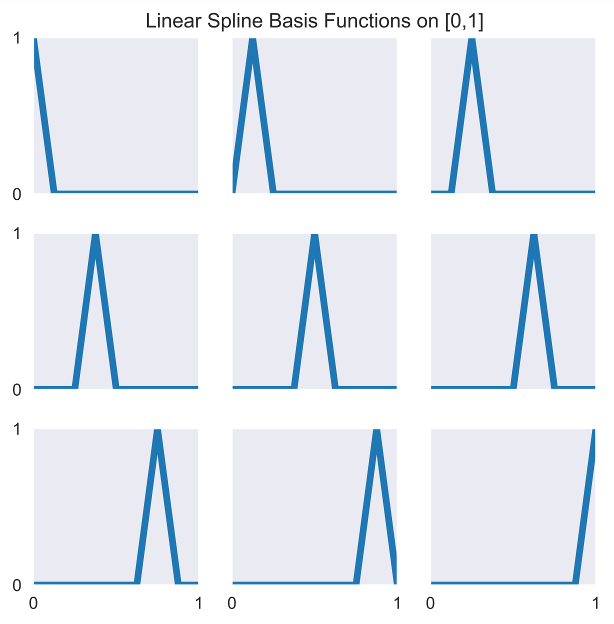



**Figure 11:** Linear Spline Basis Functions on $[0, 1]$

- Linear spline basis functions are often called the “hat” functions.
- Each basis function is zero everywhere, except over a narrow support of width $2h$.
- At most two basis functions are nonzero at any point.

- Computing basis coefficients for a linear spline approximation is a trivial matter.
- By construction, $\phi_i(x_j)$ equals one if $i = j$, but equals zero otherwise; that is, the interpolation matrix $\Phi$ is the identity matrix.
- Thus, the basis coefficients are simply the function values at the interpolation nodes, $c_i = f(x_i)$.

- Evaluating a linear spline and its derivative at an arbitrary point $x$ is straightforward.
- Since at most two basis functions are nonzero at any point, only two basis function evaluations are required.
- Specifically, if $x$ lies between $x_{i-1}$ and $x_i$, then

$$\hat{f}(x) = ((x - x_{i-1})c_i + (x_i - x)c_{i-1})/h$$

and

$$\hat{f}'(x) = (c_i - c_{i-1})/h.$$

- Linear splines, however, possess limitations that make them a poor choices in most computational economic applications.
- Linear splines possess discontinuous first derivatives and higher order derivatives that are zero almost everywhere.
- Linear splines thus do a poor job of approximating first derivatives and cannot approximate higher order derivatives.
- In many economic applications, however, derivatives are of fundamental interest to an economist.

# Cubic Splines

- A cubic spline is a series of cubic polynomials segments spliced together to form a twice continuously differentiable function.
- Cubic splines retain much of the simplicity of linear splines, but possess continuous first and second derivatives.
- Cubic splines are therefore preferred to linear splines when a smooth approximation is required.

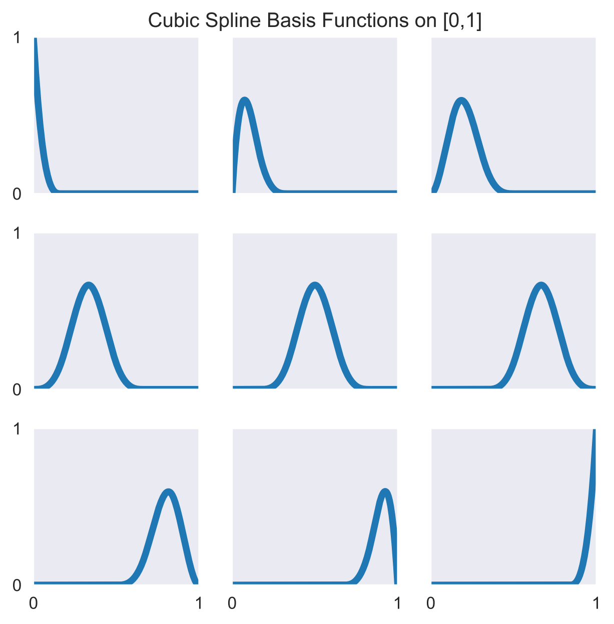



Figure 12: Cubic Spline Basis Functions on $[0, 1]$

- Cubic spline basis functions exhibit certain properties.
- Each basis function is zero everywhere, except over a narrow support.
- Each basis function and its derivatives vanish at the endpoints of its support.
- At most four basis functions are nonzero at any point.

- Computing basis coefficients for a cubic spline approximation is also relatively easy.
- By construction, at most four basis functions are nonzero at any interpolation node.
- Thus, the interpolation matrix will consist mostly of zeros, with nonzero entries concentrated around the diagonal.
- As such, the interpolation matrix may be stored in "sparse" format, reducing the required storage space and reducing the operations required to solve the interpolation equation.
- The matrix, moreover, is naturally well-conditioned.

# Additional Considerations

# Multidimensional Interpolation

- Univariate interpolation methods can be extended to higher dimensions by applying tensor product principles.
- Consider the problem of interpolating a bivariate real-valued function $f$ over an interval

$$I = \{(x, y) \mid a_x \leq x \leq b_x, a_y \leq y \leq b_y\}.$$

- Let $\phi_1^x, \phi_2^x, \ldots, \phi_{n_x}^x$ and $x_1, x_2, \ldots, x_{n_x}$ be $n_x$ univariate basis functions and $n_x$ interpolation nodes for the interval $[a_x, b_x]$.
- Let $\phi_1^y, \phi_2^y, \ldots, \phi_{n_y}^y$ and $y_1, y_2, \ldots, y_{n_y}$ be $n_y$ univariate basis functions and $n_y$ interpolation nodes for the interval $[a_y, b_y]$.

- Then an $n = n_x n_y$ bivariate function basis defined on $I$ may be obtained by forming the tensor product of the univariate basis functions:

$$\phi_{ij}(x, y) = \phi_i^x(x)\phi_j^y(y)$$

for $i = 1, 2, \ldots, n_x$ and $j = 1, 2, \ldots, n_y$.

- Similarly, a grid of $n = n_x n_y$ interpolation nodes for $I$ may be obtained by forming the Cartesian product of the univariate interpolation nodes

$$\{\, (x_i, y_j) \mid i = 1, 2, \ldots, n_x; j = 1, 2, \ldots, n_y\}.$$

- Typically, multivariate tensor product interpolation schemes inherit the favorable qualities of their univariate parents.
- Multivariate spline interpolation schemes produce sparse interpolation matrices.
- Multivariate Chebychev polynomial interpolation schemes produce orthogonal, well-conditioned interpolation matrices.

- However, multidimensional tensor product interpolation schemes suffer from the curse of dimensionality.
- Specifically, the number of basis functions and interpolation nodes grow exponentially with the dimension of the function domain.
- For example, if you choose $n$ basis functions and interpolation nodes in each of $d$ dimensions, the tensor product basis would contain $n^d$ functions and the Cartesian product interpolation grid would contain $n^d$ interpolation nodes.

- Working directly with tensor product bases requires knowledge of tensor algebra.
- However, there is no need for you to master tensor algebra.
- All mundane tensor product operations required to solve computational economic problems are handled efficiently by CompEcon utilities.

## Choosing an Approximation Method

- Chebychev polynomial interpolation tends to outperform spline interpolation when the function being approximated is very smooth.
- However, if the function possesses discontinuities in the first or second derivative, spline functions sometimes perform as well or better.
- Also, if the dimension of the problem is large, spline interpolation enjoys an advantage because of its sparse interpolation matrix.

| Function | Nodes | Linear Spline | Cubic Spline | Chebychev Polynomial |
|---|---|---|---|---|
| $e^{-x}$ | 10 | -1.82 | -4.42 | -9.22 |
| | 20 | -2.45 | -5.86 | -15.00 |
| | 30 | -2.81 | -6.64 | -15.00 |
| $\|x\|^{0.5}$ | 10 | -0.48 | -0.48 | -0.47 |
| | 20 | -0.64 | -0.67 | -0.62 |
| | 30 | -0.73 | -0.77 | -0.71 |

Log10 Approximation Errors for Smooth and Kinked Functions on $[-1, 1]$, Different Interpolation Schemes

# CompEcon Toolbox

## The Basis class

There are three classes defined in CompEcon to represent interpolation bases:

| | | |
|---|---|---|
| `BasisChebyshev` | - | defines a Chebyshev basis |
| `BasisSpline` | - | defines a spline basis |
| `BasisLinear` | - | defines a linear basis |

To work with them, we follow these steps:

1. define a basis object
2. fit the basis to a function
3. evaluate the basis at interpolation points

# To define a basis object

Step 1:

```
basis = BASIS(n,a,b,order)
```

| | |
|---|---|
| `BASIS` | - basis class ('BasisChebyshev' or 'BasisSpline') |
| `n` | - number of basis functions and nodes |
| `a` | - left endpoint of interpolation interval |
| `b` | - right endpoint of interpolation interval |
| `order` | - optional order of spline (default: 3 for cubic) |
| `basis` | - an instance of class BASIS |
| `.nodes` | - interpolation nodes |
| `.Phi()` | - interpolation matrix |
| `.c` | - basis function coefficients |

# Fitting a function

Step 2:

Either

```
basis.y = y_at_nodes
```

or

```
basis.c = new_coef
```

| | |
|---|---|
| `basis` | - an instance of class BASIS |
| `y_at_nodes` | - known value of function at nodes |
| `new_coef` | - new interpolation coefficients |
| `basis` | object is updated in place |

# Evaluate interpolant

**Step 3:**

```
y = basis(x, d)
```

| | |
|---|---|
| `basis` | - an instance of class BASIS |
| `x` | - evaluation point(s) |
| `d` | - order of differentiation |
| `y` | -approximant value or derivative |

# Evaluate basis functions

Although rarely needed when working with these classes, we can also compute the basis functions at arbitrary interpolation points.

```
phi = basis.Phi(x, d)
```

| | |
|---|---|
| `basis` | - an instance of class BASIS |
| `x` | - evaluation point(s) |
| `d` | - order of differentiation |
| `phi` | - basis functions or derivatives evaluated at `x` |

Example 1:

# Univariate Approximation

Let us construct an approximation to $f(x) = \exp(-x)$ over the interval $[-1, 1]$ and test how well it tracks the function and its first derivative.

**Step 1:** Create functions for $f$ and its derivatives:

```
def f(x): return np.exp(-x)
def d1(x): return -np.exp(-x)
def d2(x): return np.exp(-x)
```

**Step 2:** Create a Chebyshev polynomial basis and fit the `f` function:

```
n, a, b = 10, -1, 1
F = BasisChebyshev(n, a, b, f=f)
```

**Step 3:** Use F to evaluate the Chebychev polynomial interpolant and its derivatives:

```
x = np.linspace(a, b, 501)
ffit = F(x)
dfit1 = F(x, 1)
dfit2 = F(x, 2)
```

**Step 4:** Plot the approximation residuals on a refined grid:

```
plt.plot(x, ffit-f(x))
plt.plot(x, dfit1-d1(x))
plt.plot(x, dfit2-d2(x))
```

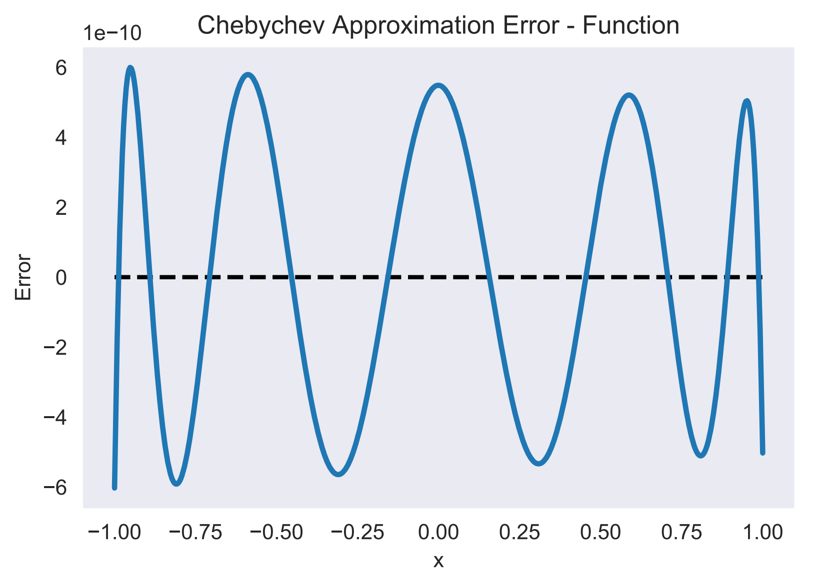



**Figure 13:** 10-node Chebychev Approximation Error for $e^{-x}$

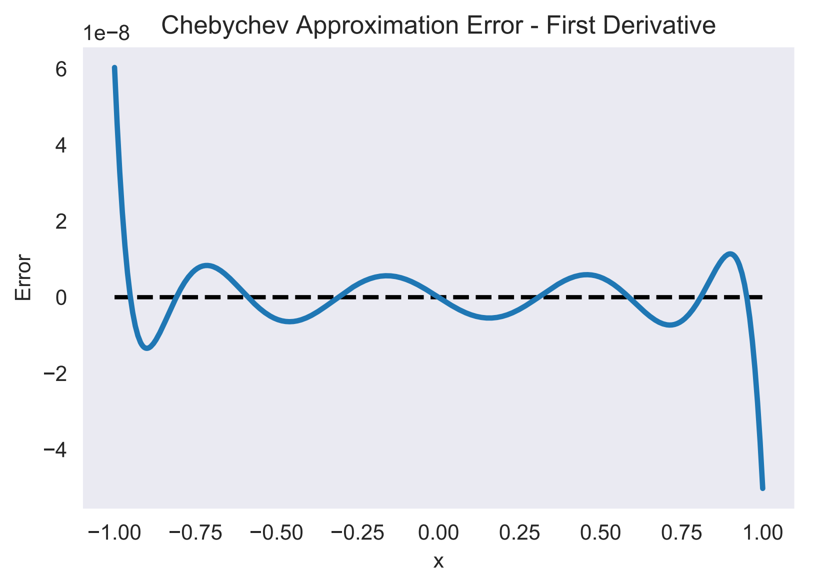



**Figure 14:** Chebychev Approximation Error for First Derivative of $e^{-x}$

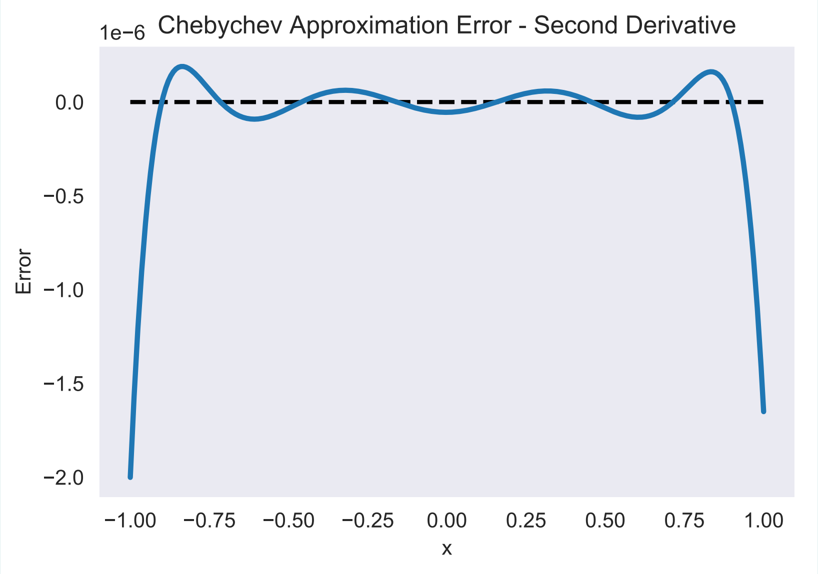



Figure 15: Chebychev Approximation Error for Second Derivative of $e^{-x}$

Example 2:

# Bivariate Approximation

Let us construct a Chebychev polynomial interpolant to the bivariate function $f(x_1, x_2) = cos(x_1)/exp(x_2)$ over unit square $[0, 1] \times [0, 1]$.

**Step 1:** Create functions for $f$ and its derivatives up to order two:

```
exp, cos, sin = np.exp, np.cos, np.sin

f   = lambda x:  cos(x[0]) / exp(x[1])
d1  = lambda x: -sin(x[0]) / exp(x[1])
d2  = lambda x: -cos(x[0]) / exp(x[1])
d11 = lambda x: -cos(x[0]) / exp(x[1])
d12 = lambda x:  sin(x[0]) / exp(x[1])
d22 = lambda x:  cos(x[0]) / exp(x[1])
```

Step 2: Create a Chebyshev polynomial basis and fit the `f` function:

```
n, a, b = 6, 0, 1
F = BasisChebyshev([n, n], a, b, f=f)
```

Step 3: To compute the partial derivatives $\frac{\partial f}{\partial x_1}$ and $\frac{\partial f}{\partial x_2}$ of the interpolant at $x = (0.5, 0.5)$ , execute

```
x = np.array([[0.5],[0.5]])
dfit1 = F(x, [1, 0])
dfit2 = F(x, [0, 1])
```

To compute the second partial derivatives $\frac{\partial^2 f}{\partial x_1^2}$, $\frac{\partial^2 f}{\partial x_1 \partial x_2}$, and $\frac{\partial^2 f}{\partial x_2^2}$ of the interpolant, execute

```
dfit11 = F(x, [2, 0])
dfit22 = F(x, [0, 2])
dfit12 = F(x, [1, 1])
```

**Step 4:** To plot the approximation residual, execute:

```
nplot = [101, 101]
X = nodeunif(nplot, [a, a], [b, b])
error = (F(x) - f(x)).reshape(nplot)
X1, X2 = X
X1.shape = nplot
X2.shape = nplot

plt.figure()
ax = fig1.add_subplot(1, 1, 1, projection='3d')
ax.plot_surface(X1, X2, error, rstride=1, cstride=1,
    cmap=cm.coolwarm, linewidth=0, antialiased=False)
```

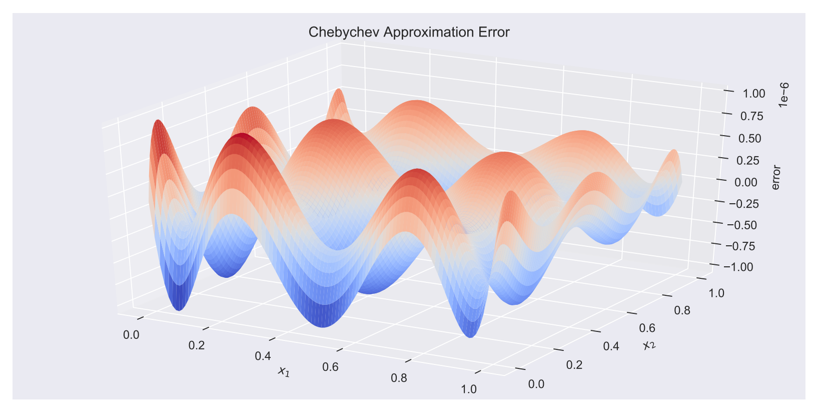



Figure 16: 6 by 6 Node Chebychev Polynomial Approximation Error for $cos(x_1)/exp(x_2)$

# Functional Equations

Functional equations are ubiquitous in dynamic economics and include

- Bellman equations
- Euler equations
- Rational expectations equilibria
- Ordinary differential equations
- Partial differential equations

- Formally, a functional equation takes the form

$$F(f, x) = 0 \text{ for all } x \in S,$$

where $f$ is an unknown real-valued function defined on a set $S \subset \Re^d$ and $F$ is a real-valued mapping with two arguments, a real-valued function $f$ defined on $S$ and an element $x$ of $S$.
- For a given function $f : S \mapsto \Re$, the real-valued mapping $x \mapsto F(f, x)$ on $S$ is called the residual of $f$.
- A solution to the functional equation is a function $f$ whose residual is zero for all $x \in S$.

- A functional equation is fundamentally difficult to solve because the unknown is an entire function $f$ that must satisfy an infinite number of conditions, one at each point $x$ of $S$.
- Although some functional equations encountered in economics posses closed-form solution, the vast majority do not.
- Accurate approximate solutions, however, can be computed numerically using natural extensions of interpolation methods.

# Collocation Method

- We will compute approximate solutions to functional equations numerically using the collocation method.
- The collocation method calls for the solution function $f$ to be approximated using a linear combination of $n$ known basis functions $\phi_1, \phi_2, \ldots, \phi_n$ defined on $S$:

$$f(x) \approx \sum_{j=1}^{n} c_j \phi_j(x).$$

- The basis coefficients $c_1, c_2, \ldots, c_n$ are fixed by requiring the approximation residual to be zero, not at all $x$ in $S$, but rather at $n$ judiciously chosen collocation nodes $x_1, x_2, \ldots, x_n$ in $S$:

$$F\left(\sum_{j=1}^{n} c_j \phi_j,\ x_i\right) = 0, \qquad i = 1, 2, \ldots, n.$$

- This equation is called the collocation equation.
- The unknown of the collocation equation is not the desired function $f$, but rather the basis coefficients $c_1, c_2, \ldots, c_n$ of its approximant.

- The collocation method replaces a fundamentally difficult infinite-dimensional functional equation problem with a finite-dimensional rootfinding problem that can be solved using standard nonlinear equation methods.
- We will use collocation to solve the dynamic economic models we encounter later in the course.
- We will introduce the collocation method by first applying it to some relatively easy examples.

Example 3:

# Implicit Function

- Given a function $g : \Re^2 \mapsto \Re$

$$g(x, y) = y^{-2} + y^{-5} - 2x$$

  find a function $f : \Re \mapsto \Re$ such that:

$$g(x, f(x)) = 0, \quad x \in [1, 5].$$

- The Implicit Function Theorem guarantees that such a function exists, is unique, and is continuously differentiable.

- To solve the functional equation numerically using collocation, approximate the unknown function using a linear combination of $n$ known basis functions $\phi_1, \phi_2, \ldots, \phi_n$:

$$f(x) \approx \sum_{j=1}^{n} c_j \phi_j(x).$$

- Then fix the basis coefficients $c_1, c_2, \ldots, c_n$ by requiring the approximant to satisfy the functional equation at $n$ judiciously chosen collocation nodes $x_1, x_2, \ldots, x_n$:

$$g(x_i, \sum_{j=1}^{n} c_j \phi_j(x_i)) = 0, \qquad i = 1, 2, \ldots, n.$$

- That is, solve the $n$ nonlinear collocation equations

$$\left(\sum_{j=1}^{n} c_j \phi_j(x_i)\right)^{-2} + \left(\sum_{j=1}^{n} c_j \phi_j(x_i)\right)^{-5} - 2x_i = 0, \quad i = 1, 2, \ldots, n$$

for the $n$ unknown basis function coefficients $c_1, c_2, \ldots, c_n$.

To solve the collocation equation in Python:

**Step 1:** Create a `BasisChebyshev` to represent $f$, and obtain its nodes

```
n, a, b = 31, 1, 5
F = BasisChebyshev(n, a, b)
x = F.nodes
```

where we use a 31 node Chebychev polynomial interpolation scheme.

Step 2: Define a function `resid` that evaluates the residual of the approximation at the basis nodes `x`, for arbitrary basis coefficient vector `c`:

```
def resid(c):
    F.c = c  # update basis coefficients
    f = F(x) # interpolate at basis nodes x
    return f ** -2 + f ** -5 - 2 * x
```

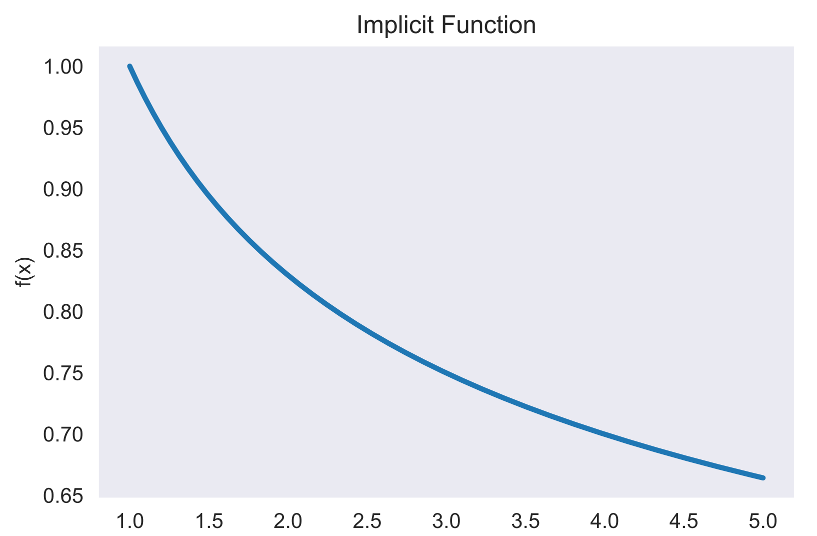
**Step 3:** Solve the collocation equation for the basis coefficients:

```
c0 = np.zeros(n) #initial guess for coeffs
c0[0] = 0.2
F.c = NLP(resid).broyden(c0)
```

Here, we use `broyden` to solve for a coefficient vector `c` that sets the residual to zero at the collocation nodes.

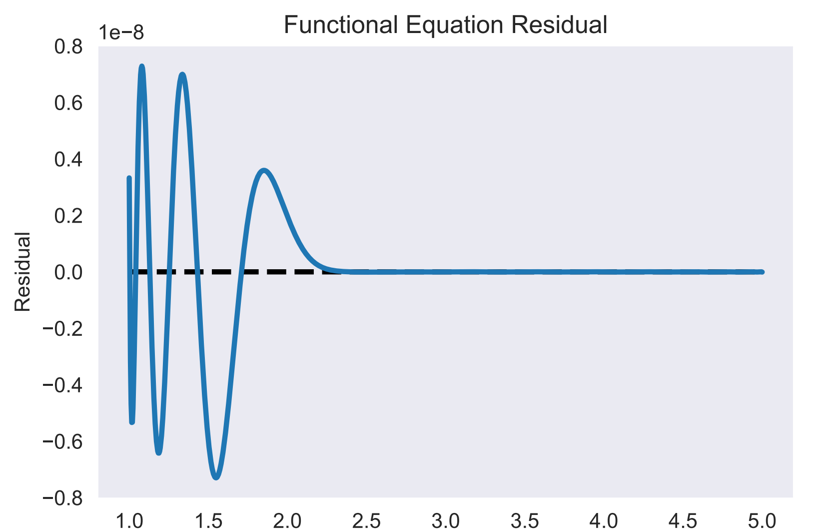
**Step 4:** Plot the approximant on a refined grid:

```
x = np.linspace(a, b, 1000)
plt.plot(x, F(x))
```



**Step 5:** Plot the residual on a refined grid of nodes to assess the quality of the approximation:

```
plt.plot(x, resid(F.c))
```



Example 4:

# Monopolistic Supply

- A monopolist facing a demand curve $q = d(p)$ sets output $q$ so as to maximize marginal profit, implying that

$$\frac{d\pi}{dq} = p + q\frac{dp}{dq} - k(q) = 0$$

where $p$ is price, $dp/dq$ is the marginal effect of the monopolists's output on price, and $k(q)$ is marginal cost.
- The monopolist's effective supply curve $q = s(p)$, which gives the quantity $q$ he is willing to produce at a given price $p$, is characterized by the functional equation

$$p + s(p)/d'(p) - k(s(p)) = 0, \quad p > 0.$$

- To solve the functional equation numerically using collocation, approximate the unknown effective supply curve using a linear combination of $n$ known basis functions $\phi_1, \phi_2, \ldots, \phi_n$:

$$s(p) \approx \sum_{j=1}^{n} c_j \phi_j(p).$$

- Then fix the basis coefficients $c_1, c_2, \ldots, c_n$ by requiring the approximant to satisfy the first-order optimality condition at $n$ judiciously chosen collocation nodes $p_1, p_2, \ldots, p_n$:

$$p_i + \sum_{j=1}^{n} c_j \phi_j(p_i) / d'(p_i) - k \left( \sum_{j=1}^{n} c_j \phi_j(p_i) \right) = 0, \quad i = 1, 2, \ldots, n.$$

- Let us derive the monopolist's effective supply curve for $p \in [0.5, 2.5]$ when
$$d(p) = p^{-3.5}$$
and
$$k(q) = \sqrt{q} + q^2.$$

To solve the collocation equation in Python:

**Step 1:** Create a `BasisChebyshev` to represent the quantity $q$, and obtain its nodes $p$ (prices):

```
n, a, b = 21, 0.5, 2.5
Q = BasisChebyshev(n, a, b)
p = Q.nodes
```

where we use a 21 node Chebychev polynomial interpolation scheme.

**Step 2:** Define a function `resid` that evaluates the residual of the approximation at the basis nodes `p`, for arbitrary basis coefficient vector `c`:

```
def resid(c):
    Q.c = c
    q = Q(p)
    return p + q/(-3.5*p**(-4.5)) - np.sqrt(q) - q**2
```

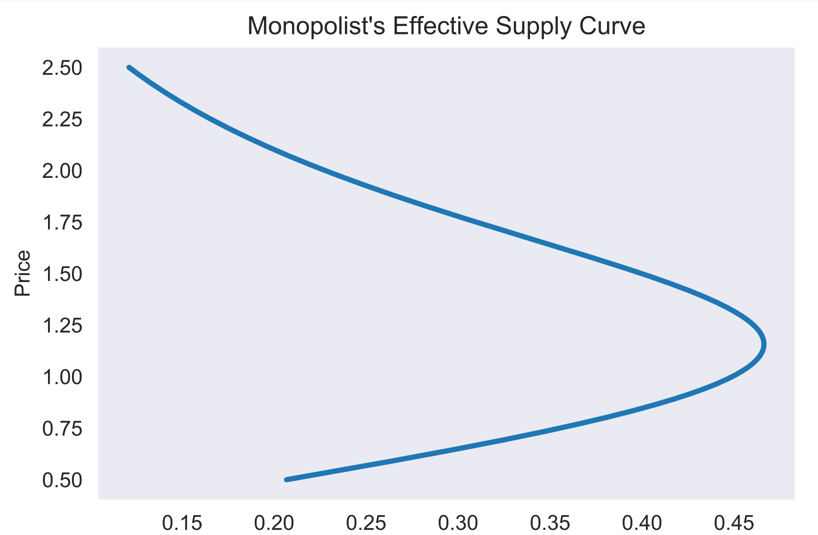
Step 3: Solve the collocation equation for the basis coefficients:

```
c0 = np.zeros(n) #initial guess for coeffs
c0[0] = 2
monopoly = NLP(resid)
Q.c = monopoly.broyden(c0)
```

Here, we use `broyden` to solve for a coefficient vector `c` that sets the residual to zero at the collocation nodes.

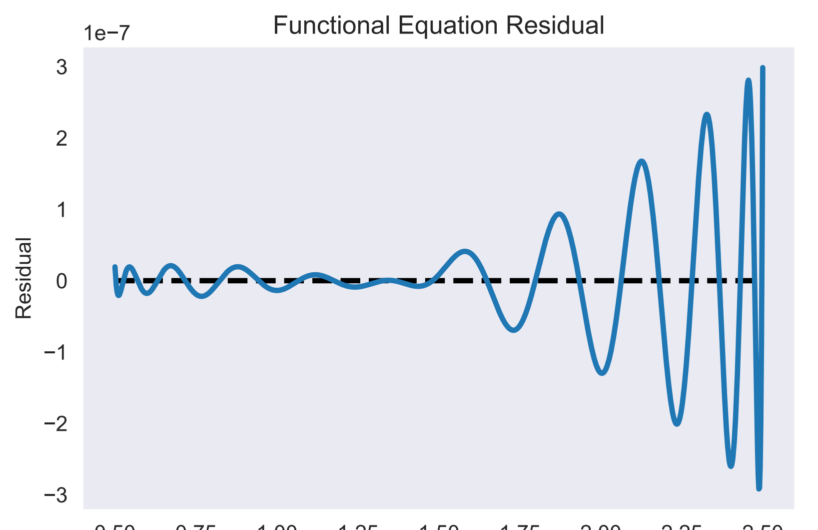
Step 4: Plot the approximant on a refined grid:

```
p = np.linspace(a, b, 1000)
plt.plot(Q(p), p)
```



Step 5: Plot the residual on a refined grid of nodes to assess the quality of the approximation:

```
plt.plot(p, resid(Q.c))
```



Example 5:

# Cournot Equilibrium

- Consider an oligopolistic market consisting of $m$ identical firms facing a common demand curve $q = d(p)$.
- Under the Cournot equilibrium assumption, each firm $i$ takes its competitors' output as fixed when determining its output.
- That is, firm $i$ assumes that the marginal impact of its output decision $q_i$ on market price $p$ is given by

$$\frac{dp}{dq_i} = \frac{1}{d'(p)}.$$

- Under the Cournot equilibrium assumption, firm $i$'s profit maximization condition thus reduces to

$$\frac{d\pi}{dq_i} = p + \frac{q_i}{d'(p)} - k(q_i) = 0,$$

where $k(\cdot)$ is the representative firm's marginal cost function.

- The representative firm's effective supply curve $q = f(p)$, which gives the quantity $q$ it is willing to produce at a given price $p$, is thus characterized by the functional equation

$$p + f(p)/d'(p) - k(f(p)) = 0, \qquad p > 0.$$

- To solve the functional equation numerically by collocation, approximate the representative firm's effective supply curve using a linear combination of $n$ known basis functions $\phi_1, \phi_2, \ldots, \phi_n$:

$$f(p) \approx \sum_{j=1}^{n} c_j \phi_j(p).$$

- Then fix the basis coefficients $c_1, c_2, \ldots, c_n$ by requiring that

$$p_i + \sum_{j=1}^{n} c_j \phi_j(p_i)/d'(p_i) - k\Big(\sum_{j=1}^{n} c_j \phi_j(p_i)\Big) = 0$$

at $n$ judiciously chosen price collocation nodes $p_1, p_2, \ldots, p_n$.

- Let us derive the representative firm's effective supply curve for $p \in [1, 2]$ if

$$d(p) = p^{-\eta}$$

and

$$k(q) = \alpha\sqrt{q} + q^2,$$

where $\alpha = 1$ and $\eta = 3.5$.

To solve the collocation equation in Python:

**Step 1:** Create a BasisChebyshev to represent the quantity supplied $s$, and obtain its nodes $p$ (prices):

```
n, a, b =  25, 0.5, 2.0
S = BasisChebyshev(n,a,b,
                   labels=['price'],y=np.ones(n))
p = S.nodes
```

where we use a 25 node Chebychev polynomial interpolation scheme.

**Step 2:** Define a function `resid` that evaluates the residual of the approximation at the basis nodes `p`, for arbitrary basis coefficient vector `c`:

```
alpha, eta = 1.0, 3.5

def resid(c):
    S.c = c  # update interpolation coefficients
    q = S(p) # compute quantity supplied at price nodes
    return p - q*(p**(eta+1)/eta) - alpha*np.sqrt(q) - q**2
```

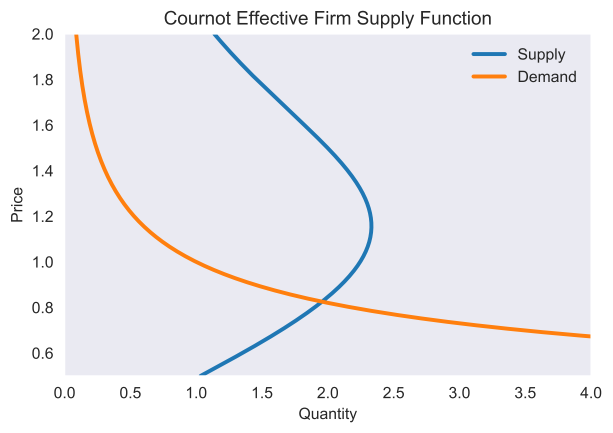
Step 3: Solve the collocation equation for the basis coefficients:

```
cournot = NLP(resid)
S.c = cournot.broyden(S.c, tol=1e-12)
```

Here, we use `broyden` to solve for a coefficient vector `c` that sets the residual to zero at the collocation nodes.

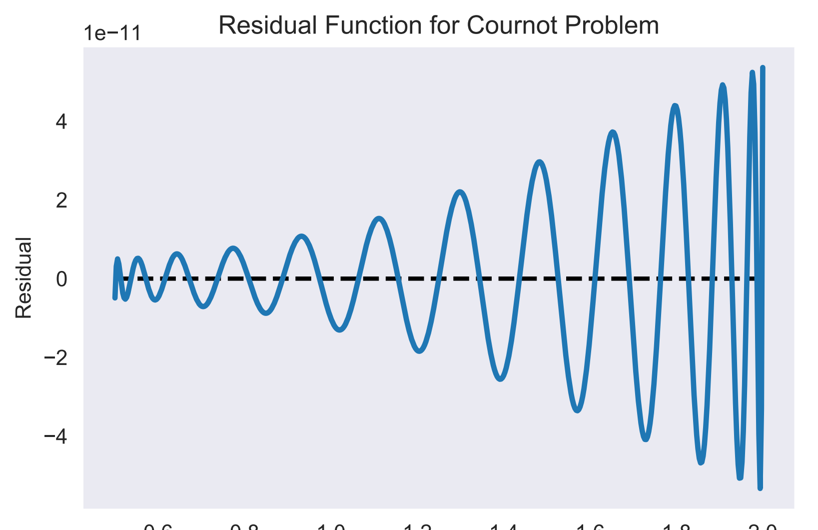
Step 4: Plot the demand and supply of 5 firms, on a refined grid:

```
D = lambda p: p**(-eta) # demand function
prices = np.linspace(a, b, 501)
plt.plot(5*S(prices),prices, D(prices),prices)
```



Step 5: Plot the residual on a refined grid of nodes to assess the quality of the approximation:

```
plt.plot(prices, resid(S.c))
```



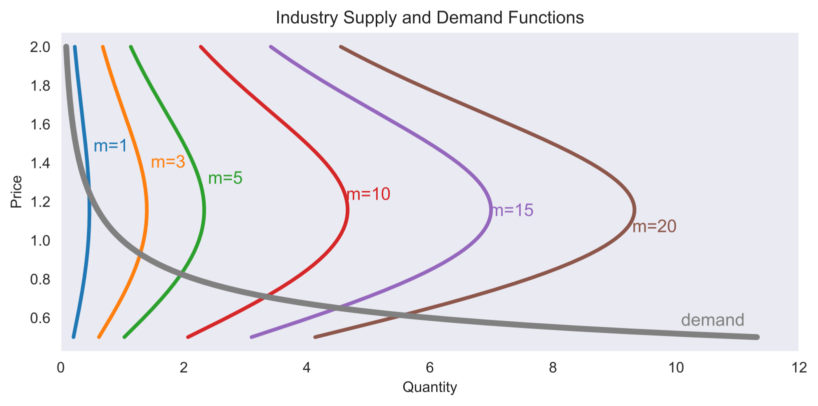



**Figure 23:** Market Demand and Effective Supply with Varying Number of Identical Firms

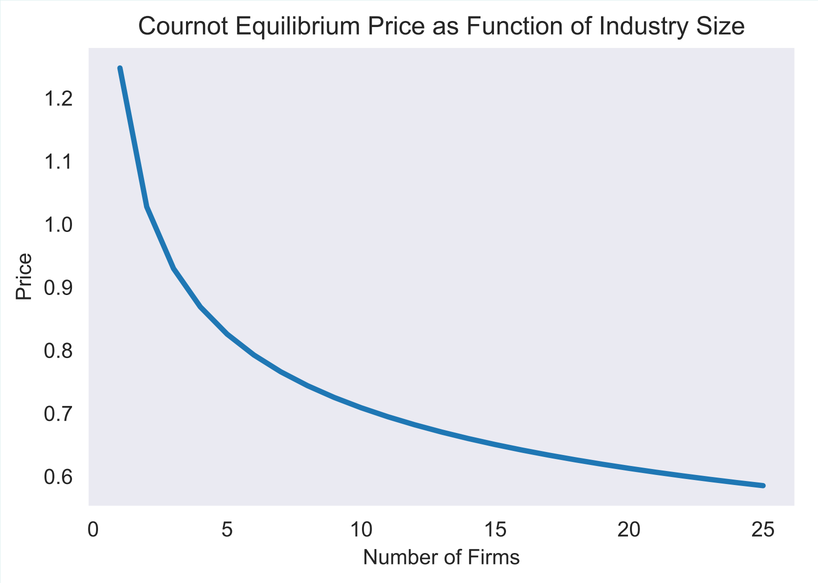



Figure 24: Equilibrium Price as a Function of Number of Firms