Numerical Methods and Symbolic Math

Consider a simple economic model of consumption, where an agent decides between consumption and investment in capital while maximizing his utility:

$$\max \sum_{t=0}^{\infty} \beta^t U(c_t)$$
$$\text{subject to } \begin{cases} c_t = k_t - k_{t+1}, \forall t >= 0 \\ k_0 > 0 \end{cases}$$

where c_t is the time t consumption, k_t is the return of the capital invested at $t-1$ and k_{t+1} is the investment choice at time t. In this simple model there is no rate of return in capital, nor depreciation.

We can transform this problem into a recursive problem using Bellman's equation:

$$\begin{aligned} V(k) = \max_{k'} \quad & U(c) + \beta V(k') \\ \text{s.t.:} \quad & c = k - k' \end{aligned}$$

And we wish to find a solution for the optimal c and k' given the previous investment k:

$$c = g^c(k)$$
$$k' = g^k(k)$$

We can find the optimal policies g^c and g^k by solving the Bellman equation. Solving the problem requires either approximating the value function or obtaining the Euler equation and then approximating c. In either case, we need to approximate functions to solve the problem. We can approximate functions either locally, around a specific point, or globally, over the entire domain of the function.

This lecture discusses methods for approximating functions and how to use them in Matlab. If we also consider a economic model of consumption where there is uncertainty regarding future states of the economy, then we will notice that to solve the maximization problem we will need to integrate functions. Therefore, this lecture also discusses methods for integrating functions and the implementations available in Matlab. We will also discuss the symbolic math capabilities of Matlab, which can help with deriving the equations we need to approximate.

1 Approximating Functions

Given a function f we want to approximate it by a finite number n of basis functions $\{p_k\}_{k=1}^n$. We need to choose weights such that:

$$\begin{aligned} f(x) &\approx \beta_1 p_1(x) + \beta_2 p_2(x) + \cdots + \beta_n p_n(x) \\ &= \underbrace{\begin{pmatrix} p_1(x) & p_2(x) & \cdots & p_n(x) \end{pmatrix}}_{p(x)'} \underbrace{\begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{pmatrix}}_{\beta} \\ &= p(x)'\beta \end{aligned}$$

If we have a set of points $\{x_i\}_{i=1}^d$ where we know the value of f, then we can write:

$$\underbrace{\begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_d) \end{pmatrix}}_{F} \approx \underbrace{\begin{pmatrix} p_1(x_1) & p_2(x_1) & \cdots & p_n(x_1) \\ p_1(x_2) & p_2(x_2) & \cdots & p_n(x_2) \\ & & \vdots & \\ p_1(x_d) & p_2(x_d) & \cdots & p_n(x_d) \end{pmatrix}}_{P} \underbrace{\begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_d \end{pmatrix}}_{\beta} \tag{1}$$

If the number of basis functions and the number of points is the same, that is $n = d$, then there is an exact solution for β if P is invertible: $\beta = P^{-1}F$.

If we have more points than basis functions, then we can compute a least-squares approximation to f:

$$\hat{\beta} \equiv \underset{\beta \in \mathbb{R}^n}{\operatorname{argmin}} \sum_{i=1}^{d} (f(x_i) - p(x_i)'\beta)^2 \tag{2}$$

We know that the "OLS" solution is:

$$\hat{\beta} = (P'P)^- P'F \tag{3}$$

Notice the use of a generalized inverse in $(P'P)^-$. This is necessary because depending on choice of the basis function, we might have a column of zeros in P (this happens with splines when there are more knots relative to points). Now, the function that approximates f is:

$$\hat{f}(x) \equiv p(x)'\hat{\beta} \tag{4}$$

This approximating function can be computed given a choice of basis functions $\{p_k\}_{k=1}^n$ and a collection of data points $\{x_i\}_{i=1}^d$.

As an aside, consider the case where the original function we wanted to estimate was a conditional mean, $f(x) = \mathbb{E}[Y|X = x]$. If we have i.i.d. observations of f and x, then the $\hat{f}$ above is exactly the non-parametric Sieves estimator for the conditional mean. In this case, there is theory justifying the consistency and asymptotic normality of $\hat{f}$.

Next, we discuss the possible choices for basis functions.

1.1 Jacobi Polynomials

The Jacobi polynomials are a family of orthogonal polynomials defined over the interval $[-1, 1]$. Let's denote the family of Jacobi polynomials by $\mathcal{P}$, and the members of this family by $P_n^{(\alpha,\beta)}$ (to be defined later), where n denotes the highest degree of the polynomial. The Jacobi polynomials are parameterized by α and β, which satisfy $\alpha > -1$ and $\beta > -1$.

We say that a family of polynomials is orthogonal if any two of its members are orthogonal to each other under some inner product. In the case of Jacobi polynomials, there is a weight function w such that $\forall P_n^{(\alpha,\beta)}, P_m^{(\alpha,\beta)} \in \mathcal{P}$:

$$\int w(x) P_n^{(\alpha,\beta)}(x) P_m^{(\alpha,\beta)}(x) dx = 0 \text{ if } n \neq m$$

The Jacobi polynomials are defined as (see Equation 22.3.1 in Abramowitz and Setgun (1964)):

$$P_n^{(\alpha,\beta)} = \frac{1}{2^n} \sum_{m=0}^{n} \binom{n+\alpha}{m} \binom{n+\beta}{n-m} (x-1)^{n-m} (x+1)^m$$

These polynomials are orthogonal with respect to the weight function:

$$w^{(\alpha,\beta)} \equiv (1-x)^\alpha (1+x)^\beta$$

The Jacobi polynomials are available in Matlab's Symbolic Math Toolbox via the function `jacobiP`. This function evaluates the Jacobi polynomial of degree n and parameters (α, β) at a point x. We can plot the functions to get a sense of their shape and of the influence of the parameters α and β:

```
% read documentation
help jacobiP
% create a jacobi polynomial for different degrees
alpha = 1;
beta = 1;
jacobi = @(k, x) jacobiP(k, alpha, beta, x);
% graph a few of the polynomials
fig = figure;
hold on;
for k = 1:5
    fplot(@(x) jacobi(k, x), [-1, 1]);
end
% compare changes in alpha
figure;
hold on;
for alpha = [-0.5, 0, 0.5, 1, 2]
    jp = @(x) jacobiP(3, alpha, 1, x);
    fplot(jp, [-1, 1]);
end
legend(["$\alpha=-0.5$", "$\alpha=0$", "$\alpha=0.5$", "$\
   alpha=1$", ...
        "$\alpha=2$"], 'Interpreter', 'latex', ...
```

```
       'location', 'southeastoutside');
% compare changes in beta
figure;
hold on;
for beta = [-0.5, 0, 0.5, 1, 2]
    jp = @(x) jacobiP(3, 1, beta, x);
    fplot(jp, [-1, 1]);
end
legend(["$\beta=-0.5$", "$\beta=0$", "$\beta=0.5$", "$\beta
   =1$", ...
         "$\beta=2$"], 'Interpreter', 'latex', ...
        'location', 'southeastoutside');
```

We can verify that the Jacobi polynomials are indeed orthogonal:

```
alpha = 1;
beta = 1;
jacobi = @(k, x) jacobiP(k, alpha, beta, x);
% verify orthogonality
w = @(x) (1-x).^alpha.*(1+x).^beta;
integral(@(x) (w(x).*jacobi(2, x).*jacobi(4, x)), -1, 1)
% what about orthonormality?
integral(@(x) (w(x).*jacobi(2, x).*jacobi(2, x)), -1, 1)
```

1.1.1 Expanding the Domain

The Jacobi polynomials are initially defined over $[-1, 1]$, but we can extend the domain to some compact interval $[a, b]$ via a linear change of variables:

$$y = -1 + 2\frac{x-a}{b-a}$$

In this case, we can write the polynomials as:

$$\tilde{P}_k^{(\alpha,\beta)}(x) = P_k^{(\alpha,\beta)}\left(-1 + 2\frac{x-a}{b-a}\right)$$

These polynomials are now orthogonal with respect to a different weight function:

$$\tilde{w}^{(\alpha,\beta)} = (b-x)^\alpha (x-a)^\beta$$

Indeed:

```
% extend jacobi polynomial to [a, b]
alpha = 1;
beta = 1;
a = 0;
b = 100;
jacobi = @(k, x) jacobiP(k, alpha, beta, -1+2.*(x-a)./(b-a));
% modified weight function
w = @(x) (b-x).^alpha.*(x-a).^beta;
% orthogonality
```

```
integral(@(x) (w(x).*jacobi(2, x).*jacobi(3, x)), -1, 1) % wrong domain
integral(@(x) (w(x).*jacobi(2, x).*jacobi(3, x)), a, b)
```

1.1.2 Orthonormal Jacobi Polynomials

It is also possible to make the Jacobi polynomials orthonormal, instead of just orthogonal. We just need to scale the polynomials by a constant that depends on the degree of the polynomial, on the parameters α and β, and on the length of the domain.

We can define the orthonormal Jacobi polynomials as:

$$\tilde{P}_k^{(\alpha,\beta)}(x) \equiv c^{(\alpha,\beta)}(k) P_k^{(\alpha,\beta)}\left(-1 + 2\frac{x-a}{b-a}\right)$$

$$c^{(\alpha,\beta)}(k) \equiv \left(\frac{1}{b-a}\right)^{\frac{\alpha+\beta+1}{2}} \sqrt{\frac{(2k+\alpha+\beta+1)\Gamma(k+1)\Gamma(k+\alpha+\beta+1)}{\Gamma(k+\alpha+1)\Gamma(k+\beta+1)}}$$

Where Γ is the Gamma function, which is available in Matlab via the function `gamma`. These polynomials are orthonormal with respect to the weight function:

$$\tilde{w}^{(\alpha,\beta)} = (b-x)^{\alpha}(x-a)^{\beta}$$

We can verify the orthonormality of the Jacobi polynomials defined by $\tilde{P}_k^{(\alpha,\beta)}(x)$:

```
a = 0;
b = 10;
alpha = 1;
beta = 1;
% define weight function
w = @(x) ((b-x).^alpha).*((x-a).^beta);
% define scaling constant
c = @(k) ((1/(b-a))^((alpha+beta+1)/2)).*((((2.*k+alpha+beta+1).*gamma(k+1).*gamma(k+alpha+beta+1))./(gamma(k+alpha+1).*gamma(k+beta+1))).^0.5);
% define orthonormal jacobi function
jacobi = @(k, x) (c(k).*jacobiP(k, alpha, beta, -1 + 2.*(x-a)./(b-a)));
% visualize
fig = figure;
hold on;
for k = 1:5
    fplot(@(x) jacobi(k, x), [a, b]);
end
% check orthonormality
integral(@(x) (w(x).*jacobi(2, x).*jacobi(3, x)), a, b) % 0
integral(@(x) (w(x).*jacobi(2, x).*jacobi(2, x)), a, b) % 1
```

1.1.3 Special Case: Legendre Polynomials

The Legendre polynomials are special cases of Jacobi polynomials. We can obtain Legendre polynomials by setting $\alpha = \beta = 0$. In this case we have:

$$P_k^{(0,0)}(x) = \frac{1}{2^k}\sum_{m=0}^{k}\binom{k}{m}^2 (x-1)^{k-m}(x+1)^m$$
$$c^{(0,0)} = \sqrt{\frac{2k+1}{b-a}}$$
$$L_k(x) \equiv \tilde{P}_k^{(0,0)}(x) = c^{(0,0)}(k)P_k^{(0,0)}\left(-1+2\frac{x-a}{b-a}\right)$$

Notice that the weight function for the Legendre polynomials is simply 1.

The Legendre polynomials can also be computed recursively:

$$P_0^{(0,0)}(x) = 1$$
$$P_1^{(0,0)}(x) = x$$
$$P_k^{(0,0)}(x) = \left(x \cdot \frac{2k-1}{k}\right) P_{k-1}^{(0,0)}(x) + \left(\frac{k-1}{k}\right) P_{k-2}^{(0,0)}(x)$$

Notice that the recursion is for the orthogonal Legendre polynomials defined over $[-1, 1]$. We can extend the recursion to the orthonormal Legendre polynomials by simply multiplying the $P_k^{(0,0)}$ above by the scaling constant $c^{(0,0)}(k)$.

We can obtain the Legendre polynomials in Matlab via the function `legendreP`. This function is similar to `jacobiP` and evaluates the Legendre polynomial of degree n at the point x.

```
% define interval [a, b]
a = 0;
b = 10;
% define scaling constant
c = @(k) ((2.*k+1)./(b-a)).^0.5;
% define orthonormal legendre polynomials
lp = @(k, x) c(k).*legendreP(k, -1 + 2.*(x-a)./(b-a));
% check orthonormality (good for catching bugs)
l1 = @(x) lp(1, x);
l2 = @(x) lp(2, x);
integral(@(x) l1(x).*l2(x), a, b)        % 0
integral(@(x) l2(x).*l2(x), a, b)        % 1
```

Let's visualize the first few Legendre polynomials:

```
a = -1;
b = 1;
% define scaling constant
c = @(k) ((2.*k+1)./(b-a)).^0.5;
% define orthonormal legendre polynomials
lp = @(k, x) c(k).*legendreP(k, -1 + 2.*(x-a)./(b-a));
% visualize the basis
f = figure;
```

```
for k = 1:6
    subplot(2, 3, k)
    fplot(@(x) lp(k, x), [a b]);
    grid on;
    legend(strcat("$L_", string(k), "(x)$"), 'Interpreter', '
       latex');
end
```

Figure 1 displays the orthonormal Legendre polynomials.

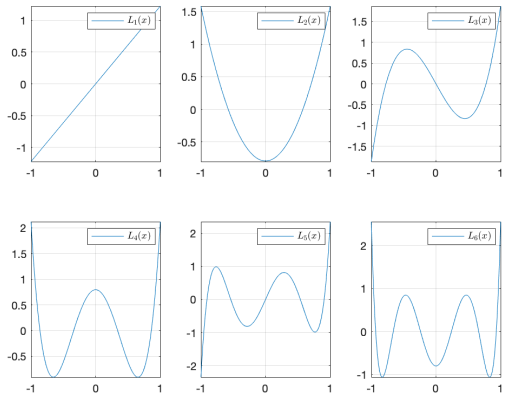

Figure 1: Orthonomal Legendre Polynomials.

1.1.4 Special Case: Chebyshev Polynomials

The Chebyshev polynomials are also a special case of the Jacobi polynomials. There are two families of Chebyshev polynomials depending on the values of α and β. If we set $\alpha = \beta = -\frac{1}{2}$, then we get the Chebyshev polynomials of the first kind, denoted by $T_k(x)$. If we set $\alpha = \beta = \frac{1}{2}$, then we get the Chebyshev polynomials of the second kind, denoted by $U_k(x)$.

These polynomials can be computed recursively. For the first kind, we have:

$$\begin{aligned} T_0(x) &\equiv P_0^{(-0.5,-0.5)} = 1 \\ T_1(x) &\equiv P_1^{(-0.5,-0.5)} = x \\ T_k(x) &\equiv 2xT_{k-1}(x) - T_{k-2}(x) \end{aligned}$$

For the second kind, we have:

$$\begin{aligned} U_0(x) &\equiv P_0^{(0.5,0.5)} = 1 \\ U_1(x) &\equiv P_1^{(0.5,0.5)} = 2x \\ U_k(x) &\equiv 2xU_{k-1}(x) - U_{k-2}(x) \end{aligned}$$

The Chebyshev polynomials are available in the functions `chebyshevT` (first kind) and `chebyshevU` (second kind).

```
a = -1;
b = 1;
% define weight function
```

```
wT = @(x) ((b-x).*(x-a)).^(-0.5);
% orthogonality of chebyshev polynomials
integral(@(x) wT(x).*chebyshevT(2, x).*chebyshevT(3, x), a, b)
integral(@(x) wT(x).*chebyshevT(3, x).*chebyshevT(3, x), a, b)

% define constant for orthonormality
cT = @(k) (2.*k.*gamma(k+1).*gamma(k)./(gamma(k+0.5).*gamma(k+0.5))).^(0.5);
% define orthonormal chebyshev polynomials
orthoC = @(k, x) cT(k).*jacobiP(k, -0.5, -0.5, -1 + 2.*(x-a)./(b-a));
% orthonormality
integral(@(x) wT(x).*orthoC(2, x).*orthoC(3, x), a, b)
integral(@(x) wT(x).*orthoC(2, x).*orthoC(2, x), a, b)
```

Let's visualize the first few orthonormal Chebyshev (first kind) polynomials:

```
f = figure;
for k = 1:6
    subplot(2, 3, k)
    fplot(@(x) orthoC(k, x), [a b]);
    grid on;
    legend(strcat("$T_", string(k), "(x)$"), 'Interpreter', 'latex');
end
```

Figure 2 displays the orthonormal Legendre polynomials.

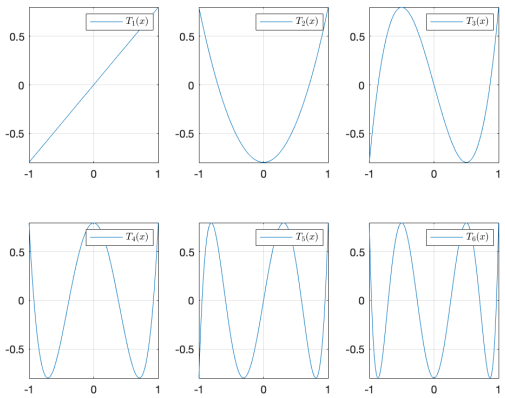

Figure 2: Orthonomal Chebyshev Polynomials (first kind).

1.1.5 Example

Let's approximate the logistic function:

$$f(x) = \frac{1}{1 + \mathrm{e}^{-x}}$$

To approximate f, we need to choose a basis of polynomials. In this example, we will use the orthonormal Legendre polynomials. We also need to compute the generalized inverse of the matrix $P'P$. In Matlab, the generalized inverse (Moore-Penrose) of a matrix can be computed with the function `pinv`.

```
% consider the logistic function
f = @(x) 1./(1+exp(-x));
% consider 5 points
x = linspace(-5, 5, 5)';
% select a basis of polynomials: orthonormal Legendre
   polynomials
% use 2 polynomials
c = @(k) ((2.*k+1)./(b-a)).^0.5;
p = @(x) [c(0).*legendreP(0, x) c(1).*legendreP(1, x)];
% visualize basis
fplot(p, [-5 5]);
% compute the matrices F, P and the generalized inverse of P'
   P
F = f(x);
P = p(x);
beta = pinv(P'*P)*P'*F;
% approximation for the function f
fhat = @(x) p(x)*beta;
% compare original to approximation
fplot(f);
hold on;
fplot(fhat);
```

We can make the approximation more precise by increasing the number of polynomials.

```
% use 4 polynomials
p = @(x) [c(0).*legendreP(0, x) c(1).*legendreP(1, x) ...
          c(2).*legendreP(2, x) c(3).*legendreP(3, x)];
F = f(x);
P = p(x);
beta = pinv(P'*P)*P'*F;
% approximation for the function f
fhat = @(x) p(x)*beta;
% compare original to approximation
fplot(f);
hold on;
fplot(fhat);
```

At this point there are a few observations to be made. First, coding the polynomials by hand is surprisingly hard, since it is quite easy to make mistakes but hard to debug. If you ever need to do so, the orthogonality and orthonomality conditions are useful to check if everything is correct. Second, the approximation of continuous functions over compacts is justified by Jackson's theorems. Third, better approximations require a higher number of basis functions. Fourth, to approximate more complex functions you need a higher

number of $(x, f(x))$ observations.

1.2 Hermite Polynomials

The Jacobi polynomials are defined to be orthogonal on a compact interval. The motivation for Hermite polynomials is to find a family of polynomials that is orthogonal on the real line. We can define Hermite polynomials recursively. Let H_k denote the Hermite polynomial of order k, then:

$$\begin{aligned} H_0(x) &\equiv 1 \\ H_1(x) &\equiv 2x \\ H_k(x) &\equiv 2xH_{k-1}(x) - 2(k-1)H_{k-2}(x) \end{aligned}$$

The Hermite polynomials are orthogonal on the real line with respect to the weight function:

$$w(x) = \mathrm{e}^{-x^2}$$

In Matlab, Hermite polynomials can be computed with the function `hemiteH`:

```
% visualize
figure;
hold on;
for k = 3:5
    fplot(@(x) hermiteH(k, x));
end
% weight function
w = @(x) exp(-x.^2);
% verify orthogonality
integral(@(x) w(x).*hermiteH(2, x).*hermiteH(1, x), -Inf, Inf
   )
integral(@(x) w(x).*hermiteH(2, x).*hermiteH(2, x), -Inf, Inf
   )
```

It is possible to show that:

$$\int_{-\infty}^{\infty} w(x)H_k(x)H_k(x)dx = 2^k k!\sqrt{\pi}$$

Thus, we can define Hermite polynomials that are orthonormal and with weight function $w(x) = 1$, by defining:

$$\begin{aligned} \tilde{H}_k(x) &= \sqrt{\frac{w(x)}{2^k k!\sqrt{\pi}}} H_k(x) \\ &= (2^k k!\sqrt{\pi})^{-\frac{1}{2}} \mathrm{e}^{-\frac{x^2}{2}} H_k(x) \end{aligned}$$

The family of polynomials $\left\{\tilde{H}_k : k \in \mathbb{N} \cup \{0\}\right\}$ is orthonormal.

```
% build the orthonormal hermite polynomials
orthoH = @(k, x) hermiteH(k, x)./((2^k*factorial(k)*sqrt(pi)
   .*exp(x.^2)).^(0.5));
```

```
% verify orthonormality (good way to debug)
integral(@(x) orthoH(2, x).*orthoH(3, x), -Inf, Inf)
integral(@(x) orthoH(3, x).*orthoH(3, x), -Inf, Inf)
```

Let's visualize the first few orthonormal Hermite polynomials:

```
f = figure;
for k = 1:6
    subplot(2, 3, k)
    fplot(@(x) orthoH(k, x), [-5 5]);
    grid on;
    legend(strcat("$H_", string(k), "(x)$"), 'Interpreter', '
       latex');
end
```

Figure 3 displays the first few orthonormal Hermite polynomials.

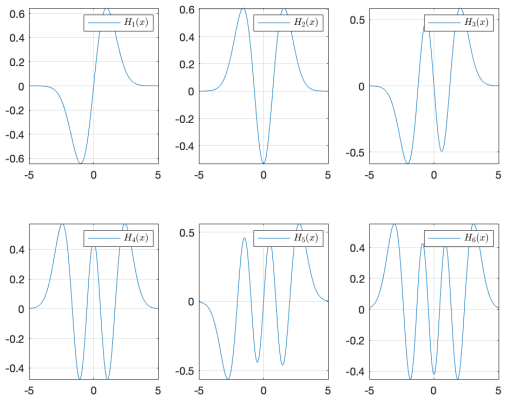

Figure 3: Orthonomal Hermite Polynomials.

1.2.1 Example

Let's approximate the density of the Cauchy distribution:

$$f(x; x_0, \gamma) = \frac{1}{\pi\gamma\left[1 + \left(\frac{x-x_0}{\gamma}\right)^2\right]}$$

To approximate f, we need to choose a basis of polynomials. In this example, we will use the orthonormal Hermite polynomials.

```
% consider the cauchy density
x0 = -2;
gamma = 1;
f = @(x) 1./((pi*gamma)*[1+((x-x0)/gamma).^2]);
% consider 5 points
x = linspace(-5, 5, 5)';
% select a basis of polynomials: orthonormal Hermite
   polynomials
% use 2 polynomials
```

```
p = @(x) [orthoH(0, x) orthoH(1, x)];
% visualize basis
fplot(p, [-5 5]);
% compute the matrices F, P and the generalized inverse of P'
   P
F = f(x);
P = p(x);
beta = pinv(P'*P)*P'*F;
% approximation for the function f
fhat = @(x) p(x)*beta;
% compare original to approximation
fplot(f);
hold on;
fplot(fhat);
grid on;
legend(["f", "$\hat{f}$"], 'interpreter', 'latex');
```

To improve the approximation, we need more observations of the function we want to approximate and to increase the number of basis polynomials.

```
x = linspace(-5, 5, 1000)';
% use 4 polynomials
p = @(x) [orthoH(0, x) orthoH(1, x) orthoH(2, x) ...
          orthoH(3, x) orthoH(4, x) orthoH(5, x) ...
          orthoH(6, x) orthoH(7, x) orthoH(8, x)];
F = f(x);
P = p(x);
beta = pinv(P'*P)*P'*F;
% approximation for the function f
fhat = @(x) p(x)*beta;
% compare original to approximation
fplot(f, [-5 5]);
hold on;
plot(x, fhat(x));
grid on;
legend(["$f$", "$\hat{f}$"], 'interpreter', 'latex');
```

1.3 Laguerre Polynomials

The Laguerre polynomials form a class of orthonormal polynomials on the non-negative real numbers, with the weight function $w(x) = \mathrm{e}^{-x}$. These polynomials can be defined recursively:

$$
\begin{aligned}
L_0(x) &\equiv 1 \\
L_1(x) &\equiv 1 - x \\
L_k(x) &\equiv \frac{(2k - 1 - x)L_{k-1}(x) - (k-1)L_{k-2}(x)}{k}
\end{aligned}
$$

The function `laguerreL` computes the Laguerre polynomial of order k at a point x:

```
% weight function
w = @(x) exp(-x);
% orthonormality
integral(@(x) w(x).*laguerreL(3, x).*laguerreL(2, x), 0, Inf)
integral(@(x) w(x).*laguerreL(3, x).*laguerreL(3, x), 0, Inf)
% orthonormal on the non-negative numbers with w(x) = 1
orthoL = @(k, x) laguerreL(k, x).*w(x).^0.5;
integral(@(x) orthoL(3, x).*orthoL(2, x), 0, Inf)
integral(@(x) orthoL(3, x).*orthoL(3, x), 0, Inf)
```

Let's visualize the first few orthonormal Laguerre polynomials:

```
% visualize
figure;
for k = 1:6
    subplot(2, 3, k);
    fplot(@(x) orthoL(k, x), [0 10]);
    grid on;
    legend(strcat("$L_", string(k), "(x)$"), 'Interpreter', '
       latex');
end
```

Figure 4 displays the first few orthonormal Laguerre polynomials.

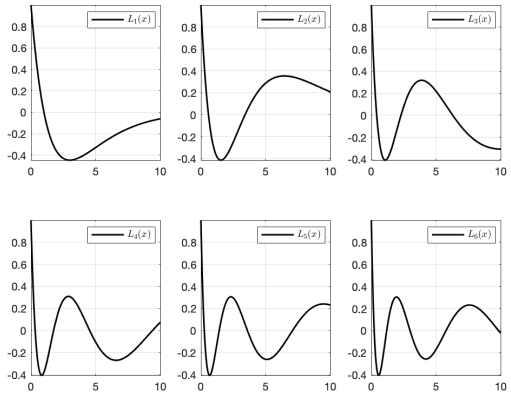

Figure 4: Orthonomal Laguerre Polynomials.

1.3.1 Example

Let's approximate the density of a Chi-squared distribution:

$$f(x,v) \equiv \begin{cases} \frac{x^{\frac{k}{2}-1}\mathrm{e}^{-\frac{x}{2}}}{2^{\frac{k}{2}}\Gamma\left(\frac{k}{2}\right)}, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

where v denotes the degrees of freedom of the distribution (also its mean in this case). We can use the Matlab function `chi2pdf` to evaluate the density. We will use the Laguerre polynomials to approximate the density, since the density of a Chi-squared is defined for non-negative numbers.

```
% Chi-squared density with 3 degrees of freedom
f = @(x) chi2pdf(x, 3);
% Create orthonormal Laguerre polynomials
w = @(x) exp(-x);
orthoL = @(k, x) laguerreL(k, x).*w(x).^0.5;
p = @(x) [orthoL(0, x) orthoL(1, x) orthoL(2, x)];
% Approximate function
x = [0:5]';
beta = pinv(p(x)'*p(x))*p(x)'*f(x);
fhat = @(x) p(x)*beta;
% Visualize
fplot(f, [0 10]);
hold on;
fplot(fhat, [0 10]);
grid on;
legend(["Chi-squared", "Approximation"]);
```

1.4 Piecewise Polynomials (Splines)

Polynomials are useful to approximate continuous functions, however the approximations often oscillate wildly. These oscillations are more apparent at the tails of the function we are approximating.

Consider the approximation of a simple function $f(x) = \frac{1}{1+x^2}$ over the interval $[-5, 5]$ using polynomials. The function we want to approximate is infinitely differentiable, but the approximation by polynomials will incur in a wild oscillation near the tails of the function. This example is known as Runge's example.

```
% Runge's example: continuous function on interval [-5, 5]
f = @(x) 1./(1+x.^2);
fplot(f, [-5  5]);
% Legrende polynomials
a = -5;
b = 5;
c = @(k) ((2.*k+1)./(b-a)).^0.5;
lp = @(k, x) c(k).*legendreP(k, -1 + 2.*(x-a)./(b-a));
% Approximation using 11 points
x = [-5:5]';
p = @(x) [lp(0, x) lp(1, x) lp(2, x) lp(3, x) lp(4, x) lp(5,
   x) ...
          lp(6, x) lp(7, x) lp(8, x) lp(9, x) lp(10, x)];
beta = pinv(p(x)'*p(x))*p(x)'*f(x);
fhat = @(x) p(x)*beta;
% Visualize
hold on;
plot(linspace(-5, 5)', fhat(linspace(-5, 5)'));
legend(["Original Function", "Approximation"]);
```

Figure 5 displays Runge's example. Notice that the approximation works well around zero, but gets increasingly worse as we move towards the ends of the intervals.

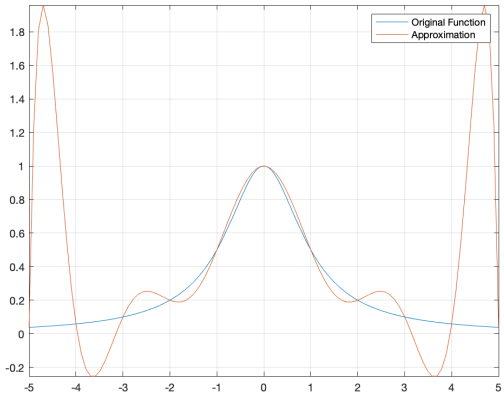

Figure 5: Runge's Example.

The idea of piecewise polynomials is that instead of using a single polynomial to approximate a function, we could use various different polynomials to approximate different parts of the function. We say that a function $s : [a, b] \mapsto \mathbb{R}$ is a **piecewise polynomial of degree k** if:

- The function s is continuous over $[a, b]$;
- There exists a finite number $n+1$ of points $\{\xi_i\}_{i=1}^{n+1}$, with $a = \xi_0 < \xi_1 < \cdots < \xi_n = b$, where s is a polynomial of degree at most k on each of the intervals $[\xi_{i-1}, \xi_i], i = 1, 2, \ldots, n$.

That is, s is a continuous function, but restricted to each interval it is also a polynomial of degree at most k. The points ξ_i are called **knots**. If the function s is differentiable up to its degree k minus 1, then we say it is a **spline function of degree k**. A spline is a piecewise polynomial that is as smooth as possible, without becoming a regular polynomial.

There are various functions that are splines. One of the simplest is the linear spline. Given a set of points $\{(x_1, y_1), \cdots, (x_m, y_m)\}$, where $x_1 = a$ and $x_m = b$, we can define s on each of the intervals $[x_i, x_{i+1}]$ by:

$$s(x) \equiv \frac{(x_{i+1} - x)y + (x - x_i)y_{i+1}}{x_{i+1} - x_i}, x \in [x_i, x_{i+1}]$$

The function s is defined as a sequence of lines connecting the intervals. Notice that the function is continuous, of degree $k = 1$, but not differentiable.

We will focus the discussion on a type of splines known as B-splines. The idea of B-splines is to write s as a linear combination of various polynomials of degree k, denoted by B_i^k:

$$s(x) = \sum_i \beta_i B_i^k(x) \text{ where } x \in [a, b]$$

Each of the B_i^k polynomials satisfy the property that they are nonzero only on a small interval, but are zero on the majority of $[a, b]$. The subscript i denotes the interval $[\xi_i, \xi_{i+1}]$

over which B_i^k is nonzero. The idea is to limit the impact of the polynomials on intervals far from where they are nonzero. An additional benefit is that B-splines incur in fewer calculations than other splines.

We can define the B_i^k polynomials by:

$$B_i^k(x) = \sum_{j=i}^{i+k+1} d_j (x - \xi_j)_+^k$$

$$d_j = \prod_{l=i, l\neq j}^{i+k+1} \frac{1}{\xi_l - \xi_j}$$

Where $(a)_+ \equiv \max(a, 0)$, and the terms $(x-\xi_j)_+^k$ are known as truncated power functions.

The subscript i indicates the interval $[\xi_i, \xi_{i+1}]$ over which the polynomial B_i^k is nonzero. However, the polynomial B_i^k is also nonzero over a slightly larger interval that extends after ξ_{i+1}. This is necessary so that the resulting function is continuous. The order of the polynomial k dictates how much longer the polynomial is extended. If the order is $k = 1$, then the polynomial extends over one more knot, that is, B_i^1 is nonzero on the interval $[\xi_i, \xi_{i+2}]$. If the order is $k = 2$, then the polynomial extends over two more knots, that is, B_i^2 is nonzero on the interval $[\xi_i, \xi_{i+3}]$. And so on for higher orders. The coefficients d_j are computed so that the polynomial B_i^k is zero outside the interval $[\xi_i, \xi_{i+k+1}]$.

The polynomials B_i^k are created for each of the knots ξ_i, for $i = 0, 1, 2, \cdots, n$. Notice that when we are at the final knots, we would need more points to the right of ξ_n to compute the coefficients d_j. In practice, we may not be able to generate these additional knots, so we need to truncate some of the coefficients to zero to generate the last few B_i^k.

We can code B-splines in Matlab:

```
% splineB.m
function b = splineB(knots, i, k)
parser = inputParser;
addRequired(parser, 'knots', @(x) size(x, 1) == 1 || size(x, 2) == 1);
addRequired(parser, 'i', @(i) mod(i, 1) == 0 && i >= 1);
addRequired(parser, 'k', @(k) mod(k, 1) == 0 && k >= 0);
parse(parser, knots, i, k);
knots = reshape(knots, [], 1);          % reshape to column vector
stop_at = min(i+k+1, length(knots));
ds = zeros(stop_at-i+1, 1);
for j = i:stop_at
    ds(j-i+1) = get_d(knots, i, j, k);
end
b = @(x) sum(ds.*(max(x - knots(i:stop_at), 0).^k));


function d = get_d(knots, i, j, k)
d = prod(1./(vertcat(knots(i:j-1), knots(j+1:min(i+k+1, length(knots)))) - knots(j)));
```

Visualize some of the B-splines:

```
% plot defaults
set(groot, 'DefaultFunctionLineLineWidth', 1.5);
set(groot, 'DefaultAxesColorOrder', [0 0 0]);
set(groot, 'DefaultAxesLineStyleOrder',' -|--|:|-.');
% define splines
knots = [0;1;2;3;4;5;6;7;8;9;10];
b1 = splineB(knots, 1, 1);
b2 = splineB(knots, 1, 2);
b3 = splineB(knots, 1, 3);
% visualize
fig = figure;
hold on;
fplot(b1, [-1 5]);
fplot(b2, [-1 5]);
fplot(b3, [-1 5]);
legend(["Linear", "Quadratic", "Cubic"]);
grid on;
```

Figure 6 displays examples of B-splines.

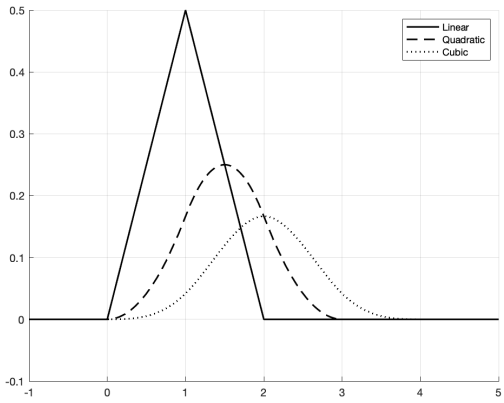

Figure 6: Linear, Quadratic and Cubic B-Splines.

We constructed splines starting at the knot 0. Observe that the linear spline extends from 0 to 2, but is zero elsewhere. The quadratic and cubic splines extend over a slightly longer interval. We can generate splines starting at other knots, and combine them to create an approximation for some function of interest.

1.4.1 Example

Let's analyze Runge's example again. Instead of using Legendre polynomials, we will now use the B-splines to approximate the function in Runge's example. We begin with a linear B-spline approximation:

```
% Runge's example: continuous function on interval [-5, 5]
f = @(x) 1./(1+x.^2);
% define B-splines of order 1 (linear)
```

```
knots = [-5:5]';
k = 1;
b1 = splineB(knots, 1, k);
b2 = splineB(knots, 2, k);
b3 = splineB(knots, 3, k);
b4 = splineB(knots, 4, k);
b5 = splineB(knots, 5, k);
b6 = splineB(knots, 6, k);
b7 = splineB(knots, 7, k);
b8 = splineB(knots, 8, k);
b9 = splineB(knots, 9, k);
b10 = splineB(knots, 10, k);
B = @(x) [b1(x) b2(x) b3(x) b4(x) b5(x) b6(x) b7(x) b8(x) b9(x) b10(x)];
% Notice that the splines do not work with vector inputs,
% so we will need for-loops to evaluate it.

% Approximate also using the 11 knots
P = zeros(length(knots), 10);
for i = 1:length(knots)
    P(i, :) = B(knots(i));
end
beta = pinv(P'*P)*P'*f(knots);
fhat = @(x) B(x)*beta;
% Visualize
fplot(f, [-5  5]);
hold on;
x = linspace(-5, 5);
fhats = zeros(length(x), 1);
for i = 1:length(fhats)
    fhats(i) = fhat(x(i));
end
plot(x, fhats);
legend(["Original Function", "Approximation"]);
ylim([0 1]);
grid on;
```

Observe that the approximation does not suffer from the problems of using a polynomial basis. The piecewise polynomials are localized, and have little effect on each other.

We can also use a cubic B-spline with the same knots, but we need more observations to approximate the function. We can also extend the first knots so that the first B-spline does not start at zero.

```
% Runge's example: continuous function on interval [-5, 5]
f = @(x) 1./(1+x.^2);
% define B-splines of order 1 (linear)
knots = [-7:5]';
k = 3;
b1 = splineB(knots, 1, k);
```

```
b2 = splineB(knots, 2, k);
b3 = splineB(knots, 3, k);
b4 = splineB(knots, 4, k);
b5 = splineB(knots, 5, k);
b6 = splineB(knots, 6, k);
b7 = splineB(knots, 7, k);
b8 = splineB(knots, 8, k);
b9 = splineB(knots, 9, k);
b10 = splineB(knots, 10, k);
B = @(x) [b1(x) b2(x) b3(x) b4(x) b5(x) b6(x) b7(x) b8(x) b9(x) b10(x)];
% Notice that the splines do not work with vector inputs,
% so we will need for-loops to evaluate it.

% Approximate also using the 11 knots
obs = linspace(-5, 5)';
P = zeros(length(obs), 10);
for i = 1:length(obs)
    P(i, :) = B(obs(i));
end
beta = pinv(P'*P)*P'*f(obs);
fhat = @(x) B(x)*beta;
% Visualize
fplot(f, [-5  5]);
hold on;
x = linspace(-5, 5);
fhats = zeros(length(x), 1);
for i = 1:length(fhats)
    fhats(i) = fhat(x(i));
end
plot(x, fhats);
legend(["Original Function", "Approximation"]);
ylim([0 1]);
grid on;
```

1.5 Approximating Functions of Many Variables

We can use the same methodology to approximate a function f of many variables. In this case, $f : \mathbb{R}^n \mapsto \mathbb{R}$, so that x is now a vector of dimension n. Let's consider a basis of power functions to approximate f. Let $p_k(x)$ denote a power function of degree k for the vector x. It consists of all interaction terms between the elements of x, such that the

sum of the exponents of each element is equal to k. For example:

$$p_0(x) = 1$$
$$p_1(x) = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$
$$p_2(x) = \begin{pmatrix} x_1^2 \\ x_1 x_2 \\ x_2^2 \end{pmatrix}$$
$$p_3(x) = \begin{pmatrix} x_1^3 \\ x_1^2 x_2 \\ x_1 x_2^2 \\ x_2^3 \end{pmatrix}$$

Given a choice for the highest order k of the polynomials, we can define $p(x)$ as before. For example, if we choose $k = 2$:

$$p(x) = \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \end{pmatrix}$$

Then, given a set of observations for x and $f(x)$, we can proceed as in Equations 1, 2, 3 and 4.

1.6 Matlab Functions

There are a few functions in Matlab that help with interpolation. The function `interp1` provides interpolation for 1-dimensional data. To use it, we must supply observations $\{(x, f(x))\}$, a set of points where we want to evaluate the interpolation and the method of interpolation (default is linear).

```
% Runge's example: continuous function on interval [-5, 5]
f = @(x) 1./(1+x.^2);
x = [-5:5];
eval_at = linspace(-5, 5);
fhat = interp1(x, f(x), eval_at, 'linear');
% Visualize
fplot(f, [-5  5]);
hold on;
plot(eval_at, fhat);
legend(["Original Function", "Approximation"]);
ylim([0 1]);
grid on;
```

We can also use piecewise cubic splines with the method `'pchip'`. There are also other methods available, see the documentation for a complete list.

The function `interp2` allows for interpolation of 2-dimensional data, and works in a similar way to `interp1`. The function `interpn` extends the interpolation to n-dimensional data.

2 Integration

We will discuss three methods for computing integrals numerically: Newton-Cotes, Quadrature and Monte Carlo.

2.1 Netwon-Cotes

We want to compute the following integral:

$$\int_a^b f(x)dx$$

Assume that we can compute the value of f at a set of equally spaced points $x_0 = a, x_1, x_2, \ldots, x_n = b$. The idea of Newton-Cotes is to interpolate the function with Lagrange polynomials and the integrate the polynomial. The approximation yields:

$$f(x) \approx \sum_{i=0}^{n} f(x_i) l_i(x)$$

Where l_i is a Lagrange polynomial, and the weights multiplying the polynomials only depend on the points $x0, \ldots, x_n$ and on f. This approximation can now be integrated:

$$\begin{aligned}\int_a^b f(x)dx &\approx \int_a^b \left(\sum_{i=0}^{n} f(x_i) l_i(x)\right) \\ &= \sum_{i=0}^{n} f(x_i) \underbrace{\int_a^b l_i(x)dx}_{w_i} \\ &= \sum_{i=0}^{n} f(x_i) w_i\end{aligned}$$

If we use a Lagrange polynomial of order 1 (linear), then we obtain the trapezoidal rule, and the approximation of the integral is simply:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} \frac{f(x_{i-1}) + f(x_i)}{2}(x_i - x_{i-1})$$

Since the points are equally spaced, we can simplify the equation above. Let $\Delta x \equiv x_i - x_{i-1}$, then:

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2}\left[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n)\right]$$

If we use a higher order Lagrange polynomial to approximate f, then we obtain different weights w_i.

Example:

```
1 f = @(x) x.^2;
2 x = [1:5];
3 delta_x = x(2) - x(1);
4 int_f = (2*sum(f(x)) - f(x(1)) - f(x(end)))*(delta_x/2);
```

The trapezoidal rule is implemented in Matlab in the function `trapz`. It assumes points are equally spaced with space equal to unit.

```
x = [1:5];
% equally spaced points, with space = 1
trapz(f(x))
% not equally spaced
x = [1;1.5;2;2.1;3;3.8;5];
trapz(x, f(x))
```

2.2 Quadrature

The Quadrature method extends Newton-Cotes method by choosing the weights w_i in a special manner. The idea is to have weights such that the approximation of the integral of f is actually an equality when f is a polynomial of some order. Matlab implements a modified Quadrature method on the function `integral`. This modified method also works when the integrand diverges.

To use `integral`, we need to pass a function handle to the function we want to integrate, and the bounds of integration.

```
f = @(x) exp(-x);
% integrate over [0 1]
integral(f, 0, 1)
% integrate over [0, infinity)
integral(f, 0, Inf)
% integrate symmetric function over [0 2*pi]
integral(@sin, 0, pi)
integral(@sin, 0, 2*pi)
```

2.3 Monte Carlo

The idea of Monte-Carlo integration is to use the law of large numbers to compute the integral of f. Consider a random variable Z with a density ϕ defined on the real line. Then, we can compute the integral of f in the following way:

$$\begin{aligned}\int_{-\infty}^{+\infty} f(x)dx &= \int_{-\infty}^{+\infty} \frac{f(x)}{\phi(x)}\phi(x)dx \\ &= \mathbb{E}_Z\left[\frac{f(Z)}{\phi(Z)}\right]\end{aligned}$$

For example, we could have ϕ be the density of the standard normal distribution. Now, we can simulate an i.i.d. sample from this density, and approximate the integral via:

$$\int_{-\infty}^{+\infty} f(x)dx \approx \frac{1}{n}\sum_{i=1}^{n}\frac{f(x_i)}{\phi(z_i)}$$

The term on the right-hand side will approximate the true value of the integral with probability one due to the law of large numbers. Example:

```
f = @(x) exp(-x.^2);                    % integral = (pi)^0.5
% generate values from the standard normal
```

```
x = normrnd(0, 1, 10, 1);
disp(mean(f(x)./normpdf(x)))
% increase number of observations
x = normrnd(0, 1, 1000000, 1);
disp(mean(f(x)./normpdf(x)))
```

If the integral is defined over a finite interval, say $[a, b]$, then we can use the uniform distribution over the same interval to generate the i.i.d. sample of x_i's. In this case, let $\mu(x) = \frac{1}{b-a}\mathbb{1}_{[a,b]}(x)$ represent the uniform density over $[a, b]$. Then:

$$\begin{aligned}\int_a^b f(x)dx &= \int_a^b \frac{f(x)}{\mu(x)}\mu(x)dx \\ &= \mathbb{E}_U\left[\frac{f(U)}{\mu(U)}\right] \\ &\approx \frac{1}{n}\sum_{i=1}^{n}\frac{f(x_i)}{\mu(x_i)} \\ &= \frac{b-a}{n}\sum_{i=1}^{n} f(x_i)\end{aligned}$$

Example:

```
% integrate over [0, 5]
f = @(x) x.^2;
% generate random values over interval [0, 5]
x = unifrnd(0, 5, 10, 1);
int_f = 5*mean(f(x))
% integral approximation converges as we increase number of
   points
% 100 observations
x = unifrnd(0, 5, 100, 1);
int_f = 5*mean(f(x))
% 1000 observations
x = unifrnd(0, 5, 1000, 1);
int_f = 5*mean(f(x))
% 10000 observations
x = unifrnd(0, 5, 10000, 1);
int_f = 5*mean(f(x))
```

This method of integration is very useful for functions of many variables and can be easily made to run in parallel.

3 Symbolic Math

For this part you will need to install the Symbolic Math Toolbox. We can use the toolbox to do analytical computations, like differentiation, integration, simplification, transformations, and equation solving.

3.1 Creating Symbolic Variables and Expressions

To create the symbolic variables we use the function `sym`.

```
% clear the workspace
clear all
% create a symbolic number
sym(1/3)
% expressions involving symbols are displayed unindented
sym(pi)
% we can do computations with symbols, and the computations are
% analytical
sin(pi)
sin(sym(pi))
% create a variable
sym('x')                                % unasigned so cannot be reused
sym('x')*sym(pi)
% create a variable and assign it
x = sym('x');
x + 2
% verify the class of x
whos x
```

We can create multiple variables at once with the function `syms`.

```
clear all
syms a b c x y
% combine symbolic variables to create expressions
z = (exp(a) + log(b) - c)/(x+y)
```

The function `syms` is a shorthand function for invoking `sym` multiple times.

We can use `sym` to create vectors and matrices of symbolic variables.

```
% pass sym the name and dimension
A = sym('a', [3, 1])
% create a matrix
X = sym('x', [2, 2])

% create symbolic matrix from numeric matrix
Y = [0.25; 1/3; pi; 0.1245];
Y = sym(Y);
```

We can use `syms` with `sym` to create multiple symbolic variables where the names are numbered:

```
clear all
% create symbolic variables a
syms(sym('a', [5, 1]))
syms(sym('x', [2, 2]))
```

Symbolic expressions can be combined:

```
clear all
syms x y z
f = x^2 + y^2 + z^2;
g = exp(f^2)/2
% symbolic expressions can be reused by restating them with
   syms
syms f
f
g
```

3.2 Creating Symbolic Functions

Symbolic functions are used to perform the analytical computations. We can create them with `syms`:

```
clear all
% create a function f of two variables, x and y
syms f(x, y)
% x and y are symbolic variables
whos x y
% f is a symbolic function
whos f
% assign an expression to f
f(x, y) = x^2 + x*y + y^2;
% we can evaluate the function
f(0, 0)
f(1, 0)
f([1, 2, 3], [-1, -2, -3])
```

3.3 Derivatives

We can use the function `diff` to differentiate a symbolic function. The function `diff` returns a symbolic function with the same arguments as the function being differentiated.

```
syms f(x, y)
f(x, y) = x^3 + 2*x*y + y^2
% differentiate with respect to x
dfdx = diff(f, x)
% evaluate the derivative
dfdx(1, 0)
% differentiate with respect to x again
dfdx2 = diff(dfdx, x)
% equivalent
dfdx2 = diff(f, x, 2)
% differentiate with respect to y and x
dfdyx = diff(f, y, x)
% differentiate with respect to x and y
```

```
dfdxy = diff(f, x, y)

% can also differentiate expressions
diff(x^2, x)
diff(x^2, x, 2)
```

3.4 Integral

The function `int` is used to integrate symbolic expressions:

```
clear all
% specify function
syms f(x)
f(x) = exp(-x^2)
% integrate from -infinity to infinity
int_f = int(f)
% evaluate integral
int_f(1)
% integrate from 0 to 1
int(f, 0, 1)
% specify a multivariate function
syms f(x, y)
f(x, y) = x^3 + x^2*y + x*y^2 + y^3
% integrate with respect to x
int(f, x)
% integrate with respect to y
int(f, y)
% if variable not specified it is implied from first variable
int(f)                                    % same as int(f, x)
   in this case
% another multivariate function
syms f(x, y) n
f(x, y) = x^n + y^n
int(f, x)
int(f, n)

% if function cannot be integrated, the integral is left
   unresolved
int(sin(sinh(x)))
```

3.5 Solve Equations

We can use the function `solve` to solve symbolic equations:

```
clear all
% solve equation on one variable
syms x
sol = solve(-x^2 + 2*x + 3 == 0, x)
```

```
% sol is still symbolic
whos sol
% if the right-hand side is not specified, then == 0 is implied
sol = solve(-x^2 + 2*x + 3, x)
% solve equations with several variables
syms x y
solve(-3*x^2 + 2*x*y + y^2 == 0, y)
% solve a system of equations
syms x y z
[sol_x, sol_y, sol_z] = solve(z == 4*x, x == y/2, z == x^2 + y^2)
```

Matlab implements solvers for more complex equations, as systems of differential equations. For a complete description, refer to the Equation Solving reference page.

3.6 Simplify Expressions

The function `simplify` can be used to simplify symbolic expressions:

```
clear all
syms x y
f = sin(x)^2 + cos(x)^2
% simplify expression
simplify(f)
% another example
g = (sym(1) + sqrt(5))/2                % need one symbol at least
simplify(g^2 - g - 1)
```

There is no clear way of simplifying a mathematical expression, and the results will vary with the way the symbolic expressions are built.

The function `expand` can be used to expand symbolic expressions:

```
% expand polynomial
f = (x-2)*(x-3)*(y-2)
f_expanded = expand(f)
% simplify expanded expression
simplify(f_expanded)
% expand expression
expand((x-1)*(x+1))
```

The function `factor` can be used to factor expressions:

```
f = (x-2)*(x-3)*(y-2)
f_expanded = expand(f)
factor(f_expanded)
```

The function `horner` factors polynomials to a form that is efficient for computation.

```
horner(f)
```

3.7 Substitutions

While symbolic functions can be evaluated as a regular function, symbolic expressions are not functions, and to be evaluated we need to use the function `subs`.

```
syms x y
f = (x-2)*(x-3)*(y-2)
f_expanded = expand(f)
% substitute x for 3
subs(f_expanded, x, 3)
% substitute y for 0
subs(f_expanded, y, 0)
% substitute a variable for another
subs(f_expanded, y, x)

% also applies to matrices
% create matrix with notation xij
X = sym('x%d%d', [3, 3])
subs(X, 'x11', 5)
```

3.8 Plotting

To plot symbolic expressions and functions we can use the functions:

- `fplot` for 2-D plots
- `fplot3` for 3-D curves
- `fsurf` for surfaces
- `fcontour` for countor plots

There is also a function for plots on polar coordinates (`ezpolar`) and one for mesh plots (`fmesh`).

3.9 Assumptions

The symbolic variables are by default assumed to be complex variables. We can use the function `assumptions` to verify it.

```
% create variable
syms z
% verify it is complex
% should return empty symbolic object
assumptions(z)
```

We can add assumptions to a symbolic variable with the function `assume`.

```
syms x
assume(x >= 0)
% assumptions impact the behavior of other functions
syms f(x)
```

```
f(x) = x^2
fplot(f)
% solve for roots
solve(-2*x^2 + 3*x + 3 == 0, x)
```

Additional assumptions can be added to the same symbolic variable with the function `assumeAlso`.

To clear assumptions restate the symbolic variable or use `assume` with the `'clear'` option:

```
% clear assumptions
assumptions(x)
syms x
assumptions(x)                                  % now empty

assume(x, 'integer')
assumptions(x)
assume(x, 'clear')
assumptions(X=x)
```

4 Assignment

Problem 1 *Consider a deterministic growth model, where an agent decides between consumption (c_t) and investment in capital (k_t), while maximizing his utility. We can write this problem as:*

$$\max \sum_{t=0}^{\infty} \beta^t U(c_t)$$
$$\text{subject to } \begin{cases} k_{t+1} = k_t^{\alpha} - c_t + (1-\delta)k_t, \forall t >= 0 \\ k_0 > 0 \end{cases}$$

Write the problem as a Bellman equation. You should obtain an equation similar to:

$$V(k) = \max_{k'} f(k, k') + \beta V(k')$$

Problem 2 *(continuation of previous problem) Consider $U(c;\sigma) = \frac{c^{1-\sigma}-1}{1-\sigma}$. Obtain the Euler equation for this problem in terms of the consumption c.*

Problem 3 *(continuation of previous problem) (Optional) Use the Symbolic toolbox to obtain the Euler equation for this problem in terms of the consumption c.*

Problem 4 *(continuation of previous problem) Assume we are in a steady state ($c = c' = c^*$ and $k = k' = k^*$), then use the Euler equation to compute the steady state value of k.*

Problem 5 *(continuation of previous problem) Solve the problem by Value Function Iteration. Consider $\sigma = 2$, $\beta = 0.95$, $\delta = 0.1$ and $\alpha = 0.33$. Use the steady state value of k to create a grid for the possible values of k, say 100 points between $0.25k^*$ and $1.75k^*$. Start with a guess for V over the grid, for example $V(k) = 0$ for all k in the grid. Use the Matlab minimization function to solve for k. You may want to add the constraint that c should always be positive.*

Problem 6 *(continuation of previous problem) Plot the value for different capital levels. Interpret.*

Problem 7 *(continuation of previous problem) Plot the optimal policy function (choice of k' given k). Plot a 45 degree line with the policy function. Interpret in terms of a steady state.*

References

Abramowitz and Setgun (1964). *Handbook of Mathematical Functions: with Formulas, Graphs, and Mathematical Tables*. Dover Publications. URL: https://isbnsearch.org/isbn/0486612724.