Métodos Numéricos con Matlab

Mishelle Seguí[1]

[1]mishelle@gmx.de
Esta es una primera versión, por lo que se agradece cualquier comentario o sugerencia.

Introducción

El propósito de estas notas es dar una introducción a los sistemas de software interactivo Matlab (**Mat**rix **lab**oratory) y Octave, los cuales son equivalentes, con la ventaja de que Octave es un software libre[2]. Esto con el objetivo de que posteriormente los alumnos puedan crear sus propios programas de programación dinámica, los cuales pueden incluir gráficas, regresiones, o cualquier computo numérico que se desee.

Dado que Matlab está diseñado para resolver problemas numéricamente, es decir, que tienen un aritmética de precisión finita, nos va a producir soluciones aproximadas y no exactas. Sin embargo, esto no resta credibilidad a este programa, ya que es uno de los programas más ultizados entre los economistas que realizan calibraciones, y es muy fácil de usar.

[2]Si tienen Windows, lo pueden bajar de $http://sourceforge.net/project/showfiles.php?group\_id = 2888$ y se llama "Octave 2.1.73 for Windows Installer".

Temario

1. Introducción a Matlab: Sintáxis, cálculos sencillos y gráficas básicas
2. Integración y diferenciación numérica
3. Resolver un sistema de ecuaciones no lineales con métodos de gradiente (Newton, secante)
4. Resolver una ecuación en diferencias con métodos numéricos (Gauss-Seidel).
5. Programación dinámica numérica:
    - Iteración de la función de valor
    - Programación dinámica determinística
    - Programación dinámica estocástica
6. Otros métodos de la aproximación: polinomios, interpolación
7. Aplicaciones económicas

1 Introducción a Matlab

1.1 Sintáxis y cálculos sencillos

Antes que nada señalemos que cuando una quiere iniciar Matlab, éste puede tardar unos segundos en abrir (de 10 a 15 seg.).

Algunas cosas que uno debe de saber de Matlab son:

- Las minúsculas y las mayúsculas no son equivalentes
- Un punto y coma al final del comando hará que no se vea en la pantalla el resultado
- Matlab utiliza paréntesis () y corchetes [], los cuales no son intercambiables
- Las teclas con las flechas hacia arriba y hacia abajo ayudan para desplazarse entre los comandos previos que se realicen
- Para seleccionar ayuda, hay que teclear `help` seguido del comando del cual se requiere la ayuda
- Matlab se puede cerrar tecleando `quit` o `exit`

Explicar las barras de herramienta

Notemos que en la ventana de comandos, cuando asignamos un nombre a una expresión, es sólo un nombre y NO representa una variable matemática, como lo hace Maple.

Veamos ahora unos ejercicios de símbolos y puntuación[3] para realizarlos en la computadora:

Ver ejercicios1.1.pdf

$$3+7$$
$$a = sqrt(3)$$
$$b = \sin(pi)$$
$$c = [1:10]$$
$$d = 1:2:8$$
$$e = [1\ \ 4];$$
$$e'$$
$$e.*e$$
$$f = [1\ \ 4; 2\ \ 5]$$
$$f\hat{}2$$

[3] http://www.indiana.edu/~statmath/math/matlab/gettingstarted/

Recordemos ahora algunas propiedades básicas de las matrices. Digamos que tenemos el vector fila $g = [1 \;\; 2 \;\; 3]$ y el vector columna $h = [4; 5; 6]$, entonces podemos como los vectores son conformables, podemos multiplicar $g * h$, o también podemos computar el producto interno con la función $dot(g, h)$.

Hacerlo y comprobar que da 32

Recordar que el producto $a * a$ no está definido, dado que las dimensiones no son compatibles, con lo cual si lo tecleamos, Matlab nos arroja un error.

Matlab tiene muchas funciones matemáticas para las matrices, por ejemplo $\exp(g)$, $\log(h)$, $sqrt(g), abs(g), \sin(g)$,etc.

Por default Matlab nos muestra el resultado con 4 dígitos decimales, aunque siempre los guarda completos en la memoria y computa el equivalente a 14 dígitos decimales. Si deseamos ver dichos dígitos, escribimos `format long,` por ejemplo $sqrt(g)$. Si deseamos regresar al formato anterior, sólo escribimos `format.`

Notemos también que Matlab nos muestra los números muy grandes y muy pequeños en notación de exponencial, con una factor de 10 denotado por `e`.Por ejemplo, $2\hat{}(-24)$.

Para redondear números, Matlab tiene los comandos *round*, fix, *ceil* y $floor$, por ejemplo $round(g)$.

Matlab también puede realizar varias funciones de análisis de datos, por ejemplo sumar, sacar la media,mediana, desviación estándar o ver la diagonal: $sum(g)$, $mean(g), median(g)$, $std(g), diag(f)$.

Veamos a continuación algunas funciones de algebra lineal que Matlab realiza. Por ejemplo, recordando que h es un vector columna, tal vez quisieramos resolver el sistema lineal $B * x = h$, en donde $B = [-3 \;\; 0 \;\; 1; 2 \;\; 5 \;\; -7; -1 \;\; 4 \;\; 8]$, a través del operador \, de la forma $x = B\backslash h$. Recordemos que la división inclinada a la derecha a/b, significa $\frac{a}{b}$, pero inclinada a la izquierda $a\backslash b$, significa $\frac{b}{a}$.

Ver que el resltado es -1.3717, 1.3874, -0.1152.

Uno podría comprobar el resultado a través de la norma euclideana del residuo: $norm(B*x - h)$, que es practicamente cero. Los valores propios (eigenvalues) se pueden encontrar usando `eig`, por ejemplo $j = eig(B)$.

Si queremos saber el tamaño de la matriz, simplemente escribimos $size(B)$. Si la queremos transponer ponemos B'.

Como ya vimos podemos generar una secuencia se números a través de los dos puntos, como en el caso c, el cual también lo podemos observar sin los corchetes. Notamos que en

este caso va del 1 al 10 en valores de uno, pero ¿cómo hacemos para que incrementen on otras unidades? Esto es muy sencillo, sólo se incorporan los incrementos especificandolo como $m : s : n$, en donde m es el valor con el que se desea inicializar, s es el paso con el cual se desea incrementar, y n es el valor final. Por ejemplo, $k = 1 : 3 : 10$ ó $l = 1 : 0.2 : 10$.

En Matlab también podemos hacer matrices más grandes usando otras que ya teníamos, por ejemplo:

$C = [B; g]$.

Si deseamos sólo un elemento de la matriz, lo señalamos como $C(i, j)$, en donde i es la fila y j es la columna, por ejemplo $C(2, 3)$, que nos de -7. Pero ¿qué tal si queremos una submatriz? Entonces sólo seleccionamos $C(i1 : i2; j1 : j2)$,por ejemplo, $C(2 : 3; 1 : 2)$. En Matlab también podemos escoger sólo las columnas o las filas, por ejemplo $C(:, j)$ es la columna j de C, y $C(i, :)$ es la fila i.

Matlab también tiene por default algunas matrices, como la identidad y la de ceros y unos, a través de `eye, zeros` y `ones.` Por ejemplo, $I3 = eye(3, 3)$, $Y = zeros(3, 5)$, $Z = ones(2)$. Si lo que deseamos es una secuencia de sólo un número repetido varias veces, por ejemplo si el número cero lo queremos repetido 3 veces, escribimos $x(3) = 0$.

Posteriormente veremos que es útil generar valores aleatorios a través de los comandos `rand` y `randn`, los cuales vienen de la distribución uniforme [0,1] y de la normal (0,1), respectivamente. Por ejemplo, $F = rand(3)$ y $G = randn(1, 5)$.

Matlab puede computar el infinito, a través del comando `inf`, sin embargo recordemos que no podemos dividir inf/inf, porque matemáticamente no está definido y en la pantalla de Matlab aparecería `NaN`(Not a Number).

Por último veamos que Matlbab, como muchos lenguajes de programación, trabaja con bucles (*loops).* Un ejemplo es:

$$1+\frac{1}{1+\frac{1}{1+\frac{1}{1+1}}}$$

Dicha solución se puede obtener generando el bucle

$m = 2$;

for $n = 1 : 3$, $m = 1 + \frac{1}{m}$;

end

m

Si deseamos ver todas nuestras variables escribimos `who` y se desplegaran todos los nombres de nuestras variables. Aunque si se desea mayor información acerca de las variables, escribimos `whos.`

1.2 Gráficas

Las gráficas más sencillas de hacer son las gráficas de puntos en el plano cartesiano. Por ejemplo,

$$x = [0; 3; 6.1; 7];$$

$$y = [3; 8; 10; 9.2];$$

$$plot(x, y)$$

Nótese que por default, Matlab junta los puntos con líneas rectas, pero para tener sólo los puntos ponemos

$$plot(x, y, 'o')$$

También podemos graficar una función con los puntos de x, por ejemplo,

$$plot(x, sin(x))$$

Podemos también dar no sólo puntos, sino intervalos, por ejemplo,

$$x = (0 : .1 : 2 * pi);$$

$$y = sin(x);$$

$$plot(x, y)$$

ó

$$x = (-5 : .1 : 5);$$

$$y = x./(1 + x.\hat{}2);$$

$$plot(x, y)$$

Para ver las diferentes opciones que tenemos para gráficar, nos podemos ir a la ayuda, donde ahí nos señalan cómo poner diferentes colores, marcadores y tipos de líneas. Veamos algunos ejemplo:

$$plot(x, y, 'g-')$$

$$plot(x, y, 'p', 'MarkerSize', 10)$$

Podemos añadirle a las gráficas título y nombre a los ejes, por ejemplo

$$plot(x, y), ...$$

$$title('Gráfica\_1'), ...$$

$$xlabel('eje\_x')$$

El comando `hold on` lo usamos para poner en una misma gráfica, otra gráfica, por ejemplo usando el ejemplo anterior, posteriormente escribimos:

$$hold\ on$$

$$v =$$

$$w =$$

$$plot(v, w)$$

`Subplot` hace que aparezcan varias gráficas en una misma ventana.
Realizemos ahora algunos ejercicios de gráficas[4]

$$x = -10 : 0.5 : 10;$$
$$y = x.\hat{}2;$$
$$plot(x, y)$$
$$t = 0 : 0.1 : 2 * pi;$$
$$x = cos(t);$$
$$y = sin(t);$$
$$plot(x, y)$$
$$t = 0 : pi/5 : 2 * pi;$$
$$u = cos(t);$$
$$v = sin(t);$$
$$figure$$
$$plot(u, v)$$
$$plot(x, y, 'r-', u, v, 'b*:')$$

$$figure$$
$$subplot(1, 2, 1)$$
$$plot(x, y)$$
$$title('Fino')$$
$$subplot(1, 2, 2)$$
$$plot(u, v)$$
$$title('Tosco')$$

Ver ejercicios1.2.pdf

[4] idem

2 Integración y diferenciación numérica

En este capítulo usaremos métodos numéricos para resolver problemas de cálculo y ecuaciones diferenciales. Veremos algunas técnicas numéricas para encontrar derivadas e integrales definidas.

Por ejemplo, cómo calculamos el área de un triangulo usando Matlab. Lo que podemos empezar a hacer es un archivo .m, en el cual definamos nuestro objetivo:

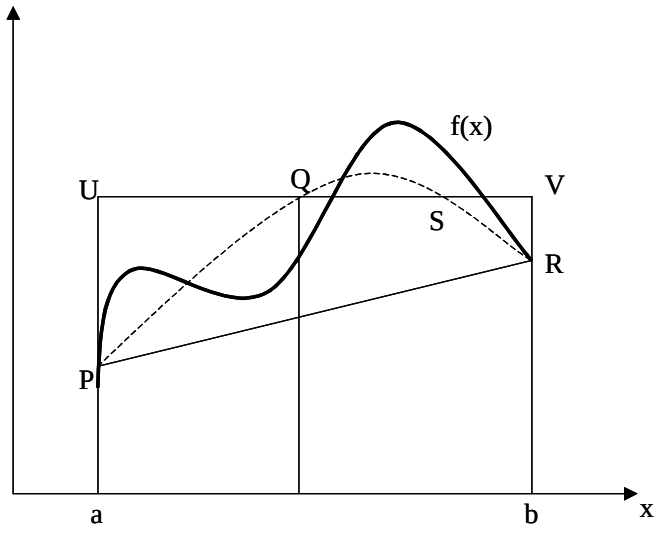
```
function A=Atri(b,h)
A=b*h/2;
```

Entonces en la ventana de Matlab sólo escribimos por ejemplo `Atri(2,5)` para saber el área de un triángulo de base 2 y altura 5.

2.1 Integración numérica

El problema general en la integración numérica, también llamado cuadratura, es el de computar $\int_D f(x)dx$ en donde $f : R^n \to R$ es una función integrable sobre el dominio $D \subset R^n$. Todas las fórmulas de integración numérica usan un número finito de evaluaciones del integrando, f, y usan una suma ponderada de estos valores para aproximar $\int_D f(x)dx$. Existen varios métodos, aunque a veces los más sencillos no son tan eficientes y los más complejos tardan mucho y necesitan varios requisitos. Nosotros no veremos los métodos muy complejos.

Las fórmulas de cuadratura de las cotas de Newton evalúan a f en un número finito de puntos, usan esta información para construir una aproximación polinomial por pedazos de f, e integran esta función f para aproximar $\int_D f(x)dx$.



Gráfica 2.1. Reglas de las cotas de Newton

En la gráfica 2.1 la función f pasa por los puntos P, Q y R. La integral $\int_a^b f(x)dx$ es el area bajo la la función f y el eje de las abscisas. Podemos observar tres aproximaciones directamente. La caja $aUQVb$ aproxima a f con una función constante de f en Q, que es el punto medio de $[a, b]$. El trapezoide $aPRb$ aproxima a f con una línea recta a través de los puntos P y R. Por último, el área bajo la curva punteada $PQSR$ aproxima a f con una parábola a través de P, Q y R. Estas aproximaciones están basadas en uno, dos y tres evaluaciones de f, respectivamente, las cuales son usadas para computar polinomios de orden uno, dos y tres. Este enfoque nos lleva a las fórmulas cuadráticas de las cotas de Newton.

Antes de ver estas fórmulas de las cotas de Newton y las propiedades de los errores, recordemos que es el teorema de Taylor.

Teorema de Taylor

Es el teorema más utilizado en el análisis numérico y veamoslo en sus versiones en R y en R^n.

Teorema 2.1. (Teorema de Taylor en R) Si $f \in C^{n+1}[a, b]$ y $x, x_0 \in [a, b]$, entonces

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x - x_0)^2}{2!}f''(x_0)$$

$$+ ... + \frac{(x - x_0)^n}{n!}f^{(n)}(x_0) + R_{n+1}(x)$$

en donde

$$R_{n+1}(x) = \frac{1}{n!}\int_{x_0}^{x}(x-t)^n f^{(n+1)}(t)dt = \frac{(x-x_0)^{n+1}}{(n+1)!}f^{(n+1)}(\xi)$$

xi

para $\xi \in [x, x_0]$

El teorema de Taylor dice, esencialmente, que uno puede usar información de las derivadas en un sólo punto para construir una aproximación polinomail de una función en un punto.

Teorema 2.2. (Teorema de Taylor en R^n) Sea $f : R^n \to R$ y es C^{k+1}, entonces para $x^0 \in: R^n$,

$$f(x) = f(x^0) + \sum_{i=1}^{n}\frac{\partial f}{\partial x_i}(x^0)(x_i - x_i^0) + \frac{1}{2!}\sum_{i=1}^{n}\sum_{j=1}^{n}\frac{\partial^2 f}{\partial x_i \partial x_j}(x^0)(x_i - x_i^0)(x_j - x_j^0)$$

$$+... + \frac{1}{k!}\sum_{i_1=1}^{n}...\sum_{i_k=1}^{n}\frac{\partial^k f}{\partial x_{i1}...\partial x_{i_k}}(x^0)(x_{i_1} - x_{i_1}^0)...(x_{i_k} - x_{i_k}^0) + \Phi(||x - x^0||^{k+1})$$

2.1.1 Regla del punto medio

Esta es la fórmula de cuadratura más sencilla, y viene dada por

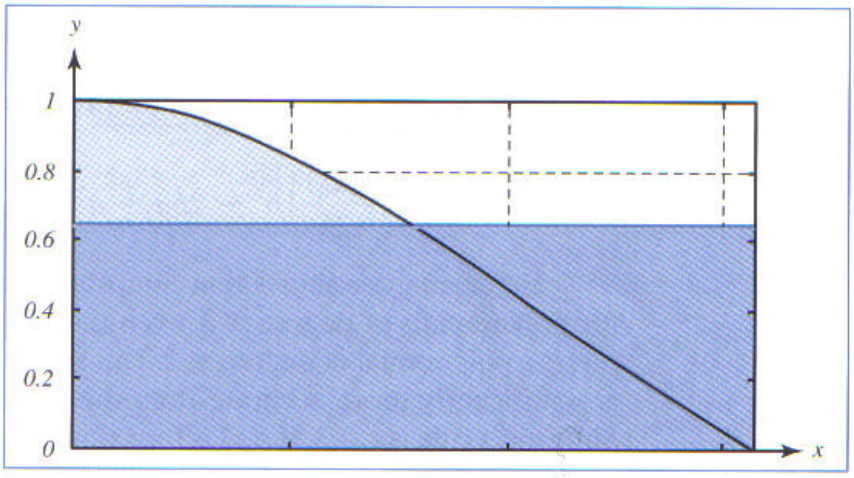
$$\int_a^b f(x)dx = (b-a)f(\frac{a+b}{2}) + \frac{(b-a)^3}{24}f''(\xi)$$

para alguna $\xi \in [a, b]$. Como haremos también más adelante, los primeros términos comprenden la regla de integración y el último término es el error de dicha regla. Esta regla es un claro ejemplo de una *regla abierta*, que es una regla que no utiliza los puntos extremos.

Por ejemplo, si queremos integrar la función $S = \int_0^\pi \frac{\sin(x)}{x}dx$ con la regla del punto medio, vamos a obtener

$$\int_0^\pi \frac{\sin(x)}{x}dx \approx \pi\frac{\sin(\frac{\pi}{2})}{\frac{\pi}{2}} = \pi\frac{1}{\frac{\pi}{2}} = 2$$

Vemos en la siguiente gráfica el verdadero valor del área con aquel encontrado con esta regla.



Gráfica 2.2. Area dada por la integral S y la aproximación usando la regla del punto medio

Sin embargo, esta regla es poco precisa para aproximar el valor deseado, por lo que es más conveniente romper el intervalo $[a, b]$ en pequeños intervalos, aproximar la integral en cada uno de estos pequeños intervalos y sumar estas aproximaciones. El resultado es una *regla compuesta*. Sea $n \geq 1$ el número de intervalos y definamos a h como $h = \frac{(b-a)}{n}$ y $x_j = a + (j - \frac{1}{2})h$, para $j = 1, 2, ..., n$. Entonces la regla compuesta del punto medio es:

$$\int_a^b f(x)dx = h\sum_{j=1}^{n} f(x_j) + \frac{h^2(b-a)}{24}f''(\xi)$$

para alguna $\xi \in [a, b]$. Notemos que el error es proporcional a h^2, es decir, que si doblamos el número de intervalos, esto reduce a la mitad el tamaño del paso h y por lo tanto reduce el error un 75 porciento. Es por esto que esta regla compuesta del punto medio converge cuadráticamente para $f \in C^2$.

Dado que esta regla es muy sencilla, pocas veces se llega a utilizar en la práctica en economía, por lo que demos énfasis a las siguientes dos reglas.

2.1.2 Regla trapezoidal

Se basa en la aproximación lineal de f usando sólo los valores de f en los puntos extremos de $[a, b]$. La regla trapezoidal consiste en:

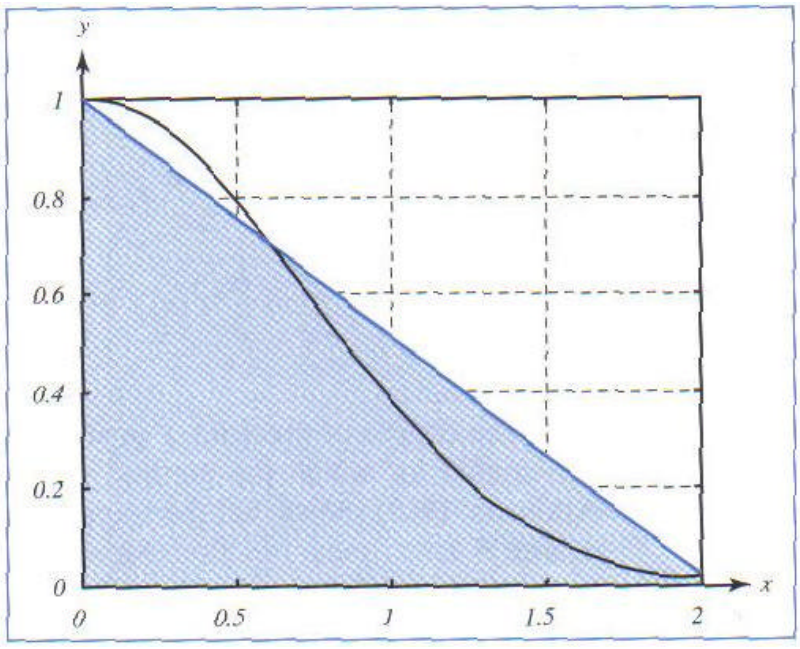
$$\int_a^b f(x)dx = \frac{(b-a)}{2}[f(a) + f(b)] - \frac{(b-a)^3}{12}f''(\xi)$$

para alguna $\xi \in [a, b]$. Esta regla es el ejemplo más simple para la *regla cerrada*, la cual es una regla que usa los puntos extremos.

Por ejemplo, $f(x) = e^{-x^2}$, con $a = 0$ y $b = 2$. Usando la regla trapezoidal tenemos que

$$\int_0^2 e^{-x^2} dx \approx 1[e^{-0^2} + e^{-2^2}] \approx 1.0183$$

La función y la aproximación lineal se pueden ver en la siguiente gráfica:



Gráfica 2.3. $f(x) = e^{-x^2}$ y una línea recta obtenida por la regla trapezoidal

En términos de Matlab, lo escribiriamos en un archivo .m de la siguiente manera:

```
function q=trapecio(f,a,b)
ya=feval(f,a);
yb=feval(f,b);
q=(b-a)*(ya+yb)/2;
```

Definiendo cualquier función en otro archivo .m, por ejemplo:

```
function y=ejemplo(x)
y=exp(-x^2);
```

Luego en la ventana de Matlab sólo escribimos `trapecio('ejemplo',0,2)`, si es que queremos integrar de 0 a 2.

Esta regla también tiene el problema de que trata de aproximar con una recta, lo cual suena un poco ilógico para aproximar funciones en C^n, en donde $n \geq 2$. Por lo tanto, aqui también querremos utilizar la regla compuesta trapezoidal. Sea $h = \frac{(b-a)}{n}$, $x_j = a + jh$, para $j = 1, 2, ..., n$, y denotemos f_i a $f(x_j)$, entonces dicha regla es:

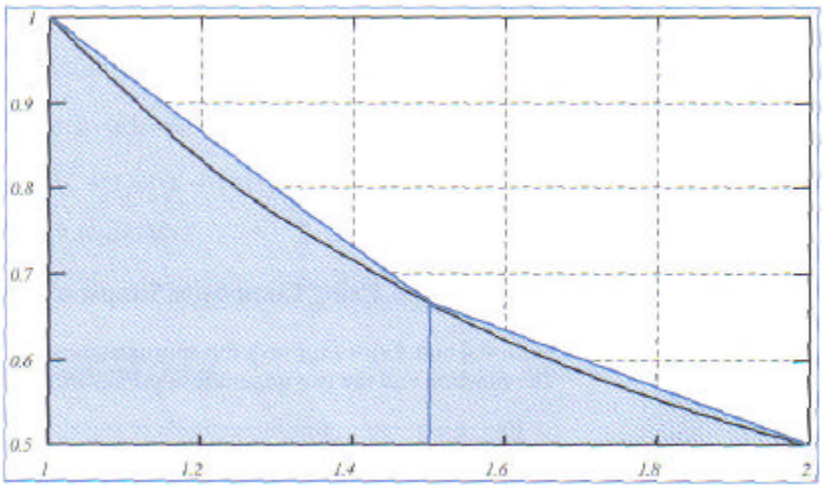
$$\int_a^b f(x)dx = \frac{h}{2}[f_0 + 2f_1 + ... + 2f_{n-1} + f_n] - \frac{h^2(b-a)}{12}f''(\xi)$$

para alguna $\xi \in [a, b]$.

Por ejemplo, $f(x) = \frac{1}{x}$, con $a = 1$, $b = 2$ y $n = 2$. Usando la regla compuesta trapezoidal tenemos que

$$\int_1^2 \frac{1}{x}dx \approx \frac{1}{4}[f(1) + 2f(1.5) + f(2)] = \frac{1}{4}[\frac{1}{1} + 2\frac{1}{1.5} + \frac{1}{2}] = \frac{17}{24} \approx 0.7083$$

La función y las aproximación lineales se pueden ver en la siguiente gráfica:



Gráfica 2.4. $y = \frac{1}{x}$ y la aproximación trapezoidal en [1,1.5] y [1.5,2]

En términos de Matlab, lo escribiriamos en un archivo .m de la siguiente manera:

```
function I=Trap(f,a,b,n)
h=(b-a)/n
S=feval(f,a);
for i=1:n-1
 x(i)=a+h*i;
```

```
 S=S+2*feval(f,x(i));
end
S=S+feval(f,b);
I=h*S/2;
```

Definiendo cualquier función en otro archivo .m, por ejemplo:

```
function y=ejemplo2(x)
y=1/x;
```

Luego en la ventana de Matlab sólo escribimos `Trap('ejemplo2',1,2,2)`, si es que queremos integrar de 1 a 2.

Checar 330.0285 M672A

2.1.3 Regla de Simpson

Una aproximación por pedazos lineal de f en la regla compuesta trapezoidal puede ser innecesaria si f es suave. Otra manera es utilizar una aproximación por pedazos cuadrática de f , la cual utiliza los valores de f en a, b y en el punto medio $\frac{1}{2}(a+b)$. La regla de Simpson en el intervalo $[a, b]$ es:

$$\int_a^b f(x)dx = \frac{(b-a)}{6}[f(a) + 4f(\frac{a+b}{2}) + f(b)] - \frac{(b-a)^5}{2880}f^{(4)}(\xi)$$

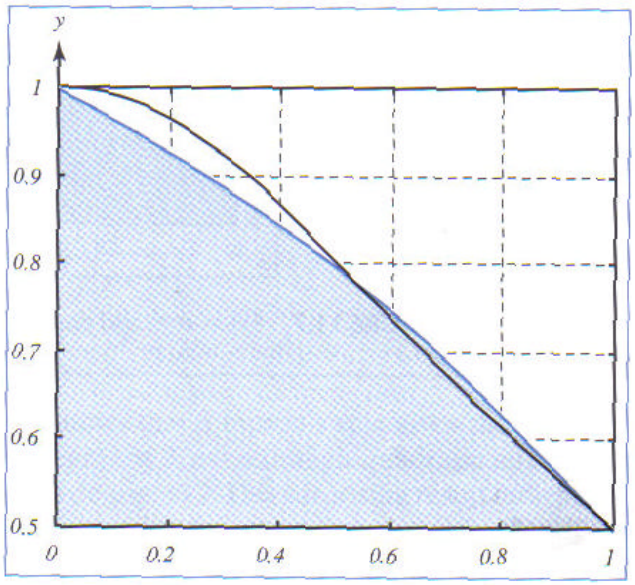
para alguna $\xi \in [a, b]$.

Por ejemplo, $f(x) = \frac{1}{1+x^2}$, con $a = 0$ y $b = 1$. Usando la regla de Simpson tenemos que

$$\int_0^1 \frac{1}{1+x^2}dx \approx \frac{1}{6}[f(0) + 4f(\frac{1}{2}) + f(1)] = \frac{1}{6}[\frac{1}{1+0^2} + 4\frac{1}{1+\frac{1}{2}^2} + \frac{1}{1+1^2}] = \frac{47}{60} \approx 0.7833$$

El valor exacto de la integral es $arctan(1) = \frac{\pi}{4} \approx 0.7853$

La función y el polinomio cuadrático que pasa por los puntos (0,1),(.5,.8) y (1,.5) se pueden ver en la siguiente gráfica:



Gráfica 2.5. $f(x) = \frac{1}{1+x^2}$ y la aproximación cuadrática usando la regla de Simpson

No es de sorprenderse que el valor aproximado de la integral con la regla de Simpson sea muy bueno, porque las dos funciones son similares.

En términos de Matlab, lo escribiriamos en un archivo .m de la siguiente manera:

```
function s=Simpson(f,a,b)
ya=feval(f,a);
yb=feval(f,b);
c=(a+b)/2;
yc=feval(f,c);
q=(b-a)*(ya+4*yc+yb)/6;
```

Definiendo cualquier función en otro archivo .m, por ejemplo:

```
function y=ejemplo(x)
y=exp(-x^2);
```

Luego en la ventana de Matlab sólo escribimos `Simpson('ejemplo',0,2)`, si es que queremos integrar de 0 a 2.

Construyamos ahora la correspondiente regla compuesta de $(n+1)$-puntos sobre $[a, b]$. Sea $n \geq 2$ un número par de intervalos, entonces $h = \frac{(b-a)}{n}$, $x_j = a + jh$, para $j = 0, 1, ..., n$,

y la regla compuesta de Simpson es:

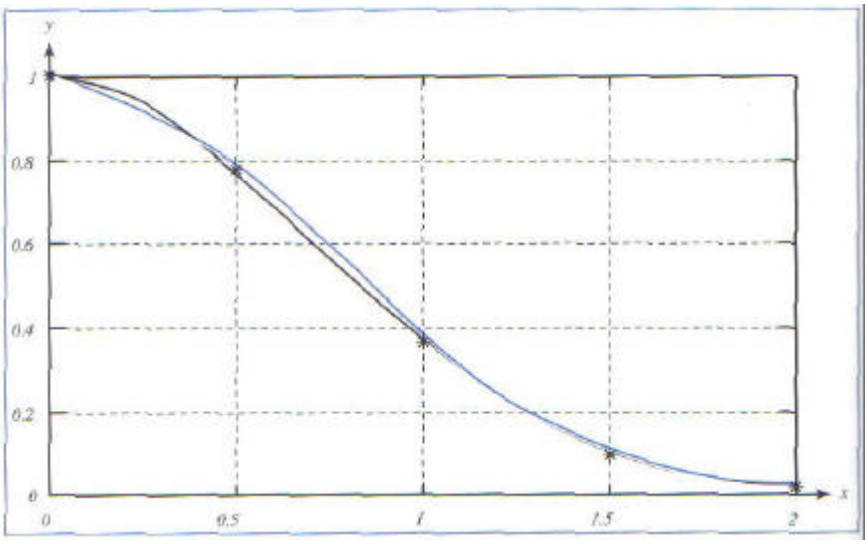
$$S_n(f) = \frac{h}{3}[f_0 + 4f_1 + 2f_2 + 4f_3 + ... + 4f_{n-1} + f_n] - \frac{h^4(b-a)}{180} f^{(4)}(\xi)$$

para alguna $\xi \in [a, b]$. Esta regla lo que hace básicamente es que toma tres particiones consecutivas de x_j, usa la función cuadrática para aproximar a f y la integra para aproximar la integral sobre el intervalo.

Por ejemplo, $f(x) = e^{-x^2}$, con $a = 0$, $b = 2$ y $n = 4$. Usando la regla compuesta de Simpson tenemos que

$$\begin{aligned} \int_0^2 e^{-x^2} dx &\approx \frac{1}{6}[f(0) + 4f(\frac{1}{2}) + 2f(1) + 4f(\frac{3}{2}) + f(2)] \\ &= \frac{1}{6}(e^{-0^2} + 4e^{-(\frac{1}{2})^2} + 2e^{-1^2} + 4e^{-(\frac{3}{2})^2} + e^{-2^2}) = 0.8818 \end{aligned}$$

La función y las aproximaciones cuadráticas se pueden ver en la siguiente gráfica:



Gráfica 2.6. $y = e^{-x^2}$ y su aproximación usando la regla de Simpson con $n = 4$

En términos de Matlab, lo escribiriamos en un archivo .m de la siguiente manera:

```
function J=Simp(f,a,b,n)
h=(b-a)/n;
S=feval(f,a);
for i=1:2:n-1
 x(i)=a+h*i;
```

```
 S=S+4*feval(f,x(i));
end
for i=2:2:n-2
 x(i)=a+h*i;
 S=S+2*feval(f,x(i));
end
S=S+feval(f,b);
J=h*S/3
```

Definiendo cualquier función en otro archivo .m, por ejemplo:

```
function y=ejemplo(x)
y=exp(-x^2);
```

Luego en la ventana de Matlab sólo escribimos `Trap('ejemplo2',0,2,4)`, si es que queremos integrar de 0 a 2.

Notemos que usando localmente una aproximación cuadrática de f tenemos un error de orden h^4, mientras que usando localmente una aproximación lineal de la regla trapezoidal tenemos un error de orden h^2. Por lo que con cualquier aproximación lineal de funciones suaves, aproximaciones de altos grados darán como resultado errores asimptóticamente menores.

Veamos un ejemplo numérico en la tabla 2.1, en donde vemos que en algunos casos se necesitan varios puntos para acercarnos a los valores verdaderos. Vemos que ambas reglas tienen dificultad en la última columna, debido al *kink* en $x = -0.05$. Si éste fuera en $x = 0$, entonces cualquier regla trapezoidal con un número de puntos impares podría computar la integral exactamente.

Ejercicio 2.2. Demostrar que esto se cumple.

Tabla 2.1. Algunas integrales sencillas

Regla	Número de puntos	$\int_0^1 x^{\frac{1}{4}} dx$	$\int_0^1 e^x dx$	$\int_{-1}^1 \max[0, x + 0.05] dx$

Trapezoidal	4	0.7212	1.7342	0.6056
	7	0.7664	1.7223	0.5583
	10	0.7797	1.7200	0.5562
	13	0.7858	1.7193	0.5542
Simpson	3	0.6496	1.4662	0.4037
	7	0.7816	1.7183	0.5426
	11	0.7524	1.6231	0.4844
	15	0.7922	1.7183	0.5528
Verdadera		0.8000	1.7183	0.5513

En general, cuando queramos una buena aproximación de la integral, va a ser necesario utilizar muchos puntos para alcanzar la convergencia con nuestra función deseada.

Si no queremos definir una regla en particular, podemos aproximar la integral en Matlab/Octave con `quad` y `quad8`, y no tenemos que definir cuantos puntos queremos aproximar. Por ejemplo, queremos integrar la función $I = \int_0^{2\pi} e^{-x}\sin(10x)dx$, entonces escribimos

```
>>f=inline('exp(-x).*sin(10*x)');
>>quad(f,0,2*pi)
ans = 0.098825
```

Esto lo podemos comprobar resolviendo la integral por partes que nos daría

$$I = -\frac{1}{101}e^{-x}[\sin(10x) + 10\cos(10x)]|_0^{2\pi} \approx 0.0988$$

Si deseamos resolver integrales con métodos de ordenes altas, entonces en vez de usar el comando `quad('f',xmin,xmax)` usamos `quad8(·)`. El primer comando usa la regla de Simpson, mientras que `quad8` usa cotas de Newton más elevadas de ordenes. También notemos que tanto Matlab como Octave ya tienen incluida el comando del trapecio a través de `trapz(x,y)`, en donde `x` y `y` son vectores.

Sin embargo, si queremos resolver la integral exactamente, escribimos en Matlab el comando `syms x t` para decirle a Matlab que queremos integrar. Si queremos una integral indefinida, pues sólo escribimo la integral, por ejemplo `int(-2*x/(1+x^2)^2)` *returns 1/(1+x^2).* Pero si queremos que nos integre en un intervalo $[a, b]$ escribimos por ejemplo, `int(x*log(1+x),0,1)` *returns 1/4.*

int

Realizemos a continuación algunos ejemplos usando:

a) La regla compuesta trapezoidal con 2 subintervalos

b) La regla compuesta trapezoidal con 10 subintervalos

c) La regla de Simpson compuesta con 2 subintervalos

d) La regla de Simpson compuesta con 10 subintervalos

1. $\int_0^4 2^x dx$
2. $\int_0^1 \frac{1+x}{1+x^3} dx$

1. *octave:1> Trap('ejem1',0,4,2)*

h = 2

ans = 25

octave:2> Trap('ejem1',0,4,10)

h = 0.40000

ans = 21.779

octave:3> Simp('ejem1',0,4,2)

h = 2

J = 22

ans = 22

octave:4> Simp('ejem1',0,4,10)

h = 0.40000

J = 21.641

ans = 21.641

octave:5> Trap('ejem2',0,1,2)

h = 0.50000

ans = 1.1667

octave:6> Trap('ejem2',0,1,10)

h = 0.10000

ans = 1.2075

octave:7> Simp('ejem2',0,1,2)
h = 0.50000
J = 1.2222
ans = 1.2222

octave:8> Simp('ejem2',0,1,10)
h = 0.10000
J = 1.2092
ans = 1.2092

2.2 Diferenciación numérica

La diferenciación numérica se usa frecuentemente en problemas numéricos. Por ejemplo, en optimización y en problemas de ecuaciones no lineales vamos a usar aproximaciones finitas de gradientes, Hesianos y Jacobianos. De hecho, esto por lo general se computa numéricamente, porque las soluciones analíticas son muy difíciles, además de que demoran mucho tiempo. Las derivadas numéricas también son muy importantes en algunos métodos de ecuaciones en diferencia. Lo que haremos será encontrar estimaciones para la derivada o pendiente de una función usando los valores de la función sólo en un conjunto de puntos discretos.

La derivada está definida como

$$f'(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

lo que sugiere la fórmula

$$f'(x) \doteq \frac{f(x+h) - f(x)}{h}$$

en donde h queremos que sea pequeña.

Veamos ahora cómo diferenciar exactamente en Matlab.

Usando también antes el comando `syms x t` escribimos análogamente como lo hicimos con la integral, pero usamos ahora el comando `diff`, por ejemplo `diff(sin(x^2))`*returns 2*cos(x^2)*x.*

2.2.1 Primeras derivadas

Diferencias adelantadas, atrasadas y centrales Las fórmulas más sencillas se basan enutilizar una línea recta para interpolar, dados los datos, es decir, que usan dos puntos dados para estimar la derivada. Por lo que vamos a suponer que x_{i-1} y x_{i+1} existen.

Sea $f(x_{i-1}) = y_{i-1}$, $f(x_i) = y_i$ y $f(x_{i+1}) = y_{i+1}$ y vamos a suponer que la distancia entre las x's es constante, es decir, $h = x_{i+1} - x_i = x_i - x_{i-1}$.

Entonces veamos nuestras tres fórmulas:

Diferencia adelanta

$$f'(x_i) \approx \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

Diferencia atrasada

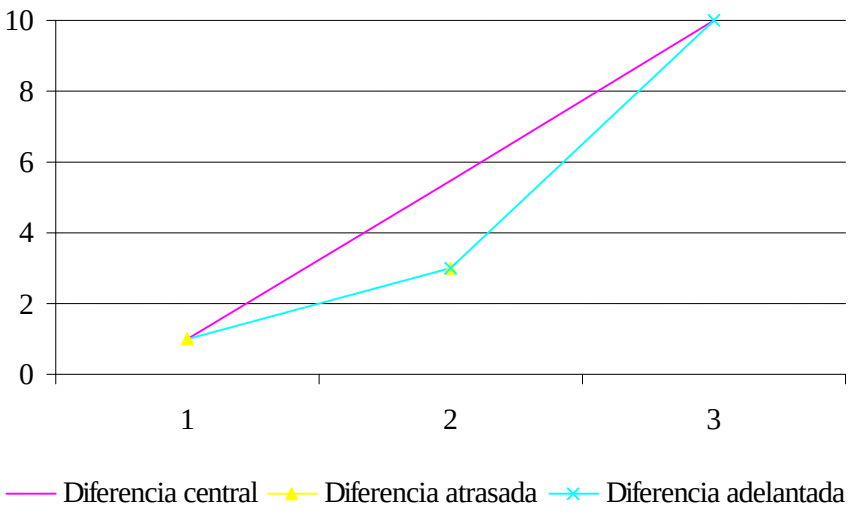
$$f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} = \frac{y_i - y_{i-1}}{x_i - x_{i-1}}$$

Un enfoque más balanceado da una aproximación de la derivada en x_i usando los valores de $f(x_{i-1})$ y $f(x_{i+1})$. Tomando el promedio de las dos diferencias arriba mencionadas, nos da la fórmula de la diferencia central.

Diferencia central

$$f'(x_i) \approx \frac{f(x_{i+1}) - f(x_{i-1})}{x_{i+1} - x_{i-1}} = \frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}}$$

En la gráfica 2.7 podemos observar estas tres fórmulas de diferencia con una función aleatoria $f'(x)$



Gráfica 2.7. Aproximaciones de fórmulas de diferencias

Para ilustrar estas tres fórmulas, consideremos los puntos (1,2), (2,4), (3,8), (4,16) y (5,32), en donde (x_i, y_i), con $i = 0, 1, 2, 3, 4$. Entonces la diferencia adelantada para $f'(x_2) = f'(3)$ con $h = 1$ nos daría

$$f'(x_2) \approx \frac{f(x_3) - f(x_2)}{x_3 - x_2} = \frac{y_3 - y_2}{1} = 16 - 8 = 8$$

Analogamente para la diferencia adelantada nos daría $f'(x_2) \approx 4$, y para la diferencia central nos da $f'(x_2) \approx 6$.

Sin embargo, podemos utilizar estas fórmulas, pero con h's más altas, por ejemplo con $h = 2$ para la diferencia central nos daría $f'(x_2) \approx 7.5$. Los datos que acabamos de utilizar son de la función $y = f(x) = 2^x$, por lo que podemos comparar las estimaciones de las derivadas con los verdaderos valores. En este caso $f'(x) = 2^x(\log 2)$, así que $f'(3) \approx 5.544$.

Fórmulas generales de tres puntos A veces interpolar una función a través de un polinomio es mejor que a través de una recta, además de que puede utilizar más puntos. Por lo que vamos a ampliar las dos primeras fórmulas antes mencionadas.

Diferencia adelanta con tres puntos

$$f'(x_i) \approx \frac{-f(x_{i+2}) + 4f(x_{i+1}) - 3f(x_i)}{x_{i+2} - x_i} = \frac{-y_{i+2} + 4y_{i+1} - 3y_i}{x_{i+2} - x_i}$$

Diferencia atrasada con tres puntos

$$f'(x_i) \approx \frac{3f(x_i) - 4f(x_{i-1}) + 2f(x_{i-2})}{x_i - x_{i-2}} = \frac{3y_i - 4y_{i-1} + 2y_{i-2}}{x_i - x_{i-2}}$$

Aplicadas al ejemplo anterior vemos que con la diferencia adelantada de tres puntos $f'(x_2) \approx 4$, y la diferencia atrasada de tres puntos nos da $f'(x_2) \approx 5$, por lo que vemos que entre más puntos utilicemos, más nos acercamos al verdadero valor.

Empero, estas aproximaciones requieren que la distancia entre las x's sea constante, pero no siempre este es el caso, por lo que el polinomio interpolador de Lagrange sólo necesita que $x_1 < x_2 < x_3$.

Recordemos la forma general del polinomio que pasa por n puntos $(x_1, y_1), ..., (x_n, y_n)$ por lo que tiene n términos que corresponden a cada un de los puntos:

$$p(x) = L_1 y_1 + L_2 y_2 + ... + L_n y_n$$

en donde

$$L_k(x) = \frac{(x - x_1)...(x - x_{k-1})(x - x_{k+1})...(x - x_n)}{(x_k - x_1)...(x_k - x_{k-1})(x_k - x_{k+1})...(x_k - x_n)}$$

El numerador es el producto

$$N_k(x) = (x - x_1)...(x - x_{k-1})(x - x_{k+1})...(x - x_n)$$

por lo que podemos escribir

$$p(x) = c_1 N_1 + c_2 N_2 + ... + c_n N_n$$

en donde c_k son los coeficientes del polinomio interpolador de Lagrange

$$c_k = \frac{y_k}{(x_k - x_1)...(x_k - x_{k-1})(x_k - x_{k+1})...(x_k - x_n)}$$

En nuestro análisis con tres puntos el polinomio interpolador de Lagrange sería:

$$L(x) = L_1(x) y_1 + L_2(x) y_2 + L_3(x) y_3$$

La aproximación a la primera derivada de f viene de $f'(x) \approx L'(x)$, que se puede

escribir como

$$L'(x) = L'_1(x)y_1 + L'_2(x)y_2 + L'_3(x)y_3$$

en donde

$$\begin{aligned} L'_1(x) &= \frac{2x - x_2 - x_3}{(x_1 - x_2)(x_1 - x_3)} \\ L'_2(x) &= \frac{2x - x_1 - x_3}{(x_2 - x_1)(x_2 - x_3)} \\ L'_3(x) &= \frac{2x - x_1 - x_2}{(x_3 - x_1)(x_3 - x_2)} \end{aligned}$$

Por lo tanto,

$$f'(x) \approx \frac{2x - x_2 - x_3}{(x_1 - x_2)(x_1 - x_3)}y_1 + \frac{2x - x_1 - x_3}{(x_2 - x_1)(x_2 - x_3)}y_2 + \frac{2x - x_1 - x_2}{(x_3 - x_1)(x_3 - x_2)}y_3$$

Por ejemplo, tenemos los puntos (-2,4), (0,2) y (2,8), entonces en un archivo .m escribimos los siguinetes comandos que nos ayudan a determinar los coeficientes del polinomio interpolador de Lagrange y el resultado de dicho polinomio.

```
function p=Lagrange_coef(x,y)
% Calcula los coeficientes de las funciones de Lagrange
n=length(x);
for k=1:n
 d(k)=1;
 for i=1:n
 if i~=k
 d(k)=d(k)*(x(k)-x(i));
 end
 c(k)=y(k)/d(k)
 end
end

function p=Lagrange_eval(t,x,c)
% Evalua el polinomio interpolador de Lagrange en x=t
m=length(x);
l=length(t);
```

```
for i=1:l
 p(i)=0;
 for j=1:m
 N(j)=1;
 for k=1:m
 if (j~=k)
 N(j)=N(j)*(t(i)-x(k));
 end
 end
 p(i)=p(i)+N(j)*c(j);
 end
end
```

En el cual en este caso nos va a dar $p(x) = x^2 + x + 2$, con lo cual evaluado en $x = 2$, nos da $p(2) = 8$.

>> x=[-2 0 2]
$x =$
-2 0 2
>> y=[4 2 8]
$y =$
4 2 8
>> Lagrange_coef(x,y)
$c =$
4
$c =$
-2
$c =$
0.5000
$c =$
0.5000 1.0000
$c =$
0.5000 1.0000
$c =$
0.5000 -0.5000

c =

0.5000 -0.5000 2.0000

c =

0.5000 -0.5000 1.0000

c =

0.5000 -0.5000 1.0000

>> Lagrange_eval(2,x,[.5 -.5 1])

ans =

8

2.2.2 Segundas derivadas

Las fórmulas de derivadas más altas se pueden encontrar diferenciando varias veces el polinomio interpolador o con las expansiones de Taylor. Por ejemplo, dados tres puntos con distancias iguales entre ellos, la fórmula de la segunda derivada es

$$f''(x_i) \approx \frac{1}{h^2}[f(x_{i+1}) - 2f(x_i) + f(x_{i-1})]$$

así que usando los puntos (2,4), (3,8), (4,16) con $h = 1$ evaluado en $x_2 = 3$, tenemos

$$f''(3) \approx [f(4) - 2f(3) + f(2)] = 4$$

Con la ayuda del teorema de Taylor, podemos escribir las siguientes fórmulas de diferencia central:

$$f'''(x_i) \approx \frac{1}{2h^3}[f(x_{i+2}) - 2f(x_{i+1}) + 2f(x_{i-1}) - f(x_{i-2})]$$

$$f''''(x_i) \approx \frac{1}{h^4}[f(x_{i+2}) - 4f(x_{i+1}) + 6f(x_i) - 4f(x_{i-1}) - f(x_{i-2})]$$

3 Resolver un sistema de ecuaciones no lineales con métodos de gradiente

Algunos conceptos de equilibrio están expresados como sistemas de ecuaciones no lineales. Estos problemas generalmente toman dos formas: ceros y puntos fijos. Si $f : R^n \to R^n$, entonces un *cero de* f es cualquier x tal que $f(x) = 0$, y un *punto fijo de* f es cualquier x tal que $f(x) = x$. Estos son básicamente el mismo poroblema, ya que x es un punto fijo de $f(x)$ si y sólo si (sii) es un cero de $f(x) - x$. En este capítulo veremos métodos numéricos para resolver ecuaciones no lineales.

El concepto de equilibrio general de Arrow-Debreu consiste esencialmente en encontrar un vector de precios tal que el exceso de demanda es cero, y este en efecto es un problema de un sistema de ecuaciones no lineales. Otro ejemplo es el de equilibrios de Nash con estrategias continuas y las trayectorias de transición de los sistemas deterministicos dinámicos. Así que como pueden ver en economía existen varios problemas de estos y nosotros lo que queremos es tratar de resolverlos con métodos numéricos.

Primero veremos métodos para resolver problemas en una dimensión y después veremos para resolver en dimensión finita. Como suele suceder con estos métodos, es que no existe el método perfecto, sini que cada uno tiene sus ventajas y sus desventajas, además de que unos son mejores para determinados modelos que otros. Para entender mejor la aplicación de estos métodos en economía, veremos algunos ejemplos interesantes, aunque todavía sencillos.

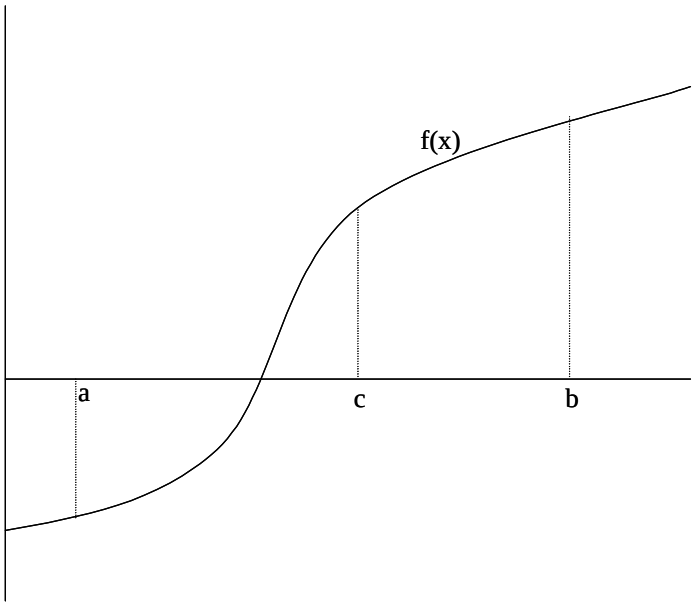
3.1 Problemas unidimensionales

Sea $f : R \to R$ y queremos resolver el problema $f(x) = 0$. Este caso es muy importante, ya que es la base para muchos métodos multidimensionales. A los problemas para encontrar cero de la función no lineal, también se le llama raíces de una ecuación no lineal.

3.1.1 Bisección

El método de la bisección es una técnica sistemática de búsqueda para encontrar el cero de una función continua. Este método se basa en encontrar un intervalo en el cual un cero se sabe que existe, dividiendo el intervalo en dos subintervalos iguales y determinar cual subintervalo contiene al cero.

Sea f continua con $f(a) < 0 < f(b)$ para alguna $a, b, a < b$, como se puede ver en la gráfica 3.1.



Gráfica 3.1. Método de la bisección

Dadas estas condiciones, el teorema del valor medio nos dice que $\exists f(\xi) = 0, \xi \in (a, b)$, entonces el método de la bisección usa este resultado varias veces para computar un cero. Consideremos $c = \frac{1}{2}(a + b)$, es decir el punto intermedio de $[a, b]$. Si $f(c) = 0$, pues ya tenemos nuestro resultado. Si $f(c) > 0$, como en la gráfica, entonces existe un cero en (a, c) y el método de la bisección continúa con (a, c). Análogo para el caso en que $f(c) < 0$. Entonces lo que hace el método de la bisección es que va a trabajar cada vez en intervalos más pequeños, aunque cabe señalar que puede haber varios ceros, y el objetivo este método es encontrar un cero. Sin embrago, podemos usar este método varias veces para encontrar más ceros.

Veamos a continuación el algorítmo de este método:

Objetivo: Encontrar $f(x) = 0$, $f : R \to R$

Paso 1: Encontrar $x^I < x^D$ tal que $f(x^I)f(x^D) < 0$ y escoger la tolerancia $\varepsilon, \delta > 0$

Paso 2: Computar el punto medio $x^M = \frac{x^I + x^D}{2}$

Paso 3: Refinar los límites, es decir, si $f(x^M)f(x^I) < 0$, entonces $x^D = x^M$ y no cambia x^I, en otro caso $x^I = x^M$ y no cambiar x^D

Paso 4: Verificar si $x^D - x^I \leq \varepsilon(|1 + |x^I| + |x^D|)$ ó $|f(x^M)| \leq \delta$, entonces detener y reportar la solución en x^M, en otro caso, volver al paso 1.

Mientras que el método de la bisección es simple, nos dice los componentes importantes para resolver cualquier ecuación no lineal, es decir, hacemos un *guess* inicial, computamos

las iteraciones y verificamos que la última iteración sea una solución aceptable. Este método tiene las ventajas de que es sencillo y siempre que existe una solución la vamos a encontrar, sin embargo, el proceso puede ser lento.

Veamos un ejemplo del método de la bisección. Queremos encontrar la aproximación numérica de $\sqrt{3}$, entonces aproximamos el cero de $y = f(x) = x^2 - 3$. Dado que $f(1) = -2$ y $f(2) = 1$, entonces tomamos como puntos iniciales $a = 1$ y $b = 2$ ($y(a) = -2$ y $y(b) = 1$), por lo que nuestro punto medio es $x_1 = 1.5$ y evaluandolo en la función nos da $y_1 = f(1.5) = -0.75$. Dado que $y(a)$ y y_1 tienen el mismo signo, y $y(b)$ y y_1 tienen signos opuestos, sabemos que existe un cero en el intervalo $[x_1, b]$. Entonces ahora tomamos $a = 1.5$ y continuando de esta manera vamos a obtener el siguiente cuadro para los primeros cinco pasos.

Cuadro 3.1.Cálculos de $\sqrt{3}$ usando el método de la bisección

Paso	a	b	x_i	$y(a)$	$y(b)$	y_i
1	1	2	1.5	-2	1	-0.75
2	1.5	2	1.75	-0.75	1	0.0625
3	1.5	1.75	1.625	-0.75	0.0625	-0.3594
4	1.625	1.75	1.6875	-0.3594	0.0625	-0.1523
5	1.6875	1.75	1.7188	-0.1523	0.0625	-0.0459

Tenemos una estimación del cero en cada iteración, y sabemos que después de k iteraciones el error va a ser a lo mucho $\frac{b_1 - a_1}{2^k}$, en donde a_1 y b_1 es el intervalo original. Como podemos observar, puede haber ocaciones en que estemos muy cerca del cero, pero que al siguiente paso nos alejemos.

Veamos ahora el método de la bisección en Matlab:

```
function [x,y] = bisect(fun,a,b,tol,max)
% Resuelve f(x)=0 usando el método de la bisección
% fun            Nombre de la función entre apóstrofes
% [a,b]      Intervalo que contiene al cero
% tol          Tolerancia permitida al computar el cero
% maxit        Número máximo de iteraciones permitido
% x              Vector de aproximaciones de cero
% y              Vector de valores de la función fun(x)
```

```
a(1)=a;
b(1)=b;
ya(1)=feval(fun,a(1));
yb(1)=feval(fun,b(1));
if ya(1)*yb(1)>0
 error( ' La función tiene el mismo signo en sus puntos extremos')
end
for i = 1:max
 x(i) = (a(i) + b(i))/2;
 y( i ) = feval (fun , x(i));
 if ((x(i)-a(i)) < tol )
 disp( ' El método de la bisección ha convergido');
 break;
 end
 if y(i) == 0
 disp('se encontró el cero exacto');
 break;
 elseif y(i)*ya(i)<0
 a(i+1)=a(i);
 ya(i+1)=ya(i);
 b(i+1)=x(i);
 yb(i+1)=y(i);
 else
 a(i+1)=x(i);
 ya(i+1)=y(i);
 b(i+1)=b(i);
 yb(i+1)=yb(i);
 end;
 iter = i;
end
if (iter >= max)
 disp('no se encontró el cero a la tolerancia deseada');
end
n= length(x);
```

```
k = 1:n;
out = [k' a(1:n)' b(1:n)' x' y'];
disp(' paso a b x y')
disp(out)
```

Para el caso que realizamos manualmente veamos la función:

```
function y=fun1(x)
y=x^2-3;
```

Con lo cual en la pantalla de Matlab sólo escribimos `Bisect('fun1',1,2,0.1,5)`.

>> Bisect('fun1',1,2,0.1,5)
El método de la bisección ha convergido
paso a b x y
1.0000 1.0000 2.0000 1.5000 -0.7500
2.0000 1.5000 2.0000 1.7500 0.0625
3.0000 1.5000 1.7500 1.6250 -0.3594
4.0000 1.6250 1.7500 1.6875 -0.1523
ans =
1.5000 1.7500 1.6250 1.6875

El método de la bisección es lento, pero seguro; y su tasa de convergencia lineal es de $\frac{1}{2}$. La ventaja de este método es que puede funcionar para problemas que causen dificultad con otros métodos.

3.1.2 Método de Newton

Otra desventaja del método de la bisección es que sólo asume que f es continua. El método de Newton usa las propiedades de suavidad de f para formular un método que es rápido cuando funciona, pero que puede no siempre convergir. El método de Newton aproxima a $f(x)$ con una sucesión de funciones lineales y aproxima un la nulidad de f con los ceros de las aproximaciones lineales. Si f es suave, entonces estas aproximaciones son crecientemente precisas, y las iteraciones sucesivas convergerán rápido al cero de f. Básicamente el método de Newton reduce un problema no lineal a una sucesión de problemas lineales, en donde los ceros son fáciles de computar.

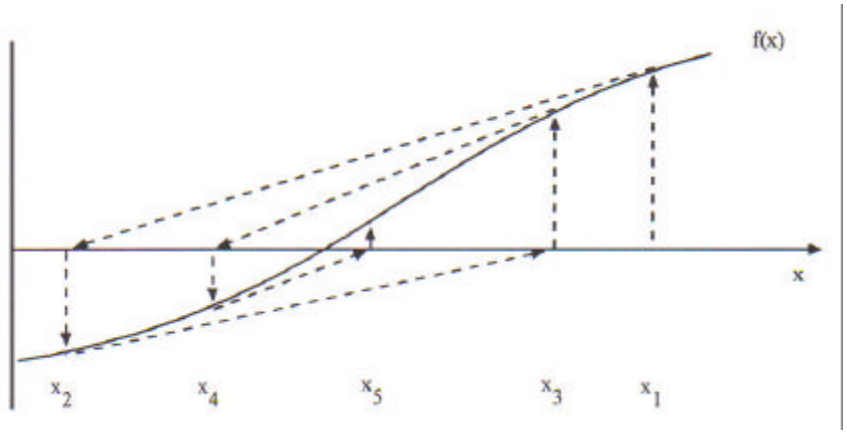
Formalmente, el método de Newton es una simple interación. Suponer que nuestro primer *guess* es x_k. Entonces en x_k construimos la aproximación lineal de f en x_k, dando como resultado la función

$$g(x) \equiv f'(x_k)(x - x_k) + f(x_k)$$

Las funciones $f(x)$ y $g(x)$ son tangentes en x_k y generalmente cerca en una vecindad de x_k. Entonces en vez de resolver la nulidad de la función f, resolvemos la nulidad de la función g, esperando que las dos tengan nulidades similares. Nuestro nuevo *guess*, x_{k+1}, será el cero de $g(x)$, implicando que

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{3.1}$$

La siguiente gráfica muestra los pasos de x_1 a x_5, en donde se ve que el nuevo *guess* no alcanza la nulidad de la función, pero esperamos que la secuencia x_k converja a la nulidad de f. El teorema 3.1 provee las condiciones suficientes para la convergencia.



Gráfica 3.2. Método de Newton

Teorema 3.1. Sea $f \in C^2$ y $f(x^*) = 0$. Si x_0 está suficientemente cerca a x^*, $f'(x^*) \neq 0$ y $|\frac{f''(x^*)}{f'(x^*)}| < \infty$, entonces la secuencia de Newton x_k definida en (3.1) converge a x^*, y es cuadráticamente convergente, es decir,

$$\limsup_{k\to\infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|^2} < \infty$$

Escoger entre estos dos métodos no es fácil, ya que por una parte el método de la bisección, aunque lento, siempre converge a cero, mientras que el método de Newton es

rápido, pero puede no converger.

Veamos a continuación el algorítmo de este método:

Objetivo: Encontrar $f(x) = 0$, $f : R \to R, f \in C^1$

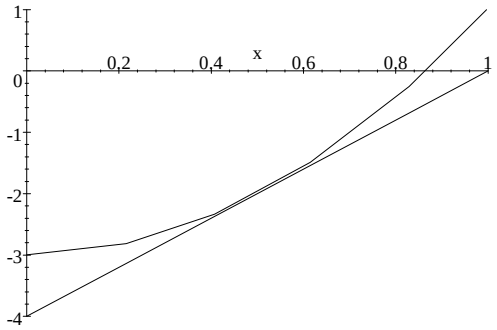
Paso 1: Escoger ε, δ y el punto inicial x_0, es decir, $k = 0$

Paso 2: Computar la siguiente iteración $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$

Paso 3: Verificar las condiciones para detenerse: si $|x_k - x_{k+1}| \leq \varepsilon(1 + |x_{k+1}|)$, pasar al siguiente paso, en otro caso pasar al paso 1

Paso 4: Reportar los resultados y detenerse: Si $|f(x_{k+1})| \leq \delta$, reportar éxito para encontrar el cero, en otro caso, reportar fracaso

¿Que pasa si queremos encontrar $\sqrt{\frac{3}{4}}$? Lo que hacemos es aproximar el cero de $y = f(x) = 4x^2 - 3$, como se ve en la gráfica 3.3. Para esto vamos a necesitar $y' = f'(x) = 8x$. Dado que $f(0) = -3$ y $f(1) = 1$, usamos como nuestro *guess* inicial $x_0 = 0.5$, con lo cual su corespondiente valor de la función es $y_0 = -2$ y el valor de la derivada es $y'_0 = 4$.



Gráfica 3.3. $y = 4x^2 - 3$ y la aproximación de la recta tangente en $x = 0.5$

Por lo tanto nuestra primera aporximación del cero es

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 0.5 - \frac{-2}{4} = 1$$

Continuando con un paso más vamos a tener

$$x_2 = x_1 - \frac{y_1}{y'_1} = 1 - \frac{1}{8} = \frac{7}{8} = 0.875$$

En Matlab podemos computar el método de Newton de la siguiente manera:

```
function [x,y]=Newton(fun,fun_pr,x1,tol,max);
```

```
% Resuelve f(x)=0 usando el método de Newton
% fun Nombre de la función entre apóstrofes
% funpr Especifica la derivada de f
% x0 Estimación inicial
% tol Tolerancia permitida al computar el cero
% maxit Número máximo de iteraciones permitido
%
% x Vector de aproximaciones de cero
% y Vector de valores de la función fun(x)
x(1)=x1;
y(1)=feval(fun,x(1));
ypr(1)=feval(funpr,x(1));
for i=2:max
 x(i)=x(i-1)-y(i-1)/ypr(i-1);
 y(i)=feval(fun,x(i));
 if abs(x(i)-x(i-1))<tol
 disp('El metodo de Newton converge');
 break;
 end
 ypr(i)=feval(funpr,x(i));
 iter=i;
end
if (iter>=max)
 disp('no se encontró el cero a la tolerancia deseada');
end
n=length(x);
k=1:n;
out=[k' x' y'];
disp(' paso x y')
disp(out)
```

Para el caso que realizamos manualmente veamos la función:

```
function y=fun2(x)
y=4*x^2-3;
```

con su correspondiente derivada:

```
function y=funpr2(x)
y=8*x;
```

Con lo cual en la pantalla de Matlab sólo escribimos `Newton('fun2','funpr2',0.5,0.1,10)`.

>> Newton('fun2','funpr2',0.5,0.1,10)
El metodo de Newton converge
paso x y
1.0000 0.5000 -2.0000
2.0000 1.0000 1.0000
3.0000 0.8750 0.0625
4.0000 0.8661 0.0003
ans =
0.5000 1.0000 0.8750 0.8661

3.1.3 Método de la secante

Un aspecto clave para el método de Newton es computar $f'(x)$, el cual puede ser costoso. El método de la secante emplea aproximaciones lineales, pero nunca evalúa f', sino que aproxima $f'(x_k)$ con la pendiente de la secante de f entre x_k y x_{k+1}, resulatndo de esta manera en la iteración

$$x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})} \tag{3.2}$$

Este método es similar al de Newton, sólo que en el algorítmo en el paso 2 se utiliza la ecuación (3.2) en vez de (3.1) para computar la siguiente iteración. El método de la secante tiene los mismos problemas de convergencia que el método de Newton, sólo que puede ahorrar tiempo debido a que no evalúa f. La tasa de convergencia es entre lineal y cuadrática.

Teorema 3.2. Sea $f(x^*) = 0$, $f'(x^*) \neq 0$ y $f'(x)$ y $f''(x)$ son continuas alrededor de x^*, entonces el método de la secante converge a una tasa $\frac{(1+\sqrt{5})}{2}$, es decir,

$$\limsup_{k\to\infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|^{\frac{(1+\sqrt{5})}{2}}} < \infty$$

Usando el ejemplo de $y = f(x) = x^2 - 3$, veamos que tal funciona para el método de la secante. Las extremos que utilizaremos son $x_0 = 1$ y $x_1 = 2$, con lo cual $y_0 = -2$ y $y_1 = 1$, respectivamente.

Nuestra primera aproximación al cero es

$$x_2 = x_1 - \frac{x_1 - x_0}{y_1 - y_0} y_1 = 2 - \frac{2-1}{1-(-2)}(1) = \frac{5}{3} \approx 1.667$$

El método de la secante no requiere que los puntos usados para computar la próxima aproximación envuelvan al cero, ni incluso que $x_0 < x_1$. Es conveniente ver los cálculos para ver como están evolucionando los resultados, los cuales los podemos ver en el cuadro 3.2.

Cuadro 3.2.Cálculos de $\sqrt{3}$ usando el método de la secante

Paso	x	y
1	1	-2
2	2	1
3	1.6667	-0.2222
4	1.7273	-0.0165
5	1.7321	0.0003
6	1.7321	-0.0000

Veamos ahora la función en Matlab para el método de la secante:

```
function [x,y] = secant(fun,a,b,tol,max)
% Resuelve f(x)=0 usando el método de la secante
% fun Nombre de la función entre apóstrofes
% a,b primeros dos estimadores de cero
% tol Tolerancia permitida al computar el cero
% max Número máximo de iteraciones permitido
% x Vector de aproximaciones de cero
% y Vector de valores de la función fun(x)
x(1)=a;
x(2)=b;
y(1)=feval(fun,x(1));
y(2)=feval(fun,x(2));
for i = 2:max
```

```
 x(i+1)=x(i)-y(i)*(x(i)-x(i-1))/(y(i)-y(i-1));
 y(i+1)=feval(fun,x(i+1));
 if abs(x(i+1)-x(i)) < tol
 disp( ' El método de la secante ha convergido');
 break;
 end
 if y(i) == 0
 disp('se encontró el cero exacto');
 break;
 end
 iter=i;
end
if (iter>=max)
 disp('no se encontró el cero a la tolerancia deseada');
end
n= length(x);
k = 1:n;
out = [k' x' y'];
disp(' paso x y')
disp(out)
```

Para el caso que realizamos manualmente veamos la función:

```
function y=fun1(x)
y=x^2-3;
```

Con lo cual en la pantalla de Matlab sólo escribimos `secant('fun1',1,2,0.001,10)`.

El método de la secante ha convergido

paso	*x*	*y*
1.0000	*1.0000*	*-2.0000*
2.0000	*2.0000*	*1.0000*
3.0000	*1.6667*	*-0.2222*
4.0000	*1.7273*	*-0.0165*
5.0000	*1.7321*	*0.0003*
6.0000	*1.7321*	*-0.0000*

ans =

1.0000 2.0000 1.6667 1.7273 1.7321 1.7321

3.1.4 Iteración del punto fijo

Como con las ecuaciones lineales, generalmente vamos a poder reescribir las problemas no lineales en formas que nos sugieran un método computacional. Cualquier problema de punto fijo $x = g(x)$ sugiere la iteración $x_{k+1} = g(x_k)$. Por ejemplo, consideremos $x^3 - x - 1 = 0$, el cual se puede reescribir en la forma del punto fijo $x = (x+1)^{\frac{1}{3}}$, que nos sugiere la iteración $x_{k+1} = (x_k + 1)^{\frac{1}{3}}$, la cual converge a una solución si $x_0 = 1$. Sin embargo, si reescribimos $x^3 - x - 1 = 0$ como $x^3 - 1 = x,$ la forma sugerida será $x_{k+1} = x_k^3 - 1$, la cual diverge a $-\infty$ cuando $x_0 = 1$.

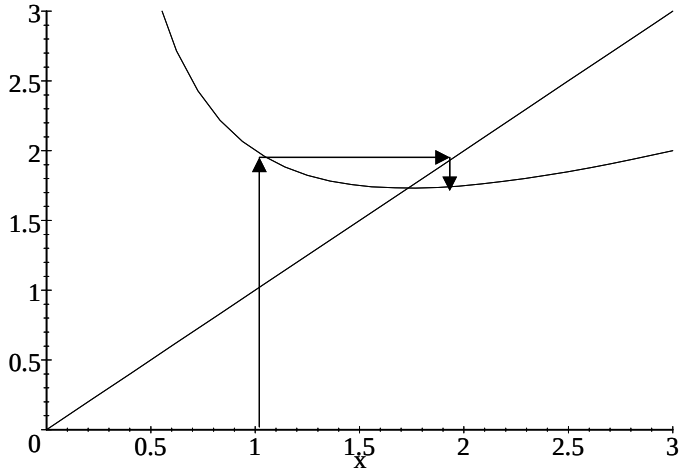
Veamos un caso sencillo, por ejemplo, queremos sacar $\sqrt{3}$ con la iteración del punto fijo. Escribimos $x^2 = 3$, y la reescribimos como

$$x = \frac{1}{2}(x + \frac{c}{x})$$

que nos da la base para la iteración, es decir que nos va a quedar algo de la forma:

$$x_k = \frac{1}{2}(x_{k-1} + \frac{c}{x_{k-1}})$$

Geometricamente hablando lo que queremos es la intersección entre $y = \frac{1}{2}(x + \frac{c}{x}) = g(x)$ con $y = x.$ Para el caso de $\sqrt{3}$ tendríamos $x_1 = \frac{1}{2}(1 + \frac{3}{1}) = 2$ y $x_k = \frac{1}{2}(2 + \frac{3}{2}) = \frac{7}{4} = 1.75$ si usamos el punto inicial $x_0 = 1$, como se ve en la siguiente gráfica.



Gráfica 3.4. 1er paso de la iteración del punto fijo de $x = \frac{1}{2}(x + \frac{3}{x})$

Hacer la gráfica a mano de una divergente, con una media parabola hacia abajo, pero muy curveada.

La función en Matlab sería:

```
function [x,gval] = fixpoint(g,x,tol,max)
% Computa el punto fijo de una función
% g Nombre de la función de la forma gval=g(x)
% x Guess inicial para el punto fijo
% x Punto fijo de g
% gval Estimación del valor de la función
for it=1:max
 gval = feval(g,x);
 if norm(gval-x)<tol, return, end
 x = gval;
end
warning('Fracaso')
```

Para el caso que realizamos manualmente veamos la función:

```
function y=g(x)
y=(x+(3/x))/2;
```

Con lo cual en la pantalla de Matlab sólo escribimos `fixpoint('g',1,0.01,10)`.

>> fixpoint('g',1,0.01,10)

ans =

1.7321

Este método es más o menos utilizado, sin embargo generalmente no se puede confiar mucho en él. Veremos métodos más confiables, aunque esto no quiere decir que debemos ignorar estos esquemas de iteraciones de punto fijo, sólo hay que tener mucho cuidado y estar pendientes de sus desventajas.

3.2 Problemas multidimensionales

Muchos problemas tienen varias ecuaciones, por lo que nosotros vamos a querer resolver el problema $f(x) = 0$, con $f : R^n \to R^n$, es decir,

$$\begin{aligned} f^1 &= (x_1, x_2, ..., x_n) = 0 \\ &\vdots \\ f^n &= (x_1, x_2, ..., x_n) = 0 \end{aligned} \tag{3.3}$$

Pero esto de extender algunos métodos que vimos en la sección anterior, no es tan fácil para algunos casos. Por ejemplo, para dos funciones de dos variables, $z = f(x, y)$ y $z = g(x, y)$, encontrar el cero del sistema requiere que encontremos la intersección de las curvas $f(x, y) = 0$ y $g(x, y) = 0$. Con lo cual es muy importante utilizar toda la información posible para determinar cual es la región en donde pueda estar el cero.

3.2.1 Método de Newton

Para resolver (3.3) con el método de Newton, vamos a reemplazar f con una aproximación lineal, y posteriormente resolver el problema lineal para generar el nuevo *guess*, tal como lo hicimos para el caso unidimensional. Por el teorema de Taylor, la aproximación lineal de f alrededor de *guess* inicial x^0 es

$$f(x) \doteq f(x^0) + J(x^0)(x - x^0)$$

Podemos resolver la nulidad de esta aproximación lineal, dando como resultado $x^1 = x^0 - J(x^0)^{-1} f(x^0)$. Este cero luego sirve como el nuevo *guess* sobre el cual vamos a volver a

linealizar. La iteración de Newton es

$$x^{k+1} = x^k - J(x^k)^{-1} f(x^k)$$

Veamos cómo sería el algorítmo de este método:

Paso 1: Escoger el criterio para detenerse ε y δ y el punto inicial x^0, es decir, $k = 0$

Paso 2: Computar la siguiente iteración: Computar el jacobiano $A_k = J(x^k)$, resolver $A_k s^k = -f(x^k)$ para s^k, y considerar $x^{k+1} = x^k + s^k$

Paso 3: Verificar los criterios para detenernos: Si $||x^k - x^{k+1}|| \leq \varepsilon(1 + ||x^{k+1}||)$ seguir al paso 4; en otro caso ir al paso 2.

Paso 4: Detenerse y reportar los resultados: Si $||f(x^{k+1})|| \leq \delta$, reportar éxito al encontrar el cero; en otro caso reportar fracaso.

```
function x=Newton_sys(F,JF,x0,param,tol,maxit)
% Resuelve el sistema no lineal F(x)=0 usando el método de Newton
% Los vectores x y x0 son vectores fila
% la función F da un vector columna [f1(x)...fn(x)]'
% detener cuando la norma del cambio en el vector solución sea menor a la
tolerancia
% la siguiente aproximación de solución es x_new=x_old+y';
x_old=x0;
disp([0 x_old]);
iter=1;
while (iter<=maxit)
 y=-feval(JF,x_old)\feval(F,x_old,param);
 x_new=x_old+y';
 dif=norm(x_new-x_old);
 disp([iter x_new dif]);
 if dif<=tol
 x=x_new;
 disp('El método de Newton ha convergido')
 return;
 else
 x_old=x_new;
 end
```

```
 iter=iter+1;
end
disp('El método de Newton no convergió')
x=x_new;
```

Usemos por ejemplo el sistema de ecuaciones:

$$\begin{aligned} f(x,y,z) &= x^3 - 10x + y - z + 3 = 0 \\ g(x,y,z) &= y^3 + 10y - 2x - 2z - 5 = 0 \\ h(x,y,z) &= x + y - 10z + 2\sin(z) + 5 = 0 \end{aligned}$$

cuyo Jacobiano es

$$J(x) = \begin{bmatrix} 3x^2 - 10 & 1 & -1 \\ -2 & 3y^2 + 10 & -2 \\ 1 & 1 & -10 + 2\cos(z) \end{bmatrix}$$

En Matlab escribimos el sistema de funciones con su respectivo Jacobiano:

```
function F=F1(x,param)
F=[x(1)^3-10*x(1)+x(2)-x(3)+3-param(1);...
x(2)^3+10*x(2)-2*x(1)-2*x(3)-5-param(2);...
x(1)+x(2)-10*x(3)+2*sin(x(3))+5-param(3)];

function JF=JF1(x)
JF=[(3*x(1)^2)-10 1 -1; -2 (3*x(2)^2)+10 -2; 1 1 -10+2*cos(x(3))];
```

Así que en Matlab sólo escriben `Newton_sys('F1','JF1',[1 1 1],[0 0 0],0.01,10)`

>> Newton_sys('F1','JF1',[1 1 1],[0 0 0],0.01,10)
0 1 1 1
1.0000 0.1359 0.6699 0.7185 0.9669
2.0000 0.2955 0.6745 0.7305 0.1601
3.0000 0.2970 0.6748 0.7307 0.0015
El método de Newton ha convergido
ans =
0.2970 0.6748 0.7307

3.2.2 Método de la secante

Como vimos para el caso univariado, el método de la secante es similar al de Newton, de hecho, algunos lo llaman el método de Newton-Raphson, sólo que utiliza aproximaciones lineales para el Jacobiano, en vez del Jacobiano *per se*.

Veamos que el ejemplo anterior también se puede resolver con este método.

```
function x = secant_sys(fun,x0,param,tol,maxit);
% secant_sys.m Programa para resolver sistemas de ecuaciones no lineales
% detener cuando la norma del cambio en el vector solución sea menor a la
tolerancia
del = diag(max(abs(x0)*1e-4,1e-8));
n = length(x0);
for i=1:maxit
 f = feval(fun,x0,param);
 for j=1:n
 J(:,j) = (f-feval(fun,x0-del(:,j),param))/del(j,j);
 end;
 x = x0-inv(J)*f;
 if norm(x-x0)<tol
 break
 end
 x0 = x;
end
if i>=maxit
 sprintf('Se alcanzaron %iteraciones',maxit)
end
```

Entonces en la ventana de Matlab escribimos

```
secant_sys('F1',[1 1 1]',[0 0 0],0.001,10)
```

ans =

0.2970

0.6748

0.7307

3.2.3 Iteración del punto fijo

Por analogía al caso unidimensional, el procedimiento más sencillo para resolver $x = f(x)$ es $x^{k+1} = f(x^k)$. A este método también se le llama *aproximación sucesiva, sustitución sucesiva* o *iteración función*. En el caso multivariado, sólo se generaliza el proceso univariado.Un ejemplo del punto fijo está en el teorema de la contracción (visto en el curso de Programación Dinámica).

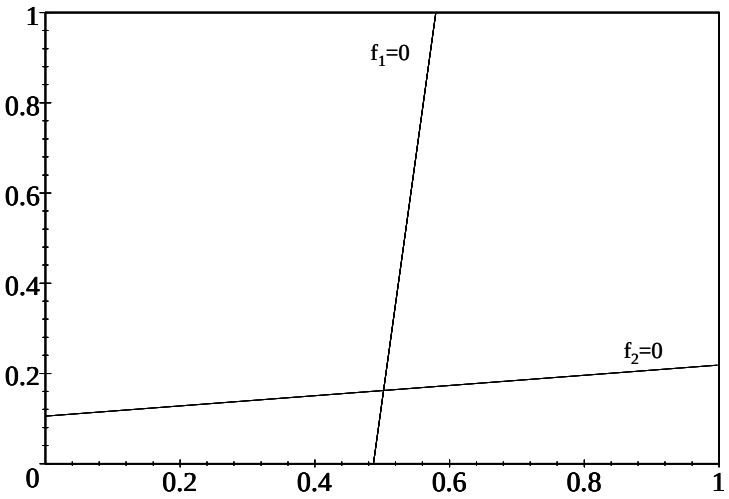
Por ejemplo, si tenemos el sistema de ecuaciones

$$\begin{aligned} f_1(x_1, x_2) &= x_1^3 + 10x_1 - x_2 - 5 = 0 \\ f_2(x_1, x_2) &= x_1 + x_2^3 - 10x_2 + 1 = 0 \end{aligned}$$

lo reescribimos de la forma $x_1 = g_1(x_1, x_2)$ y $x_2 = g_2(x_1, x_2)$ de la siguiente manera:

$$\begin{aligned} x_1 &= -0.1x_1^3 - 0.1x_2 - 0.5 \\ x_2 &= 0.1x_1 + 0.1x_2^3 + 0.1 \end{aligned}$$

La siguiente gráfica ilustra $f_1 = 0$ y $f_2 = 0$, y el cuadro muestra las soluciones aproximadas de cada iteración



Gráfica 3.5.Sistema de dos ecauciones no lineales

La función escrita en lenguaje Matlab sería:

```
function x=fixpoint_sys(G,x0,tol,max)
% Resuleve el sistema no lineal x=G(x) usando la iteración del punto fijo
% Los vectores x y x0 son vectores fila
% la función G da un vector columna [g1(x)...gn(x)]'
% detener cuando la norma del cambio en el vector solución sea menor a la
tolerancia
% la siguiente aproximación de solución es x_new=x_old+y';
disp([0 x0]);
x_old=x0;
iter=1;
while (iter<=max)
 y=feval(G,x_old)
 x_new=y';
 dif=norm(x_new-x_old);
 disp([iter x_new dif]);
 if dif<=tol
 x=x_new;
 disp('La iteración del punto fijo ha convergido')
 return;
 else
 x_old=x_new;
 end
 iter=iter+1;
end
disp('La iteración del punto fijo no convirgió')
x=x_new;
```

La función que estabamos utilizando la escribimos como

```
function G=ejem4(x)
G=[(-0.1*x(1)^3+0.1*x(2)+0.5)
 (0.1*x(1)+0.1*x(2)^3+0.1)];
```

En la ventana de Matlab sólo escribimos `fixpoint_sys('ejem4',[1 1],0.01,10)`

>> fixpoint_sys('ejem4',[1 1],0.01,10)
0 1 1

1.0000 0.5000 0.3000 0.8602
2.0000 0.5175 0.1527 0.1483
3.0000 0.5014 0.1521 0.0161
4.0000 0.5026 0.1505 0.0020
La iteración del punto fijo ha convergido
ans =
0.5026 0.1505

3.3 Ejemplos económicos

Veamos en este capítulo un ejemplo de Microeconomía, en particular de teoría de juegos. Vamos a ilustrar un ejemplo del modelo de Cournot con el método de la secante.

En el modelo vamos a tener que el inverso de la demanda de un bien es $P(q) = q^{\frac{-1}{\eta}}$ y las firmas van a producir con las siguientes funciones de producción

$$C_i(q_i) = \frac{1}{2}c_i q_i^2, \qquad i = 1, 2$$

Las ganancias de la empresa i son

$$\pi_i(q_1, q_2) = P(q_1 + q_2)q_i - C_i(q_i)$$

Si la empresa i toma como dado el producto de la otra empresa, va a optimizar su nivel de producción resolviendo

$$\frac{\partial \pi_i}{\partial q_i} = P(q_1 + q_2) + P'(q_1 + q_2)q_i - C_i'(q_i) = 0$$

Entonces los productos de equilibrio, q_1 y q_2, son las raíces de las dos ecuaciones no lineales

$$f_i(q) = (q_1 + q_2)^{\frac{-1}{\eta}} - \left(\frac{-1}{\eta}\right)(q_1 + q_2)^{\frac{-1}{\eta}-1}q_i - c_i q_i = 0$$

para $i = 1, 2$.

Vamos a suponer que usamos $\eta = 1.6$, aunque notemos que éste también lo podemos poner como parámetro de la función de Cournot, como hemos hecho para los diferentes costos.

```
function fval = cournot(q)
 c = [param(1);param(2)];
 eta = 1.6;
 e = -1/eta;
 fval = sum(q)^e + e*sum(q)^(e-1)*q - diag(c)*q;
```

Así que solo escribimos en la ventana de Matlab `secant_sys('cournot',[0.2 0.2]',[0.6 0.8],0.001,10)` ó `secant_sys('cournot',[0.2 0.2]',[0.6 0.6],0.001,10)`

>> secant_sys('cournot',[0.2 0.2]',[0.6 0.8],0.001,10)

ans =

0.8396

0.6888

>> secant_sys('cournot',[0.2 0.2]',[0.6 0.6],0.001,10)

ans =

0.8329

0.8329

4 Métodos para sistemas de ecuaciones lineales

Resolver ecuaciones lineales también es muy utilizado en la práctica, y a veces se necesita la combinación de métodos que resuleven sistemas de ecuaciones lineales con ecuaciones no lineales, como al final de este capítulo veremos un ejemplo. También ya hemos visto que algunos métodos aproximan con ecuaciones lineales algunos problemas no lineales, por lo que entender cómo se resuleven los problemas lineales y las dificultades asociadas es muy importante para el análisis numérico.

Los problemas del sistema de ecuaciones lineales, $Ax = b$, se divide basicamente en dos tipos, dependiendo de las entradas de A: i) Decimos que A es denso si $a_{ij} \neq 0$ para la mayoría de i, j. ii) Decimos que A es dispersa o poco densa (*sparse*) si $a_{ij} = 0$ para la mayoría de i, j. Nótemos que estas definiciones no son muy precisas, pero en la práctica vamos a ver que claramente podemos determinar si la matriz A es densa o no tanto.

Como vimos en el capítulo anterior, ¿qué sucede si tenemos ecuaciones no lineales? También en este caso vamos a tener varias incógnitas, por lo que necesitamos también necesitamos métodos multidimensionales. Supongamos de nuevo que $f : R^n \to R^n$ y que queremos resolver $f(x) = 0$, con lo cual tenemos lo mismo que en (3.3)

$$\begin{aligned} f^1 &= (x_1, x_2, ..., x_n) = 0 \\ &\vdots \\ f^n &= (x_1, x_2, ..., x_n) = 0 \end{aligned}$$

Vamos a ver dos métodos que se pueden aplicar tanto a problemas lineales como no lineales: Gauss-Jacobi y Gauss-Seidel.

4.1 Métodos directos

4.1.1 Eliminación Gaussiana

La eliminación Gaussiana es un método directo común para resolver un sistema lineal. La eliminación Gaussiana se basa en el hecho de que si dos ecuaciones tienen un punto en común, entonces este punto también satisface cualquier combinación lineal de esas dos ecuaciones. Si podemos encontrar combinaciones lineales de una forma más o menos simple, entonces podremos encontrar la solución del problema original de una forma más sencilla. La forma

que estaríamos buscando sería aquella en la que ciertas variables han sido eliminadas de algunas ecuaciones.

El caso más simple es el de las matrices triangulares, en donde se dice que A es triangular baja si todos los elementos que no son cero están sobre o bajo la diagonal, es decir,

$$A = \begin{pmatrix} a_{11} & 0 & ... & 0 \\ a_{21} & a_{22} & ... & 0 \\ \vdots & \vdots & \ddots & 0 \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix}$$

Análogamente para el caso en que A sea triangular alta. Una matriz diagonal es aquella que sólo tienen elementos que no son cero sobre la diagonal. Recordemos que una matriz triangular es no singular si y sólo si todos los elementos de la diagonal no son cero. Los sistemas lineales en los cuales A es triangular se pueden resolver con *sustitución hacia atrás.* Supongamos que A es triangular baja y no singular, dado que todos los elementos que están fuera de la diagonal de la primera fila son cero, el problema se reduce a resolver $a_{11}x_1 = b_1$, cuya solución sabemos que es $x_1 = \frac{b_1}{a_{11}}$. Con esta solución podemos resolver para x_2 , que implica $a_{22}x_2 + a_{21}x_1 = b_2$; y así sucesivamente en donde tendremos la sucesión

$$\begin{aligned} x_1 &= \frac{b_1}{a_{11}} \\ x_k &= \frac{b_k - \sum_{j=1}^{k-1} a_{kj}x_j}{a_{kk}} \qquad \text{para } k = 2, 3, ..., n \end{aligned}$$

que está bien definida para matrices no singulares. Si la matriz fuera triangular alta entonces empecaríamos con $x_n = \frac{b_n}{a_{nn}}$.

Pero lamentablemente en la práctica rara vez nos encontraremos con ecuaciones en esta forma, pero lo que vamos a hacer será buscar poner nuestro sistema de ecuaciones de esta manera.

Por ejemplo, suponer que tenemos el siguiente sistema de ecuaciones:

$$\begin{aligned} 2x + 6y + 10z &= 0 \\ x + 3y + 3z &= 2 \\ 3x + 14y + 28z &= -8 \end{aligned}$$

que en forma matricial la podemos escribir así:

$$\left[\begin{array}{cccc} 2 & 6 & 10 & | & 0 \\ 1 & 3 & 3 & | & 2 \\ 3 & 14 & 28 & | & -8 \end{array}\right]$$

donde recordando un poco de algebra matricial podemos realizar nuestro primer paso de la eliminación, con lo cual nos quedaría:

$$\left[\begin{array}{cccc} 2 & 6 & 10 & | & 0 \\ 0 & 0 & 4 & | & -4 \\ 0 & -10 & -26 & | & 16 \end{array}\right]$$

normalizando y cambiando filas tenemos

$$\left[\begin{array}{cccc} 1 & 3 & 5 & | & 0 \\ 0 & 1 & 2.6 & | & -1.6 \\ 0 & 0 & 1 & | & -1 \end{array}\right]$$

que ya es una matriz triangular alta, por lo que podemos dar los resultados de una vez sustituyendo en las ecuaciones, o podríamos continuar con este método para que nos quede sólo la matriz identidad.

$$\left[\begin{array}{cccc} 1 & 3 & 5 & | & 0 \\ 0 & 1 & 0 & | & 1 \\ 0 & 0 & 1 & | & -1 \end{array}\right]$$

$$\Rightarrow \left[\begin{array}{cccc} 1 & 0 & 0 & | & 2 \\ 0 & 1 & 0 & | & 1 \\ 0 & 0 & 1 & | & -1 \end{array}\right]$$

es decir, $x_1 = 2, x_2 = 1, x_3 = -1$

Veamoslo en la función de Matlab:

```
function x = gelim(A,b)
% Solución para el sistema lineal de ecuaciones Ax = b
% usando la eliminación de Gauss
```

```
% A es la matriz de coeficientes nxn
% b es el vector del lado derecho de nx1
[n,nl] = size(A);
for i=1:n-1
 [pivot,k]= max(abs(A(i:n,i)));
 % k es la posición del pivote, relativo a la fila i
 if k > 1
 temp1 = A(i,:);
 temp2 = b(i,:);
 A(i,:)  = A(i+k-1,:);
 b(i,:)  = b(i+k-1,:);
 A(i+k-1,:)  = temp1;
 b(i+k-1,:)  = temp2;
 end
 for h =i+1:nl
 m = A(h,i)/A(i,i);
 A(h,:)  = A(h,:)-m*A(i,:);
 b(h,:)  = b(h,:)-m*b(i,:);
 end;
end;
% sustitución hacia atrás
x(n,:)=b(n,:)./A(n,n);
for i= n-1:-1:1
 x(i,:)  = (b(i ,:)-A(i,i+1:n)*x(i+1:n,:))./A(i,i);
end
```

Así que en este caso escribimos `gelim(A,b)`, con A, b definidos como en el ejercicio.

4.1.2 Descomposición LU

Este es un método que utiliza matrices triangulares como base para resolver matrices no singulares en general. Para resolver $Ax = b$ para A general, primero factorizamos A en el producto de dos matrices triangulares, $A = LU$, en donde L es triangular baja (*lower*) y U es triangular alta (*upper*). A esta factorización se le llama la descomposición LU de A.

Posteriormente reemplazamos el problema $Ax = b$ con el problema equivalente $LUx = b$. En la práctica, es usual que L tenga en la diagonal 1's (método de Doolittle), por lo tanto una matriz de 3×3 queda de la siguiente manera:

$$\begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Comenzamos encontrando $u_{11} = a_{11}$ y después resolviendo para los demás elementos de la primera fila de U y la primera columna de L. Como segundo paso, encontramos u_{22} y después el resto de la segunda fila de U y de la segunda columna de L. Siguiendo así determinamos todos los elementos de U y de L.

Si queremos la factorización LU de $A = \begin{bmatrix} 1 & 4 & 5 \\ 4 & 20 & 32 \\ 5 & 32 & 64 \end{bmatrix}$ escribimos el producto deseado

$$\begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} = \begin{bmatrix} 1 & 4 & 5 \\ 4 & 20 & 32 \\ 5 & 32 & 64 \end{bmatrix}$$

Empezamos resolviendo para la primera fila de U y la primera columna de L :

$$\begin{aligned} (1)u_{11} &= a_{11} = 1 \\ (1)u_{12} &= a_{12} = 4 \\ (1)u_{13} &= a_{13} = 5 \\ l_{21}u_{11} &= a_{21} = 4 \\ l_{31}u_{11} &= a_{31} = 5 \end{aligned}$$

Usando estos valores, encontramos la segunda fila de U y la columna de L:

$$\begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 5 & l_{32} & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 & 5 \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} = \begin{bmatrix} 1 & 4 & 5 \\ 4 & 20 & 32 \\ 5 & 32 & 64 \end{bmatrix}$$

$$(4)(4) + u_{22} = 20 \Rightarrow u_{22} = 4$$
$$(4)(5) + u_{23} = 32 \Rightarrow u_{23} = 12$$
$$(5)(4) + l_{32}u_{22} = 32 \Rightarrow l_{32} = 3$$

Finalmente sólo nos queda una desconocida en U :

$$\begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 5 & 3 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 & 5 \\ 0 & 4 & 12 \\ 0 & 0 & u_{33} \end{bmatrix} = \begin{bmatrix} 1 & 4 & 5 \\ 4 & 20 & 32 \\ 5 & 32 & 64 \end{bmatrix}$$

$$(5)(5) + (3)(12) + u_{33} = 64 \Rightarrow u_{33} = 3$$

Por lo que la descomposición queda:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 5 & 3 & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} 1 & 4 & 5 \\ 0 & 4 & 12 \\ 0 & 0 & 3 \end{bmatrix}$$

La función en Matlab es:

```
function [L,U] = LU(A)
[n, m] = size(A); % La dimensión de A
U = zeros(n, n); % Inicializar U
L = eye(n); % Inicializar L
for k = 1 :  n % computar el paso k
 U(k, k) = A(k, k) - L(k, 1:k-1)*U(1:k-1, k);
 for j = k+1 :  n
 U(k, j) = A(k, j) - L(k, 1:k-1)*U(1:k-1, j);
 L(j, k) = (A(j, k) - L(j, 1:k-1)*U(1:k-1, k))/U(k, k);
 end
end
```

Así que escribimos en la ventana de Matlab/Octave `[L,U]=LU(A)`.

octave:10> A=[1 4 5;4 20 32;5 32 64]

A =

1 4 5
4 20 32
5 32 64

octave:15> [L,U]=LU(A)
L =
1 0 0
4 1 0
5 3 1
U =
1 4 5
0 4 12
0 0 3

4.2 Métodos iterativos

4.2.1 Método de Gauss-Jacobi

Los métodos de descomposición para las ecuaciones lineales son métodos directos de solución. Sin embargo, dichos métodos pueden ser muy costosos para sistemas grandes. Para estos casos existen los métodos iterativos, los cuales son rápidos y buenos. Dichos métodos son valiosos, ya que las ideas básicas se generalizan para los sitemas no lineales.

El método de Gauss-Jacobi es uno de los más sencillos para resolver ecuaciones lineales. Comienza con la observación de que cada ecuación es una sola ecuación lineal, un tipo de ecuación en la cual podemaos resolver una variable en término de las otras.

Consideremos la ecuación de la primera fila de $Ax = b$:

$$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1$$

Podemos resolver para x_1 en términos de $(x_2, ..., x_n)$ si $a_{11} \neq 0$, resultando

$$x_1 = a_{11}^{-1}(b_1 - a_{12}x_2 - ... - a_{1n}x_n)$$

En general, si $a_{ii} \neq 0$, podemos usar la fila i de A para resolver para x_i, encontrando

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j \right)$$

Aparentemente este parece una manera diferente de expresar el sistema lineal, pero vamos a convertirlo a continuación en un método iterativo. Empecemos con un *guess* para x y usemos las soluciones de las ecuaciones solas para computar un nuevo *guess* de x. Suponiendo que $a_{ii} \neq 0$ podemos verlo como:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^k \right), \qquad \text{para } i = 1, ..., n \tag{4.1}$$

Este método reduce el problema de resolver n incógnitas simultanemente en n ecuaciones a resolver repetidamente n ecucaciones con una incógnita. Esto sólo define una sucesión de *guesses*, por lo que esperamos que (4.1) converja a la verdadera solución.

Veamos un ejemplo gráfico de este método. Primero consideremos un sistema de 2×2

$$\begin{aligned} 2x + y &= 6 \\ x + 2y &= 6 \end{aligned}$$

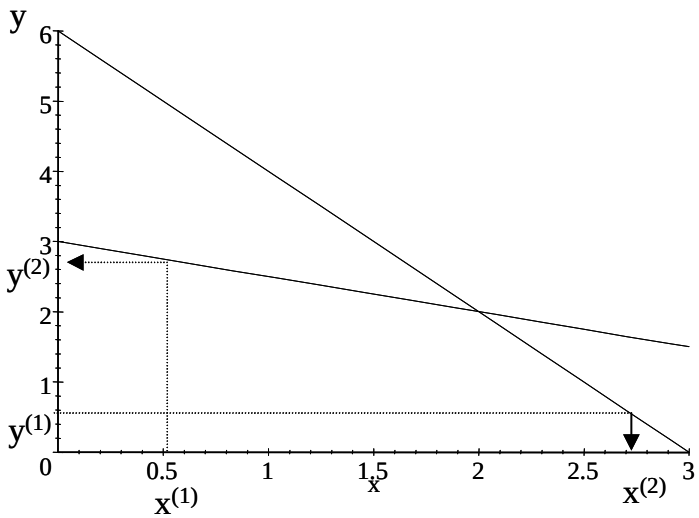
Con el método de Gauss-Jacobi, reescribimos el sistema como:

$$\begin{aligned} x &= -\frac{1}{2} y + 3 \\ y &= -\frac{1}{2} x + 3 \end{aligned}$$

Empezando con $x^1 = \frac{1}{2}$ y $y^1 = \frac{1}{2}$,la primera ecuación produce el siguiente estimado para x (usando y^1), y la segundo ecuación da el siguiente valor de y (usando x^1):

$$\begin{aligned} x^2 &= -\frac{1}{2} y^1 + 3 = -\frac{1}{4} + 3 = \frac{11}{4} \\ y^2 &= -\frac{1}{2} x^1 + 3 = -\frac{1}{4} + 3 = \frac{11}{4} \end{aligned}$$

Notemos que los nuevos valores de las variables no se usan hasta que empieza una nueva iteración. A esto se le llama *actualización simultánea.* La primera iteración se ilustra en la siguiente gráfica:



Gráfica 4.1. La primera iteración del método de Gauss-Jacobi

Veamos ahora en Matlab el siguiente sistema de 3×3

$$\begin{aligned} 2x_1 - x_2 + x_3 &= -1 \\ x_1 + 2x_2 - x_3 &= 6 \\ x_1 - x_2 - 2x_3 &= -3 \end{aligned}$$

el cual se convierte en

$$\begin{aligned} x_1 &= .5x_2 - .5x_3 - .5 \\ x_2 &= -.5x_1 + .5x_3 + 3 \\ x_3 &= -.5x_1 + .5x_2 - 1.5 \end{aligned}$$

En notación matricial, $Ax = b$, es

$$\begin{bmatrix} 2 & -1 & 1 \\ 1 & 2 & -1 \\ 1 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 6 \\ -3 \end{bmatrix}$$

que se transforma a

$$\begin{bmatrix} x_1^k \\ x_2^k \\ x_3^k \end{bmatrix} = \begin{bmatrix} 0 & .5 & -.5 \\ -.5 & 0 & .5 \\ -.5 & .5 & 0 \end{bmatrix} \begin{bmatrix} x_1^{k-1} \\ x_2^{k-1} \\ x_3^{k-1} \end{bmatrix} + \begin{bmatrix} -.5 \\ 3 \\ -1.5 \end{bmatrix}$$

Empezando con $x^0 = (0, 0, 0)$ encontramos

$$\begin{bmatrix} x_1^1 \\ x_2^1 \\ x_3^1 \end{bmatrix} = \begin{bmatrix} 0 & .5 & -.5 \\ -.5 & 0 & .5 \\ -.5 & .5 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -.5 \\ 3 \\ -1.5 \end{bmatrix} = \begin{bmatrix} -.5 \\ 3 \\ -1.5 \end{bmatrix}$$

Para la segunda iteración tendríamos

$$\begin{bmatrix} x_1^2 \\ x_2^2 \\ x_3^2 \end{bmatrix} = \begin{bmatrix} 0 & .5 & -.5 \\ -.5 & 0 & .5 \\ -.5 & .5 & 0 \end{bmatrix} \begin{bmatrix} -.5 \\ 3 \\ -1.5 \end{bmatrix} + \begin{bmatrix} -.5 \\ 3 \\ -1.5 \end{bmatrix} = \begin{bmatrix} 1.75 \\ 2.5 \\ .25 \end{bmatrix}$$

El método de Gauss-Jacobi va a converger en 20 iteraciones al vector

$$x = [1.0002 \quad 2.0001 \quad -0.9997]'$$

usando $\varepsilon = 0.001$, siendo la solución verdadera $x^* = [1 \quad 2 \quad -1]'$.

La función en Matlab es:

```
function x=gjacobi(A,b,x0,tol,max)
% Solución al sistema de ecuaciones lineales Ax = b
% usando el algoritmo iterativo de Gauss-Jacobi
% Inputs:
% A matriz de coeficientes (nxn)
% b lado derecho (nx1)
% x0 solución inicial (nx1)
% tol Tolerancia permitida al computar el cero
% max Número máximo de iteraciones permitido
% Output:
% x Vector solución (nx1)
[n m] = size (A);
```

```
xold = x0;
C =-A;
for i = 1 :  n
 C(i ,i) = 0;
 end
for i = 1 :n
 C(i ,:)  =C(i,:)/A(i,i);
end
for i = 1 :n
 d(i ,1) = b(i)/A(i ,i);
end
i = 1;
disp( ' i x1 x2 x3 ....');
while (i <= max)
 xnew = C*xold + d;
 if norm(xnew-xold) <=tol
 x = xnew;
 disp('El método de Gauss-Jacobi ha convergido');
 return;
 else
 xold=xnew;
 end
 disp([i xnew']);
 i=i+1;
end
disp('El método de Gauss-Jacobi NO ha convergido');
disp('Resulatdos después del número máximo de iteracione');
x=xnew;
```

Veamoslo para nuestro ejercicio, en donde escribimos `gjacobi(A,b,[0;0;0],0.001,20)`

octave:23> gjacobi(A,b,[0;0;0],0.001,20)
i x1 x2 x3 ....
1.00000 -0.50000 3.00000 -1.50000
2.00000 1.75000 2.50000 0.25000

3.00000 0.62500 2.25000 -1.12500
4.00000 1.18750 2.12500 -0.68750
5.00000 0.90625 2.06250 -1.03125
6.00000 1.04688 2.03125 -0.92188
7.00000 0.97656 2.01562 -1.00781
8.00000 1.01172 2.00781 -0.98047
9.00000 0.99414 2.00391 -1.00195
10.00000 1.00293 2.00195 -0.99512
11.00000 0.99854 2.00098 -1.00049
12.00000 1.00073 2.00049 -0.99878
13.00000 0.99963 2.00024 -1.00012
El método de Gauss-Jacobi ha convergido
ans =
1.00018
2.00012
-0.99969

Generalización para problemas no lineales

Ahora generalicemos este método para problemas no lineales. Dado el valor de la iteración k, x^k, usamos la ecuación i para computar el componente i de la incógnita x^{k+1}, nuestra siguiente iteración. Formalemente, x^{k+1} se define en términos de x^k por las ecuaciones:

$$\begin{aligned} f^1 &= (x_1^{k+1}, x_2^k, x_3^k, ..., x_n^k) = 0 \\ f^2 &= (x_1^k, x_2^{k+1}, x_3^k, ..., x_n^k) = 0 \\ &\vdots \\ f^n &= (x_1^k, x_2^k, x_3^k, ..., x_{n-1}^k, x_n^{k+1}) = 0 \end{aligned} \tag{4.2}$$

Cada ecuación en (4.2) es una sola ecuación no lineal con una incógnita, por lo que tenemos las mimas ventajas que para el caso lineal.

El método de Gauss-Jacobi se ve afectado por el esquema de indexación de las variables y de las ecuaciones, por lo que no hay una elección natural para decidir qué variable será la primera y cuál ecuación la primera. Por lo tanto tenemos $\frac{n(n-1)}{2}$ esquemas diferentes de Gauss-Jacobi, y es muy difícil determinar cual es el mejor esquema. Sin embargo, hay algunas cosas que podemos hacer, por ejemplo, si alguna ecuación sólo depende de una incógnita,

entonces esa ecuación debería ser la primera y esa variable la primera. En general, lo que vamos a queres es que (4.2) se asemeje a una forma no lineal de sustitución hacia atrás.

Cada paso en el método de Gauss-Jacobi es una ecuación no lineal y se resuelve por lo general con un método iterativo. No tienen tanto sentido resolver cada ecuación exactamente, dado que queremos resolver cada ecuación en la siguiente iteración. Por lo que podríamos simplemente resolver aproximando cada ecuación en (4.2), lo cual conlleva al método lineal Gauss-Jacobi, el cual utiliza un sólo paso del Newton para aproximar los componentes de x^{k+1}, resultando en el siguiente esquema:

$$x_i^{k+1} = x_i^k - \frac{f^i(x^k)}{f^i_{x_i}(x^k)}, \qquad i = 1, ..., n$$

Dadas las desventajas del método de Gauss-Jacobi, pasemos a continuación al siguiente método, donde daremos un ejemplo.

4.2.2 Algorítmo de Gauss-Seidel

Para el caso de un sistema de ecuaciones lineales, el método de Gauss-Jacobi usamos un nuevo *guess* para x_i^k, x_i^{k+1}, sólo después de que hemos computado todo el vector de los nuevos valores x^{k+1}. Este retraso para usar la nueva información puede no ser sensible. Supongamos que x^* es la solución a $Ax = b$. Si, como esperamos, x_i^{k+1} es una mejor estimación de x_i^* que x_i^k, entonces usar x_i^{k+1} para computar x_{i+1}^{k+1} en (4.1) parecería que es mejor que usar x_i^k. El argumento intuitivo es que la nueva información sobre x_i^* debe ser usarse lo más pronto posible.

La idea básica del método de Gauss-Seidel es usar una nueva aproximación de x_i^* en cuanto esté disponible. Es decir, si nuestro nuevo *guess* para x^* es x^k, entonces compuatmos nuestro nuevo *guess* para x_1 como en (4.1):

$$x_1^{k+1} = a_{11}^{-1}\left(b_1 - a_{12}x_2^k - ... - a_{1n}x_n^k\right)$$

pero usamos este nuevo *guess* para x_1^* inmediantamente cuando computemos los otros nuevos componentes de x^{k+1}. En particular, nuestra ecuación para x_2^{k+1} derivada de la segunda fila de A se volvería

$$x_2^{k+1} = a_{22}^{-1}\left(b_2 - a_{21}x_1^{k+1} - a_{23}x_3^k - ... - a_{2n}x_n^k\right)$$

En general, para resolver $Ax = b$ con el método de Gauss-Seidel, definimos la sucesión $\left\{x^k\right\}_{k=1}^{\infty}$ por la iteración

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^k \right), \qquad \text{para } i = 1, ..., n \tag{4.3}$$

Podemos observar claramente cómo los nuevos componentes de x^{k+1} son usado inmediatamente de que son computados, dado que x_i^{k+1} es usado para computar x_j^{k+1} para $j > i$.

Notémos que en este método el orden en el cual resolvemos para los componentes sucesivos de x importa, mientras que el orden no importa en el método de Gauss-Jacobi. Por lo que esto le da al método de Gauss-Seidel más flexibilidad, en particular si un orden no converge, entonces tratamos otro orden.

Veamos el mismo ejemplo de 2×2 que vimos en la sección anterior, es decir,

$$\begin{aligned} 2x + y &= 6 \\ x + 2y &= 6 \end{aligned}$$

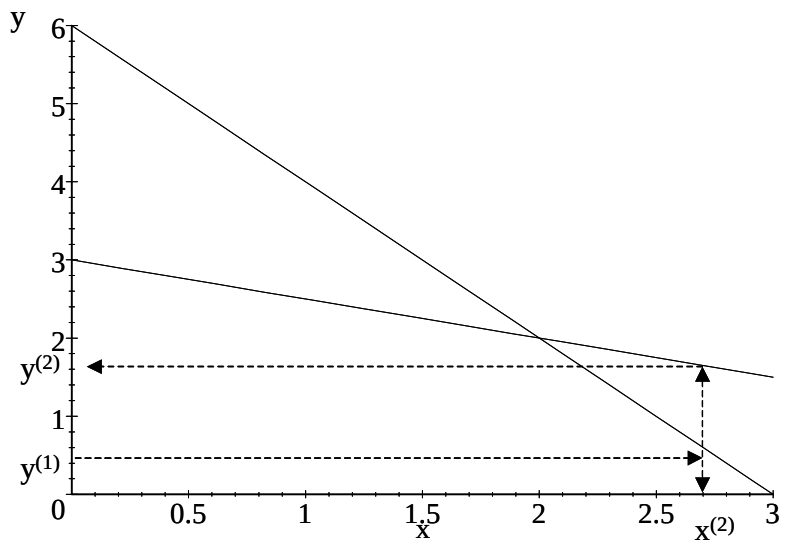
Con el método de Gauss-Seidel, reescribimos el sistema como:

$$\begin{aligned} x &= -\frac{1}{2}y + 3 \\ y &= -\frac{1}{2}x + 3 \end{aligned}$$

Empezando con $x^1 = \frac{1}{2}$ y $y^1 = \frac{1}{2}$,la primera ecuación produce el siguiente estimado para x (usando y^1), y la segundo ecuación da el siguiente valor de y (usando x^2):

$$\begin{aligned} x^2 &= -\frac{1}{2}y^1 + 3 = -\frac{1}{4} + 3 = \frac{11}{4} \\ y^2 &= -\frac{1}{2}x^2 + 3 = -\frac{11}{8} + 3 = \frac{13}{8} \end{aligned}$$

La primera iteración se ilustra en la siguiente gráfica:



Gráfica 4.2. La primera iteración del método de Gauss-Seidel

Veamos la función en Matlab de Gauss-Seidel usando el mismo ejercicio que para Gauss-Jacobi:

```
function x=gseidel(A,b,x0,tol,max)
% Solución al sistema de ecuaciones lineales Ax = b
% usando el algoritmo iterativo de Gauss-Seidel
% Inputs:
% A matriz de coeficientes (nxn)
% b lado derecho (nx1)
% x0 solución inicial (nx1)
% tol Tolerancia permitida al computar el cero
% max Número máximo de iteraciones permitido
% Output:
% x Vector solución (nx1)
[n m] = size (A);
x = x0;
C=-A;
for i = 1 :  n
 C(i ,i) = 0;
end
for i = 1 :n
```

```
 C(i,1:n) = C(i,1:n)/A(i,i);
end
for i = 1:n
 r(i,1) = b(i)/A(i,i);
end
i = 1;
disp( ' i x1 x2 x3 ....');
while (i <= max)
 xold =x;
 for j = 1:n
 x(j) = C(j, :)*x + r(j);
 end
 if norm(xold - x) <= tol
 disp('El método de Gauss-Seidel ha convergido');
 return;
 end
 disp([i x']);
 i=i+1;
end
disp('El método de Gauss-Seidel NO ha convergido');
disp('Resulatdos después del número máximo de iteracione');
x=xnew;
```

En la ventana de Matlab escribimos `gseidel(A,b,[0;0;0],0.001,10)` y observamos que nos sale igual que para el caso de Gauss-Jacobi.

octave:31> gseidel(A,b,[0;0;0],0.001,10)
i x1 x2 x3 ....
1.00000 -0.50000 3.25000 0.37500
2.00000 0.93750 2.71875 -0.60938
3.0000 1.1641 2.1133 -1.0254
4.0000 1.0693 1.9526 -1.0583
5.0000 1.0055 1.9681 -1.0187
6.00000 0.99339 1.99395 -0.99972
7.00000 0.99684 2.00172 -0.99756

8.00000 0.99964 2.00140 -0.99912

9.00000 1.00026 2.00031 -0.99997

El método de Gauss-Seidel ha convergido

ans =

1.0001

1.9999

-1.0001

Generalización para problemas no lineales

En el método de Gauss-Jacobi para ecuaciones no lineales usamos otra vez el nuevo *guess* de x_i, x_i^{k+1} , sólo después de haber computado todo el vector de nuevos valores x^{k+1}. La idea básica del método de Gauss-Seidel es de nuevo, usar el nuevo *guess* de x_i en cuanto esté disponible. En el caso no lineal, esto implica que dado x^k, construimos x^{k+1} componente a componente resolviendo los problemas unidimensionales en sucesión:

$$\begin{aligned} f^1 &= (x_1^{k+1}, x_2^k, x_3^k, ..., x_n^k) = 0 \\ f^2 &= (x_1^{k+1}, x_2^{k+1}, x_3^k, ..., x_n^k) = 0 \\ &\vdots \\ f^{n-1} &= (x_1^{k+1}, x_2^{k+1}, ..., x_{n-1}^{k+1}, x_n^k) = 0 \\ f^n &= (x_1^{k+1}, x_2^{k+1}, ..., x_{n-1}^{k+1}, x_n^{k+1}) = 0 \end{aligned}$$

Resolvemos $f^1, f^2, ..., f^n$ en sucesión, usando inmediatamente cada nuevo componente. Notemos que ahora la notación de índices importa, ya que afecta la manera en que los últimos resultados dependen de los primeros. Cabe señalar que también podemos dar una forma lineal del método de Gauss-Seidel

$$x_i^{k+1} = x_i^k - \left(\frac{f^i}{f_{x_i}^i}\right)(x_1^{k+1}, ..., x_{i-1}^{k+1}, x_i^k, ..., x_n^k),$$

donde $i = 1, 2, ..., n$, con el fin de optimizar en los costos de optimización de cada iteración.

Estos métodos Gaussianos son usados con frecuencia, pero tienen algunos problemas. Como en el caso lineal, son métodos riesgosos para usar si el sistema no es diagonalmente dominante, es decir, si cada elemento de la diagonal es mayor a la suma de las magnitudes

de los elementos no diagonales en su fila, matemáticamente hablando,

$$\sum_{j \neq i} |a_{ij}| < |a_{ii}| \qquad\qquad i = 1, ..., n$$

Teorema 4.1 Si A es diagonalmente dominante, tanto el esquema iterativo de Gauss-Jacobi como el de Gauss-Seidel convergen para todos los valores iniciales.

Aplicación para sistemas de ecuaciones en diferencia de segundo orden

Recordemos un poco de los sistemas de ecuaciones en diferencia de segundo orden no lineal, en donde tenemos una ecuación de la forma $g(x_t, x_{t+1}, x_{t+2})$ con una condición inicial, es decir, x_0 dado y una condición terminal $\lim_{t\to\infty} x_t = x^*$. Si suponemos que en un tiempo finito T se llega a x^*, entonces vamos a tener el siguiente sistema de ecuaciones no lineal:

$$\begin{aligned} g(x_0, x_1, x_2) &= 0 \\ g(x_1, x_2, x_3) &= 0 \\ &\vdots \\ g(x_{t-2}, x_{t-1}, x^*) &= 0 \end{aligned}$$

que ya viene siendo un sistema con $T-2$ ecuaciones y $T-2$ incóngitas, con lo cual podemos ahora si aplicar nuestros metodos numéricos.

Dado que en economía nosotros veremos funciones que dependerán de varias variables, como el capital, el consumo, etc., y en donde el sistema de funciones será no lineal, veamos a continuación un ejemplo sencillo del modelo de crecimiento neoclásico con planificador social.

El planficador central va a querer maximizar lo siguiente:

$$\begin{aligned} & & \max \sum_{t=0}^{\infty} \beta^t u(c_t) & & (4.4) \\ s.a. \qquad c_t + i_t &= f(k_t) & & \forall t \\ k_{t+1} &= (1-\delta)k_t + i_t & & \forall t \end{aligned}$$

La función en Matlab quedaría así:

```
% gseide.m Programa para resolver una simple versión del problema del planificad
social
% usando Gauss-Seidel.  El problema a resolver es:
```

```
%
%          inf
% max sum beta^t ln(ct)
%          t=0
%
% s.a.  ct+it = A*k^alpha
% kt+1 = (1-delta)*kt+it
%
% con k0 dado
%
%
clear
% Parámetros del modelo
alpha = 0.35;
beta = 0.95;
delta = 0.06;
A = 10;
% Otros parámetros (máximo de iteraciones, tolerancia deseada y
% número de periodos antes de alcanzar el estado estacionario)
maxit = 250;
crit = 1e-2;
T = 100;
% Computa el capital de estado estacionario (kss), el capital inicial (k0)
y
% el guess inicial para la sucesión de kt (kg)
kss = ((A*beta*alpha)/(1-(1-delta)*beta))^(1/(1-alpha));
k0 = (1/2)*kss;
kg = zeros(T+1,1);
ksol = zeros(T,1);
kg(1) = k0;
ksol(1) = k0;
for t=2:T+1
 kg(t) = kg(t-1)+(kss-k0)/T;
end
```

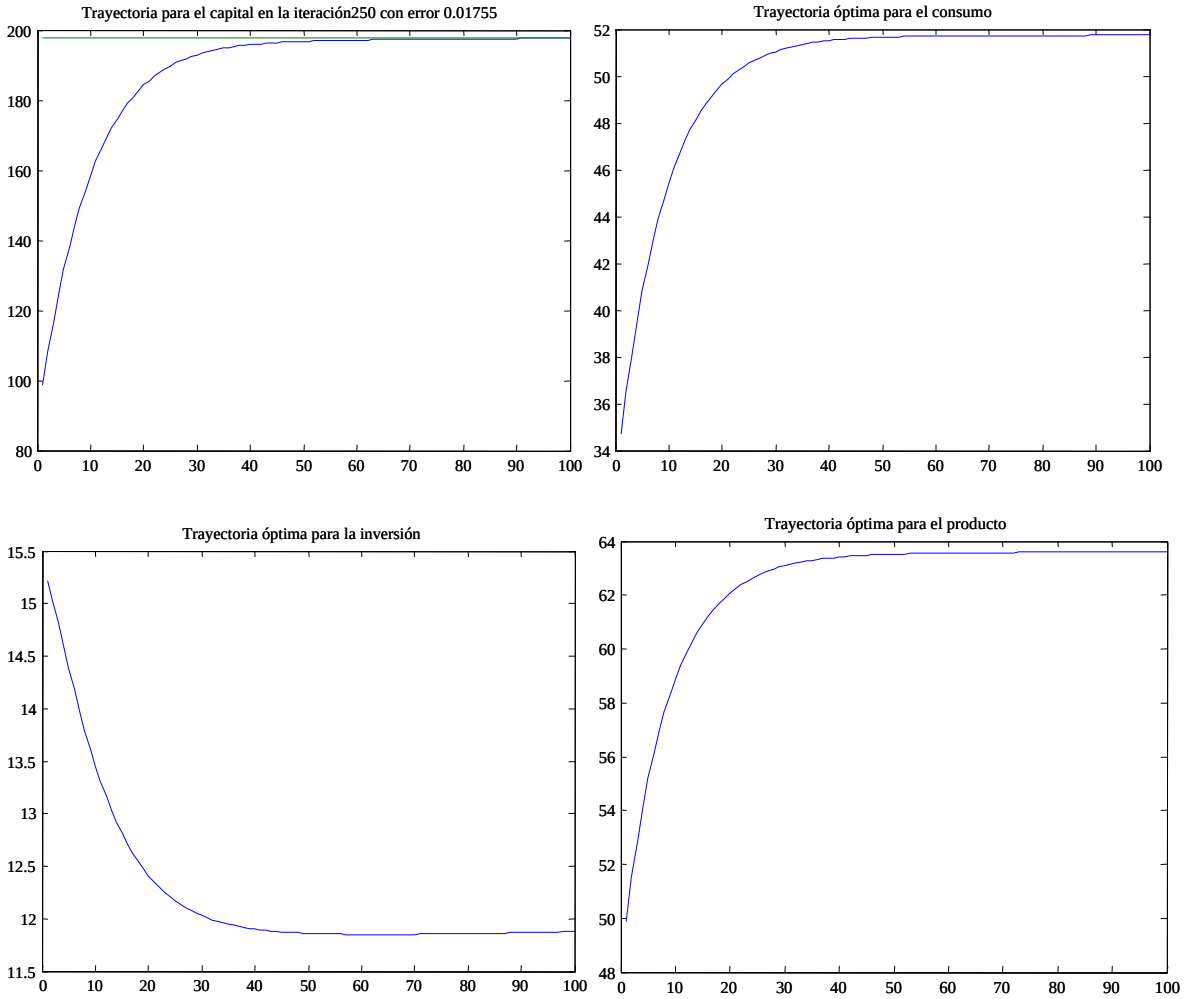
```
% Iteraciones de Gauss-Seidel
for i=1:maxit
 for t=2:T
 param = [alpha;beta;delta;A;ksol(t-1);kg(t+1)];
 ksol(t) = secant_sys('fgs',kss/2,param,1e-5,20);
 end
 dist = norm(kss-ksol(T));
 plot(1:T,ksol(1:T),1:T,kss*ones(T,1))
 title(['Trayectoria para el capital en la iteración',int2str(i),...
 ' con error ',num2str(dist)])
 pause(0.01)
 if dist<crit
 break
 end
 kg(1:T) = ksol(1:T);
end
% Gráficas de las trayectorias óptimas para el producto (yt),
% la inversión (it) y el consumo (ct)
kt = kg(1:T+1);
figure(2)
yt = A*kt(1:T).^alpha;
plot(1:T,yt)
title('Trayectoria óptima para el producto')
figure(3)
it = kt(2:T+1)-(1-delta)*kt(1:T);
plot(1:T,it)
title('Trayectoria óptima para la inversión')
figure(4)
ct = yt-it;
plot(1:T,ct)
title('Trayectoria óptima para el consumo')
```

Y la función de producción ncesesaria para correr este programa la llamaremos `fgs.m` que viene dada por:

```
function y = fgs(kt1,p)
```

```
alpha = p(1);
beta = p(2);
delta = p(3);
A = p(4);
kt = p(5);
kt2 = p(6);
y = A*kt1^alpha+(1-delta)*kt1-beta*(A*kt^alpha-kt1+ ...
 (1-delta)*kt)*(A*alpha*kt1^(alpha-1)+(1-delta))-kt2;
```

Notemos que estamos incluyendo todos los parámetros en nuestro archivo `gseide.m`, por lo que en la ventana de Matlab sólo escribimos `gseide.m`. Los resultado del capital con las 250 iteraciones dadas sería que $k_t = 197.7$ y las gráficas quedarían:



5 Programación dinámica numérica

5.1 Función de valor

Esta sesión tiene como objetivo presentar el método de iteración de la función de valor. Para ello, empezamos reformulando la definiciones de equilibrio en términos recursivos y presentando algunos resultados de la literatura sobre programación dinámica. Estos resultados constituyen la base teórica para el método de iteración de la función de valor, que se presenta a continuación para resolver el problema del planificador social en un contexto determinístico. Posteriormente se verá la extensión para resolver y simular un modelo estocástico con shocks tecnológicos.

Recordando la ecuación (4.4) y resolviendo el problema del planificador central, nos vamos a encontrar con la siguiente ecuación de Euler

$$\beta f'(k_t)u'[f(k_t) - k_{t+1}] = u'[f(k_{t-1}) - k_t], \qquad t = 1, 2, ..., T \tag{5.1}$$

y las condiciones iniciales y finales

$$k_{T+1} = 0, \qquad k_0 > 0 \text{ dado}$$

La ecuación (5.1) es una ecuación en diferencia de segundo orden en k_t, por lo tanto su solución tiene una familia de soluciones de dos parámetros. Sin embargo, el único óptimo para el problema de maximización es aquella solución en esta familia que además satisface las condiciones iniciales y terminales.[5] Una manera de resolver el problema es que dado que supusimos que el tiempo es finito, resolvemos el problema y la trayectoria de capital será

$$k_{t+1} = \alpha\beta \frac{1 - (\alpha\beta)^{T-1}}{1 - (\alpha\beta)^{T-t+1}} k_t^{\alpha}, \qquad t = 0, 1, ..., T$$

si lo queremos convertir en un problema de horizonte infinito, sólo tomamos límites y nos queda

$$k_{t+1} = \alpha\beta k_t^{\alpha}, \qquad t = 0, 1, ... \tag{5.2}$$

De hecho esta conjetura va a ser la correcta y esta va a ser la única solución del

[5] Capítulo 2 de SLP

problema. A la ecuación (5.2) se le llama función de política y se puede escribir de también como $k_{t+1} = g(k_t)$ para $t = 0, 1, ...$, en donde $g : R_+ \to R_+$.

Aunque empezamos nuestro problema escogiendo sucesiones infinitas para el consumo y el capital $\{(c_t, k_{t+1})\}_{t=0}^{\infty}$, el problema que en realidad enfrenta el planificador central en el periodo $t = 0$ es el de escoger sólo el consumo actual, c_0, y el capital de mañana k_1, el resto puede esperar hasta mañana. Así que si conocemos las preferencias sobre estos dos bienes, entonces podemos realizar nuestro sencillo problema de maximizar (c_0, k_1) sujerto a nuestra restricción presupuestal.

Dado k_0 podemos definir una función $v : R_+ \to R$ tomando $v(k_0)$ como el valor de la función objetivo maximizada para cada $k_0 \geq 0$. Una función de este tipo se le llama *función de valor*. Con v definida de esta manera, $v(k_1)$ daría el valor de de la utilidad del periodo 1 usando k_1 y $\beta v(k_1)$ sería el valor de esta utilidad descontada al periodo 0. Entonces el problema del planeador central en el periodo 0 usando esta función de valor sería

$$\max_{c_0, k_1} [u(c_0) + \beta v(k_1)]$$

$$\begin{aligned} s.a. \quad c_0 + k_1 &\leq f(k_0) \\ c_0, k_1 &\geq 0 \\ k_0 &> 0 \text{ dado} \end{aligned} \tag{5.3}$$

Si (5.3) provee una solución para el problema, entonces $v(k_0)$ debe de ser la función objetivo maximizadora de (5.3) también. Por lo tanto, v debe de satifacer

$$v(k_0) = \max_{0 \leq, k_1 \leq f(k_0)} \{u[f(k_0) - k_1] + \beta v(k_1)\}$$

Notemos que cuando vemos el problema de una manera recursiva, los subíndices de tiempo no importan, por lo que podemos reescribir el problema como

$$v(k) = \max_{0 \leq, k' \leq f(k)} \{u[f(k) - k'] + \beta v(k')\}$$

Esta ecuación en la función desconocida v se llama *ecuación funcional*. El estudio de los problemas de optimización dinámica a través del análisis de estas ecuaciones funcionales se llama *programación dinámica*.

5.2 Formulación Recursiva y Programación Dinámica[6]

Un *Equilibrio General Competitivo Recursivo* es un conjunto de funciones $v(k,K)$, $c(k,K)$, $i(k,K)$, $k'(k,K)$, precios $w(K)$ y $r(K)$ y ley de movimiento $\Gamma(K)$ tales que:

i) Dadas las funciones w, r y Γ, la función de valor $v(k,K)$ resuelve la *ecuación de Bellman*:

$$v(k,K) = \max_{c,i,k'} \{u(c) + \beta v(k',K')\} \tag{5.4}$$

$$\begin{aligned} s.t. \quad c+i &= w(K) + r(K)\,k \\ k' &= (1-\delta)\,k + i \\ K' &= \Gamma(K) \end{aligned}$$

y las funciones $c(k,K)$, $i(k,K)$ y $k'(k,K)$ son *reglas de decisión óptimas* para este problema.

ii) Para todo K, los precios satisfacen las condiciones marginales:

$$r(K) = f'(K) \tag{5.5}$$

$$w(K) = f(K) - f'(K)\,K \tag{5.6}$$

iii) Para todo K, hay igualdad entre oferta y demanda:

$$f(K) = c(K,K) + i(K,K) \tag{5.7}$$

iv) Para todo K, ley de movimiento agregada y comportamiento individual son consistentes:

$$\Gamma(K) = k'(K,K) \tag{5.8}$$

[6]**Esta sección fue tomada de las notas de Carlos Urrutia "Métodos Numéricos para Resolver Modelos Macroeconómicos Dinámicos".**

Resolver un Equilibrio Competitivo en su formulación recursiva consiste entonces en encontrar un conjunto de funciones que satisfagan las condiciones anteriores. En particular, una vez encontradas las reglas de decisión óptimas y partiendo de un k_0 dado, podemos reconstruir las secuencias para las principales variables de manera recursiva:

$$\begin{aligned} k_1 &= k'(k_0, k_0) \\ k_2 &= k'(k_1, k_1) = k'(k'(k_0, k_0), k'(k_0, k_0)) \end{aligned}$$

y así sucesivamente para k_t, $t = 3, 4, ...$ El *Principio de Optimalidad* asegura que la secuencia resultante es igual a la que obtendríamos resolviendo el equilibrio en forma de secuencias que vimos en la sesión anterior.

De manera similar, definimos un *Optimo de Pareto Recursivo* como un conjunto de funciones $v(k)$, $c(k)$, $i(k)$, $k'(k)$ que resuelven la ecuación de Bellman del planificador social:

$$v(k) = \max_{c,i,k'} \{u(c) + \beta v(k')\} \tag{5.9}$$

$$\begin{aligned} s.t. \quad c + i &= f(k) \\ k' &= (1-\delta)k + i \end{aligned}$$

La equivalencia entre el equilibrio competitivo y el problema del planificador social se sigue manteniendo en este contexto.

5.2.1 Programación Dinámica

Las ecuaciones de Bellman (5.4) y (5.9) son ejemplos de un problema más general. Sea x un vector columna de n componentes, X un subconjunto de R^n, la función de retorno $F : X \times X \to R$ y la correspondencia - $: X \to X$. Con esos elementos, definimos la función de valor $v : X \to R$ como la solución de la ecuación de Bellman:

$$v(x) = \max_{y} \{F(x, y) + \beta v(y)\} \tag{5.10}$$

$$s.t. \quad y \in \text{-}\ (x)$$

para todo $x \in X$. Dada la función de valor, podemos definir la regla de decisión óptima $g : X \to X$ como:

$$g(x) = \arg\max_{y} \quad \{F(x, y) + \beta v(y)\} \tag{5.11}$$

$$s.t. \quad y \in \text{-}\ (x)$$

es decir, la función que satisface:

$$v(x) = F(x, g(x)) + \beta v(g(x))$$

Las propiedades de las funciones v y g han sido analizadas en la literatura sobre programación dinámica[7]. En ella, se demuestra que si (i) X es un conjunto convexo; (ii) - (x) es un conjunto compacto y no-vacío, para todo $x \in X$; (iii) la correspondencia - es convexa y continua; (iv) la función F es cóncava, acotada y continua; y (v) $\beta < 1$; entonces:

1. existe una única función v que satisface (5.10);
2. v es continua, acotada y estrictamente cóncava;
3. existe una única función g que resuelve (5.11);
4. g es continua.

En la mayor parte de ejemplos prácticos que veremos, incluyendo el Modelo de Crecimiento Neoclásico, las restricciones impuestas en las preferencias y tecnología aseguran que los supuestos (i)-(v) se cumplen.

Aparte de asegurarnos existencia y unicidad de la solución, el método de programación dinámica nos ofrece un procedemiento para encontrar dicha solución. Definamos el operador $T : B(X) \to B(X)$, en donde $B(X)$ es el espacio de funciones acotadas definidas en X, de la siguiente manera:

[7]Una referencia completa, aunque bastante técnica, puede encontrarse en Stokey y Lucas (1989).

$$T\left[v\left(x\right)\right] = \max_{y} \quad \left\{F\left(x,y\right) + \beta v\left(y\right)\right\} \tag{5.12}$$

$$s.t. \quad y \in \text{-} \ \left(x\right)$$

Encontrar la función de valor que resuelve (5.10) es equivalente a encontrar un punto fijo del operador T, es decir, una función v tal que $T\left[v\right] = v$.

Bajo ciertas condiones técnicas, el operador T es una *contracción*[8]. Por lo tanto, partiendo de cualquier función $v^0 \in B\left(X\right)$, la secuencia v^n definida por:

$$\begin{aligned} v^1 &= Tv^0 \\ v^2 &= Tv^1 = T^2v^0 \\ &\text{.....} \end{aligned}$$

converge hacia el punto fijo v.

5.3 Iteración de la Función de Valor

Volvamos ahora al problema del planificador social (5.9) en su froma recursiva. Reemplazando las restricciones en la función objetivo, tenemos:

$$v\left(k\right) = \max_{k'} \quad \left\{u\left[f\left(k\right) + \left(1-\delta\right)k - k'\right] + \beta v\left(k'\right)\right\} \tag{5.13}$$

$$s.t. \quad k' \in \left[0, f\left(k\right) + \left(1-\delta\right)k\right]$$

[8]El operador T es una contracción con módulo β si existe un número $\beta \in (0,1)$ tal que $\left\|Tf\left(x\right) - Tg\left(x\right)\right\| \leq \beta \left\|f\left(x\right) - g\left(x\right)\right\|$ para todo $f, g \in B\left(X\right)$, $x \in X$.

Las llamadas *condiciones de Blackwell* suficientes para una contracción son: monotonicidad del operador T, en el sentido que $Tf \leq Tg$ si $f\left(x\right) \leq g\left(x\right)$ para todo $x \in X$, y descuento, en el sentido que existe un número $\beta \in (0,1)$ tal que $T\left[f + a\right]\left(x\right) \leq Tf\left(x\right) + \beta a$ para todo $f \in B\left(X\right)$, $x \in X$ y $a \geq 0$. Nuevamente, los supuestos (i)-(v) mencionados anteriormente garantizan su cumplimiento.

un problema equivalente a la ecuación de Bellman general (5.10) con $x = k$, $y = k'$, $F(x, y) = u[f(x) + (1-\delta)x - y]$ y - $(x) = [0, f(x) + (1-\delta)x]$.

De acuerdo con la teoría de la programación dinámica que revisamos brevemente, partiendo de cualquier función v^0 (por ejemplo, $v^0(k) = 0$) la secuencia v^n definida por

$$v^{n+1}(k) = \max_{k'} \quad \{u[f(k) + (1-\delta)k - k'] + \beta v^n(k')\} \tag{5.14}$$

$$s.t. \quad k' \in [0, f(k) + (1-\delta)k]$$

converge a la solución v cuando n tiende a infinito. Esto constituye de por sí un método iterativo en un espacio de funciones; en esta sección veremos como implementarlo numéricamente.

5.3.1 Implementación Numérica

Empezamos definiendo una malla de valores para k, es decir, un vector $K = (K_1, K_2, ..., K_p)$ con $K_1 = K_{\min}$ y $K_p = K_{\max}$. Por simplicidad, podemos usar puntos igualmente espaciados, con $K_2 = K_{\min} + \eta$, $K_3 = K_{\min} + 2\eta$ y así sucesivamente, en donde $\eta = (K_{\max} - K_{\min})/(p-1)$ Mientras más puntos usemos, es decir mientras p sea mayor y η más pequeño, la aproximación será más precisa.

Luego, definimos una matriz M de la siguiente manera:

$$M = \begin{bmatrix} F(K_1, K_1) & F(K_1, K_2) & ..... & F(K_1, K_p) \\ F(K_2, K_1) & F(K_2, K_2) & ..... & F(K_2, K_p) \\ ..... & ..... & ..... & ..... \\ F(K_p, K_1) & F(K_p, K_2) & ..... & F(K_p, K_p) \end{bmatrix}$$

en donde $F(x, y) = u[f(x) + (1-\delta)x - y]$. La matriz M contiene la función de retorno evaluada para cada posible $k \in K$ y $k' \in K$. Sin embargo, sabemos que no todos esos valores son posibles. La restricción $k' \in [0, f(k) + (1-\delta)k]$ nos permite eliminar algunas celdas de la matriz que son inalcanzables para el planificador social, imponiéndole un valor cero[9]. Hacemos entonces:

[9]O un valor negativo (como por ejemplo -1000). El punto es impedir que el programa escoja esa celda al calcular la maximización.

$$M_{ij} \equiv F\left(K_i, K_j\right) = 0 \qquad \text{si} \quad K_j > f\left(K_i\right) + \left(1-\delta\right) K_i$$

Una vez construída la matriz M, el resto del procedimiento es una simple iteración. Empezamos con cualquier vector columna $V^0 \in R^p$ (por ejemplo, $V^0 = 0$) e iniciamos el siguiente algoritmo con $s = 0$:

1. Dado el vector $V^s = \left(V_1^s, V_2^s, ..., V_p^s\right)$ y la matriz M, calcular V^{s+1} de la siguiente manera:

$$V^{s+1} = \max \begin{bmatrix} M_{11} + \beta V_1^s & M_{12} + \beta V_2^s & ... & M_{1p} + \beta V_p^s \\ M_{21} + \beta V_1^s & M_{22} + \beta V_2^s & ... & M_{2p} + \beta V_p^s \\ ..... & ..... & ... & ..... \\ M_{p1} + \beta V_1^s & M_{p2} + \beta V_2^s & ... & M_{pp} + \beta V_p^s \end{bmatrix}$$

   donde el máximo es calculado por fila, es decir:

$$V^{s+1} = \begin{bmatrix} \max\left\{M_{11} + \beta V_1^s, M_{12} + \beta V_2^s, ..., M_{1p} + \beta V_p^s\right\} \\ \max\left\{M_{21} + \beta V_1^s, M_{22} + \beta V_2^s, ..., M_{2p} + \beta V_p^s\right\} \\ ..... \\ \max\left\{M_{p1} + \beta V_1^s, M_{p2} + \beta V_2^s, ..., M_{pp} + \beta V_p^s\right\} \end{bmatrix}$$

2. Evaluar $\|V^{s+1} - V^s\|$. Si la norma es mayor que el criterio de tolerancia permitido, regresar al paso 1 con $s = s + 1$. En caso contrario, el procedimiento termina con la solución $V = V^{s+1}$.

Una vez obtenida la aproximación a la función de valor V (evaluada en cada uno de los p puntos de la malla), la correspondiente regla de decisión óptima G se obtiene calculando:

$$G = \arg\max \begin{bmatrix} M_{11} + \beta V_1 & M_{12} + \beta V_2 & ... & M_{1p} + \beta V_p \\ M_{21} + \beta V_1 & M_{22} + \beta V_2 & ... & M_{2p} + \beta V_p \\ ..... & ..... & ... & ..... \\ M_{p1} + \beta V_1 & M_{p2} + \beta V_2 & ... & M_{pp} + \beta V_p \end{bmatrix}$$

Es decir, G es un vector columna de n componentes, en donde $G_i \in \{1, ..., p\}$ nos indica el número de la columna que maximiza la fila i.

Podemos entonces reconstruir la secuencia óptima de capital, partiendo de $k_0 = K_i$ de la siguiente manera:

$$\begin{aligned} k_1 &= K_j \qquad \text{con} \quad j = G_i \\ k_2 &= K_l \qquad \text{con} \quad l = G_j \\ &\ldots\ldots\ldots\ldots\ldots \end{aligned}$$

y así sucesivamente.

5.3.2 Resolviendo el Equilibrio Competitivo

El método de iteración de la función de valor es particularmente apropiado para resolver el problema del planificador social. Sin embargo, su utilidad es menor al momento de resolver directamente el Equilibrio Competitivo. La razón no es que existan dos variables de estado, el capital individual y el agregado, pues eso sería una simple extensión del método anterior. El principal problema es que la ley de movimiento del capital agregado (la función Γ en la ecuación de Bellman (5.4)) debe ser conocida al momento de realizar la optimización. Sin embargo, este es un objeto que solo conocemos luego de resolver el equilibrio.

Una forma de evitar este problema guarda cierta analogía con el metodo de doble iteración que vimos en la sesión anterior. Presentaremos solo la idea general, pues su utilización es limitada en la práctica. Suponemos que la ley de movimiento tiene cierta forma funcional, por ejemplo un polinomio de grado n:

$$K' = \Gamma(K) = a_1 + a_2 K^2 + ... + a_n K^n$$

y proponemos un vector inicial de parámetros $(a_1, a_2, ..., a_n)$.

Dada esa ley de movimiento, la ecuación (5.4) puede resolverse iterando la función de valor. Obtenemos entonces la regla de decisión óptima y, dado k_0, calculamos la secuencia óptima resultante $k_0, k_1, ..., k_T$. Con esos datos, corremos la regresión:

$$k_{t+1} = a_1 + a_2 k_t^2 + ... + a_n k_t^n$$

y estimamos un nuevo juego de parámetros $(\hat{a}_1, \hat{a}_2, ..., \hat{a}_n)$. Si estos estan razonablemente

cercanos a los que propusimos, termina el procedimiento. En caso contrario, volvemos a resolver la ecuación de Bellman usando $(\hat{a}_1, \hat{a}_2, ..., \hat{a}_n)$ como los nuevos parámetros de la ley de movimiento.

Evidentemente, este método de doble iteración será más preciso conforme mayor sea el grado n del polinomio que se use para aproximar la ley de movimiento del capital agregado. Pero aún con un n grande, nada garantiza la convergencia del algoritmo.

Veamos ahora un ejemplo numérico para entender mejor la iteración de la función de valor:

Suponer que queremos encontrar y graficar las trayectorias óptimas para el capital, consumo, producto, salario real y tasa de interés en el Modelo de Crecimiento Neoclásico Simple, con las siguientes formas funcionales:

$$u(c) = \log(c) \qquad f(k) = Ak^{\alpha}$$

los siguientes valores para los parámetros:

$$\beta = 0.95 \qquad A = 10 \qquad \alpha = 0.35 \qquad \delta = 0.06$$

y un stock de capital inicial k_0 igual a dos tercios del capital de estado estacionario k^*, utilizando el método de iteración de la función de valor. Usar una malla para el stock de capital de 500 puntos igualmente espaciados desde $0.5k^*$ hasta $1.2k^*$ y graficar las funciones de valor obtenidas en cada una de las iteraciones.

Para esto vamos a escribir la función `vfiterm.m` como:

```
% vfiterm.m Programa para resolver una versión simple del problema
del planeador social
% usando la iteración de la función de valor.  La ecuación de
Bellman a resolver es:
%
% v(k) = max {ln(A*k^alpha+(1-delta)*k-k') + beta v(k')}
% k'
%
```

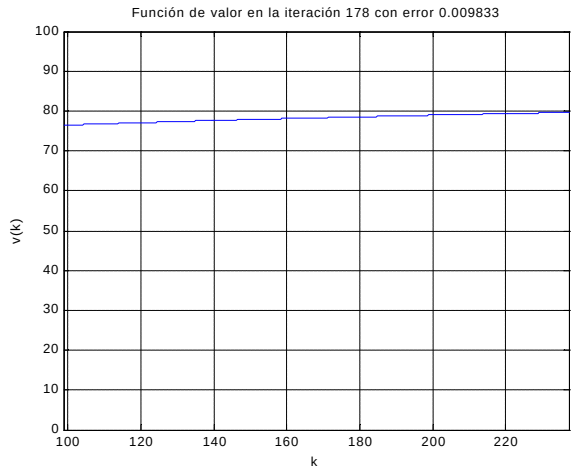
```
    % con k0 dado
    %
    % Output:  file vfout.mat con la función de valor, regla de decisión óptima,
    % y sucesiones de las variables principales.
    clear
    % Parametros del modelo
    alpha = 0.35;
    beta = 0.95;
    delta = 0.06;
    A = 10;
    % Otros parámetros (máximo número de iteraciones, tamaño del grid,
    % criterio de tolerancia, número de periodos para la simulación)
    maxit = 1000;
    p = 500;
    crit = 1e-2;
    T = 100;
    % Computa el capital de estado estacionario (kss), el capital inicial (k0)

    % y define el grid para k
    kss = ((A*beta*alpha)/(1-(1-delta)*beta))^(1/(1-alpha));
    k0 = (1/2)*kss;;
    Kmin = 0.5*kss;
    Kmax = 1.2*kss;
    K(1)=k0;
    K = [Kmin+(0:p-2)*(Kmax-Kmin)/(p-1) Kmax]';
    % Ponemos el guess inicial para la función de valor (V0=0) y construye la
matriz M
    V0 = 0*K;
    e = ones(1,p);
    M = log(max((A*K.^alpha+(1-delta)*K)*e-e'*K',1e-8));
    % Iteración de la función de valor
    figure(1)
    iconv = 0;
    it = 1;
```

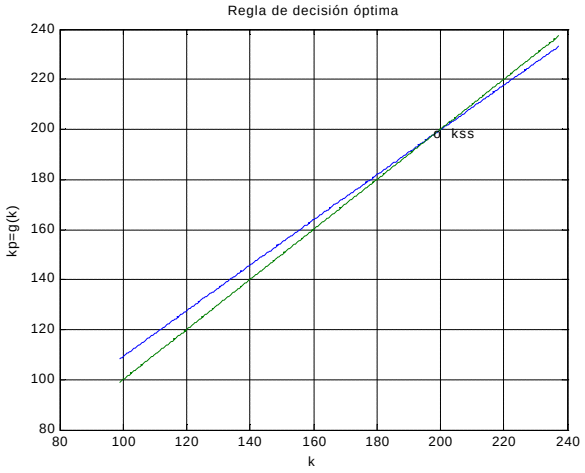
```
while (iconv==0 & it<maxit)
 [V1,G] = max((M+beta*(V0*e)')');
 V1 = V1';
 G = G';
 if norm(V0-V1)<crit
 iconv = 1;
 end
 plot(K,V1)
 axis([Kmin Kmax 0 100])
 title(['Función de valor en la iteración ',int2str(it),...
 ' con error ',num2str(norm(V0-V1))]),...
 xlabel('k'),...
 ylabel('v(k)'),...
 grid
 pause(0.01)
 V0 = V1;
 it = it+1;
end
figure(2)
plot(K,K(G),K,K,':')
title('Regla de decisión óptima'),...
xlabel('k'),...
ylabel('kp=g(k)'),...
grid,...
text(kss,kss,'o kss')
%Veamos los estados estacionarios
 yss=A*kss^alpha
 css=yss+[(1-delta)-1]*kss
 iss=yss-css
 sss=iss/yss %la tasa de ahorro
% Simula trayectorias óptimas para el capital (kt), consumo (ct),
% inversión (it) y producto (yt)
figure(3)
kt = zeros(T+1,1);
```

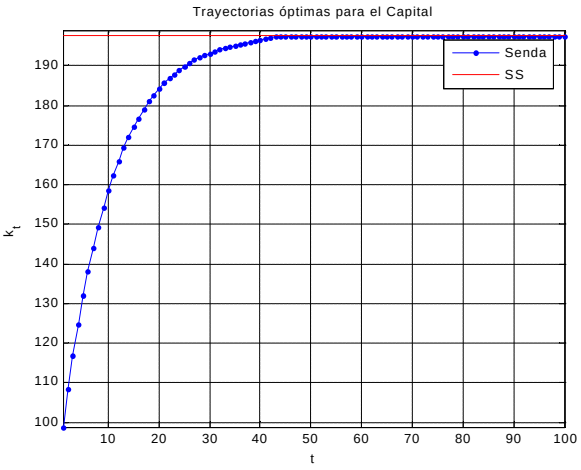
```
ind = zeros(T,1);
[aux,ind(1)] = min(K<k0);
kt(1) = K(ind(1));
for t=1:T
 ind(t+1) = G(ind(t));
 kt(t+1) = K(ind(t+1));
end
kss2=kss*ones(T,1)';
z=[1,T,K(1),kss+1];
plot(1:1:T,kt(1:T),'.-',1:1:T,kss2,'r')
title('Trayectorias óptimas para el Capital'),...
xlabel('t'),...
ylabel('k_t'),...
grid,...
axis(z)
legend('Senda', 'SS')
yss2=yss*ones(T-1,1)';
figure(4)
yt = A*kt(1:T).^alpha;
plot(1:1:T,yt,'.-',1:1:T-1,yss2,'r')
title('Trayectorias óptimas para el Producto'),...
xlabel('t'),...
ylabel('y_t'),...
grid,...
legend('Senda', 'SS')
iss2=iss*ones(T-1,1)';
figure(5)
it = kt(2:T+1)-(1-delta)*kt(1:T);
plot(1:1:T,it,'.-',1:1:T-1,iss2,'r')
title('Trayectorias óptimas para la inversión'),...
xlabel('t'),...
ylabel('i_t'),...
grid,...
legend('Senda', 'SS')
```

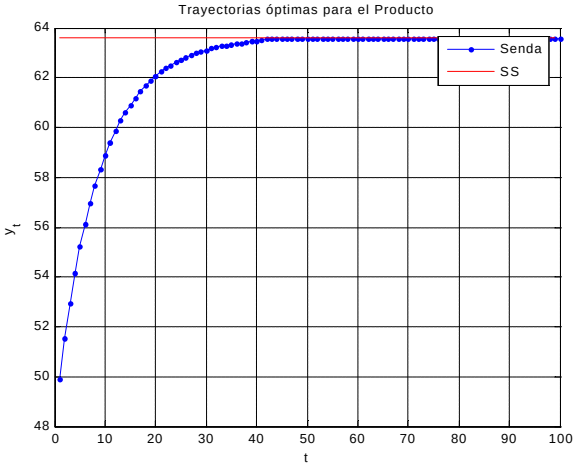
```
css2=css*ones(T-1,1)';
figure(6)
ct = yt-it;
plot(1:1:T,ct,'.-',1:1:T-1,css2,'r')
title('Trayectorias óptimas para el consumo'),...
xlabel('t'),...
ylabel('c_t'),...
grid,...
legend('Senda', 'SS')
% Guarda los resultados principales en el file vfout.mat
save E:/vfoutm V0 G kt ct it yt
```

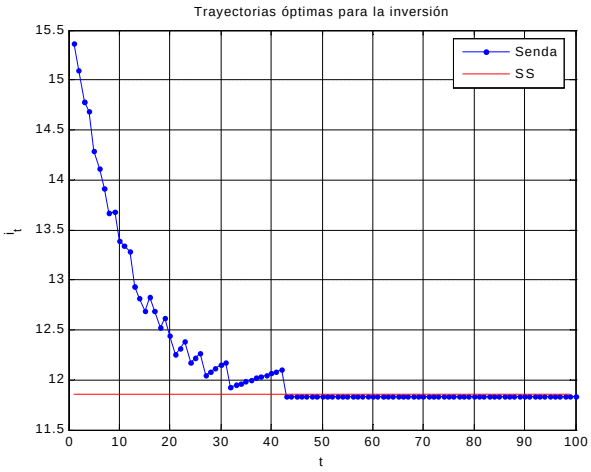
En la ventana de Matlab escribimos `vfiterm` y vamos a obtener:

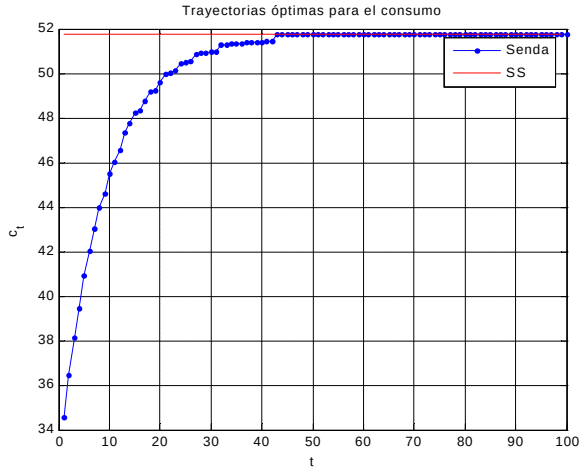









Trayectorias óptimas para la inversión
Senda
SS
i_t
t
15.5
15
14.5
14
13.5
13
12.5
12
11.5
0
10
20
30
40
50
60
70
80
90
100


Trayectorias óptimas para el consumo
Senda
SS
c_t
t
52
50
48
46
44
42
40
38
36
34
0
10
20
30
40
50
60
70
80
90
100

6 Problemas de Optimización

En economía lo que los agentes quieren es optimizar, ya sea que las empresas quieran minimizar sus costos y maximizar sus ganacias, los consumidores quieres maximizar su utilidad, los jugadores maximizar sus pagos y el planeador social maximizar el bienestar social. Los económetras también utilizan métodos de optimización, como procedimientos de mínmos cuadrados ordinarios, métodos de momentos y estimación de máxima verosimilitud.

Vamos a examinar problemas de minimización, ya que si queremos maximizar f, es lo mismo que minimizar $-f$, además de que Matlab tiene un comando para minimizar.

El problema de optimización más general es el de minimizar una función objetivo sujeto a restricciones de igualdad o desigualdad:

$$\begin{aligned} & \min_x f(x) \\ s.a.\ g(x) &= 0 \\ h(x) &\leq 0 \end{aligned}$$

en donde $f : R^n \to R$ es nuestra función objetivo, $g : R^n \to R^m$ es el vector de $m-$ restricciones con igualdad y $h : R^n \to R^l$ es el vector de $l-$ restricciones con desigualdad. Algunos ejemplos son la maximización de la utilidad sujeto a una restricción presupuestal lineal, o problemas más complejos de agente-principal. Vamos a suponer que f, g y h son continuas, aunque este supuesto veremos que no siempre es tan necesario.

Todos los métodos de optimización buscan dentro del espacio de posibles opciones, generando una sucesión de *guesses* que deben de converger a la verdadera solución. Veremos algunos métodos para resolver estos problemas, señalando también sus ventajas y desventajas.

6.1 Minimización unidimensional

El problema de optimización más sencilla es el problema sin restricciones escalar

$$\min_{x \in R} f(x)$$

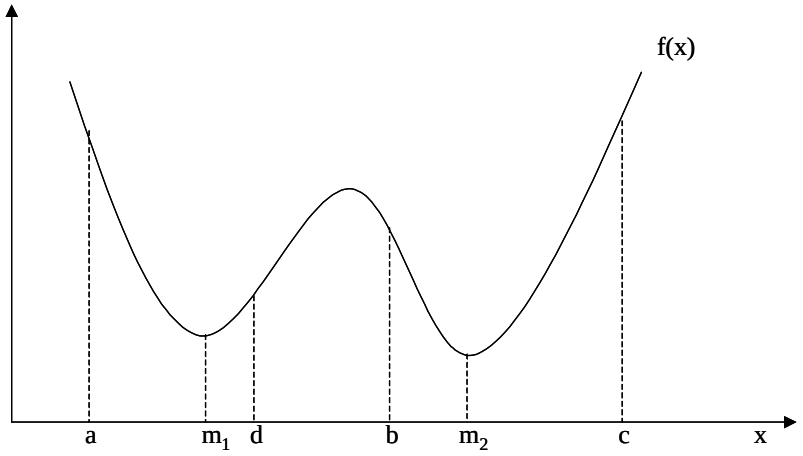
en donde $f : R \to R$. Veremos estos problemas unidimensionales porque ilustran las técnicas básicas de una manera clara y porque muchos problemas multidimensionales se reducen a resolver problemas unidimensionales.

6.1.1 Método de acorchetamiento

Este es el más sencillo dentro de los métodos de comparación. Suponer que hemos encontrado los puntos a, b y c tal que

$$a < b < c, f(a), f(c) > f(b) \tag{6.1}$$

.



Gráfica 6.1. El método de acorchamiento

La gráfica 6.1 ilustra este caso, en el cual dados los supuestos sabemos que existe un mínimo local en alguna parte del intervalo $[a, c]$, es decir que tenemos un mínimo acorchado. Lo que haremos es hacer otro subintervalo en donde se encuentre el mínimo, y lo hacemos escogiendo otro punto $d \in (a, c)$. Supongamos que $d \in (a, b]$, entonces si $f(d) > f(b)$, entonces sabemos que existe un mínimo en alguna parte de $[d, c]$. Si $f(d) < f(b)$, como en la gráfica, entonces existe un mínimo en $[a, b]$. Analogamente para el caso en que $d \in [b.c)$. Por lo que para encontrar nuestro mínimo vamos a proceder de esta manera hasta determinar un criterio de cuando detenernos. Veamos el algorítmo:

Paso 1: Encontrar $a < b < c$, tal que $f(a), f(c) > f(b)$, y escoger un criterio para detenernos ε

Paso 2: Escoger d: Si $b - a < c - b$, entonces poner $d = \frac{b+c}{2}$, en otro caso $d = \frac{a+b}{2}$

Paso 3: Computar $f(d)$

Paso 4: Escoger los nuevos (a, b, c): Si $d < b$ y $f(d) > f(b)$, entonces reemplazar (a, b, c) con (d, b, c). Si $d < b$ y $f(d) < f(b)$, entonces reemplazar (a, b, c) con (a, d, b). Si

$d > b$ y $f(d) < f(b)$, entonces reemplazar (a, b, c) con (b, d, c). En otro caso, reemplazar (a, b, c) con (a, b, d).

Paso 5: Detener si $c - a < \varepsilon$, en otro caso volver al paso 2.

¿Pero cómo encontrar a, b y c? Lo que hacemos es escoger un punto x_0, una contante $\alpha > 1$ y un paso Δ, luego computar $f(x_0), f(x_0 \pm \alpha\Delta), f(x_0 \pm \alpha^2\Delta), ...$,hasta que tengamos nuestros tres puntos que satisfacen las condiciones (6.1).

Las ventajas de este método es que funciona con cualquier función continua y acotada en un intervalo finito,pero el método es lento, además de que sólo encuentra mínimos locales, por ejemplo en la gráfica vemos que puede converger a m_1 en vez de al mínimo global m_2.

6.1.2 Método de Newton-Raphson

Para funciones C^2 este método es muy usado, y la idea principal es empezar en un punto a y encontrar el polinomio cuadrático $p(x)$ que aproxima a $f(x)$ en a en segundo grado, implicando que

$$p(x) \equiv f(a) + f'(a)(x - a) + \frac{f''(a)}{2!}(x - a)^2$$

Si $f''(a) > 0$, entones p es convexo. Ahora vamos a aproximar f econtrando el punto x_m que minimiza $p(x)$. El valor de minimización es $x_m = a - \frac{f'(a)}{f''(a)}$. Si $p(x)$ es una buena aproximación global de $f(x)$, entonces x_m estará cerca del mínimo de f, o al menos, esperamos que x_m esté más cerca del mínimo que a. En general, vamos a repetir este paso hasta que estemos muy cerca de nuestro objetivo.

Estas consideraciones motivan el método de Newton-Raphson:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

Notemos que a este método no le importa si la función es cóncava o convexa, es decir que el método de Newton-Raphson está tratando de encontra puntos críticos en general, es decir, soluciones a $f'(x) = 0$. Si este método converge a x^*, tenemos que comprobar con $f''(x^*)$ si x^* es un máximo o mínimo local.

La ventaja de este método es que si converge, lo hace muy rápido si tenemos un buen *guess* inicial, sin embargo puede no converger o puede ser que $f''(x)$ sea muy difícil de calcular.

El algorítmo del método Newton-Raphson es muy sencillo:

Paso 1: Escoger el *guess* inicial y los parámetros para detenernos $\delta > 0$ y $\varepsilon > 0$

Paso 2: $x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$

Paso 3: Si $|x_{k+1} - x_k| < \varepsilon(1 + |x_k|)$ y $|f'(x_k)|$

Ejemplo: Restricción presupuestaria del consumidor

Consideremos ahora un sencillo ejemplo de maximización de utilidad para ver estos dos métodos. Suponer que el consumidor tiene \$1 para gastar en el bien x y en el bien y. El precio de x es de \$2, el de y es de \$3, y su función de utilidad es $x^{0.5} + 2y^{0.5}$. Sea θ la cantidad gastada en x, y $1 - \theta$ la cantidad gastada en y. Por lo tanto el problema del consumidor se reduce a

$$\max_{\theta} \left(\frac{\theta}{2}\right)^{0.5} + 2\left(\frac{1-\theta}{3}\right)^{0.5}$$

cuya solución es $\theta^* = \frac{3}{11} = 0.272727...$

Para usar el método de comparación, necesitamos acotar o acorchar el valor óptimo de θ. En este caso sabemos que $\theta^* \in [0, 1]$, por lo tanto escogermos $a = 0, b = 0.5, c = 1$ inicialmente y generamos una sucesión de (a, b, c) triples, en donde en cada iteración b es el valor mínimo de θ que se ha probado. Los nuevos mínimos van a formar la sucesión

$0.5, 0.5, 0.25, 0.25, 0.25, 0.25, 0.281, 0.281, 0.266, 0.2737$

Notemos que la tasa de convergencia es lenta como lo habíamos dicho.

Veamos este ejemplo, pero usando el método de Newton-Raphson: Si $\theta_0 = 0.5$ es nuestro *guess* inicial entonces la iteración nos daría:

```
>> NewRaph('funNR','funNRpr','funNRpr2',.5,0.01,10)
El metodo de Newton-Raphson converge
 paso             x             ypr
 1.0000     0.5000     0.3165
 2.0000     0.2596     0.0229
 3.0000     0.2724     0.0005
 4.0000     0.2727     0.0000
ans =
 0.5000 0.2596 0.2724 0.2727
```

Claramente vemos que el método de Newton-Raphson converge más rápido, siempre y cuando nuestro *guess* sea cercano a la verdadera solución.

Ahora tratemos en clase de replicar esta solución escribiendo los alumnos sus propias funciones de Newton-Raphson y de la función objetivo con sus derivadas.

```
function [x,y]=NewRaph(fun,funpr,funpr2,x1,tol,max);
% Resuelve f(x)=0 usando el método de Newton
% fun Nombre de la función entre apóstrofes
% funpr Especifica la derivada de f
% x0 Estimación inicial
% tol Tolerancia permitida al computar el cero
% maxit Número máximo de iteraciones permitido
%
% x Vector de aproximaciones de cero
% y Vector de valores de la función fun(x)
x(1)=x1;
y(1)=feval(fun,x(1));
ypr(1)=feval(funpr,x(1));
ypr2(1)=feval(funpr2,x(1));
for i=2:max
 x(i)=x(i-1)-ypr(i-1)/ypr2(i-1);
 ypr(i)=feval(funpr,x(i));
 if abs(x(i)-x(i-1))<tol
 disp('El metodo de Newton-Raphson converge');
 break;
 end
 ypr2(i)=feval(funpr2,x(i));
 iter=i;
end
if (iter>=max)
 disp('no se encontró el cero a la tolerancia deseada');
end
n=length(x);
k=1:n;
out=[k' x' ypr'];
disp(' paso x ypr')
disp(out)
```

```
function y=funNR(x)
y=(x/2)^(1/2)+2*((1-x)/3)^(1/2);
```

```
function y=funNRpr(x)
y=1/4*2^(1/2)/x^(1/2)-1/3/(1/3-1/3*x)^(1/2);
```

```
function y=funNRpr2(x)
y=-1/8*2^(1/2)/x^(3/2)-1/18/(1/3-1/3*x)^(3/2);
```

6.2 Minimización multidimensional

Vimos la sección anterior para darnos una idea básica de estos métodos, pero pasemos ahora a verlo con varias variables, como sucede más en la práctica.

El problema sin restricciones sería

$$\min_x f(x)$$

para $f : R^n \to R^n$. Por conveniencia vamos a suponer que f es continua , aunque los siguientes métodos pueden funcionar aún sin este supuesto.

6.2.1 Búsqueda de la malla (*grid*)

Es el procedimiento más primitivo para encontrar un mínimo de una función, ya que se especifica una malla de puntos, de digamos, 100 a 1000 puntos, se evalua $f(x)$ en estos puntos y se escoge el mínimo. Aunque este método parace lento y no muy sofisticado, a veces es bueno primero intentar con este método, porque tal vez tengamos suerte.

Aunque muchas veces no nos conformemos con estas soluciones, si son muy útiles, ya que por ejemplo si estamos tratando de maximizar la función de máxima verosimilitud, los resultados de estos cálculos pueden indicar la curvatura general de la función de verosimilitud. Si la búsqueda de la malla indica que la función es plana en un gran intervalo, no tiene mucho sentido buscar métodos más sofisticados. Si la búsqueda de la malla nos indica que hay varios óptimos locales, entonces sabes que vamos a tener que trabajar más para encontrar el óptimo global. Además si el Hessiano en el mejor punto de la malla no es muy bueno, entonces es

no es muy probable que los siguientes métodos que veamos sean más efectivos. Sin embargo, la ventaja de este método es que puede dar muy buenos *guesses* iniciales.

Si sabemos que el objetivo es suave con una buena curvatura y con un único óptimo local, entonces este paso preliminar no es muy útil. Pero fuera de estos casos especiales, necesitamos la información que nos da esta búsqueda de la malla.

6.2.2 Método de Newton-Raphson

Algunas funciones objetivo, como las funciones de utilidad o las funciones de ganacias son suaves y con varias derivadas, por lo que podemos sacar mucho provecho usando esta información acerca de la función objetivo.

Denotemos

$$\begin{aligned} \nabla f(x) &= \left(\frac{\partial f}{\partial x_1}(x), ..., \frac{\partial f}{\partial x_n}(x)\right) \\ H(x) &= \left(\frac{\partial^2 f}{\partial x_i \partial x_j}(x)\right)_{i,j=1}^{n} \end{aligned}$$

como el gradiente y el Hessiano de f en x, respectivamente. Notemos que x es un vector columna, mientras que el gradiente es un vector fila.

El método de Newton-Raphson multidimensional es un procedimiento sencillo que nos da resultados rapidamente a problemas C^2.

Si f es convexa, pero no cuadrática, generalmente no vamos a poder resolver para el mínimo. Sin embargo, podemos reemplazar f localmente con una aproximación cuadrática y resolver para el mínimo de la aproximación. En este método examinamos una sucesión de puntos, x^k, en donde en cada iteración reemplazamos $f(x)$ con una aproximación cuadrática de f en x^k y escoge x^{k+1} como el punto crítico de la aproximación local. En x^k esta aproximación cuadrática es $f(x) \doteq f(x^k) + \nabla f(x^k)(x - x^k) + \frac{1}{2!}(x - x^k)' H(x^k)(x - x^k)$. Si f es convexa, entonces $H(x^k)$ será definida positiva y la aproximación va a tener un mínimo en $x^k - H(x^k)^{-1}(\nabla f(x^k))'$. Por lo tanto el método de Newton-Raphson será el siguiente esquema iterativo:

$$x^{k+1} = x^k - H(x^k)^{-1}(\nabla f(x^k))'$$

El método multivariante de Newton-Raphson puede ser muy costoso, porque computar y guardar el Hessiano es muy espacioso, por lo que se utiliza un nuevo paso, definido como

$s^k = x^{k+1} - x^k$, que es la solución al problema lineal

$$H(x^k)s^k = -\nabla(f(x^k))' \tag{6.2}$$

Por lo que el procedimiento sería computar $H(x^k)$, resolver el sistema lineal (6.2) para s^k y poner $x^{k+1} = x^k + s^k$.

Este método en su forma multivariante tiene los mismos problemas que el univariante, aunque también tiene la ventaja de que cuando converge, converge cuadráticamente.

6.2.3 Encontrar el cero de un sistema no lineal con minimización

Vamos a ver que los métodos de minimización también nos ayudan a encontrar el cero de un sistema de ecuaciones. Empezamos definiendo una nueva función que es la suma de las cuadrados de las funciones cuyo cero en común se desea encontrar. Por ejemplo, si queremos el cero de las funciones

$$\begin{aligned} f(x,y) &= x^2 + y^2 - 1 \\ g(x,y) &= x^2 - y \end{aligned}$$

definimos la función $h(x,y)$ como

$$h(x,y) = (x^2 + y^2 - 1)^2 + (x^2 - y)^2$$

El valor mínimo de $h(x,y) = 0$, lo cual sólo ocurre cuando $f(x,y) = 0$ y $g(x,y) = 0$. Las derivadas para el gradiente son

$$\begin{aligned} h_x(x,y) &= 2(x^2 + y^2 - 1)2x + 2(x^2 - y)2x \\ h_y(x,y) &= 2(x^2 + y^2 - 1)2y - 2(x^2 - y) \end{aligned}$$

Con la siguiente función de Matlab obtenemos los siguientes valores:

```
>> ffmin('ex_min','ex_min_g',[0.5 0.5],0.00001,100)

0          0.5000     0.5000
 1.0000    0.8750     0.6250     -0.2683
 2.0000    0.7451     0.6113     -0.0360
```

```
 3.0000     0.7925     0.6190     -0.0080
 4.0000     0.7845     0.6178     -0.0002
 5.0000     0.7866     0.6181     -0.0000
las iteraciones han convergido
ans =
 0.7860 0.6180
```

en donde `ffmin`, `ex_min` y `ex_min_g` están definidas como:

```
function xmin = ffmin(my_func, my_func_g, x0,tol, max_it)
% Encuentra el mínimo de una sistema de ecuaciones
iter = 1;
x_old = x0;
disp([ 0 x0])
while (iter <= max_it)
 dx = feval(my_func_g, x_old);
 x_new = x_old + dx;
 z0 = feval(my_func, x_old);
 z1 = feval(my_func, x_new);
 ch = z1- z0;
 while( ch >= 0)
 dx = dx/2;
 if ( norm(dx) < 0.00001)
 break;
 else
 x_new = x_old + dx;
 z1 = feval(my_func, x_new);
 ch = z1 -z0;
 end
 end
 if (abs(ch) < tol)
 disp('las iteraciones han convergido')
 break;
 end
 disp([ iter x_new ch])
```

```
 x_old = x_new;
 iter = iter +1;
end
xmin = x_new;


function y=ex_min(x)
y=(x(1)^2+x(2)^2-1)^2+(x(1)^2-x(2))^2;


function dy=ex_min_g(x)
dy=[-(2*(x(1)^2+x(2)^2-1)*2*x(1)+2*(x(1)^2-x(2))*2*x(1));
    -(4*(x(1)^2+x(2)^2-1)*x(2)-2*(x(1)^2-x(2)))]';
```

6.2.4 Optimización sin restricciones

En las siguientes secciones vamos a ver las funciones que Matlab ya tiene escritas en su *Optimization Toolbox.*

Vamos a comenzar con un sencillo ejemplo de un problema de minimización sin restricciones. Por ejemplo,

$$\min f(x_1, x_2) = (x_1 - 2)^4 + (x_1 - 2x_2)^2$$

En este ejemplo, claramente $f(x_1, x_2) \geq 0$ y $f(2, 1) = 0$, por lo tanto $(2, 1)$ es un óptimo global. El gradiente es:

$$\nabla f(x_1, x_2) = \begin{bmatrix} 4(x_1 - 2)^3 + 2(x_1 - 2x_2) \\ -4(x_1 - 2x_2) \end{bmatrix}$$

y en este caso es muy obvio que $\nabla f(2, 1) = 0$.

En esta caja de herramientas que tenemos de optimización en Matlab tenemos dos opciones para resolver estos problemas sin restricciones:

- `fminsearch`
- `fminunc`

Ambas funciones necesitan un archivo .m o una función *inline* para evaluar la función objetivo y un *guess* inicial de la solución.

Supongamos para nuestro ejemplo un *guess* inicial de $x_0 = 0$, así

```
>>f=inline('(x(1)-2)^4+(x(1)-2*x(2))^2');
>>x=fminsearch(f,[0 0])
```

Los resultados serán $x_1 = 2$ y $x_2 = 1$, con $f(x) = 2.9563e - 017$

Probemos ahora `fminunc`:

```
>>x=fminunc(f,[0 0])
```

En este caso los resultados son $x_1 = 1.9897$ y $x_2 = 0.9948$, con $f(x) = 1.1274e - 008$

Podemos ver que este resultado no es exacto, debido a que la función es plana alrededor del minimizador. De hecho la función objetivo es cercana al cero en la solución reportada por Matlab. Podríamos cambiar los parámetros de tolerancia para mejorar nuestra solución, pero en la práctica no hace mucho sentido hacer esto. Podemos notar que Matlab también se queja por la falta de la información del gradiente, por lo que no puede aplicar algún método de región de confianza[10]. Sin embargo, esto no es ningún problema, porque el gradiente se puede estimar numéricamente. Notemos que nosotros también podemos darle el gradiente a Matlab para que lo utilice en la computación de la solución. Sin embargo, para incorporar el gradiente tenemos que crear un archivo .m con la función objetivo y su gradiente:

```
function [f,g] = myfun(x)
f=(x(1)-2)^4+(x(1)-2*x(2))^2;
g=[4*(x(1)-2)^3+2*(x(1)-2*x(2)), -4*(x(1)-2*x(2))];

>> options = optimset('GradObj','on','tolfun',1e-13);
>> x = fminunc(@(x) myfun(x),[0 0],options)
```

El resultado es $x_1 = 1.9997$ y $x_2 = 0.9998$, el cual podemos notar que es un poco más preciso.

Sin embargo, como hemos visto a través del curso, computar el gradiente analíticamente puede ser complicado, por lo cual le podemos preguntar a Matlab que compare eñ gradiente

[10]Si hay tiempo se verá estos métodos en el curso.

que nosotros le demos con una estimación numérica. Lo único que tenemos que hacer *reset* las opciones y activar en las opciones el comando `derivativecheck`. Es decir,

```
>>options=optimset;
>>options=optimset('gradobj','on','derivativecheck','on','tolfun',1e-13);
>>x = fminunc(@(x) myfun(x),[0 0],options)
```

Podemos observar que la diferencia entre los gradientes es muy chica $(3.72e-007)$, por lo que preferiríamos en la práctica dejar que Matlab evalue el gradiente, en vez de nosotros calcularlo numéricamente. Veamos los valores si Matlab evalúa el gradiente:

```
>> options = optimset('tolfun',1e-13);
>> [X,FVAL,EXITFLAG,OUTPUT,GRAD]=fminunc(f,[0 0],options)

GRAD =
 1.0e-008 *
 0.4604
 -0.9182
```

Mientras que el gradiente evaluado nos da:

```
>> g=inline('[4*(x(1)-2)^3+2*(x(1)-2*x(2)), -4*(x(1)-2*x(2))]');
>> g(x)
ans =
 1.0e-009 *
 -0.1256 0.0000
```

Notemos que también podemos darle otro valor del gradiente y checar si las derivadas están cerca, y cuán cerca están. Como de costumbre si queremos obtener más información de la función ponemos `help fminunc`.

6.2.5 Optimización con restricciones de igualdad y desigualdad

Consideremos el problema convexo de

$$\begin{aligned} & & \min x_1^2 + x_2^2 \\ s.a. \quad x_1 & \geq & 0 \\ x_2 & \geq & 3 \\ x_1 + x_2 & = & 4 \end{aligned}$$

Primero vamos a comenzar escribiendo el Lagrangeano, recordando que como es una minimización las restricciones de desigualdad deberían de reescribirse como $-x_1 \leq 0$ y $-x_2 + 3 \leq 0$

$$L(\mathbf{x}, \boldsymbol{\mu}, \lambda) = x_1^2 + x_2^2 - \mu_1 x_1 - \mu_2(x_2 - 3) + \lambda(x_1 + x_2 - 4)$$

Las condiciones de Kuhn-Tucker serían:

$$\begin{aligned} 2x_1 - \mu_1 + \lambda & = & 0 \\ 2x_2 - \mu_2 + \lambda & = & 0 \\ x_1 & \geq & 0 \\ x_2 & \geq & 3 \\ x_1 + x_2 & = & 4 \\ \mu_1 x_1 & = & 0, \quad \mu_1 \geq 0 \\ \mu_2(x_2 - 3) & = & 0, \quad \mu_2 \geq 0 \end{aligned}$$

Podemos observar que vamos a tener varios casos, por lo que vamos a ver cada uno de los casos de manera analítica y luego procederemos a ver como lo podemos resolver en Matlab.

Caso 1: $\mu_1 = \mu_2 = 0$

En este caso las restricciones de desigualdad no se toman en cuenta en el Lagrangeano,

y las condiciones de estacionaridad del sistema son:

$$\begin{aligned} 2x_1 + \lambda &= 0 \\ 2x_2 + \lambda &= 0 \\ x_1 + x_2 &= 4 \end{aligned}$$

Con la cual $x_1 = x_2 = 2$, sin embargo se viola la segunda restricción de desigualdad.

Caso 2: $\mu_1, \mu_2 \neq 0$

Las condiciones de factibilidad y exclusión[11] inmediatamente conllevan a $x_1 = 0, x_2 = 3$ violando la restricción de igualdad.

Caso 3: $\mu_1 \neq 0, \mu_2 = 0$

En este caso tenemos:

$$\begin{aligned} x_1 &= 0 \\ x_2 &= 4 \\ \lambda &= -2x_2 = -8 \\ \mu_1 &= \lambda = -8 \end{aligned}$$

Se violan las condiciones de Kuhn-Tucker, porque μ_1 es negativa.

Caso 4: $\mu_1 = 0, \mu_2 \neq 0$

Vamos a obtener:

$$\begin{aligned} x_2 &= 3 \\ x_1 &= 1 \\ \lambda &= -2 \\ \mu_2 &= 4 \end{aligned}$$

[11]Ver notas de Ricard Torres "Resultados de Optimización"

El cual satisface todas las condiciones!!!

Dado que es un problema conexo, hemos obtenido el óptimo global.

Veamos ahora cómo lo podemos resolver con Matlab. El comando `quadprog` resuleve problemas cuadráticos tal que

$$\min \frac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{f}^T\mathbf{x}$$

$$\begin{aligned} s.a. \mathbf{A}\mathbf{x} &\leq \mathbf{b} \\ \mathbf{A}_{eq}\mathbf{x} &= \mathbf{b}_{eq} \\ \mathbf{l} &\leq \mathbf{x} \leq \mathbf{u} \end{aligned}$$

Para nuestro problema algunas entradas están vacías. Definamos nuestro problema de la siguiente manera:

```
>>H=2*eye(2);
>>f=[0 0];
>>Aeq= [1 1];
>>beq=4;
>>lb=[0 3];
>>[x,f,exitflag,output,lambda]=quadprog(H,f,[],[],Aeq,beq,lb);
>>x
>>lambda.eqlin
>>lambda.lower
```

Los argumentos de salida incluyen las variables de decisión óptimas, el valor óptimo de la función objetivo e información adicional, como el valor de lambda. Podemos ver que los resultados son los mismos que nosotros calculamos analíticamente, pero en Matlab lo resolvimos muy rápido.

7 Optimización dinámica

Hemos visto lo que es la programación dinámica numérica para un ejemplo de crecimiento neoclásico. Vamos a ver a continuación otro ejemplo clásico para estos problemas de opti-

mización: Problema de comerse un pastel (cake-eating)

7.1 Cake-eating example[12]

Supongamos que se nos presenta un pastel de tamaño W_1. En cada momento del tiempo $t = 1, 2, ...$ uno puede comer un poco del pastel, pero debe guadar el resto. Sea c_t el consumo en el periodo t, y sea $u(c_t)$ la utilidad de este consumo. Notemos que la función de utilidad no está indexada con el tiempo, es decir, que las preferencias son estacionarias. Podemos suponer que $u(\cdot)$ vive en los reales, es diferenciable, estricatmente creciente y estricatmente cóncava. Representemos la utilidad a lo largo del ciclo de vida como

$$\max_{\{c_t\}_1^\infty, \{W_t\}_2^\infty} \sum_{t=1}^{\infty} \beta^t u(c_t)$$

en donde $0 \leq \beta \leq 1$ es el factor de descuento.

Vamos a suponer que el pastel no se deprecia, es decir, que no se hecha a perder, pero tampoco crece, por lo que la evolución del pastel a través del tiempo, es decir, la ecuación de transición es $W_{t+1} = W_t - c_t$ para $t = 1, 2, ...$

Escribiendo este problema en programación dinámica nos quedaría la siguiente ecuación de Bellman:

$$V(W) = \max_{c \in [0,W]} u(c) + \beta V(W - c) \qquad \forall W$$

en donde $V(W)$ es el valor del problema de horizonte infinito empezando con el tamaño de un pastel W. Por lo tanto en cada periodo el agente escoge su consumo actual y por lo tanto reduce el tamaño del paste a $W' = W - cm$ como en la ecuación de transición. Notemos que en este problema las variables de estado son el tamaño del pastel W dado al principio de cada periodo. La variable de control es el consumo actual.

Alternativamente podemos especificar el problema para que en vez de escoger el consumo actual, podamos escoger el estado de mañana:

$$V(W) = \max_{W' \in [0,W]} u(W - W') + \beta V(W') \qquad \forall W$$

[12] **Ejemplo tomado de Adda y Cooper (2003).**

Claramente cualquiera de las dos especificaciones nos da el mismo resultado. Sin embargo, esta última expresión es más fácil algegraicamente, por lo que trabajaremos con ésta. A esta expresión se le llama *ecuación funcional.* Recordemos que la ecuación de Bellman no está indexada por el tiempo, característica de la estacionaridad. Notemos que toda la información pasada ya está registrada en W.

Las condiciones de primer orden son:

$$u'(c) = \beta V'(W')$$

Resolviendo el problema[13], vamos a obtener la función de política:

$$c = \phi(W), \quad W' = \varphi(W) \equiv W - \phi(W)$$

En general, no es muy posible que podamos encontrar las soluciones de las función de valor y de política, por lo que vamos a tratar de proponer unas formas específicas y ver que nuestras conjeturas son correctas. Por ejemplo, vamos a suponer que $u(c) = \ln(c)$, por lo que podemos conjeturar que la ecuación funcional es de la forma

$$V(W) = A + B\ln(W)$$

por lo que sólo tenemos que determinar dos parámetros. Resolviendo vamos a obtener que:

$$\begin{aligned} c &= W(1-\beta) \\ W' &= \beta W \end{aligned}$$

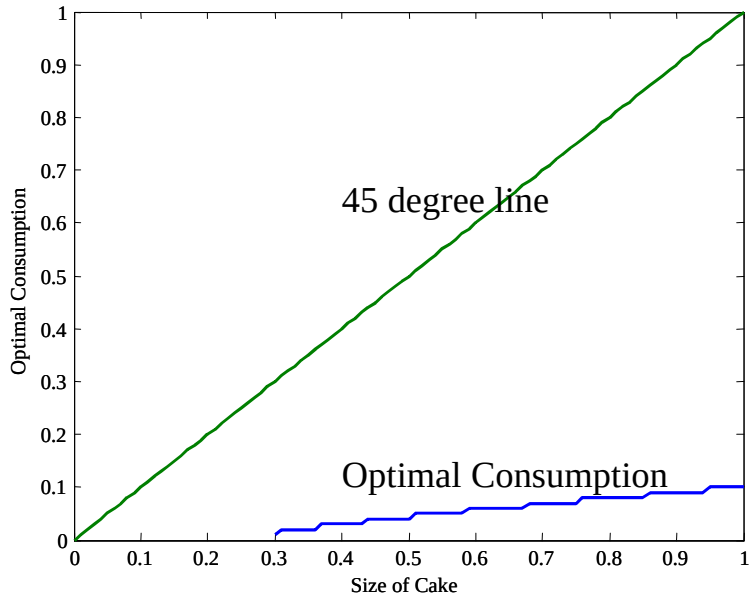
Por lo que la política óptima nos dice que ahorrarremos una fracción constante del pastel y comeremos la fracción restante.El programa propuesto por Adda y Cooper (2003) es el siguiente:

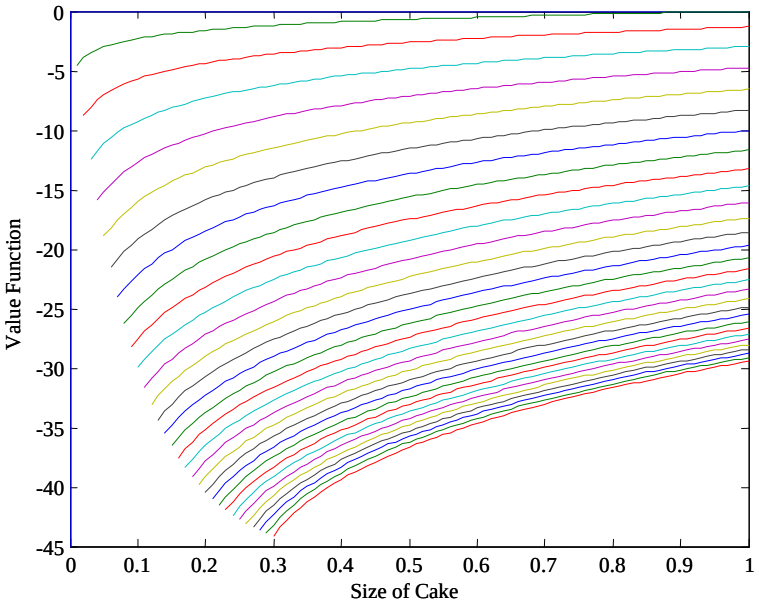
```
%%%%%%%%%% DETERMINISTIC CAKE EATING MODEL %%%%%%%%%%%%%%%%%%%%
%%% THIS FILE SOLVES THE DETERMINISTIC CAKE EATING MODEL PRESENTED
%%% IN CHAPTER 2 OF THE ADDA/COOPER BOOK (SEE SECTION 2.4.1, PAGE 16)
clear
dimIter=30; % number of iterations
```

[13]Para más detalles, ver Adda y Cooper (2003).

```
beta=0.9; % discount factor
K=0:0.01:1; % grid over cake size
[rk,dimK]=size(K);
V=zeros(dimK,dimIter); % initialize the value function.  Rows are cake size,
columns the iteration
iter=1;
for iter=1:dimIter % loop over all current cake sizes
 iter
 aux=zeros(dimK,dimK)+NaN;
 for ik=1:dimK % loop over next period cake size
 for ik2=1:(ik-1)
 aux(ik,ik2)=log(K(ik)-K(ik2))+beta*V(ik2,iter);
 end
 end
 V(:,iter+1)=max(aux')'; % optimizing over size of next period cake
end
% computing the optimal consumption as a function of initial cake size
[Val,Ind]=max(aux');
optK=K(Ind);
optK=optK+Val*0;
optC=K'-optK';
% Plotting the value function after each iterations
figure(1)
plot(K,V);
xlabel('Size of Cake');
ylabel('Value Function');
figure(2)
plot(K,[optC K'],'LineWidth',2)
xlabel('Size of Cake');
ylabel('Optimal Consumption');
text(0.4,0.65,'45 degree line','FontSize',18)
text(0.4,0.13,'Optimal Consumption','FontSize',18)
```

Cuyas gráficas correspondientes son:





Para ver cuáles fueron los valores óptimos, sólo escribimos:

>>`optK`

>>`optC`

8 El Modelo Estocástico

Consideremos ahora un modelo un poco distinto, en el cual la productividad de la firma representativa esta sujeta a un shock tecnológico θ. Es decir, en cada período t, tenemos:

$$Y_t = \theta_t f(K_t)$$

en donde θ_t es una variable aleatoria que sigue un determinado proceso estocástico. Las decisiones se toman en cada período antes de que el shock correspondiente sea observado.

Para poder aplicar el método de programación dinámica, suponemos que los shocks toman un número finito q de valores y siguen un *proceso de Markov* de primer orden. Es decir, $\theta \in \left(\theta^1, \theta^2, ..., \theta^q\right)$ y $Prob\left[\theta_{t+1} = \theta^j \middle| \theta_t = \theta^i\right] = \pi_{ij}$. Evidentemente, debe cumplirse que $\sum_{j=1}^{q} \pi_{ij} = 1$. La matriz Π de orden $q \times q$ con todas las probabilidades π_{ij} se conoce como la *matriz de transición*.

Un *Equilibrio General Competitivo, Recursivo y Estocástico* para esta economía es un conjunto de funciones (o planes contingentes) $v(k, K, \theta)$, $c(k, K, \theta)$, $i(k, K, \theta)$, $k'(k, K, \theta)$, precios $w(K, \theta)$ y $r(K, \theta)$ y ley de movimiento $\Gamma(K, \theta)$ tales que:

i) Dadas las funciones w, r y Γ, la función de valor $v(k, K, \theta)$ resuelve la ecuación de Bellman:

$$v(k, K, \theta) = \max_{c,i,k'} \quad \{u(c) + \beta E_\theta v(k', K', \theta')\} \tag{8.1}$$

$$\begin{aligned} s.t. \quad c + i &= w(K, \theta) + r(K, \theta) k \\ k' &= (1 - \delta) k + i \\ K' &= \Gamma(K, \theta) \end{aligned}$$

y las funciones $c(k, K)$, $i(k, K)$ y $k'(k, K)$ son reglas de decisión óptimas para este problema.

ii) Para todo K y θ, los precios satisfacen las condiciones marginales:

$$r(K, \theta) = \theta f'(K) \tag{8.2}$$

$$w(K,\theta) = \theta f(K) - \theta f'(K) K \tag{8.3}$$

iii) Para todo K y θ, hay igualdad entre oferta y demanda:

$$\theta f(K) = c(K,K,\theta) + i(K,K,\theta) \tag{8.4}$$

iv) Para todo K y θ, ley de movimiento agregada y comportamiento individual son consistentes:

$$\Gamma(K,\theta) = k'(K,K,\theta) \tag{8.5}$$

Nótese que, dada la forma en que hemos definido el proceso estocástico para los shocks tecnológicos, podemos expresar el valor esperado en (8.1) como:

$$E_{\theta^i} v(k',K',\theta') = \sum_{j=1}^{q} \pi_{ij} v\left(k',K',\theta^j\right) \tag{8.6}$$

Nótese también que las reglas de decisión óptimas son planes contigentes. Para obtener las trayectorias correspondientes es necesario simular una historia completa de realización de los shocks tecnológicos.

Similarmente, definimos un *Optimo de Pareto Recursivo y Estocástico* como un conjunto de funciones $v(k,\theta)$, $c(k,\theta)$, $i(k,\theta)$, $k'(k,\theta)$ que resuelven la ecuación de Bellman del planificador social:

$$v(k,\theta) = \max_{c,i,k'} \quad \{u(c) + \beta E_\theta v(k',\theta')\} \tag{8.7}$$

$$\begin{aligned} s.t. \quad c + i &= \theta f(k) \\ k' &= (1-\delta)k + i \end{aligned}$$

y nuevamente, en aquellos casos en que se cumplan los supuestos de los Teoremas del Bi-

enestar, resolveremos este problema como una manera indirecta de calcular el Equilibrio Competitivo.

8.1 Iteración de la Función de Valor

La ecuación de Bellman (8.7) puede resolverse numéricamente usando el método de iteración de la función de valor. Dada una malla $(K_{1,}K_{2,}...,K_p)$ para el stock de capital y la malla $(\theta_1,\theta_2,...,\theta_q)$ para los shocks tecnológicos, definimos la matriz M de orden $pq\times p$ de la siguiente manera:

$$M=\left[\begin{array}{cccc} F(K_1,K_1,\theta_1) & F(K_1,K_2,\theta_1) & ..... & F(K_1,K_p,\theta_1)\\ ..... & ..... & ..... & .....\\ F(K_p,K_1,\theta_1) & F(K_p,K_2,\theta_1) & ..... & F(K_p,K_p,\theta_1)\\ F(K_1,K_1,\theta_2) & F(K_1,K_2,\theta_2) & ..... & F(K_1,K_p,\theta_2)\\ ..... & ..... & ..... & .....\\ F(K_p,K_1,\theta_2) & F(K_p,K_2,\theta_2) & ..... & F(K_p,K_p,\theta_2)\\ .....\\ F(K_1,K_1,\theta_q) & F(K_1,K_2,\theta_q) & ..... & F(K_1,K_p,\theta_q)\\ ..... & ..... & ..... & .....\\ F(K_p,K_1,\theta_q) & F(K_p,K_2,\theta_q) & ..... & F(K_p,K_p,\theta_q) \end{array}\right]$$

en donde $F(x,y,\theta)=u[\theta f(x)+(1-\delta)x-y]$. Nuevamente, imponemos las restricciones tecnológicas en la matriz M haciendo::

$$M_{p(l-1)+i,j}\equiv F(K_i,K_j,\theta_l)=0 \qquad \text{si} \quad K_j>\theta_l f(K_i)+(1-\delta)K_i$$

Una vez construída M, escojemos cualquier vector columna $V^0\in R^{pq}$ (por ejemplo, $V^0=0$) e iniciamos el siguiente algoritmo con $s=0$:

1. Dado el vector $V^s=\left(V_1^s,V_2^s,...,V_{pq}^s\right)$, calcular una matriz W^s de $q\times p$ de acuerdo a:

$$W^s=\left[\begin{array}{cccc} \sum_{j=1}^q \pi_{1j}V^s_{j(p-1)+1} & \sum_{j=1}^q \pi_{1j}V^s_{j(p-1)+2} & ..... & \sum_{j=1}^q \pi_{1j}V^s_{j(p-1)+p}\\ \sum_{j=1}^q \pi_{2j}V^s_{j(p-1)+1} & \sum_{j=1}^q \pi_{2j}V^s_{j(p-1)+2} & ..... & \sum_{j=1}^q \pi_{2j}V^s_{j(p-1)+p}\\ ..... & ..... & ..... & .....\\ \sum_{j=1}^q \pi_{qj}V^s_{j(p-1)+1} & \sum_{j=1}^q \pi_{qj}V^s_{j(p-1)+2} & ..... & \sum_{j=1}^q \pi_{qj}V^s_{j(p-1)+p} \end{array}\right]$$

2. Dados el vector W^s y la matriz M, calcular la matriz M^* de orden $pq \times p$ de la siguiente manera:

$$M^* = \begin{bmatrix} M_{11} + \beta W_{11}^s & M_{12} + \beta W_{12}^s & ..... & M_{1p} + \beta W_{1p}^s \\ ..... & ..... & ..... & ..... \\ M_{p1} + \beta W_{11}^s & M_{p2} + \beta W_{12}^s & ..... & M_{pp} + \beta W_{1p}^s \\ M_{p+1,1} + \beta W_{21}^s & M_{p+1,2} + \beta W_{22}^s & ..... & M_{p+1,p} + \beta W_{2p}^s \\ ..... & ..... & ..... & ..... \\ M_{2p,1} + \beta W_{21}^s & M_{2p,2} + \beta W_{22}^s & ..... & M_{2p,p} + \beta W_{2p}^s \\ ..... & ..... & ..... & ..... \\ M_{(q-1)p+1,1} + \beta W_{q1}^s & M_{(q-1)p+1,2} + \beta W_{q2}^s & ..... & M_{(q-1)p+1,p} + \beta W_{qp}^s \\ ..... & ..... & ..... & ..... \\ M_{qp,1} + \beta W_{q1}^s & M_{qp,2} + \beta W_{q2}^s & ..... & M_{qp,p} + \beta W_{qp}^s \end{bmatrix}$$

3. Dada la matriz M^*, calcular el vector V^{s+1} como:

$$V^{s+1} = \max\ M^*$$

donde el máximo es calculado por fila.

4. Evaluar $\|V^{s+1} - V^s\|$. Si la norma es mayor que el criterio de tolerancia permitido, regresar al paso 1 con $s = s + 1$. En caso contrario, el procedimiento termina con la solución $V = V^{s+1}$.

La regla de decisión óptima se calcula como antes:

$$G = \arg\max\ M^*$$

en donde M^* es la matriz obtenida en el segundo paso de la última iteración.

8.2 Simulación de las Trayectorias Optimas

Con el método anterior, obtenemos la función de valor V y la regla de decisión óptima G evaluadas para cada stock de capital y cada shock tecnológico. Estos vectores están ordenados de la siguiente manera:

$$V = \begin{bmatrix} v(K_1,\theta_1) \\ v(K_2,\theta_1) \\ \ldots\ldots\ldots\ldots \\ v(K_p,\theta_1) \\ v(K_1,\theta_2) \\ \ldots\ldots\ldots\ldots \\ v(K_p,\theta_2) \\ \ldots\ldots\ldots\ldots \\ v(K_1,\theta_q) \\ \ldots\ldots\ldots\ldots \\ v(K_p,\theta_q) \end{bmatrix} \qquad G = \begin{bmatrix} g(K_1,\theta_1) \\ g(K_2,\theta_1) \\ \ldots\ldots\ldots\ldots \\ g(K_p,\theta_1) \\ g(K_1,\theta_2) \\ \ldots\ldots\ldots\ldots \\ g(K_p,\theta_2) \\ \ldots\ldots\ldots\ldots \\ g(K_1,\theta_q) \\ \ldots\ldots\ldots\ldots \\ g(K_p,\theta_q) \end{bmatrix}$$

en donde $l = g(K_i, \theta_j)$ nos indica que, dado el stock de capital y la realización del shock tecnológico, el planificador social escogerá un stock de capital para el siguiente período igual a K_l.

Para obtener una secuencia óptima de capital $k_0, k_1, ..., k_T$ debemos entonces simular primero una historia de realización de shocks tecnológicos $\theta_0, \theta_1, ..., \theta_T$, usando por supuesto la información contenida en la matriz de transición Π. Cualquier lenguaje de programación escoje números aleatrios de acuerdo a una distribución específica. Por ejemplo, podemos pedir que escoja θ_0 asignando la misma probabilidad $1/q$ a cada uno de los posibles valores en la malla $\left(\theta^1, \theta^2, ..., \theta^q\right)$. A continuación, pedimos que simule recursivamente una secuencia $\theta_1, ..., \theta_T$ de manera tal que, dado $\theta_t = \theta^i$, escoja θ_{t+1} entre los valores $\left(\theta^1, \theta^2, ..., \theta^q\right)$ con una distribución de probabilidad $(\pi_{i1}, \pi_{i2}, .., \pi_{iq})$.

Una vez conocida la secuencia de shocks tecnológicos, podemos simular una secuencia óptima para el capital $k_0, k_1, ..., k_T$ y para el resto de variables, usando la regla de decisión óptima G. Estas secuencias representan *series de tiempo*, cuyos principales estadísticos (media, varianza, patrón de correlaciones) pueden ser calculados y comprados con los datos. Nótese, sin embargo, que las series de tiempo obtenidas dependen de la particular realización de los shocks tecnológicos. Por eso, se recomienda hacer un número grande de simulaciones y usar el promedio (entre simulaciones) de cada uno de los estadísticos mencionados.

8.3 Limitaciones

El método de iteración de la función de valor tiene la ventaja de ser bastante preciso cuando la malla de valores para las variables de estado es suficientemente fina. Además, permite incorporar incertidumbre de una manera natural. Sin embargo, este método tiene a su vez ciertas limitaciones. Primero, su aplicabilidad es mayor para economías en las que los Teoremas del Bienestar se cumplen y podemos entonces resolver el problema del planificador social. Ya hemos visto las dificultades que se presentan al tratar de resolver directamente el Equilibrio Competitivo.

En segundo lugar, al aumentar el número de variables de estado y/o el número de puntos en la malla, la dimensión del problema (de la matriz M) crece exponencialmente volviendo este método muy lento y costoso en términos computacionales.

Por último, los métodos basado en la programación dinámica solo permiten introducir shocks estocásticos de una determinada estructura, básicamente procesos de Markov de primer orden con número finito de estados. Otros procesos estocásticos más generales requieren métodos distintos, como se verá más adelante.

8.4 Ejemplo

Consideremos la versión simple del modelo del agente representativo, con las formas funcionales:

$$u(c) = \log(c) \qquad f(k) = \theta A k^{\alpha}$$

y los valores de los parámetros:

$$\beta = 0.95 \qquad A = 10 \qquad \alpha = 0.35 \qquad \delta = 0.06$$

y un stock de capital inicial k_0 igual a la mitad de su valor k^* en un equilibrio estacionario. En cada período la función de producción se enfrenta a un shock tecnológico θ_t. Este shock puede tomar tres valores: $\theta \in \{0.5, 1, 1.5\}$ y sigue un proceso de Markov discreto de orden uno con la siguiente matriz de transición:

$$\Pi = \begin{bmatrix} 0.5 & 0.3 & 0.2 \\ 0.1 & 0.7 & 0.2 \\ 0 & 0.4 & 0.6 \end{bmatrix}$$

Tenemos que resolver el problema del planeador usando iteración de la función de valor con una malla de 300 puntos igualmente espaciados de 0 a $1.5k^*$, en donde k^* representa el stock de capital de estado estacionario en la economía determinpistica. Comenzando con un stock de capital inicial $k_0 = k^*$ y un shock $\theta_0 = 0.5$, vamos a simular y graficar la realización (series de tiempo) de las trayectorias óptimas del capital, consumo, producto, salario real y tasa de retorno del capital. Finalmente, usando un promedio de 100 simulaciones, computar la desviación estándar del logaritmo de cada una de las series obtenidas y su correlación con el producto.

El programa en un archivo .m sería:

```
% vfiter2.m Program to solve a simple version of the planner´s problem
% with technology shocks using value function iteration.  The Bellman
% equation to solve is:
%
```

```
    % v(k,z) = max {ln(th*A*k^alpha+(1-delta)*k-k') + beta*E(v(k',th'))}
    % k'
    %
    % with k0 given
    %
    % Output:  file vfout2.mat with value function, optimal decision rule,
    % and sequences for main variables.
    % Programa obtenido de Carlos Urrutia.
    %
    % Needs function shock.m to simulate shocks
    clear
    % Parameters of the model
    alpha = 0.35;
    beta = 0.95;
    delta = 0.06;
    A = 10;
    sh = [0.5 1 1.5];
    pi = [0.5 0.3 0.2;0.1 0.7 0.2;0 0.4 0.6];
    % Other required parameters (maximum number of iterations, size of the
    % grid for k, tolerance criterion, number of periods for simulations, size
of the
    % shock vector)
    maxit = 1000;
    p = 300;
    crit = 1e-2;
    T = 100;
    q = 3;
    % Computes steady state capital (kss) for an economy without uncertainty,
    % initial capital (k0) and defines a grid for k
    kss = ((A*beta*alpha)/(1-(1-delta)*beta))^(1/(1-alpha));
    k0 = kss;
    k = linspace(0,1.5*kss,p);
    % Sets initial guess for value function (V0=0) and constructs matrix M
    V0 = zeros(p*q,1);
```

```
e = ones(1,p);
f = ones(1,q);
g = ones(1,p*q);
state = ones(p*q,2);
aux = f'*k;
aux = aux(:);
state(:,1) = aux;
aux = sh'*e;
aux = aux(:);
state(:,2) = aux;
M = log(max((A*state(:,2).*(state(:,1).^alpha)+...
 (1-delta)*state(:,1))*e-g'*k,1e-8));
I = eye(q);
E = I;
for i=1:p-1
 E=[E;I];
end
% Value function iteration
iconv=0;
it=1;
while (iconv==0 & it<maxit)
 [V1,G] = max((M+beta*(E*(pi*reshape(V0,q,p))))');
 V1 = V1';
 G = G';
 if norm(V0-V1)<crit
 iconv=1;
 end
 disp(['Iteration= ',num2str(it),' Error= ',num2str(norm(V0-V1))])
 V0 = V1;
 it = it+1;
end
% Simulates a history of shocks (uses function shock.m)
theta = shock(2,pi,T);
% Simulates optimal paths for capital (kt), consumption (ct),
```

```
% investment (it) and output (yt)
G = reshape(G,q,p);
kt = zeros(T+1,1);
ind = zeros(T,1);
[aux,ind(1)] = min(k<k0);
kt(1) = k(ind(1));
for t = 1:T
 ind(t+1) = G(theta(t),ind(t));
 kt(t+1) = k(ind(t+1));
end
figure(1)
plot(1:T,sh(theta(1:T)))
title('Realizations of Technology Shock')
xlabel('t')
ylabel('z_t')
figure(2)
plot(1:T,kt(1:T))
title('Optimal Path for Capital')
xlabel('t')
ylabel('k_t')
figure(3)
yt = A*sh(theta(1:T))'.*kt(1:T).^alpha;
plot(1:T,yt)
title('Optimal Path for Output')
xlabel('t')
ylabel('y_t')
figure(4)
it = kt(2:T+1)-(1-delta)*kt(1:T);
plot(1:T,it)
title('Optimal Path for Investment')
xlabel('t')
ylabel('i_t')
figure(5)
ct = yt-it;
```

```
plot(1:T,ct)
title('Optimal Path for Consumption')
xlabel('t')
ylabel('c_t')
figure(6)
rt = alpha*A*sh(theta(1:T))'.*kt(1:T).^(alpha-1);
plot(1:T,rt)
title('Optimal Path for the Interest Rate')
xlabel('t')
ylabel('r_t')
figure(7)
wt = yt-rt(1:T).*kt(1:T);
plot(1:T,wt)
title('Optimal Path for the Real Wage')
xlabel('t')
ylabel('w_t')
% Saves main results in file vfout2.mat
save vfout2 V0 G kt ct it yt
% Computes statistics averaging 100 simulations
theta1 = zeros(T,100);
kt1 = zeros(T+1,100);
yt1 = zeros(T,100);
ct1 = zeros(T,100);
it1 = zeros(T,100);
rt1 = zeros(T,100);
wt1 = zeros(T,100);
ind1 = zeros(T+1,100);
for i=1:100
 x = rand; % Draws initial th_0 assigning same probability
 if(x<=0.333) % to each possible state
 aux=1;
 elseif(x>0.333 & x<=0.666)
 aux=2;
 else
```

```
 aux=3;
 end
 theta1(:,i) = shock(aux,pi,T)';
 [aux,ind1(1,i)] = min(k<k0);
 kt1(1,i) = k(ind1(1,i));
 for t=1:T
 ind1(t+1,i) = G(theta1(t,i),ind1(t,i));
 kt1(t+1,i) = k(ind1(t+1,i));
 end
 yt1(:,i) = A*sh(theta1(1:T,i))'.*kt1(1:T,i).^alpha;
 it1(:,i) = kt1(2:T+1,i)-(1-delta)*kt1(1:T,i);
 ct1(:,i) = yt1(:,i)-it1(:,i);
 rt1(:,i) = alpha*A*sh(theta1(1:T,i))'.*kt1(1:T,i).^(alpha-1);
 wt1(:,i) = yt1(:,i)-rt1(:,i).*kt1(1:T,i);
end
% Standard Deviationsof the log and Correlations with Output
dtk = std(log(kt1));
dtc = std(log(ct1));
dti = std(log(it1));
dty = std(log(yt1));
dtr = std(log(rt1));
dtw = std(log(wt1));
aux = zeros(2,2);
for i=1:100
 aux = corrcoef(kt1(1:T,i),yt1(:,i));
 corrk(i) = aux(1,2);
 aux = corrcoef(ct1(:,i),yt1(:,i));
 corrc(i) = aux(1,2);
 aux = corrcoef(it1(:,i),yt1(:,i));
 corri(i) = aux(1,2);
 aux = corrcoef(rt1(:,i),yt1(:,i));
 corrr(i) = aux(1,2);
 aux = corrcoef(wt1(:,i),yt1(:,i));
 corrw(i) = aux(1,2);
```

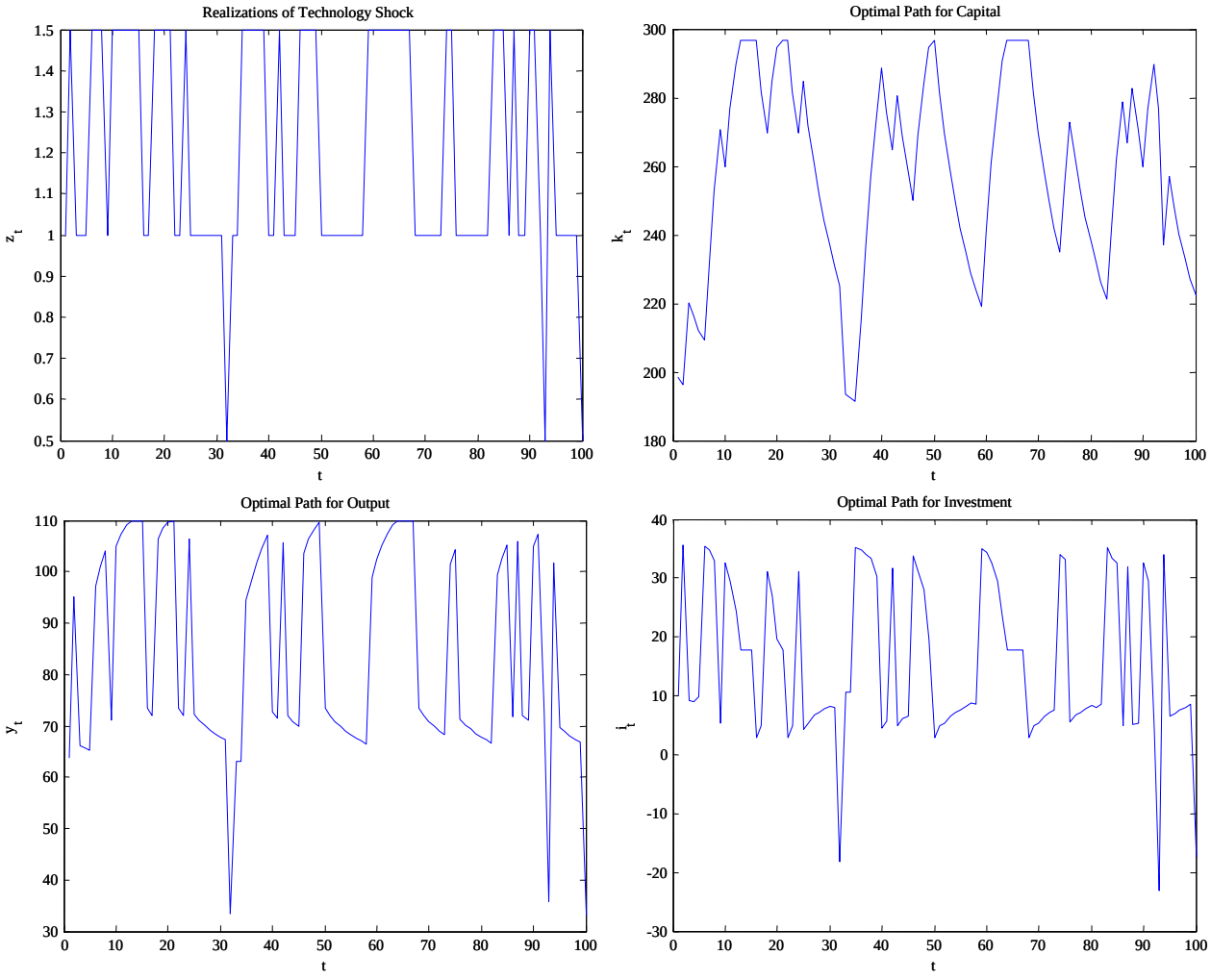
```
end
desvk = sum(dtk)/100;
desvc = sum(dtc)/100;
desvi = sum(dti)/100;
desvy = sum(dty)/100;
desvr = sum(dtr)/100;
desvw = sum(dtw)/100;
correlk = sum(corrk)/100;
correlc = sum(corrc)/100;
correli = sum(corri)/100;
correlr = sum(corrr)/100;
correlw = sum(corrw)/100;
disp(' ')
disp('Standard Deviation of the log:  ')
disp(' ')
disp(['- Output = ',num2str(desvy)])
disp(['- Consumption = ',num2str(desvc)])
disp(['- Investment = ',num2str(desvi)])
disp(['- Capital = ',num2str(desvk)])
disp(['- Interest Rate = ',num2str(desvr)])
disp(['- Real Wage = ',num2str(desvw)])
disp(' ')
disp('Correlation with Output of:  ')
disp(' ')
disp(['- Consumption = ',num2str(correlc)])
disp(['- Investment = ',num2str(correli)])
disp(['- Capital = ',num2str(correlk)])
disp(['- Interest Rate = ',num2str(correlr)])
disp(['- Real Wage = ',num2str(correlw)])
disp(' ')
```

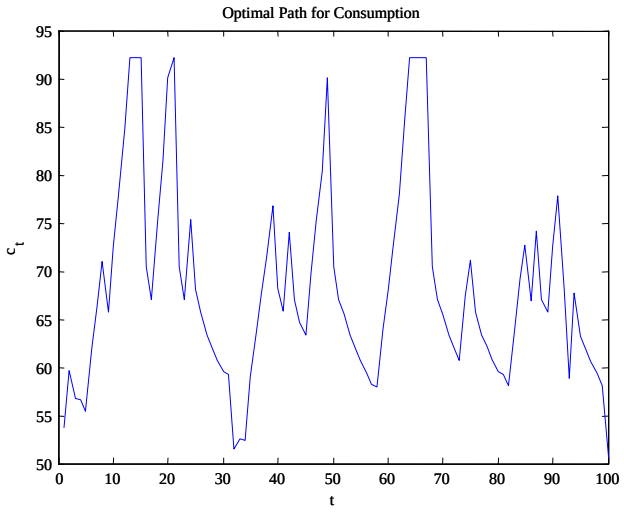
Notemos que necesitamos el archivo shock.m para obtener los shocks:

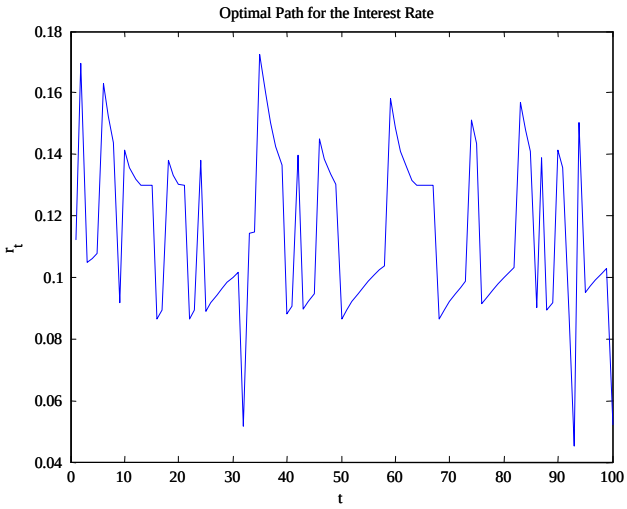
```
function th = shock(th0,pi,T)
% Function to simulate a history of T shocks, given an initial value of
% the shock equal to th0 and Markov transition matrix pi
```

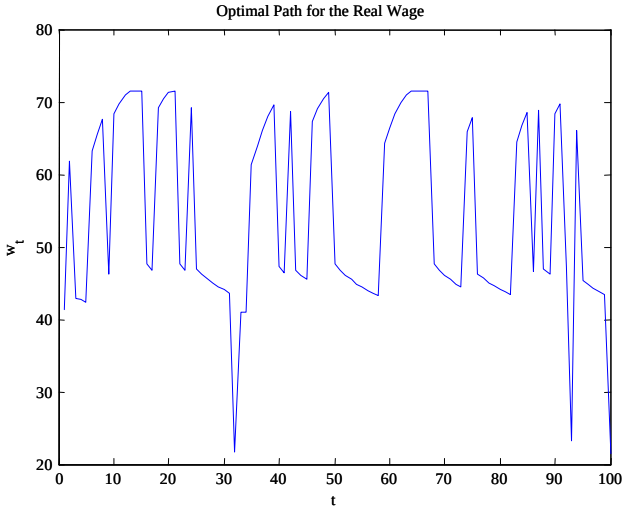
```
th = ones(1,T);
th(1) = th0;
cum = cumsum(pi')';
for i = 2:T
 x = find(rand < cum(th(i-1),:));
 th(i) = x(1);
end
```

Por lo tanto los resultados obtenidos serán:









Iteration= 184 Error= 0.0098782

Standard Deviation of the log:

- Output = 0.35472
- Consumption = 0.19007
- Investment = 1.2214
- Capital = 0.21661
- Interest Rate = 0.32783
- Real Wage = 0.35472

Correlation with Output of:

- Consumption = 0.78726

- Investment = 0.88858
- Capital = 0.4203
- Interest Rate = 0.72377
- Real Wage = 1

9 Aproximación

9.1 Aproximación por mínimos cuadrados

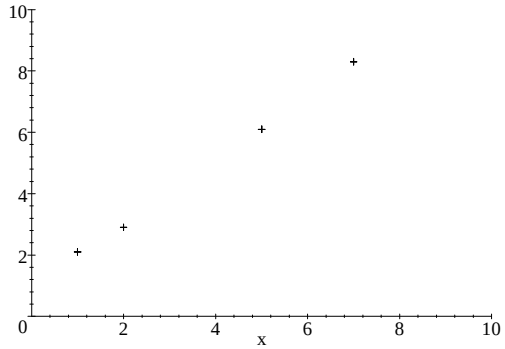
9.1.1 Lineal

Como es bien sabido, uno de los métodos más comunes para aproximar datos es el de querer minimizar alguna medida de la diferencia entre la función de aproximación y los datos. El método de mínimos cuadrados intenta minimizar la suma (sobre todos los puntos de la base de datos) de los cuadrados de las diferencias entre el valor de la función y los puntos de los datos.

Las ventajas de los mínimos cuadrados ya son bien conocidas, por ejemplo, que las diferencias negativas no se cancelan con las positivas, es fácil diferenciar la función y las diferencias pequeñas se hacen más chicas y las grandes las magnifica.

Veamos primero como podemos hacer con Matlab una aproximación lineal con algunos datos, para después ver una aproximación cuadrática y una cúbica. Dado que queremos una aproximación lineal, vamos a querer encontrar la mejor función $f(x) = ax + b$ que aproxime los datos.

Consideremos los puntos $(1, 2.1), (2, 2.9), (5, 6.1)$ y $(7, 8.3)$ como se ven en la siguiente gráfica:



Gráfica 9.1. Datos para el ejemplo de la aproximación de mínimos cuadrados lineal

Para encontrar los mejores coeficientes de la línea recta que mejor aproxima tenemos que minimizar la función de los errores al cuadrado, es decir,

$$\begin{aligned} E &= [f(x_1) - y_1]^2 + [f(x_2) - y_2]^2 + [f(x_3) - y_3]^2 + [f(x_4) - y_4]^2 \\ &= [ax_1 + b - y_1]^2 + [ax_2 + b - y_2]^2 + [ax_3 + b - y_3]^2 + [ax_4 + b - y_4]^2 \end{aligned}$$

El mínimo de esta función diferenciable sabemos que se va a encontrar en donde la derivada sea cero, es decir,

$$\begin{aligned} \frac{\partial E}{\partial a} &= 0 \\ \frac{\partial E}{\partial b} &= 0 \end{aligned}$$

La solución depende de varias cantidades que se pueden computar con los datos. Las cuatro sumatorias necesarias son:

$$\begin{aligned} S_{xx} &= \sum_{i=1}^{4} x_i^2 \\ S_x &= \sum_{i=1}^{4} x_i \\ S_{xy} &= \sum_{i=1}^{4} x_i y_i \\ S_y &= \sum_{i=1}^{4} y_i \end{aligned}$$

Las desconocidas a y b se pueden encontrar resolviendo el siguiente sistema de ecuaciones conocidas como ecuaciones normales:

$$\begin{aligned} aS_{xx} + bS_x &= S_{xy} \\ aS_x + b(4) &= S_y \end{aligned}$$

La solución a este sistema de ecuaciones es:

$$\begin{aligned} a &= \frac{4S_{xy} - S_x S_y}{4S_{xx-} S_x S_x} \\ b &= \frac{S_{xx} S_y - S_{xy} S_x}{4S_{xx-} S_x S_x} \end{aligned}$$

Para nuestro ejemplo, el sistema de ecuaciones que tenemos que resolver es:

$$\begin{aligned} a(79) + b(15) &= 96.5 \\ a(15) + b(4) &= 19.4 \end{aligned}$$

Cuya solución es $a = 1.044$ y $b = 0.9352$.

Veamos ahora cual es la función en Matlab que nos va a ayudar a encontrar la mejor aproximación lineal de mínimos cuadrados:

```
function s = Lin_LS(x, y)
% función de regresión lineal
% input x y y como vectores columna o fila
% (se convierten en forma de columnas si es necesario)
m = length(x); x = x(:); y = y(:);
sx = sum(x); sy = sum(y);
sxx = sum(x.*x); sxy = sum(x.*y);
a = ( m*sxy -sx*sy) / (m*sxx - sx^2)
b = ( sxx*sy - sxy*sx) / (m*sxx - sx^2)
table = [x y (a*x+b) (y - (a*x+b))];
disp(' x y (a*x+b) (y - (a*x+b))')
disp(table), err = sum(table( :  , 4) .^2)
s(1) = a; s(2) = b;
n=m;
xx=[x(1):(x(n)-x(1))/100:x(n)];
plot(x, y, 'r*', xx, a*xx+b, 'b-');
grid on;
```

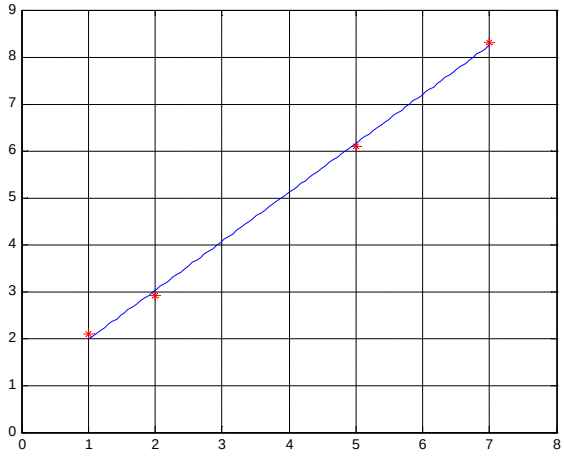
Por lo que en la ventana de Matlab escribimos:

```
>> x=[1,2,5,7]
>> y=[2.1,2.9,6.1,8.3]
```

y nos arroja los siguientes resulatdos:

```
>> Lin_LS(x,y)
a =
 1.0440
b =
 0.9352
 x y (a*x+b) (y - (a*x+b))
 1.0000 2.1000 1.9791 0.1209
 2.0000 2.9000 3.0231 -0.1231
 5.0000 6.1000 6.1549 -0.0549
```

```
 7.0000 8.3000 8.2429 0.0571
err =
 0.0360
ans =
 1.0440 0.9352
```



9.1.2 Cuadrática

Usando el mismo enfoque que para el caso lineal, ahora vamos a querer aproximar nuestros datos a la función cuadrática $f(x) = ax^2 + bx + c$, cuya función de error es la misma que para el caso anterior, es decir,

$$E = \sum_{i=1}^{n} [f(x_i) - y_i]^2$$

Igualando las derivadas de E con respecto a a, b y c a cero, obtenemos las siguientes ecuaciones normales para a, b y c :

$$\begin{aligned} a\sum_{i=1}^{n} x_i^4 + b\sum_{i=1}^{n} x_i^3 + c\sum_{i=1}^{n} x_i^2 &= \sum_{i=1}^{n} x_i^2 y_i \\ a\sum_{i=1}^{n} x_i^3 + b\sum_{i=1}^{n} x_i^2 + c\sum_{i=1}^{n} x_i &= \sum_{i=1}^{n} x_i y_i \\ a\sum_{i=1}^{n} x_i^2 + b\sum_{i=1}^{n} x_i + c(n) &= \sum_{i=1}^{n} y_i \end{aligned}$$

La función en Matlab es:

```
function z = Quad_LS(x,y)
% regresión cuadrática
% input x y y como vectores columna o fila
% (se convierten en forma de columnas si es necesario)
n = length(x); x = x(:); y = y(:);
sx = sum(x); sx2 = sum(x.^2);
sx3 = sum(x.^3); sx4 = sum(x.^4);
sy = sum(y); sxy = sum(x.* y);
sx2y = sum(x.*x.*y);
 A = [ sx4 sx3 sx2,
 sx3 sx2 sx,
 sx2 sx n]
r = [sx2y sxy sy]'
z = A\r;
a = z(1), b = z(2), c = z(3)
table = [x y (a*x.^2 + b*x + c) (y - (a*x.^2 + b*x + c))];
disp(' x y (a*x.^2 + b*x + c) (y - (a*x.^2 + b*x + c))')
disp(table)
err = sum(table( :  , 4) .^2)
xx=[x(1):(x(n)-x(1))/100:x(n)];
plot(x, y, 'r*', xx, a*xx.^2+b*xx+c, 'b-');
grid on;
```

Consideremos los datos

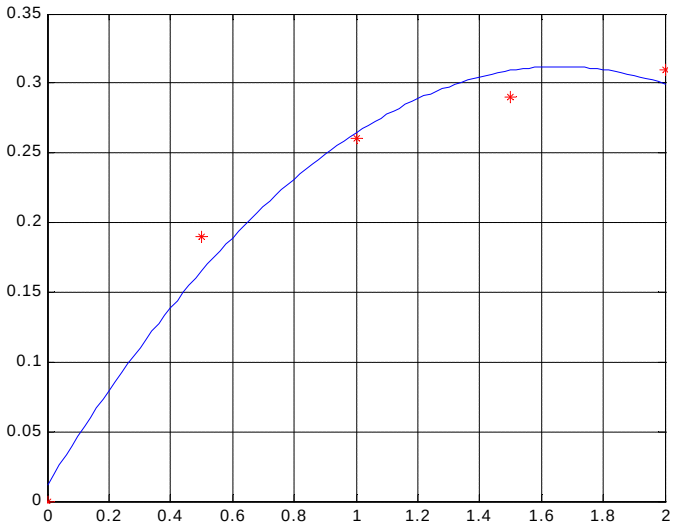
$x = [0, 0.5, 1, 1.5, 2]$

$y = [0, 0.19, 0.26, 0.29, 0.31]$

Cuyos resultados son:

```
>> Quad_LS(x,y)
A =
 22.1250 12.5000 7.5000
 12.5000 7.5000 5.0000
 7.5000 5.0000 5.0000
r =
```

```
 2.2000
 1.4100
 1.0500
a =
 -0.1086
b =
 0.3611
c =
 0.0117
 x y (a*x.^2 + b*x + c) (y - (a*x.^2 + b*x + c))
 0 0 0.0117 -0.0117
 0.5000 0.1900 0.1651 0.0249
 1.0000 0.2600 0.2643 -0.0043
 1.5000 0.2900 0.3091 -0.0191
 2.0000 0.3100 0.2997 0.0103
err =
 0.0012
ans =
 -0.1086
 0.3611
 0.0117
```



9.1.3 Cúbica

Por último veamos cómo sería para el caso cúbica, , en donde el sistema de ecuaciones para determinar los coeficientes de para el mejor ajuste de la función $f(x) = ax^3 + bx^2 + cx + d$ es:

$$\begin{aligned}
a\sum_{i=1}^{n} x_i^6 + b\sum_{i=1}^{n} x_i^5 + c\sum_{i=1}^{n} x_i^4 + d\sum_{i=1}^{n} x_i^3 &= \sum_{i=1}^{n} x_i^3 y_i \\
a\sum_{i=1}^{n} x_i^5 + b\sum_{i=1}^{n} x_i^4 + c\sum_{i=1}^{n} x_i^3 + d\sum_{i=1}^{n} x_i^2 &= \sum_{i=1}^{n} x_i^2 y_i \\
a\sum_{i=1}^{n} x_i^4 + b\sum_{i=1}^{n} x_i^3 + c\sum_{i=1}^{n} x_i^2 + d\sum_{i=1}^{n} x_i &= \sum_{i=1}^{n} x_i y_i \\
a\sum_{i=1}^{n} x_i^3 + b\sum_{i=1}^{n} x_i^2 + c\sum_{i=1}^{n} x_i + d\sum_{i=1}^{n} 1 &= \sum_{i=1}^{n} y_i
\end{aligned}$$

Cuya función en Matlab es:

```
function z = Cubic_LS(x,y)
% regresión cúbica,
% input x y y como vectores columna o fila
% (se convierten en forma de columnas si es necesario)
n = length(x); x = x( :  ); y = y( :  );
sx = sum(x); sx2 = sum(x .^2);
sx3 = sum(x .^3); sx4 = sum(x .^4);
sx5 = sum(x .^5); sx6 = sum(x .^6);
sy = sum(y); syx = sum(y.* x);
syx2 = sum(y.*x.^2); syx3 = sum(y.*x.^3);
A = [ sx6 sx5 sx4 sx3;
 sx5 sx4 sx3 sx2;
 sx4 sx3 sx2 sx;
 sx3 sx2 sx n]
r = [syx3 syx2 syx sy]'
z = A \ r ;
a = z(1); b = z (2); c = z(3); d = z(4);
p = a*x.^3 + b*x.^2 + c*x + d;
table = [ x y p (y - p)];
disp(' x y p(x) = a x ^3 + b x ^2 + c x + d y - p(x)')
```

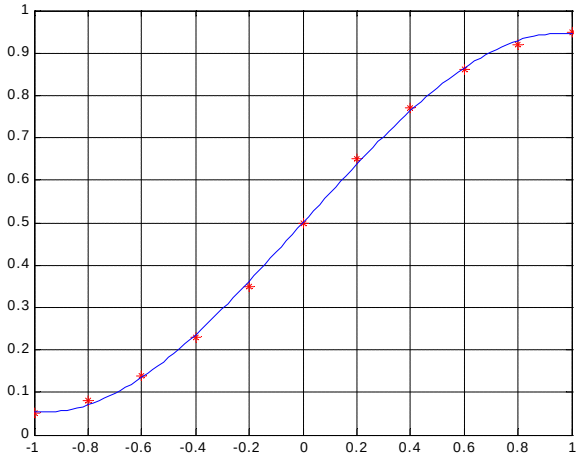
```
disp( table )
err = sum(table( :  , 4) .^2)
xx=[x(1):(x(n)-x(1))/100:x(n)];
plot(x, y, 'r*', xx, a*xx.^3+b*xx.^2+c*xx+d, 'b-');
grid on;
```

Usando los siguientes datos:
$x = [-1 : .2 : 1]$
$y = [0.05, 0.08, 0.14, 0.23, 0.35, 0.5, 0.65, 0.77, 0.86, 0.92, 0.95]$

Obtenemos:

```
>> Cubic_LS(x,y)
A =
 2.6259 0 3.1328 0.0000
 0 3.1328 0.0000 4.4000
 3.1328 0.0000 4.4000 0.0000
 0.0000 4.4000 0.0000 11.0000
r =
 1.5226
 2.2000
 2.2800
 5.5000
 x y p(x) = a x ^3 + b x ^2 + c x + d y - p(x)
 -1.0000 0.0500 0.0552 -0.0052
 -0.8000 0.0800 0.0708 0.0092
 -0.6000 0.1400 0.1352 0.0048
 -0.4000 0.2300 0.2364 -0.0064
 -0.2000 0.3500 0.3621 -0.0121
 0 0.5000 0.5000 -0.0000
 0.2000 0.6500 0.6379 0.0121
 0.4000 0.7700 0.7636 0.0064
 0.6000 0.8600 0.8648 -0.0048
 0.8000 0.9200 0.9292 -0.0092
 1.0000 0.9500 0.9448 0.0052
err =
```

```
 6.4615e-004
ans =
 -0.2550
 -0.0000
 0.6997
 0.5000
```



9.1.4 Función de Matlab `polyfit`

La función de Matlab `polyfit` encuentra los coeficientes de un polinomio de algún grado en específico que mejor se ajusta alos datos, en el sentido de mínimos cuadrados. La función es del estilo `[p,S]=polyfit(x,y,n)`, en donde **x** y **y** son los vectores de los datos y n es el grado del polinomio deseado.

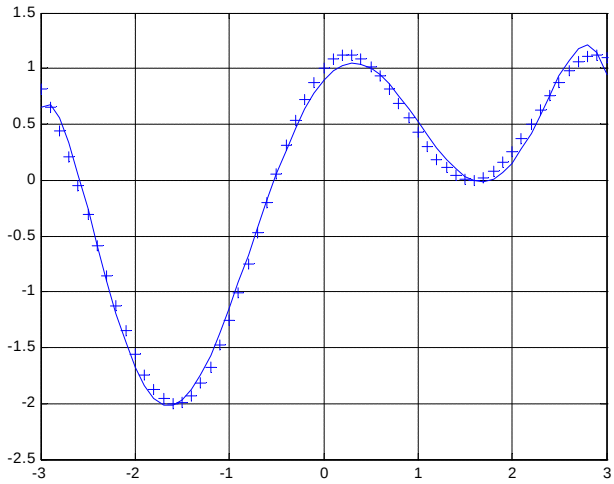
Veamos un ejemplo paso a paso:

```
% regresión polinomial
% generar los datos
x = -3 :  0.1 :  3 ;
yy1 = sin(x);
yy2 = cos(2*x);
yy = yy1 + yy2;
y = 0.01*round(100*yy);
% encontrar mínimos cuadrados del polinomio de grado 6
```

```
%para ajustar los datos
z = polyfit(x, y, 6)
% evalua el polinomio
p = polyval(z, x);
% graficar el polinomio
plot(x, p);
grid on;
hold on;
% graficar los datos
plot(x, y, '+');
hold off;
```

Los resultados son:

```
>> use_pfit
z =
 -0.0232 0.0058 0.3819 -0.1567 -1.5712 0.9907 0.8974
```

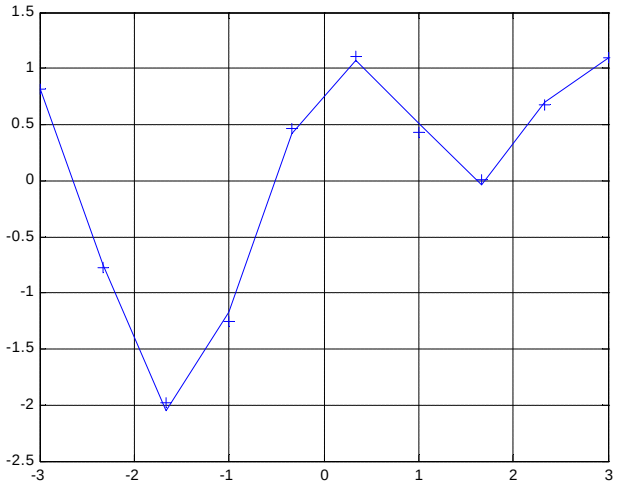


Notemos que en vez de utilizar 61 puntos, podemos hacer una buena aproximación con sólo 10 puntos, es decir, usemos

```
x=[-3:2/3:3] ;
y =[.82,-.77,-1.98,-1.26,.46,1.11,.43,.01,.68,1.1];
```

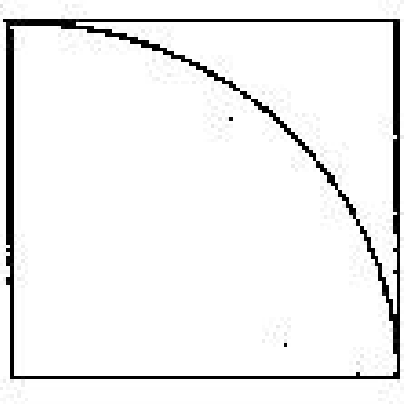
y obtenemos los siguientes resultado, aunque vemos que no es tan suave la grafica:

```
>> use_pfit2
z =
 -0.0241 0.0057 0.3983 -0.1562 -1.6272 0.9908 0.9199
```



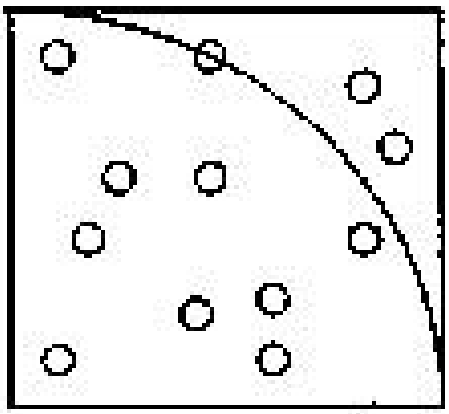
10 Integración de Monte Carlo

Esta es una técnica de integración numérica muy intuitiva, y la mejor manera de explicarla es a través de un ejemplo. Consideremos el caso en el que queremos determinar el área de un cuarto de círculo, por lo cual es una integración de dos variables:



Ahora escogemos un número de puntos dentro de esta caja, primero comenzando eligiendo un número aleatorio repreesentando la distancia en el eje-x y posteriormente escogiendo un número aleatorio representando la distancia en el eje-y. Estos dos números

representan un par (x, y) que localiza un punto en la caja. Realizando este proceso repetidamente conlleva a una imagen como la siguiente:



Si suponemos que el área del cuarto de círculo relativamente al área de la caja está determinada por el número de puntos que caen dentro del círculo en relación al número de puntos del cuadrado, entonces podemos escribir:

$$A_{círculo} = A_{cuadrado} \frac{N_{círculo}}{N_{cuadrado}}$$

Entonces podemos aproximar la integral escogiendo un número de puntos aleatorios en el cuadrado, contando cuántos caen dentro del área de interés y luego realizar la ecuación. La técnica de Monte Carlo es muy útil para casos multidimensionales cuyas formas son algo extrañas.

El comando en Matlab para obtener la integral numérica de este ejercicio es:

```
% this is a plot of the function to be solved, x=0 to 1
for i=1:100 %notar que en esta escala, la gráfica de 0 a 1
%se verá como de 0 a 100
 x(i)=i*.01;
 f(i)=sqrt(1-x(i)^2);
end
plot(f)
%
T=10;
for i=1:1000
```

```
 x=rand;
 y=2*rand;
 f=sqrt(1-x^2);
 if y<=f
 T=T+1;
 end
end
integral=2*T/1000
```

Podemos ver una simulación en:
http://homepages.cae.wisc.edu/~blanchar/eps/quad/quadframe.htm

11 Bibliografía

Adda, J. & Cooper, R. (2003): Dynamic Economics: Quantitative Methods and Applications, MIT Press.

Fausett, L.V. (1999)Applied Numerical Analysis Using MATLAB. Prentice Hall, L.

Judd, Kenneth (1998): Numerical Methods in Economics, MIT Press.

Lucas, R. Stokey, N. & Prescott, E.(1989) Recursive Methods in Economic Dynamics Harvard University Press.

Miranda, M. and Fackler, P. (2002) Applied Computational Economics MIT Press

12 Apéndice

En Matlab para trabajar en la carpeta donde están los archivos que nos interesan, sólo vamos a la barra de herramientas que tiene tres puntos ..., y ahí buscamos nuestra carpeta de interés.

En Octave escribimos:

```
cd '/cygdrive/c/'
cd..
ls
```

y ahí vamos buscando nuestra carpeta escribiendo cada vez `cd`, por ejemplo, si tenemos algo en nuestro USB, escribimos `cd d`