Tema 3
Ecuaciones no lineales

Randall Romero Aguilar, PhD

Universidad de Costa Rica
SP6534 - Economía Computacional

I Semestre 2020

Última actualización: 10 de marzo de 2020

UCR
UNIVERSIDAD DE COSTA RICA

ESCUELA de ECONOMÍA
UNIVERSIDAD DE COSTA RICA

Tabla de contenidos

1. Introducción

La ecuación no lineal toma una de dos formas.

Problema de búsqueda de raíz

Dada una función $f : \Re^n \mapsto \Re^n$, calcular un n-vector x^*, llamado la raíz de f, tal que

$$f(x^*) = 0.$$

Problema de punto fijo

Dada una función $g : \Re^n \mapsto \Re^n$, calcular un n-vector x^*, llamado el punto fijo de g, tal que

$$g(x^*) = x^*.$$

Las dos formas son equivalentes:

- Una raíz de f es un punto fijo de $g(x) = x - f(x)$.
- Un punto fijo de g es una raíz de $f(x) = x - g(x)$.

- Las ecuaciones no lineales surgen naturalmente en economía:
  - modelos de equilibrio de mercados multi-mercancía
  - modelos estáticos de juegos multi-jugador
  - modelos de optimización no restringida

- Las ecuaciones no lineales también aparecen indirectamente cuando se resuelven numéricamente modelos económicos que tienen ecuaciones funcionales:
  - modelos de optimización dinámicos
  - modelos de expectativas racionales
  - modelos de precios de arbitraje

2. Iteración de funciones

- La iteración de funciones es un algoritmo para calcular un punto fijo de una función $g : \Re^n \mapsto \Re^n$.
- Adivinamos un valor inicial x_0 y sucesivamente se forman las iteraciones

$$x_{k+1} = g(x_k)$$

hasta que converjan.

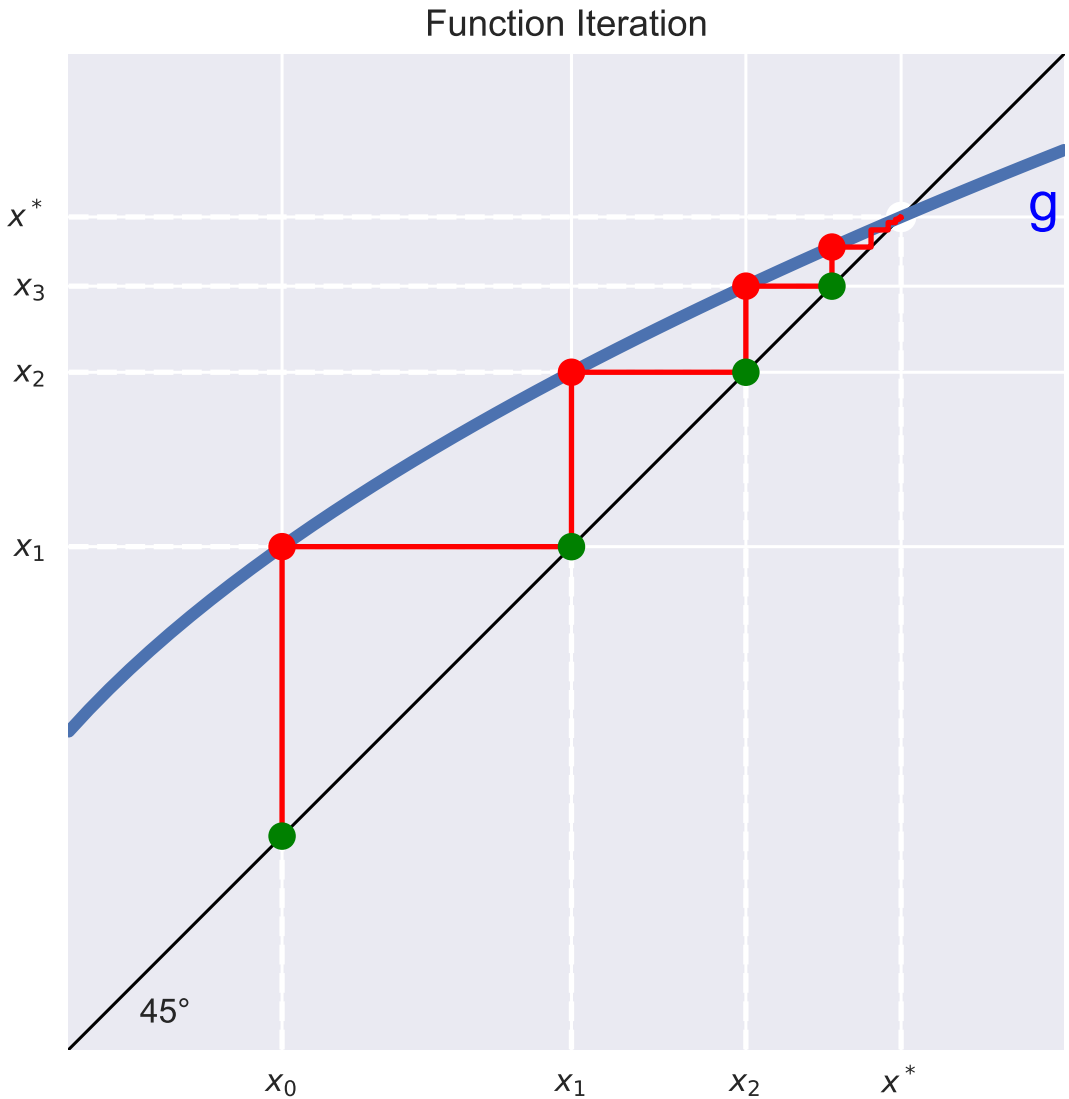

Figura 3.1: Calculando un punto fijo de g usando iteración de funciones

- £Cuando está garantizado que la iteración de funciones converja?
- Por el teorema de contracción de mapas, si para algún $\delta < 1$,

$$\|g(x_1) - g(x_2)\| \leq \delta \|x_1 - x_2\|$$

para todo x_1 y x_2, entonces g posee un único punto fijo x^* y la iteración de funciones converge a él desde cualquier valor inicial x_0.

- Además,

$$\|x_k - x^*\| \leq \frac{\delta}{1-\delta} \|x_k - x_{k-1}\|.$$

- En general, la iteración de funciones también convergirá si el valor inicial x_0 está cerca de un punto fijo x^* de g en el cual

$$\|g'(x^*)\| < 1.$$

- No hay una manera práctica general de determinar qué es cerca.

Ejemplo 1:

Punto fijo de $g(x) = \sqrt{x + 0.2}$

- Para calcular el punto fijo de

$$g(x) = \sqrt{x + 0.2}$$

usando iteración de funciones, uno emplea la regla de iteración

$$x_{k+1} = g(x_k) = \sqrt{x_k + 0.2}.$$

- En Python, ejecutemos este código

```
x=0.4
for it in range(50):
    xold, x = x, np.sqrt(x+0.2)
    if abs(x-xold)<1.e-10: break
```

- Luego de 28 iteraciones, x converge a 1.1708.

3. Método de Newton

- El método de Newton es un algoritmo para calcular la raíz de una función $f : \Re^n \mapsto \Re^n$.
- Adivinamos un valor inicial x_0 y sucesivamente formamos las iteraciones

$$x_{k+1} = x_k - f'(x_k)^{-1} f(x_k)$$

  hasta que converjan.
- Si $n = 1$, la regla de iteración tiene una forma más simple

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

- El método de Newton utiliza una estrategia de linealización sucesiva.
- La estrategia consistente en aproximar la función no lineal f con una secuencia de funciones lineales cuyas raíces son calculadas fácilmente e, idealmente, converge a la raíz de f.
- En particular, la iteración $k+1^{th}$

$$x_{k+1} = x_k - f'(x_k)^{-1} f(x_k)$$

es la raíz de la aproximación lineal de Taylor de f en la iteración precedente x_k:

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k).$$

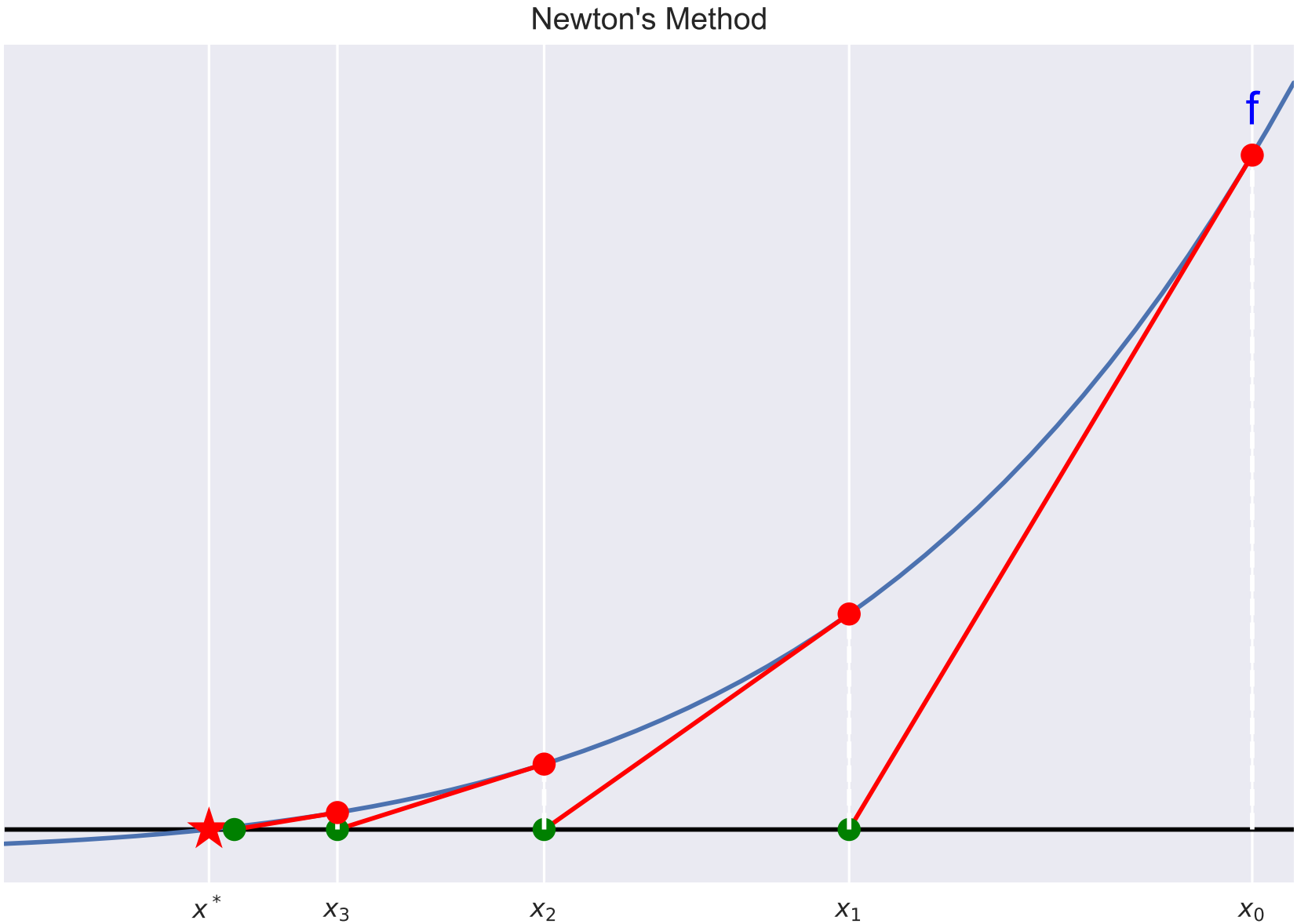

Figura 3.2: Calculando la raíz de f usando el método de Newton

- £Cuándo está garantiza la convergencia del método de Newton?
- En teoría, el método de Newton convergirá si el valor inicial x_0 está cerca de una raíz x^* de f en la cual $f'(x^*)$ es no-singular.
- No obstante, la teoría no ofrece una definición práctica de cerca.
- En la práctica, en método de Newton no convergirá a x^* si $f'(x^*)$ es mal condicionada, i.e., casi singular.

Ejemplo 2:

Raíz de la función $f(x) = x^4 - 2$

- Para calcular la raíz de $f(x) = x^4 - 2$ usando el método de Newton, usamos la regla de iteración:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} = x_k - \frac{x_k^4 - 2}{4x_k^3}$$

- En Python, ejecutamos este código

```
x = 2.3
for it in range(50):
    print(it, x)
    step = -(x**4-2)/(4*x**3)
    x += step
    if abs(step)<1.e-10: break
```

- Después de 8 iteraciones, x converge a 1.1892.

4. Métodos cuasi-Newton

- Los métodos cuasi-Newton reemplazan el jacobiano en el método de Newton con una estimación que es más fácil de calcular.
- Específicamente, los métodos cuasi-Newton usan una regla de iteración

$$x_{k+1} = x_k - A_k^{-1} f(x_k)$$

donde A_k es una estimación del jacobiano $f'(x_k)$.

Método de la secante

- El método cuasi-Newton para problemas de búsqueda de raíz univariado se llama método de la secante.
- El método de la secante reemplaza la derivada en el método de Newton con la estimación

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}},$$

lo que lleva a la regla de iteración

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

- El método de la secante se llama así porque aproxima la función f usando líneas secantes dibujadas a través de sucesivos pares de puntos en su gráfico.

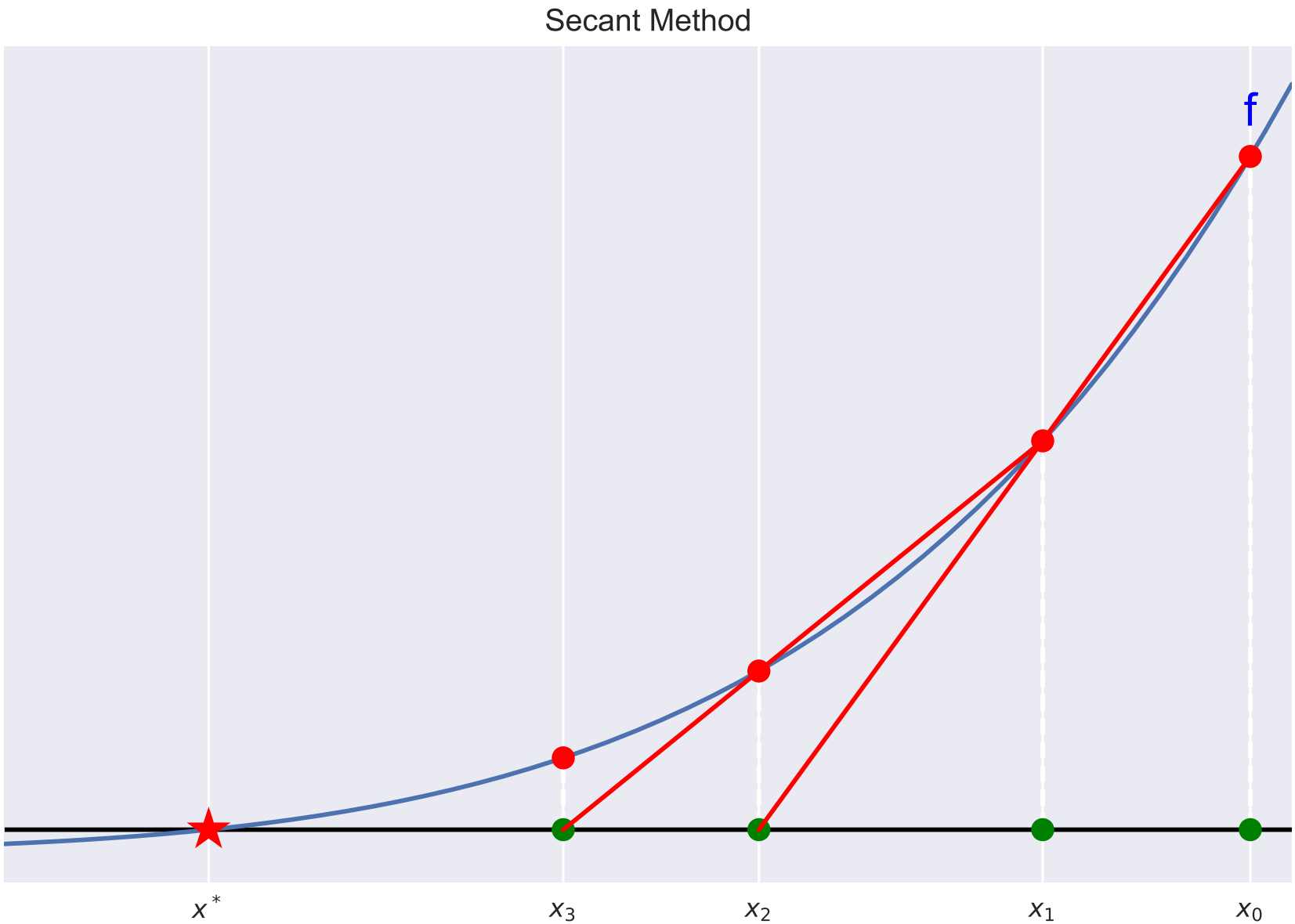

Figura 3.3: Calculando la raíz de f usando el método de la secante

Ejemplo 3:

Raíz de la función $f(x) = x^4 - 2$

- Para calcular la raíz de $f(x) = x^4 - 2$ usando el método de la secante, ejecutamos el código

```
f = lambda z: z**4-2
x, xlag = 2.3, 2.4

for it in range(50):
    d = (f(x)-f(xlag)) / (x-xlag)
    x, xlag = x-f(x)/d, x
    if abs(x-xlag)<1.e-10: break
```

- Luego de 10 iteraciones, x converge a 1.1892.

Método de Broyden

- El método de Broyden es la generalización multivariada más popular del método de la secante.
- El método de Broyden reemplaza el jacobiano en el método de Newton con una estimación A_k que es actualizada haciendo el cambio más pequeño posible que sea consistente con la condición de la secante:

$$f(x_{k+1}) - f(x_k) = A_{k+1}(x_{k+1} - x_k).$$

- Esta da por resultado la regla de iteración

$$A_{k+1} = A_k + \left(f(x_{k+1}) - f(x_k) - A_k d_k\right) \frac{d_k'}{d_k' d_k}$$

donde $d_k = x_{k+1} - x_k$.

- £Cuándo está garantizada la convergencia del método de Broyden?
- En teoría, el método de Broyden convergirá si el valor inicial x_0 está cerca de una raíz x^* de f donde el jacobiano $f'(x^*)$ es no-singular y si la estimación inicial del jacobiano A_0 está cerca de $f'(x^*)$.
- Sin embargo, la teoría no ofrece una definición práctica de cerca.
- Además, en la práctica el método de Broyden no convergirá a x^* si los jacobianos estimados A_k se vuelven mal condicionados, i.e., casi singulares.
- Adicionalmente, incluso si x_k converge a x^*, las estimaciones del jacobiano A_k en la iteración final a menudo no es una aproximación aceptable del jacobiano $f'(x^*)$ en la raíz.

5. Ejemplos numéricos usando CompEcon

La clase NLP

- El paquete CompEcon contiene la clase `NLP` (non-linear problem) para calcular la raíz de la función $f : \Re^n \mapsto \Re^n$.
- Una ecuación no lineal se crea así:

```
from compecon import NLP

def f(x):  #objective function
    fval = ...  #function value
    fjac = ...  #Jacobian value
    return fval, fjac

problem = NLP(f)
```

Resolviendo con el método de Newton

- Para resolver con el método de Newton, simplemente llamamos el método `.newton` del objeto `NLP`, dándole un valor inicial `x0`

```
x = problem.newton(x0)
```

- Al ejecutarse, `x` es la raíz del problema.
- Para verificar el valor de f, obtenemos el atributo `problem.fx`
- Estos pasos pueden combinarse como en:

```
x = NLP(f).newton(x0)
```

Resolviendo con el método de Broyden

- Para resolver con el método de Broyden, simplemente llamamos el método `.broyden` del objeto `NLP`, dándole un valor inicial `x0`

```
x = problem.broyden(x0)
```

- Al ejecutarse, `x` es la raíz del problema.
- Para verificar el valor de f, obtenemos el atributo `problem.fx`
- Estos pasos pueden combinarse como en:

```
x = NLP(f).broyden(x0)
```

- Para usar el método de Broyden, no es necesario que la función `f(x)` retorne el jacobiano de la función objetivo.

Ejemplo 4:
Raíces de una función en $\mathscr{R}^2$

- Calculemos la raíz de

$$f(x,y) = \begin{bmatrix} y\exp(x) - 2y \\ xy - y^3 \end{bmatrix}$$

- Para usar `broyden` con valor inicial $(1.0, 0.5)$

```
def f(z):
    x, y = z
    fval =[y*np.exp(x) - 2*y, x*y - y**3]
    return np.array(fval)

x0 = np.array([1.0, 0.5])
x = NLP(f).broyden(x0)
```

- De salida, digitando `print(x)` obtenemos

```
[ 0.69314718  0.8325546 ]
```

- Para usar `newton`, necesitamos calcular el jacobiano

```
def f(z):
    x, y = z
    fval = [y*np.exp(x) - 2*y, x*y - y**3]
    fjac = [[y*np.exp(x), np.exp(x)-2],
            [y, x-3*y**2]]
    return np.array(fval), np.array(fjac)

x0 = np.array([1.0, 0.5])
x = NLP(f).newton(x0)
```

- De salida, digitando `print(x)` obtenemos

```
[ 0.69314718  0.83255461]
```

- Predeterminadamente, tanto `broyden` como `newton` trabajan silenciosamente.
- Para imprimir las iteraciones basta con fijar la opción `print=True`. Por ejemplo,

```
x = NLP(f).newton(x0, print=True)
```

imprime el siguiente resultado

```
Solving nonlinear equations by Newton's method
it    bstep  change
--------------------
   0     5  3.69e-01
   1     0  6.76e-02
   2     0  4.69e-03
   3     0  2.89e-05
   4     0  1.11e-09
```

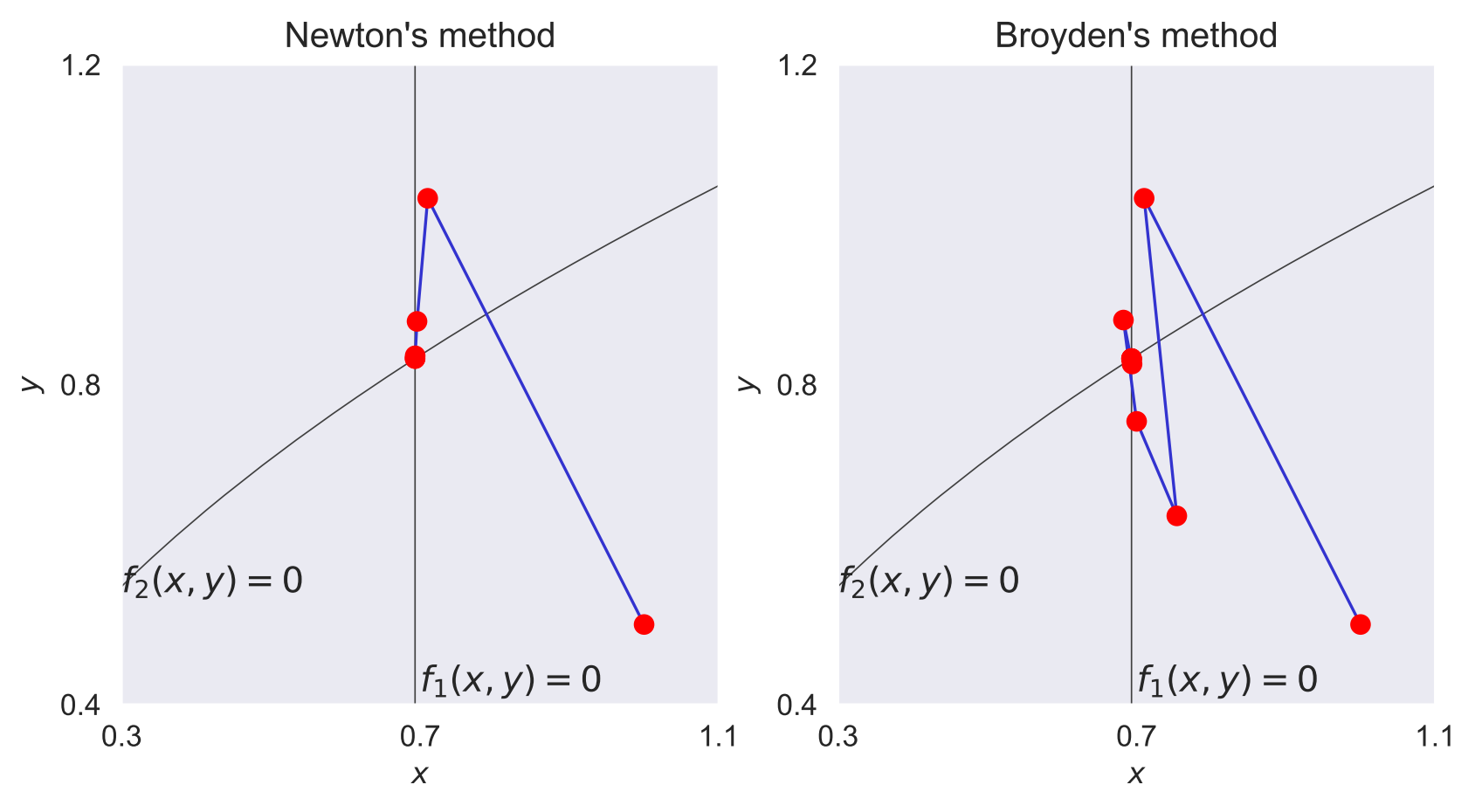

Figura 3.4: Senderos de convergencia para los métodos de Newton y Broyden

Ejemplo 5:
Un equilibrio de Cournot

- Consideremos un mercado con dos empresas que producen el mismo bien.
- El costo total de producción de la empresa i es una función

$$C_i(q_i) = \frac{\beta_i}{2} q_i^2$$

de la cantidad q_i que produce.

- El precio de equilibrio es una función

$$P(q_1 + q_2) = (q_1 + q_2)^{-\alpha}$$

de la cantidad total producida por ambas empresas.

- La empresa i escoge producir q_i para maximizar su ganancia

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

tomando el nivel de producción de la otra empresa como dado.

- Así, en equilibrio, para las empresas $i = 1, 2$:

$$\frac{\partial \pi_i}{\partial q_i} = P + P'q_i - C_i' = 0$$

- Es decir, el vector de producción que equilibra el mercado $q^* = (q_1^*, q_2^*)$ es una raíz de la función no lineal $f : \Re^2 \mapsto \Re^2$ dada por

$$f(q) = \begin{bmatrix} P + P'q_1 - C_1' \\ P + P'q_2 - C_2' \end{bmatrix}$$

cuyo jacobiano es

$$f'(q) = \begin{bmatrix} 2P' + P''q_1 - C_1'' & P' + P''q_1 \\ P' + P''q_2 & 2P' + P''q_2 - C_2'' \end{bmatrix}$$

- Supongamos que $\alpha = 0.6$, $\beta_1 = 0.6$, y $\beta_2 = 0.8$.
- Para resolver el problema usando el método de Broyden:

```
alpha, beta = 0.6,  np.array([0.6, 0.8])

def cournot(q):
    qsum = q.sum()
    P = qsum**(-alpha)
    P1 = -alpha*qsum**(-alpha-1)
    return P + (P1-beta)*q

NLP(cournot).broyden([0.2,0.2])
```

- Aquí, `cournot` calcula la ganancia marginal de ambas empresas.
- Luego de 9 iteraciones, q converge a $q^* = (0.8562, 0.700)$.

Para calcular el equilibrio usando el método de Newton, definamos la función `cournot2` tal que además retorne el jacobiano `J`:

```
alpha, beta = 0.6,  np.array([0.6, 0.8])

def cournot2(q):
    qsum = q.sum()
    P = qsum**(-alpha)
    P1 = -alpha*P/qsum
    P2 = (-alpha-1)*P1/qsum
    f = P + (P1-beta)*q
    J = P1 + P2*np.repeat(q,[2]).reshape(2,2) + np.diag(P1-beta)
    return f, J
```

Calculamos el equilibrio de mercado llamando el método `newton` de `NLP`:

```
NLP(cournot2).newton([0.2,0.2])
```

Después de 5 iteraciones, q converge a $q_1^* = 0.856$ and $q_2^* = 0.700$.

Por supuesto, si no contamos con el paquete CompEcon, siempre podemos calcular el equilibrio de mercado directamente usando el método de Newton:

```
alpha, beta = 0.6,  np.array([0.6, 0.8])
q = np.array([0.2,0.2])

for it in range(40):
    f, J = cournot2(q)
    step = -np.linalg.solve(J,f)
    q += step
    if np.linalg.norm(step)<1e-10: break
```

Al ejecutarse, digitando `print(q)` obtenemos

```
[0.8562 0.7   ]
```

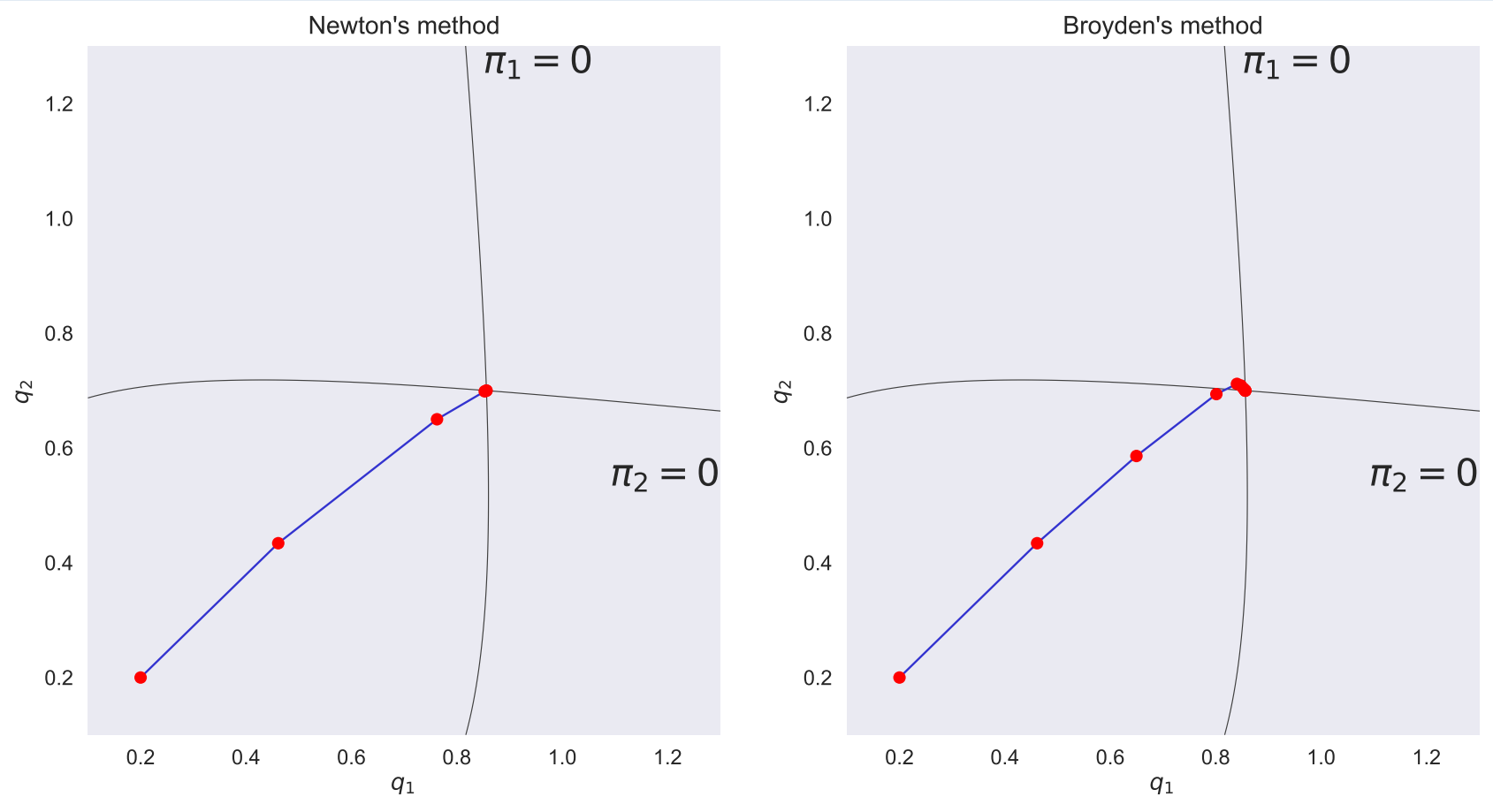

Figura 3.5: Senderos de convergencia para los métodos de Newton y Broyden

6. Asuntos prácticos

No convergencia

En la práctica, los algoritmos de ecuaciones no lineales pueden fallar por varias razones:

- error humano
- mal valor inicial
- mal condicionamiento

Error humano

- Error matemático: analista deriva incorrectamente la función o el jacobiano.
- Error de codificación: analista codifica incorrectamente la función o el jacobiano.
- Errores más probables con el método de Newton porque hay que derivar y codificar correctamente el jacobiano.
- Errores menos probables con iteración de funciones y el método de Broyden porque no necesitan derivadas.

Mal valor inicial

- Los algoritmos de ecuaciones no lineales requieren valores iniciales.
- Si el valor inicial está lejos de la raíz deseada, algoritmo puede divergir o converger a la raíz equivocada.
- La teoría no ofrece ningúna guía sobre cómo especificar un valor inicial.
- Analista debe adivinar un buen valor a partir de su conocimiento del modelo.
- Si las iteraciones divergen, intentamos con otro valor inicial.
- Funciones bien portadas son más robustas al valor inicial.
- Funciones mal portadas son más sensibles al valor inicial.

Mal condicionamiento

- El cálculo cada iteración en los métodos de Newton y Broyden requiere resolver una ecuación lineal en la que aparece el jacobiano o su estimación.
- Si el jacobiano o su estimación son mal condicionados cerca de la solución, el paso de la iteración no puede calcularse con precisión.
- Muy poco puede hacerse en estos caso.
- Este inconveniente se presenta más frecuentemente de lo que quisiéramos.

Velocidad de ejecución

Dos factores determinan la velocidad con la cual un algoritmo apropiadamente codificado e inicializado convergirá a una solución:

- la tasa asintótica de convergencia
- el esfuerzo computacional por iteración

Tasa asintótica de convergencia

- La tasa asintótica de convergencia mide la mejora obtenida por iteración cerca de la solución.
- Una secuencia x_k converge a x^* a una tasa asintótica de orden p si hay una constante $C > 0$ tal que para k suficientemente grande,

$$\|x_{k+1} - x^*\| \leq C\|x_k - x^*\|^p.$$

Método	Tasas de convergencia
Iterar f	lineal: $p = 1$ y $C < 1$—relativamente despacio
Broyden	superlineal: $p \approx 1.62$ —relativamente rápido
Newton	cuadrática: $p = 2$ —extremadamente rápido

Ejemplo 6:
Tasas de convergencia

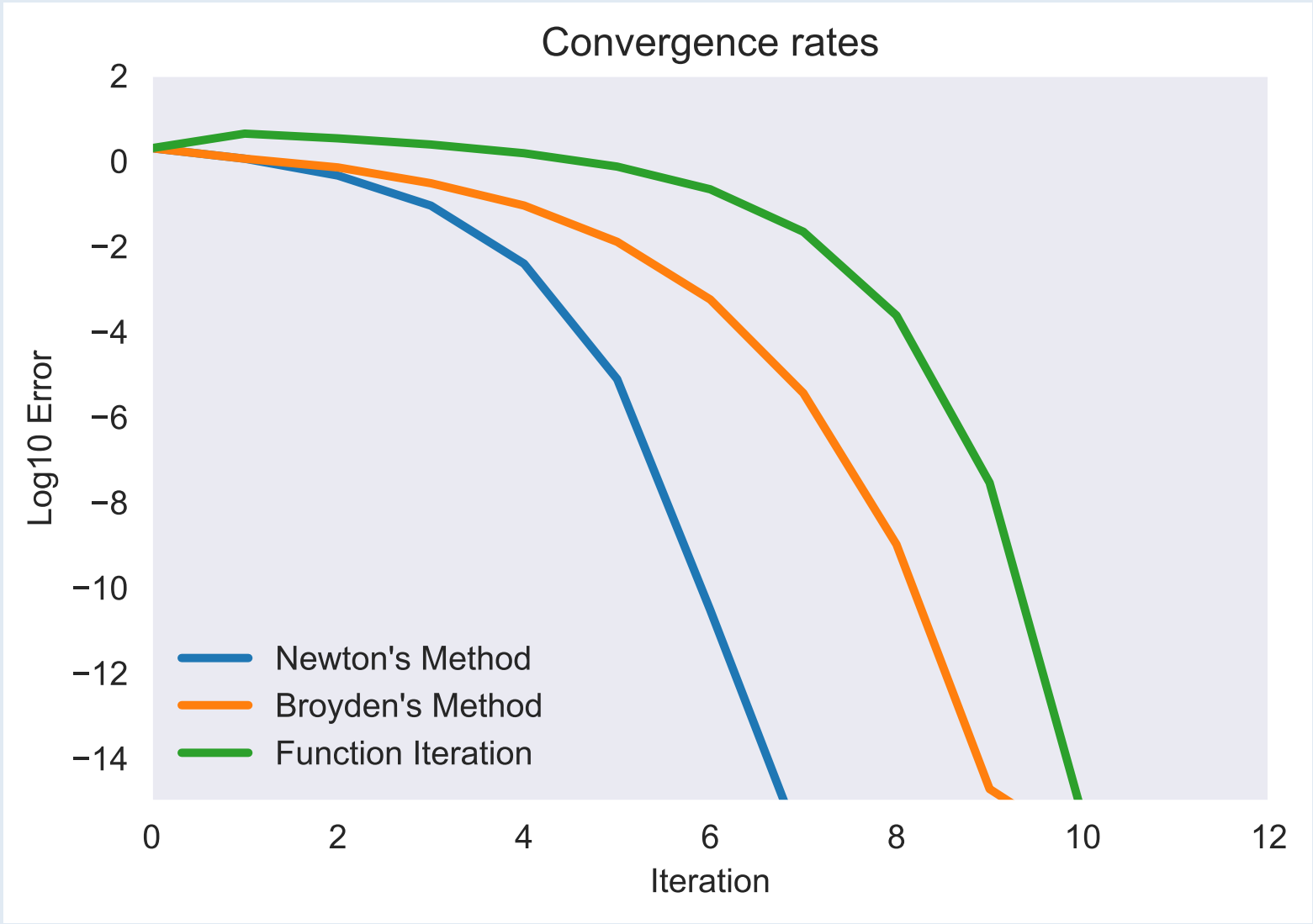

Figura 3.6: Tasa de convergencia al calcular el punto fijo de $x - \exp(x) + 1$ usando varios métodos, $x_0 = 2$

Esfuerzo computacional por iteración

- No obstante, los algoritmos difieren en cálculos por iteración.
- Iteración de funciones requiere una evaluación de la función.
- El método de Broyden adicionalmente requiere una solución lineal.
- El método de Newton adicionalmente requiere una evaluación de jacobiano.
- Así, una tasa de convergencia más rápida típicamente solo puede conseguirse invirtiendo un mayor esfuerzo computacional por iteración.
- La compensación óptima entre tasa de convergencia y esfuerzo computacional varía entre aplicaciones.

Escogiendo un método de solución

- La preocupación por la velocidad de ejecución puede resultar exagerada.
- El tiempo que el analista debe invertir en escribir y corregir el código es típicamente mucho más importante.
- Los método sin derivadas, como la iteración de funciones y el método de Broyden, pueden implementarse más rápidamente en tiempo real y más confiablemente que el método de Newton.
- El método de Newton debe usarse solo si
  - la dimensión es baja o las derivadas son simples,
  - otros métodos no han logrado converger, o
  - en la producción de código reutilizable o de propósito general.

7. Problemas de complementariedad

Problemas de complementariedad

Muchos fenómenos en economía y finanzas son naturalmente modelados como pares de desigualdades que deben satisfacerse de manera complementaria, es decir, tal que al menos una de las dos desigualdades se cumpla con igualdad estricta. Entre ellos

- Modelos de equilibrio parcial en los cuales precios y cantidades están restringidos (cuotas, límites de capacidad, precios piso, precios techo, no-negatividad).
- Modelos de optimización estáticos en los cuales una función debe maximizarse sujeta a restricciones.

General setup

El problema de complemetariedad tiene la siguiente forma: Dada una función f de $\Re^n$ a $\Re^n$, $a \in \Re^n$ y $b \in \Re^n$, encontrar $x \in \Re^n$ tal que para todo $i = 1, \ldots, n$,

$$a_i \leq x_i \leq b_i$$

$$x_i > a_i \Rightarrow f_i(x) \geq 0$$

$$x_i < b_i \Rightarrow f_i(x) \leq 0.$$

En breve, denotamos el problema de complementariedad

$$f(x) \perp [a, b].$$

Búesqueda de raíces vs. problemas de complementariedad

- El problema de búsqueda de raíces es un caso especial del problema de complementariedad en el cual $a_i = -\infty$ y $b_i = \infty$ para todo i.
- No obstante, el problema de complementariedad no es la búsqueda de una raíz dentro de unos límites especificados.
- $f_i(x)$ puede ser positivo en la solución, aunque solo si x_i es igual a su límite superior.
- $f_i(x)$ puede ser negativo en la solución, aunque solo si x_i es igual a su límite inferior.

Condiciones de arbitraje como problemas de complementariedad

Los problemas de complementariedad en economía y finanzas típicamente admiten una interpretación de arbitraje:

- Hay n actividades económicas.
- El nivel de actividad i, denotado x_i, está limitado abajo por a_i y arriba por b_i.
- La ganancia marginal de la actividad i es $f_i(x)$.
- Las ganacias pueden aumentarse bajando x_i si $x_i > a_i$ y $f_i(x) < 0$.
- Las ganancias pueden disminuise aumentando x_i si $x_i < b_i$ y $f_i(x) > 0$.
- Se alcanza el equilibrio si y sólo si todas las oportunidades de ganancias han sido eliminadas, es decir, si y sólo si x resuelve $f(x) \perp [a, b]$.

El teorema de Karush-Kuhn-Tucker

El teorema de Karush-Kuhn-Tucker afirma que x maximiza una función $f : \Re^n \mapsto \Re$ sujeta a la restricción $a \leq x \leq b$ solo si resuelve el problema de complementariedad $f'(x) \perp [a, b]$, es decir, solo si, para todo i,

$$
\begin{aligned}
& a_i \leq x_i \leq b_i \\
& x_i > a_i \Rightarrow f_i'(x) \geq 0 \\
& x_i < b_i \Rightarrow f_i'(x) \leq 0.
\end{aligned}
$$

Formas alternativas de las condiciones Karush-Kuhn-Tucker

A partir del lagrangeano

$$\mathscr{L} = f(x) + \lambda(x-a) + \mu(b-x)$$

la condición de primer orden es

$$f'(x) = \mu - \lambda$$

y las condiciones de holgura son

- $\lambda \geq 0$ $\quad x \geq a$ $\quad \lambda(x-a) = 0$
- $\mu \geq 0$ $\quad x \leq b$ $\quad \mu(b-x) = 0$

Versión 1

$$a \leq x \leq b$$
$$x > a \Rightarrow f'(x) \geq 0$$
$$x < b \Rightarrow f'(x) \leq 0$$

Versión 2

$$a < x < b \Rightarrow f'(x) = 0$$
$$x = a \Rightarrow f'(x) \leq 0$$
$$x = b \Rightarrow f'(x) \geq 0$$

Versión 3

$$f'(x) = 0 \Rightarrow a \leq x \leq b$$
$$f'(x) < 0 \Rightarrow x = a$$
$$f'(x) > 0 \Rightarrow x = b$$

Ejemplo 7:
Precio techo

- El exceso de demanda de un bien (cantidad demandada menos cantidad ofrecida) es una function $E(p)$ del precio de mercado del bien.
- Si el mercado del bien es competitivo, el precio de equilibrio está caracterizado por el problema de búsqueda de raíz $E(p) = 0$.
- Sin embargo, si el gobierno impone un precio techo $\bar{p}$, el precio de equilibrio estaría caracterizado por el problema de complementariedad $E(p) \perp [-\infty, \bar{p}]$:

$$
\begin{aligned}
-\infty \leq p \leq \bar{p} \\
E(p) \geq 0 \\
p < \bar{p} \Rightarrow E(p) \leq 0.
\end{aligned}
$$

- En equilibrio puede existir un exceso de demanda del bien, $E(p) > 0$, si el precio techo es vinculante.

Ejemplo 8:
Salario mínimo

- El exceso de demanda de trabajo (trabajo demandado menos trabajo ofrecido) es una función $E(w)$ de la tasa salarial.
- Si el mercado laboral es competitivo, el salario de equilibrio está caracterizado por el problema de búsqueda de raíz $E(w) = 0$.
- Sin embargo, si el gobierno impone una salario mínimo $\bar{w}$, el salario de equilibrio estaría caracterizado por el problema de complementariedad $E(w) \perp [\bar{w}, \infty]$

$$\begin{aligned} \bar{w} \leq w \leq \infty \\ E(w) \leq 0 \\ w > \bar{w} \ \Rightarrow \ E(w) \geq 0. \end{aligned}$$

- En equilibrio puede existir un exceso de oferta laboral, $E(w) < 0$, si el salario mínimo es vinculante.

Ejemplo 9:
Precio de equilibrio espacial

- Una mercancía es producida y consumida en n regiones.
- La cantidad x_{ij} producida en la región i y consumida en la región j no puede ser negativa ni exceder la capacidad de envío b_{ij}.
- El costo c_{ij} de transportar una unidad de la mercancía de productores en la región i a consumidores en la región j es constante.
- En la región i, una función inversa de oferta indica el precio p_i^s que debe pagarse a los productores para que produzcan la cantidad $q_i^s = \sum_j x_{ij}$.
- En la región i, una función inversa de demanda indica el precio p_i^d que los consumidores están dispuestos a pagar para comprar la cantidad $q_i^d = \sum_j x_{ji}$.

- La ganancia de producir una unidad de la mercancía en la región i y venderla a consumidores en la región j es

$$\pi_{ij}(x) = p_j^d(x) - p_i^s(x) - c_{ij}$$

- El equilibrio se alcanza solo si todas las oportunidades de ganancias han sido eliminadas, es decir, solo si los flujos de mercancías x resuelven el problema de complementariedad $\pi(x) \perp [0, b]$:

$$\begin{aligned} 0 \leq x_{ij} &\leq b_{ij} \\ x_{ij} > 0 &\Rightarrow \pi_{ij}(x) \geq 0 \\ x_{ij} < b_{ij} &\Rightarrow \pi_{ij}(x) \leq 0. \end{aligned}$$

Existencia y unicidad de la solución

- Está garantizado que el problema de complementariedad $f(x) \perp [a, b]$ posee una solución única si f es estrictamente monótona negativa, es decir, si $(x-y)'(f(x)-f(y)) < 0$ siempre que $x, y \in [a, b]$ y $x \neq y$.
- Esta es una generalización multidimensional de la condición unicondicional de que f sea estrictamente decreciente.
- La mayoría de los modelos económicos satisfacen esta condición.
- Esta condición es satisfecha por las condiciones Karush-Kuhn-Tucker de un problema de maximización restringida de una función objetivo estrictamente cóncava.

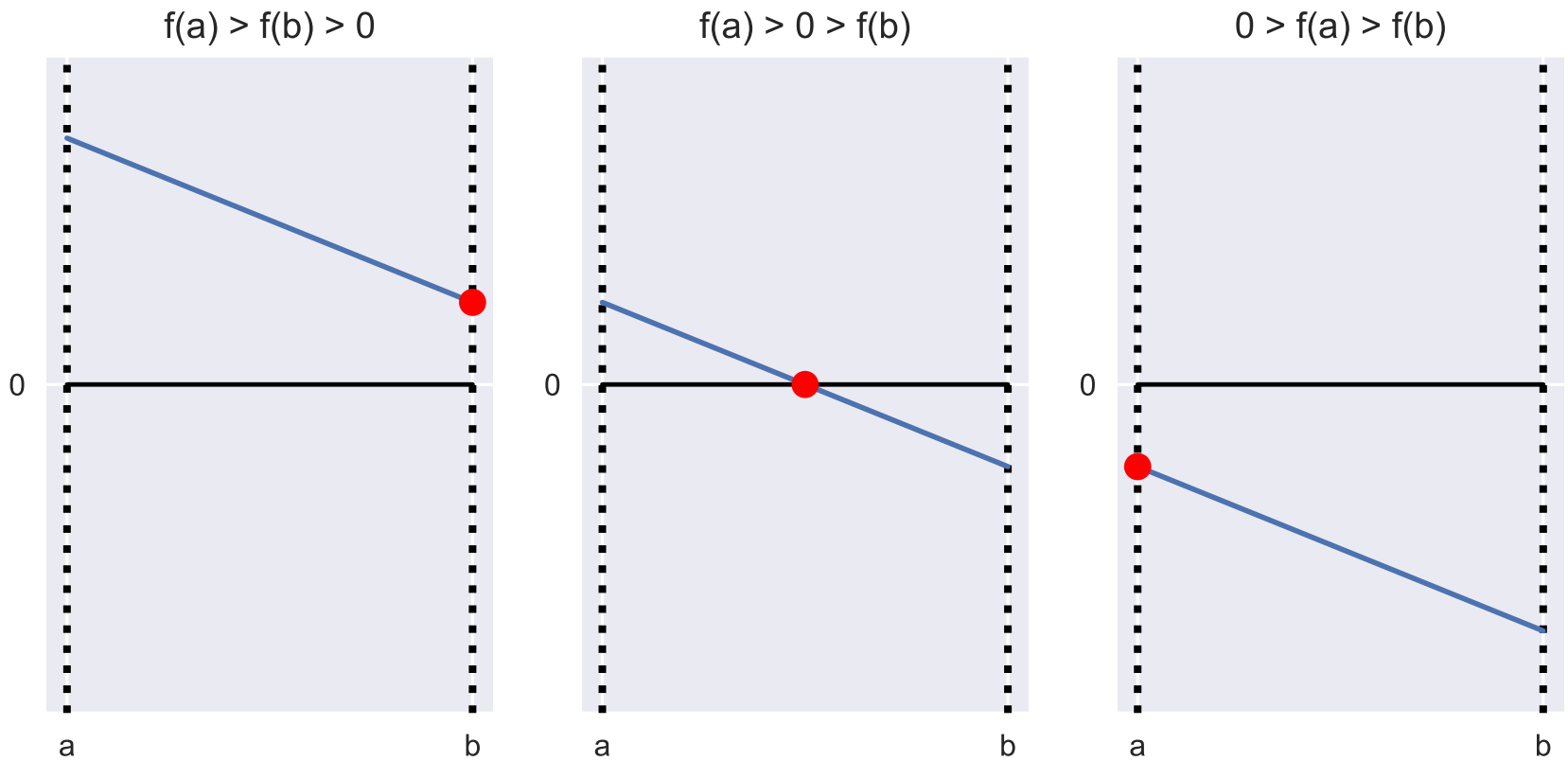

Figura 3.7: Posibles soluciones al problema de complementariedad, f estrictamente decreciente

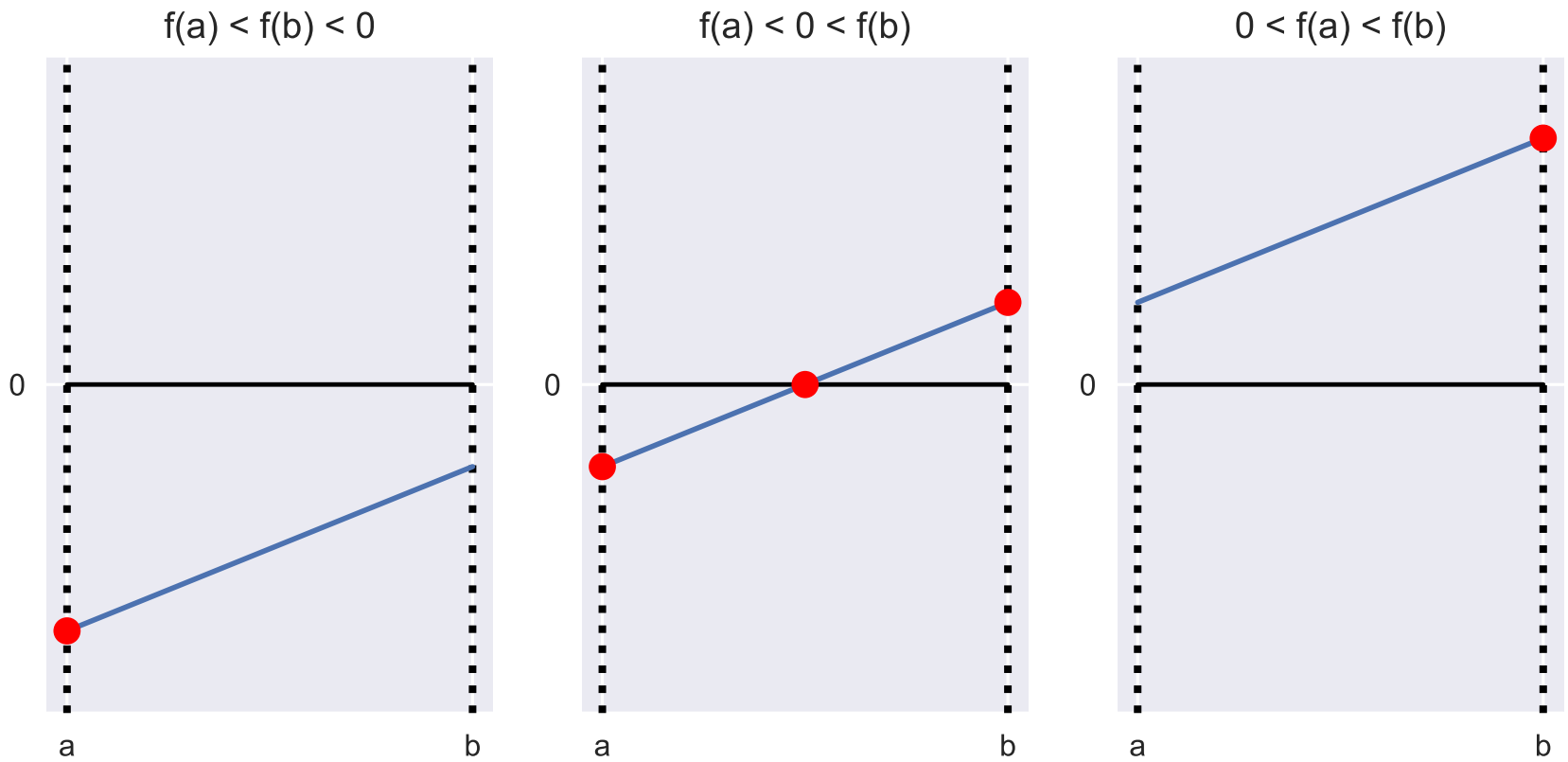

Figura 3.8: Posibles soluciones al problema de complementariedad, f estrictamente creciente

Métodos numéricos de solución

- Un problema de complementariedad puede reformularse como un problema equivalente de búsqueda de raíces y ser resuelto usando métodos estándar de ecuaciones no lineales, tales como los métodos de Newton o Broyden.
- En particular, x resuelve el problema de complementariedad $f(x) \perp [a, b]$ si y sólo si resuelve el problema de búsqueda de raíces

$$\hat{f}(x) = \min(\max(f(x), a - x), b - x) = 0$$

  donde min y max se aplican por fila.
- Intentar resolver $f(x) \perp [a, b]$ calculando la raíz de $\hat{f}(x)$, llamada la formulación min-max, es computacionalmente barato y a menudo funciona bien en la práctica.

- Sin embargo, la formulación min-max conlleva un problema puntiagudo que a veces encuentra problemas computacionales.
- Puede mostrarse también que x resuelve el problema de complementariedad $f(x) \perp [a, b]$ si y sólo si resuelve el problema de búsqueda de raíces

$$\tilde{f}(x) = \phi^{-}(\phi^{+}(f(x), a - x), b - x) = 0$$

donde

$$\phi_i^{\pm}(u, v) = u_i + v_i \pm \sqrt{u_i^2 + v_i^2}.$$

- Intentar calcular la raíz de $\tilde{f}(x)$, llamada la formulación semi-suave, es computacionalmente más caro, pero presenta un problema más suave que a menudo puede resolverse cuando el otro no se puede.

La clase MCP

- El paquete CompEcon define la clase `MCP` para resolver el problema no lineal de complementariedad $f(x) \perp [a, b]$ donde $f : \Re^n \mapsto \Re^n$.
- Creamos un problema no lineal de complementariedad así

```
from compecon import MCP
a, b = ...   #lower and upper bounds
def f(x):    #objective function
    fval = ...   #function value
    fjac = ...   #function Jacobian
    return fval, fjac

problem = MCP(f, a, b)
```

Resolviendo con la transformación min-max

- Para usar la transformación min-max, solo llamamos el método `.zero` del objeto `MCP`, dando un valor inicial `x0`, y fijando la opción `transform='minmax'`

```
x = problem.zero(x0, transform='minmax')
```

- De respuesta, `x` es la solución del problema.
- Para verificar el valor de f, obtenemos el atributo `problem.fx`
- Los pasos pueden combinarse así:

```
x = MCP(f, a, b).zero(x0, transform='minmax')
```

Resolviendo con la transformación semi-suave

- Para usar la transformación semi-suave, de nuevo llamamos `.zero` del objeto `MCP`, dando un valor inicial `x0`, y fijando `transform='ssmooth'`

```
x = problem.zero(x0, transform='ssmooth')
```

- De respuesta, `x` es la raíz del problema.
- Para verificar el valor de f, obtenemos el atributo `problem.fx`
- De nuevo, los pasos pueden combinarse así:

```
x = MCP(f,a,b).zero(x0,transform='ssmooth')
```

Ejemplo 10:
Un problema de complementariedad sencillo

- Consideremos el problema de complementariedad $f(x, y) \perp [a, b]$ donde

$$f(x, y) = \begin{bmatrix} 1 + xy - 2x^3 - x \\ 2x^2 - y \end{bmatrix}$$

y $0 \leq x \leq 1$, $0 \leq y \leq 1$, es decir,

$$a = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

- Para resolver el problema usando la clase `MCP` de CompEcon, primero codificamos la función, y la pasamos a `MCP` junto con los límites:

```
def func(z):
    x, y = z
    return np.array([1 + x*y - 2*x**3 - x,
                     2*x**2 - y])
F = MCP(func, [0, 0], [1,1])
```

- Luego, para resolverlo empezando en $(x, y) = (0.5, 0.5)$ y usando la transformación minmax

```
x0 = [0.5, 0.5]
x = F.zero(x0, transform='minmax')
```

- Luego de 8 iteraciones, una solution $x = (0.7937, 1)$ es encontrada.

- La solución de arriba fue calculada sin necesidad de programar el jacobiano.
- Para calcular la solución con el jacobiano, cambiamos la defición de `func` a

```
def func2(z):
    x, y = z
    f = [1 + x*y - 2*x**3 - x, 2*x**2 - y]
    J = [[y-6*x**2-1, x],[4*x, -1]]
    return np.array(f), np.array(J)

F2 = MCP(func2, [0, 0], [1,1])
x = F2.zero(x0, transform='minmax')
```

- Luego de 4 iteraciones, una solución $x = (0.7937, 1)$ es encontrada.

Ejemplo 11:
Comercio entre 3 países

- Una mercancía es producida y consumida en tres países.
- La demanda y la oferta en los tres países esta dada por:

	Demanda	Oferta
País 1:	$p = 42 - 2q$	$p = 9 + 1q$
País 2:	$p = 54 - 3q$	$p = 3 + 2q$
País 3:	$p = 51 - 1q$	$p = 18 + 1q$

- Los costos unitarios de transporte son:

Desde / hacia	País 1	País 2	País 3
País 1:	0	3	9
País 2:	3	0	3
País 3:	6	3	0

Para resolver el modelo, fijamos los parámetros y definimos una función `market` que toma como insumo un vector de flujos `x` y devuelve las ganancias potenciales de arbitraje`fval`, con `fjac` vacía:

```
import numpy as np
from compecon import MCP

A = np.array
as_, bs = A([9, 3, 18]), A([1, 2, 1])  #supply
ad, bd = A([42, 54, 51]), A([2, 3, 1]) #demand
c = A([[0, 3, 9], [3, 0, 3],[6, 3, 0]]) #transp.cost

def market(x, jac=False):
    quantities = x.reshape((3,3))
    ps = as_ + bs * quantities.sum(0)
    pd = ad - bd * quantities.sum(1)
    ps, pd = np.meshgrid(ps, pd)
    fval = (pd - ps - c).flatten()
    return (fval, None) if jac else fval
```

Luego creamos un objeto MCP

```
a = np.zeros(9)
b = np.full(9, np.inf)
Market = MCP(market, a, b)
```

y resolvemos empezando desde cantidades cero:

```
x0 = np.zeros(9)
x = Market.zero(x0, transform='minmax')

quantities = x.reshape(3,3)
prices = as_ + bs * quantities.sum(0)
exports = quantities.sum(0) - quantities.sum(1)
```

Después de 31 iteraciones, encontramos la solución

```
quantities =
 [[ 9.     -0.      0.    ]
 [ 1.6347  7.3653  0.    ]
 [ 4.3653  4.6347 12.    ]]

prices =
 [24. 27. 30.]

exports =
 [ 6.  3. -9.]
```

References I


Miranda, Mario J. y Paul L. Fackler (2002). *Applied Computational Economics and Finance*. MIT Press. isbn: 0-262-13420-9.


Romero-Aguilar, Randall (2016). *CompEcon-Python*. url: http://randall-romero.com/code/compecon/.