Notas Económicas Regionales
Secretaría Ejecutiva del Consejo Monetario Centroamericano
No. 93, diciembre 2017

Python para economistas

Randall Romero-Aguilar[1]
rromero@secmca.org

En muchos de los trabajos que realizamos los economistas se hace cada vez más necesario el uso de lenguajes de programación, sea porque los modelos teóricos que desarrollamos carecen de soluciones analíticas, porque el nuevo estimador econométrico que deseamos emplear no se encuentra aún disponible en un programa con interface gráfica (donde podamos interactuar con el clic de un ratón), o bien porque no es eficiente analizar grandes volúmenes de datos con hojas de cálculo.

Quienes desean explotar las ventajas de la programación para realizar estas tareas primero deben decidir cuál de los muchos lenguajes de programación aprender. Por ejemplo, los lenguajes R, Python, Julia, Fortran, Gauss, y MATLAB son utilizados en mayor o menor medida por economistas. MATLAB ha sido especialmente popular en este campo, y muchas herramientas se han desarrollado para ejecutarse en este programa, entre ellas DYNARE e IRIS (para resolver y estimar modelos de equilibrio general estocásticos, DSGE), CompEcon (para economía computacional), y Econometrics (para econometría espacial).

A pesar de que Python aún no es tan popular como MATLAB entre economistas, ciertamente en años recientes su popularidad ha crecido enormemente. Por ejemplo, los siguientes libros usan Python para realizar labores típicas de los economistas:

- QuantEcon de Thomas Sargent y John Stachurski.
- Economic Dynamics: Theory and Computation, de Stachurski.
- Python for Econometrics, de Kevin Sheppard.

Python es un lenguaje versátil y fácil de aprender —de hecho es muy utilizado en las mejores universidades de Estados Unidos para enseñar cursos introductorios de informática. Su sintaxis es muy clara, lo que facilita el desarrollo y mantenimiento del código. Debido a que es uno de los lenguajes más populares entre programadores informáticos,

[1]Economista Consultor de la Secretaría Ejecutiva del Consejo Monetario Centroamericano (SECMCA) y profesor de economía en la Universidad de Costa Rica (UCR). Doctor en Economía por la Ohio State University (OSU) y Master en Econometría por la London School of Economics (LSE). Las opiniones expresadas son las del autor y no necesariamente representan la posición de la SECMCA, ni de los miembros del CMCA.

existen múltiples facilidades para aprenderlo (libros, páginas de Internet). Es una excelente herramienta para ejecutar tareas de cálculos científicos (gracias a paquetes como Numpy y Scipy), manejo de datos (pandas), visualización (Matplotlib) y modelación econométrica (Statsmodels).

Otra ventaja de utilizar Python es que, a diferencia de programas propietarios, Python y muchos de estos paquetes complementarios son completamente gratuitos. La mejor forma de conseguir Python es a través de Anaconda, una distribución gratuita que incluye más de 300 paquetes de gran utilidad en ciencias, matemática, ingeniería, y análisis de datos. Además de Python, Anaconda incluye herramientas como IPython (para ejecutar Python de manera interactiva), Jupyter (un editor que permite integrar texto, código y resultados en un sólo archivo, excelente para documentar trabajos), Spyder (una interfaz gráfica para editar código, similar a la interfaz de MATLAB) y Conda (permite instalar y actualizar paquetes).

Quienes deseen empezar a trabajar en Python deben tener en cuenta dos asuntos. Primero, actualmente existen dos versiones de Python que no son enteramente compatibles entre sí, la 2 (cuya última actualización es la 2.7) y la 3 (actualmente actualizada a 3.6). En lo personal, recomiendo trabajar con la versión 3.6 porque tiene mejoras importantes con respecto a la versión 2.7, y además la mayoría de los paquetes necesarios para trabajar en aplicaciones típicas de un economista ya han sido portadas a la 3.6.

Segundo, aunque Spyder facilita la edición de código, usuarios más avanzados podrían preferir PyCharm, un excelente editor de Python cuya versión "Community" puede utilizarse gratuitamente. Este editor facilita mucho la edición de programas, ya que cuenta con funciones como autocompletado (especialmente útil cuando aún no hemos memorizado las funciones de Python), resaltado de sintaxis (muestra palabras claves de color distinto, para que sea más fácil entender la lógica del programa escrito), y depurador (para ejecutar parcialmente un programa cuando es necesario encontrar un error).

El objetivo de esta nota es ilustrar algunas de las tareas comunes que los economistas pueden ejecutar utilizando Python. Primero, se replican dos modelos de competencia de Cournot presentados por Miranda y Fackler (2002) que se resuelven con técnicas numéricas utilizando el paquete "CompEcon-python"[2], el cual está disponible libremente en Github[3]. Segundo, se ilustra cómo automatizar la obtención de datos de Internet y su presentación en tablas y gráficos. Tercero, se muestran algunos ejemplos de modelos econométricos estimados en Python.

[2]Este paquete fue desarrollado por el autor y está basado precisamente en el *toolbox* "CompEcon" para MATLAB de Miranda y Fackler (2002).

[3]Los lectores interesados en el tema de economía computacional encontrarán más de estos ejemplos en Romero-Aguilar (2016).

Para cada uno de los problemas que se presentan, se incluye el código de Python que lo resuelve, así como breves explicaciones de cómo trabaja este código. No obstante, esta nota no pretende enseñar a programar en Python, porque como se mencionó anteriormente, existen ya muchos recursos didácticos de gran calidad para este fin, entre ellos el sitio de desarrolladores de Google, el sitio learnpython, y varios cursos en línea en edx. De igual manera, en los dos primeros ejemplos se presentan concisamente los métodos numéricos que se implementan en Python, pero se recomienda a los lectores interesados en este tema consultar los libros de texto de Miranda y Fackler (2002), Judd (1998), y Press, Teukolsky y Brian P. Flannery (2007) (todos ellos disponibles en inglés).

Ejemplo 1: Un modelo de Cournot con 2 empresas

Suponga que el mercado de un producto está dominado por dos empresas que compiten entre sí. Para este duopolio, la inversa de la función de demanda está dada por

$$P(q) = q^{-\alpha}$$

y ambas empresas tienen costos cuadráticos

$$C_1 = \tfrac{1}{2}\beta_1 q_1^2$$
$$C_2 = \tfrac{1}{2}\beta_2 q_2^2$$

Las ganancias de las empresas son

$$\pi_1(q_1, q_2) = P(q_1 + q_2)\, q_1 - C_1(q_1)$$
$$\pi_2(q_1, q_2) = P(q_1 + q_2)\, q_2 - C_2(q_2)$$

En un equilibrio de Cournot, cada empresa maximiza sus ganancias tomando como dadas la producción de la otra empresa. Así, debe cumplirse que

$$\frac{\partial \pi_1(q_1, q_2)}{\partial q_1} = P(q_1 + q_2) + P'(q_1 + q_2)\, q_1 - C_1'(q_1) = 0$$
$$\frac{\partial \pi_2(q_1, q_2)}{\partial q_2} = P(q_1 + q_2) + P'(q_1 + q_2)\, q_2 - C_2'(q_2) = 0$$

De tal manera que los niveles de producción de equilibrio de este mercado viene dado por la solución a este sistema de ecuaciones no lineales

$$f(q_1, q_2) = \begin{bmatrix} (q_1 + q_2)^{-\alpha} - \alpha q_1 (q_1 + q_2)^{-\alpha - 1} - \beta_1 q_1 \\ (q_1 + q_2)^{-\alpha} - \alpha q_2 (q_1 + q_2)^{-\alpha - 1} - \beta_2 q_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{1}$$

El método de Newton

Para encontrar la raíz de la función definida en (1) utilizaremos el método de Newton. En general, este método se aplica a la función $f : \Re^n \to \Re^n$ para encontrar algún[4] valor x^* tal que $f(x^*) = 0$. Para ello, partimos de un valor $x_0 \in \Re^n$ y formamos la recursión

$$x_{i+1} = x_i - J^{-1}(x_i) f(x_i) \tag{2}$$

donde $J(x_i)$ corresponde al jacobiano de f evaluado en x_0. En teoría, siguiendo esta recursión x_i converge a x^* siempre y cuando la función f sea continuamente diferenciable y el valor inicial x_0 esté "suficientemente cercano" a la raíz x^*.

[4]Nótese que, dependiendo de la función, podría haber más de una solución, o ninguna solución del todo.

Resolviendo el modelo con Python

Primero, iniciamos una sesión de Python e importamos `compecon`

```
import numpy as np
import matplotlib.pyplot as plt

from compecon import NLP, gridmake
from compecon.demos import demo
```

Para resolver este modelo computacionalmente, es necesario asignar valores a los parámetros, por lo que fijamos los valores $\alpha = 0.625$, $\beta_1 = 0.6$ y $\beta_2 = 0.8$.

```
alpha = 0.625
beta = np.array([0.6, 0.8])
```

Las incógnitas de nuestro problema son los niveles de producción de cada empresa, q_1 y q_2. Definimos la función `market` que nos dice la cantidad total de producción y el precio resultante del bien, dado los niveles de q_1 y q_2. Nótese que ambas cantidades son pasadas a esta función en el vector `q`

```
def market(q):
    quantity = q.sum()
    price = quantity ** (-alpha)
    return price, quantity
```

Luego, definimos la función `cournot`, la cual retorna una tupla con dos elementos: la función objetivo y su jacobiano, ambos evaluados en un par de cantidades contenidas en el vector `q`. Para facilitar el código, nótese que la función (1) puede escribirse más sucintamente como

$$f\left(q_1, q_2\right) = \begin{bmatrix} P + \left(P' - c_1\right) q_1 \\ P + \left(P' - c_2\right) q_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

y su jacobiano es

$$J\left(q_1, q_2\right) = \begin{bmatrix} 2P' + P''q_1 - c_1 & P' + P''q_1 \\ P' - P''q_2 & 2P' + P''q_2 - c_2 \end{bmatrix}$$

Si definimos la producción total como $Q = q_1 + q_2$, note además que

$$P' = -\alpha \frac{P}{Q} \qquad \text{y que} \qquad P'' = -(\alpha + 1)\frac{P'}{Q}$$

```
def cournot(q):
    P, Q = market(q)
    P1 = -alpha * P/Q
    P2 = (-alpha - 1) * P1 / Q
    fval = P + (P1 - beta) * q
    fjac = np.diag(2*P1 + P2*q - beta) + np.fliplr(np.diag(P1 + P2*q))
    return fval, fjac
```

A continuación, calculamos el equilibrio usando el método de Newton (ecuación (2)) para encontrar la raíz de la función `cournot`. Partimos de $q_0 = \begin{bmatrix}0.2 & 0.2\end{bmatrix}'$ como nuestro valor inicial e iteramos hasta que la norma del cambio entre dos valores sucesivos de la recursión sea menor que 10^{-10}.

```
q = np.array([0.2, 0.2])

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

price, quantity = market(q)
print(f'\nCompany 1 produces {q[0]:.4f} units, while' +
 f' company 2 produces {q[1]:.4f} units.')
print(f'Total production is {quantity:.4f} and price is {price:.4f}')
```

Luego de solo 5 iteraciones, el método de Newton converge a la respuesta, la cual Python muestra en pantalla:

```
Company 1 produces 0.8396 units, while company 2 produces 0.6888 units. Total production is 1.5284 and price is 0.7671
```

Vemos que el programa ha encontrado el equilibrio de este mercado.

La librería `compecon` provee la clase `NLP` (non-linear problem), con la cual podemos resolver el problema anterior sin necesidad de programar el algoritmo de Newton. Para utilizarla, creamos una instancia del `NLP` a partir de la función `cournot`, y simplemente ejecutamos el método `newton`, usando `q0` como punto de partida.

```
q0 = np.array([0.2, 0.2])
cournot_problem = NLP(cournot)
q = cournot_problem.newton(q0)
```

```
price, quantity = market(q)
print(f'\nCompany 1 produces {q[0]:.4f} units, while' +
 f' company 2 produces {q[1]:.4f} units.')
print(f'Total production is {quantity:.4f} and price is {price:.4f}')
```

Al completar este bloque, Python muestra lo siguiente en pantalla:

```
Company 1 produces 0.8396 units, while company 2 produces 0.6888 units. Total production is 1.5284 and price is 0.7671
```

Como es de esperar, hemos obtenido el mismo resultado.

El problema que hemos resuelto se ilustra en la figura 1, cuyos ejes representan los niveles de producción de cada empresa. La línea blanca cuasi-vertical representa el nivel que maximiza las ganancias de la empresa 1, tomando como dada la producción de la empresa 2. De manera similar, la línea blanca cuasi-horizontal representa el nivel que maximiza las ganancias de la empresa 2, dada la producción de la empresa 1. La solución del problema corresponde a la intersección de estas dos líneas. Observe además la trayectoria de convergencia (línea azul) desde el punto inicial $q_0 = \begin{bmatrix} 0.2 & 0.2 \end{bmatrix}'$ hasta la solución.

```
n = 100
q1 = np.linspace(0.1, 1.5, n)
q2 = np.linspace(0.1, 1.5, n)
z = np.array([cournot(q)[0] for q in gridmake(q1, q2).T]).T
```

```
steps_options = {'marker': 'o',
                 'color': (0.2, 0.2, .81),
                 'linewidth': 2.5,
                 'markersize': 9,
                 'markerfacecolor': 'white',
                 'markeredgecolor': 'red'}

contour_options = {'levels': [0.0],
                   'colors': 'white',
                   'linewidths': 2.0}


Q1, Q2 = np.meshgrid(q1, q2)
Z0 = np.reshape(z[0], (n,n), order='F')
Z1 = np.reshape(z[1], (n,n), order='F')
```

```
methods = ['newton', 'broyden']
cournot_problem.opts['maxit', 'maxsteps', 'all_x'] = 10, 0, True

qmin, qmax = 0.1, 1.3
x = cournot_problem.zero(method='newton')
demo.figure("Convergence of Newton's method", '$q_1$', '$q_2$',
 [qmin, qmax], [qmin, qmax])
plt.contour(Q1, Q2, Z0, **contour_options)
plt.contour(Q1, Q2, Z1, **contour_options)

plt.plot(*cournot_problem.x_sequence, **steps_options)

demo.text(0.85, qmax, '$\pi_1 = 0$', 'left', 'top')
demo.text(qmax, 0.55, '$\pi_2 = 0$', 'right', 'center')
```

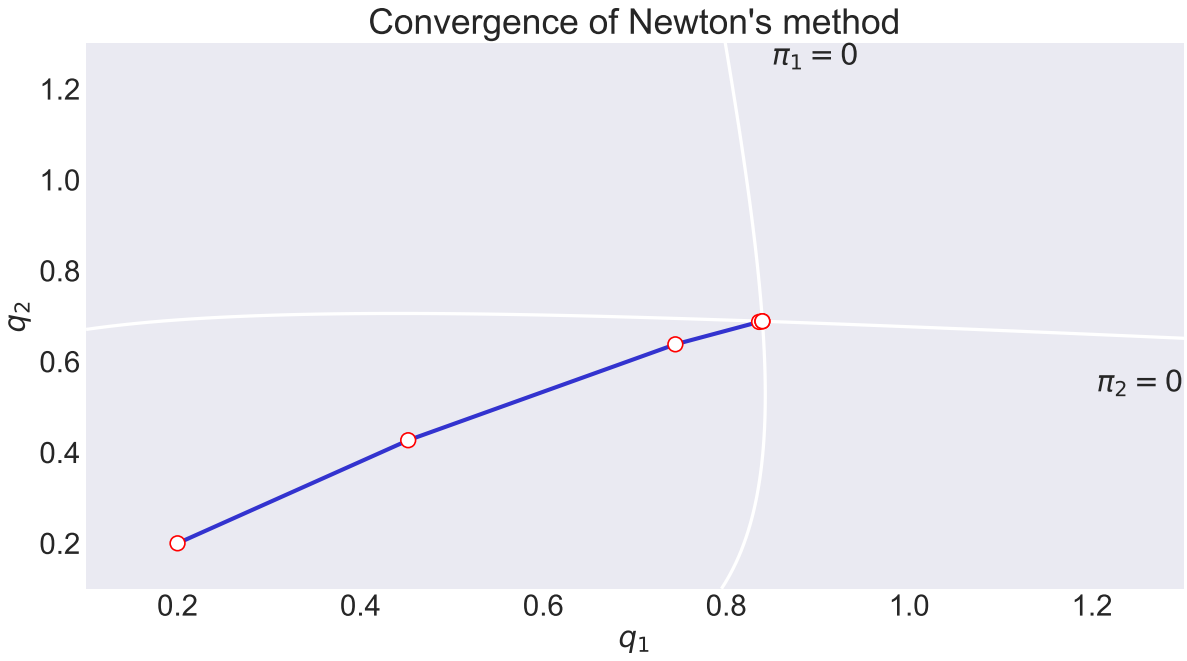

Figura 1: Convergencia hacia la raíz del sistema

Ejemplo 2: Resolviendo un oligopolio de Cournot vía colocación

Para ilustrar la implementación del método de colocación para problemas de función implícita, considere el caso del oligopolio de Cournot. En el modelo microeconómico estándar de la empresa, la empresa maximiza sus ganancias al igualar el ingreso marginal al costo marginal (MC). Una empresa oligopolística, reconociendo que sus acciones afectan al precio, sabe que su ingreso marginal es $p + q\frac{dp}{dq}$, donde p es el precio, q la cantidad producida, y $\frac{dp}{dq}$ es el impacto marginal del producto sobre el precio de mercado. El supuesto de Cournot es que la empresa actúa como si ninguno de sus cambios de producción provocaría una reacción de sus competidores. Esto implica que:

$$\frac{dp}{dq} = \frac{1}{D'(p)} \tag{3}$$

donde $D(p)$ es la curva de demanda del mercado.

Suponga que deseamos derivar la función de oferta efectiva de la empresa, la cual especifica la cantidad $q = S(p)$ que proveerá a cada precio. La función de oferta efectiva de la empresa está caracterizada por la ecuación funcional

$$p + \frac{S(p)}{D'(p)} - MC(S(p)) = 0 \tag{4}$$

para todo precio $p > 0$. En casos simples, puede hallarse esta función explícitamente. No obstante, en casos más complicados, no existe solución explícita. Suponga por ejemplo que la demanda y el costo marginal vienen dados por

$$D(p) = p^{-\eta} \qquad\qquad CM(q) = \alpha\sqrt{q} + q^2$$

con lo que la ecuación funcional a resolver para $S(p)$ es

$$\left[p - \frac{S(p)p^{\eta+1}}{\eta}\right] - \left[\alpha\sqrt{S(p)} + S(p)^2\right] = 0 \tag{5}$$

El método de colocación

En la ecuación (5), la incógnita es la *función* de oferta $S(p)$, lo que convierte a (5) en una ecuación de infinitas dimensiones. En vez de resolver la ecuación directamente, aproximaremos la solución utilizando n polinomios de Chebyshev $\phi_i(x)$, los cuales se definen recursivamente para $x \in [0, 1]$ como:

$$\begin{aligned}
\phi_0(x) &= 1 \\
\phi_1(x) &= x \\
\phi_{k+1}(p_i) &= 2x\phi_k(x) - \phi_{k-1}(x), \qquad \text{para } k = 1, 2, \ldots
\end{aligned}$$

Además, en vez de requerir que la ecuación se cumpla exactamente sobre todo el dominio de $p \in \Re^+$, escogeremos n nodos de Chebyshev p_i pertenecientes al intervalo $[a, b]$:

$$p_i = \frac{a+b}{2} + \frac{b-a}{2} \cos\left(\frac{n-i+0.5}{n}\pi\right), \qquad \text{para } i = 1, 2, \ldots, n \tag{6}$$

Así, la oferta se aproxima como

$$S(p_i) = \sum_{k=0}^{n-1} c_k \phi_k(p_i)$$

Sustituyendo esta última expresión en (5) para cada uno de los nodos de colocación (Chebyshev en este caso) resulta en un sistema no lineal de n ecuaciones (una por cada nodo) en n incógnitas c_k (una por cada polinomio de Cheybshev), el cual en principio puede ser resuelto por el método de Newton, como en el ejemplo pasado. Así, en la práctica el sistema a resolver es

$$\left[p_i - \frac{\left(\sum_{k=0}^{n-1} c_k \phi_k(p_i)\right) p_i^{\eta+1}}{\eta}\right] - \left[\alpha \sqrt{\sum_{k=0}^{n-1} c_k \phi_k(p_i)} + \left(\sum_{k=0}^{n-1} c_k \phi_k(p_i)\right)^2\right] = 0 \tag{7}$$

para $i = 1, 2, \ldots, n$ y para $k = 1, 2, \ldots, n$.

Resolviendo el modelo con Python

Para resolver este modelo de nuevo iniciamos una sesión de Python:

```
import numpy as np
import matplotlib.pyplot as plt

from compecon import BasisChebyshev, NLP, nodeunif
from compecon.demos import demo
```

y fijamos los parámetros α y η

```
alpha, eta = 1.0, 3.5
```

Por conveniencia, creamos una función `lambda` para la demanda

```
D = lambda p: p**(-eta)
```

La solución la aproximaremos para precios en el rango $p \in [\frac{1}{2}, 2]$, usando 25 nodos de colocación. La librería `compecon` ofrece la clase `BasisChebyshev` para realizar operaciones con bases de Chebyshev:

```
n, a, b =  25, 0.5, 2.0
S = BasisChebyshev(n, a, b, labels=['price'], l=['supply'])
```

Vamos a suponer que nuestra primera adivinanza es que $S(p) = 1$. Para ello fijamos el valor de `S` igual a uno en cada uno de los nodos

```
p = S.nodes
S.y = np.ones_like(p)
```

Es importante recalcar que en este problema, la incógnita son los coeficientes c_k de la base de Chebyshev; no obstante, un objeto de clase `BasisChebyshev` ajusta automáticamente el valor de los coeficientes que son consistentes con los valores que asignemos a la función en los nodos (indicados acá por la propiedad `.y`)

Estamos ahora en posición de definir la función objetivo, que llamaremos `resid`. Esta función toma por argumento los 25 coeficientes de la base de Chebyshev y retorna el valor del lado izquierdo de las 25 ecuaciones definidas por (7).

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

Note que la función `resid` toma un único argumento (los coeficientes para la base de Chebyshev). Los demás parámetros (`Q, p, eta, alpha` deben ser declarados como tales en el código principal, donde Python podrá encontrar sus valores.

Para utilizar el método de Newton, es indispensable contar con el jacobiano de la función cuyos ceros estamos buscando. En ocasiones, como en el problema que estamos resolviendo, escribir este jacobiano correctamente puede resultar engorroso. La clase `NLP` ofrece además del método de Newton (que utilizamos en el ejemplo anterior) el método de Broyden, el cual tiene la ventaja de que no requiere conocer el jacobiano (el método mismo lo aproximará).

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

Después de 17 iteraciones el método de Broyden converge a la solución deseada. Esto lo podemos visualizar en la figura 3, la cual muestra el valor de la función en 501 puntos distintos dentro del rango de aproximación. Obsérvese que la gráfica del residuo cruza 25 veces el eje horizontal; esto sucede precisamente en los nodos de colocación (puntos rojos). Esta figura muestra además la calidad de la aproximación: fuera de los nodos, la función no difiere de cero por más de 5×10^{-11}.

Una de las ventajas de trabajar con la clase `BasisChebyshev` es que, una vez encontrados los coeficientes de colocación, podemos evaluar la función de oferta tratando al objeto `S` como si fuese una función de Python. Así, por ejemplo, para saber la cantidad ofrecida por una empresa cuando el precio es 1.2 simplemente evaluamos `print(S(1.2))` lo cual retorna 0.4650. Esta funcionalidad la utilizamos a continuación para calcular la oferta efectiva cuando hay 5 empresas idénticas en el mercado; el resultado se muestra en la figura 2.

```
pplot = nodeunif(501, a, b)
demo.figure('Cournot Effective Firm Supply Function',
            'Quantity', 'Price', [0, 4], [a, b])
plt.plot(5 * S(pplot), pplot, D(pplot), pplot)
plt.legend(('Supply','Demand'))
```

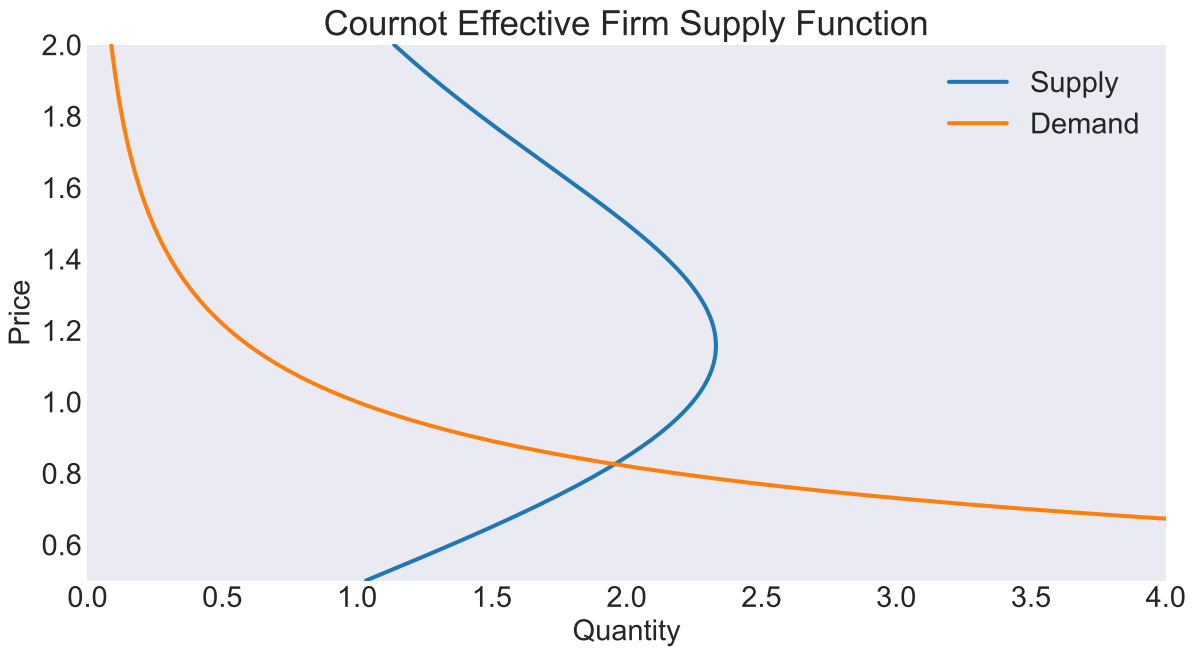

Figura 2: Oferta y demanda cuando hay 5 empresas

El código que se muestra en el siguiente bloque genera la figura 3.

```
p = pplot
demo.figure('Residual Function for Cournot Problem',
            'Quantity', 'Residual')
plt.hlines(0, a, b, 'k', '--', lw=2)
plt.plot(pplot, resid(S.c))
plt.plot(S.nodes,np.zeros_like(S.nodes),'r*')
```

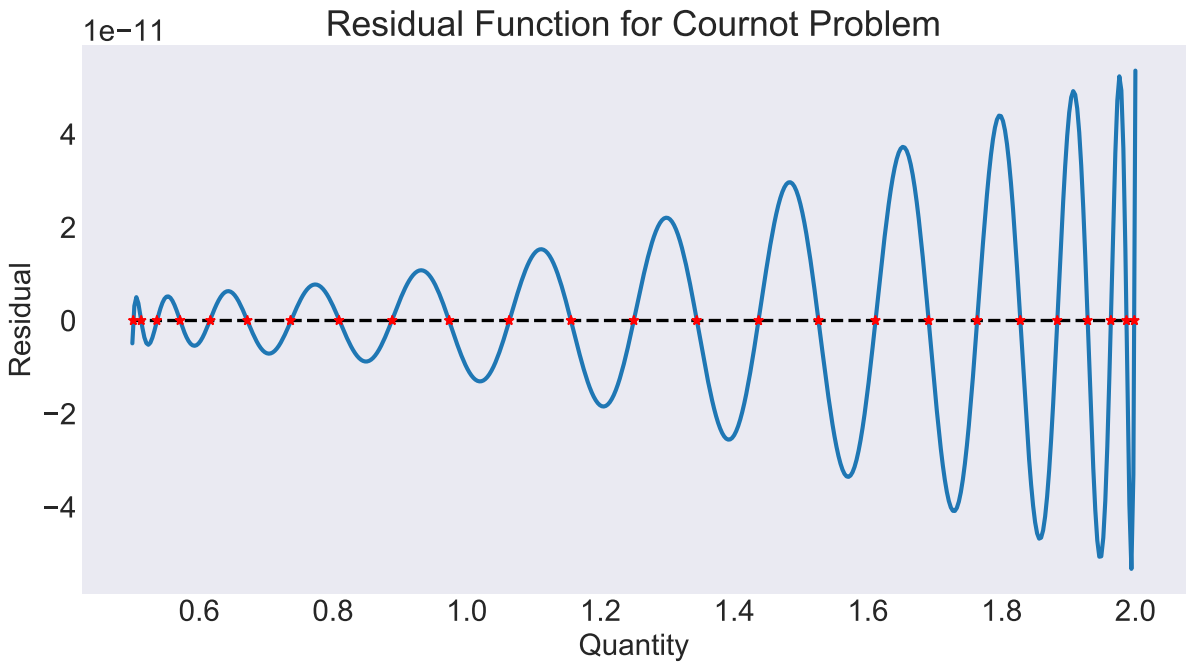

Figura 3: Residuos de aproximar la ecuación (5)

Ahora, graficamos la demanda efectiva para distintos números de empresas; el resultado se muestra en la figura 4.

```
m = np.array([1, 3, 5, 10, 15, 20])
demo.figure('Supply and Demand Functions', 'Quantity', 'Price', [0, 13])
plt.plot(np.outer(S(pplot), m), pplot)
plt.plot(D(pplot), pplot, linewidth=4, color='black')
plt.legend(['m=1', 'm=3', 'm=5', 'm=10', 'm=15', 'm=20', 'demand'])
```

Notése en la figura 4 cómo el precio y la cantidad de equilibrio cambian conforme aumenta el número de empresas. En la última figura de este ejemplo (figura 5), se muestra el precio de equilibrio en función del número de empresas.

```
pp = (b + a) / 2
dp = (b - a) / 2
m  = np.arange(1, 26)
for i in range(50):
    dp /= 2
    pp = pp - np.sign(S(pp) * m - D(pp)) * dp

demo.figure('Cournot Equilibrium Price as Function of Industry Size',
```

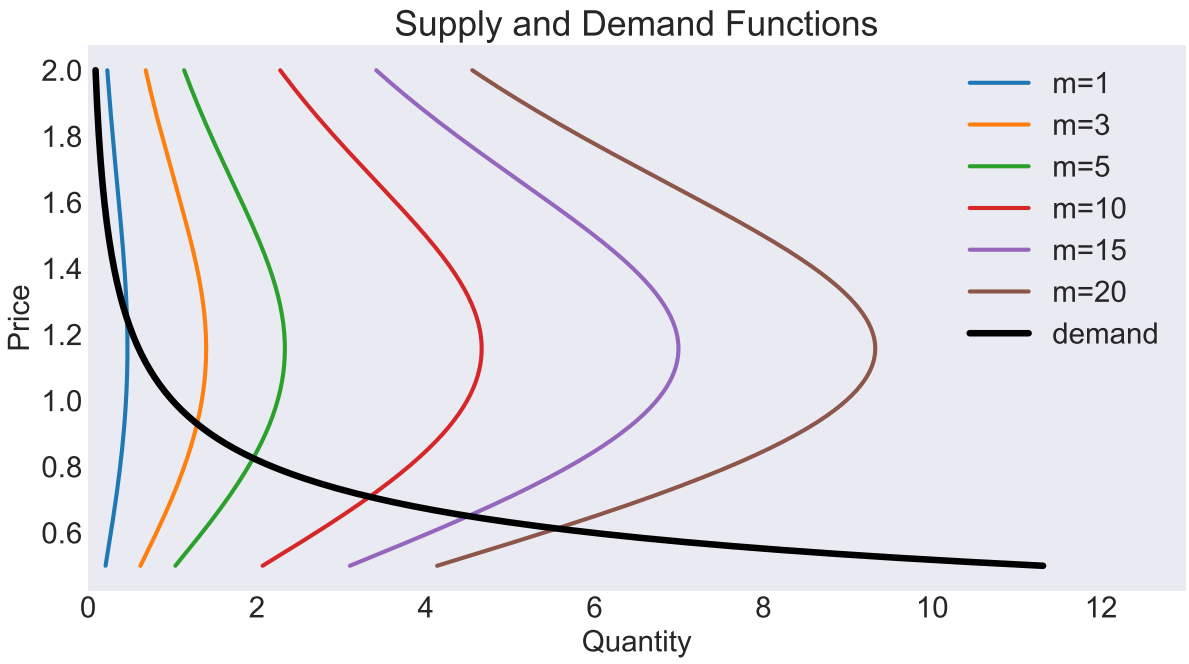

Figura 4: Cambios en la oferta efectiva conforme aumenta el número de empresas

```
            'Number of Firms', 'Price')
plt.bar(m, pp)
```

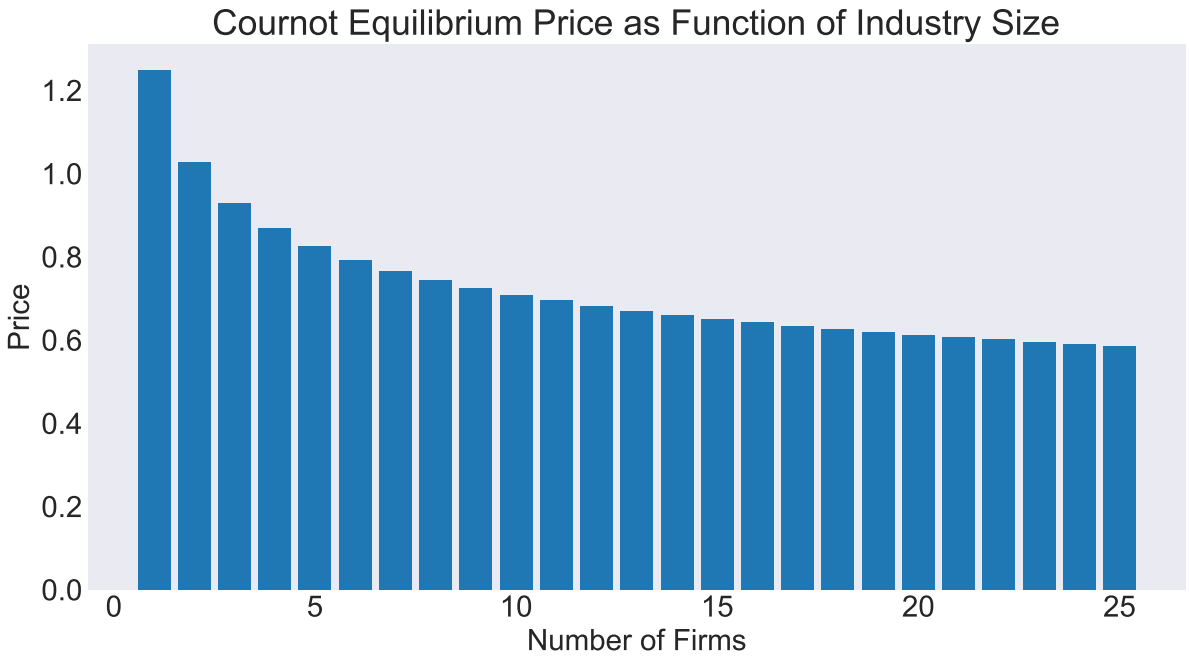

Figura 5: Precio de equilibrio en función del número de empresas

Ejemplo 3: Importando datos de Internet

A menudo necesitamos dar seguimiento a algunos indicadores económicos. Esta labor usualmente requiere visitar el sitio de Internet de algún proveedor de datos, buscar los indicadores de interés, bajar los datos (posiblemente en varios archivos distintos), copiarlos a un archivo común, acomodarlos apropiadamente, y solamente después de completar estas engorrosas tareas, graficarlos. Si esta labor hay que realizarla periódicamente entonces también es necesario documentar bien cada uno de estos pasos para poder replicarlos exactamente en un futuro. Sobra decir que, si es necesario hacer todas estas tareas con numerosos indicadores, la labor termina demandando una cantidad considerable de tiempo y está propensa a muchos errores.

Para facilitar esta labor, Python también permite obtener directamente datos disponibles en Internet, gracias a paquetes como pandas-datareader. Esto se logra más fácilmente cuando los proveedores de datos ofrecen un API –application program interface—que especifica la manera en que un programa como Python pueden encontrar los datos de interés.

Ilustremos esto con un ejemplo. Suponga que deseamos contar con información reciente acerca del crecimiento de las economías de los países miembros del CMCA. El Banco Mundial ofrece los datos relevantes en su "World Database", los cuales podemos leer con el módulo `wb` de `pandas_datareader`.

```
from pandas_datareader import wb
```

Para poder leer datos del Banco Mundial, primero necesitamos saber el código exacto del indicador que deseamos leer. La primera vez que realizamos esta labor no sabemos este código, pero podemos buscarlo en la página del Banco Mundial o más fácilmente desde Python mismo. Así, para encontrar datos acerca del PIB real per cápita, ejecutamos lo siguiente con la función `.search`:

```
wb.search('gdp.*capita.*const').iloc[:,:2]
```

donde el punto seguido de asterisco (.*) indica que puede aparecer cualquier texto en su posición. Esta función retorna una tabla de datos con información de indicadores que cumplen el criterio de búsqueda. En la línea anterior, el código `.iloc[:,:2]` lo utilizamos para que Python solo imprima las primeras dos columnas de esa tabla.

Luego de ejecutar esta búsqueda, escogemos la variable 'NY.GDP.PCAP.KD', cuya descripción es "GDP per capita (constant 2010 US$)". Definimos una variable con una lista de los códigos de país de los países del CMCA:

```
paises = ['CR', 'DO', 'GT', 'HN', 'NI', 'SV']
```

y procedemos a leer los datos desde 1991:

```
datos = wb.download(indicator='NY.GDP.PCAP.KD',
  country=paises,start=1991, end=2016)
```

Es posible obtener datos de más de un indicador en una sola llamada a la función `wb.download`, escribiendo sus códigos en una lista (tal como hemos hecho para obtener datos de los seis países de una sola vez). En todos los casos, lo que obtenemos es una tabla de datos en formato de panel, donde cada columna corresponde a uno de los indicadores. Para este ejemplo en particular, donde solo tenemos una variable, sería útil que la tabla se organice de forma tal que cada fila corresponda a un año y cada columna a un país. Esto lo logramos con esta instrucción:

```
GDP = datos.reset_index().pivot('year','country')
```

Una vez que los datos están organizados de esta manera, es muy sencillo calcular el crecimiento de todos los países en un solo paso:

```
GROWTH = 100 * GDP.pct_change()
```

o bien generar una tabla con los datos, para incluirla en documento de LaTeX

```
GROWTH.tail(6).round(2).to_latex('micuadro.tex')
```

En la instrucción anterior, la parte `.tail(6)` indica que solo queremos las últimas 6 observaciones, mientras que la parte `.to_latex('micuadro.tex')` exporta esa tabla a un archivo con nombre 'micuadro.tex', el cual puede posteriormente incluirse en un documento. El resultado (ligeramente editado) de esta parte está en el cuadro 1.

Finalmente, graficamos los resultados en la figura 6. Cabe apuntar que es posible mejorar la presentación estética de esta figura, por ejemplo, cambiando la leyenda de posición. Estas mejoras no se presentan acá por consideraciones de espacio.

	Costa Rica	Dominican Republic	El Salvador	Guatemala	Honduras	Nicaragua
2011	3.06	1.77	1.76	1.94	1.89	5.03
2012	3.59	1.50	1.41	0.80	2.24	5.24
2013	1.13	3.45	1.37	1.54	0.99	3.72
2014	2.54	6.32	0.93	2.03	1.29	3.60
2015	3.63	5.79	1.79	2.03	1.89	3.68
2016	3.27	5.44	1.85	1.02	1.88	3.55

Cuadro 1: Crecimiento del PIB per cápita en CARD, 2011-2016

Figura 6: Crecimiento del PIB per capita en CARD, 1992-2016

```
GROWTH.columns = paises
GROWTH.plot()
```

También es posible graficar cada una de las series en un eje separado, con la instrucción

```
GROWTH.plot(subplots=True, layout=[2,3], sharey=True)
```

donde hemos especificado que las series deben graficarse por separado (`subplots=True`), presentarse en dos filas y tres columnas (`layout=[2,3]`), y que todas las gráficas deben tener el mismo eje "y" (`sharey=True`, para facilitar la comparación entre países). En la figura 7 se muestra el resultado.

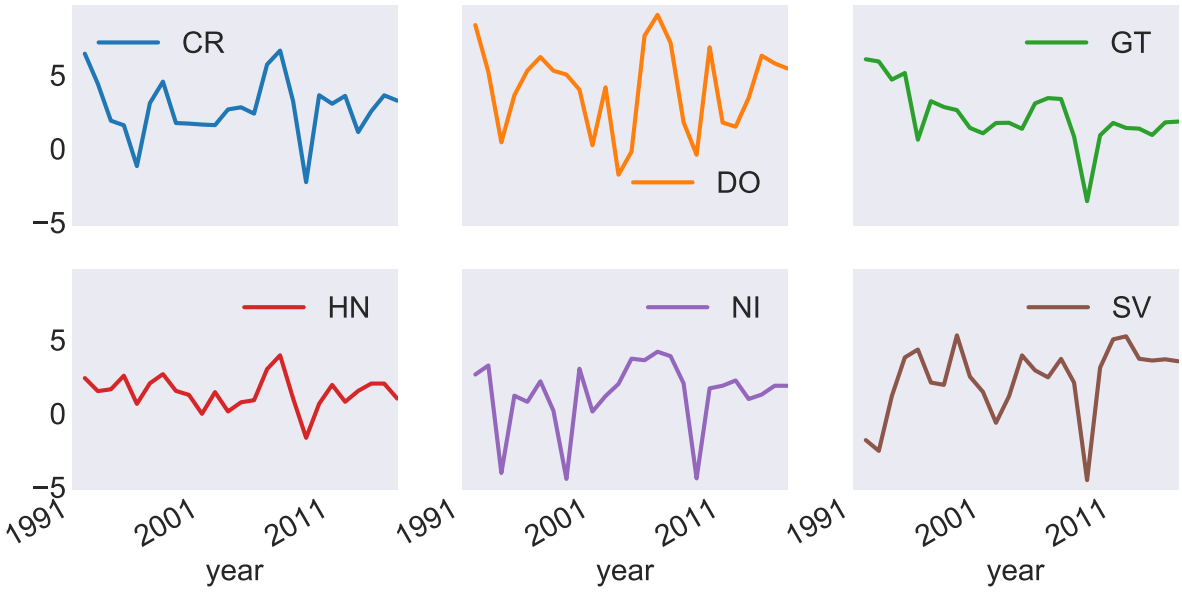

Figura 7: Crecimiento del PIB per capita en CARD, por país, 1992-2016

Ejemplo 4: Estimaciones econométricas

La librería `statsmodels` de Python permite estimar muchos tipos de modelos econométricos, aunque quizá no tantos como los que se pueden estimar con R. Una ilustración sencilla es la estimación de una función de consumo keynesiana,

$$\ln(c_t) = \beta_0 + \beta_1 \ln(y_t) + \epsilon_t$$

donde c_t representa el consumo, y_t el ingreso, ϵ es el término estocástico. En este caso β_1 corresponde a la elasticidad-ingreso del consumo.

Al igual que en el ejemplo anterior, utilizaremos `pandas-datareader` para importar datos de Internet. En este ejemplo importamos además la función `log` de la librería `numpy` para calcular el logaritmo de los datos, así como el módulo `formula.api` de `statsmodels` para hacer la estimación del modelo.

```
import pandas_datareader.data as web
from numpy import log
import statsmodels.formula.api as smf
```

Hecho esto, estamos listos para importar los datos. Para este ejemplo, utilizamos datos trimestrales de consumo y producción de Estados Unidos disponibles en FRED, una base de datos del Banco de la Reserva Federal de San Luis. Para "consumo" usamos la serie "PCEC" (Personal Consumption Expenditures) y como "ingreso" utilizamos "GDP" (Gross Domestic Product).

```
usdata = web.DataReader(['PCEC','GDP'],'fred', 1947, 2017)
```

Al ejecutar este comando, la variable `usdata` es una tabla de datos `pandas`, en la que cada columna corresponde a una variable y cada fila a un trimestre. Ahora estimamos el modelo por mínimos cuadrados ordinarios (`.ols`) e imprimimos un resumen de los resultados

```
mod = smf.ols('PCEC ~ GDP', log(usdata)).fit()
print(mod.summary())
```

Obsérvese que la función `.ols` toma dos argumentos, la fórmula con la especificación del modelo, y el nombre de la tabla de datos que contiene las variables. En el bloque de código anterior especificamos los datos como `log(usdata)`, lo cual le indica a Python que deseamos los datos en logaritmos, y nos evita la tarea de generar las series transformadas de forma previa (como sería necesario en, por ejemplo, Stata). Alternativamente, esa línea puede escribirse también como

```
mod = smf.ols('log(PCEC) ~ log(GDP)', usdata).fit()
```

que resulta conveniente en casos donde no todas las variables de la tabla de datos deban ser transformadas.

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   PCEC   R-squared:                       1.000
Model:                            OLS   Adj. R-squared:                  1.000
Method:                 Least Squares   F-statistic:                 5.912e+05
Date:                Tue, 12 Dec 2017   Prob (F-statistic):               0.00
Time:                        13:30:02   Log-Likelihood:                579.06
No. Observations:                 281   AIC:                            -1154.
Df Residuals:                     279   BIC:                            -1147.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -0.6799      0.011    -63.505      0.000      -0.701      -0.659
GDP            1.0280      0.001    768.865      0.000       1.025       1.031
==============================================================================
Omnibus:                       49.070   Durbin-Watson:                   0.070
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               78.376
Skew:                           1.009   Prob(JB):                     9.57e-18
Kurtosis:                       4.619   Cond. No.                         47.2
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

Cuadro 2: Estimación de una función de consumo, series en niveles

Los resultados de la regresión están en el cuadro 2. Como es de esperar en una regresión con series de tiempo con tendencia, el estadístico R^2 es prácticamente igual a uno, y el Durbin-Watson señala alta posibilidad de autocorrelación en los residuos. Aunque este documento no busca ser una guía de prácticas apropiadas de econometría, consideremos un último modelo en donde el crecimiento del consumo depende del crecimiento del ingreso:

$$\Delta \ln(c_t) = \beta_0 + \beta_1 \Delta \ln(y_t) + \epsilon_t$$

el cual estimamos en Python con

```
smf.ols('PCEC ~ GDP', log(usdata).diff()).fit().summary()
```

Los resultados de esta segunda estimación están en el cuadro 3. Observamos que ahora el R^2 ya no está cercano a uno, y que el estadístico Durbin-Watson está más cercano a 2.0, lo cual indicaría ausencia de autocorrelación.

Esta última línea de código, en la que estimamos el modelo en diferencias, resalta una de las razones del por qué el código escrito en Python es muy conciso: no siempre

es necesario almacenar los resultados de operaciones intermedias en variables, sino que podemos simplemente encadenar varias operaciones. En nuestro caso particular, hemos especificado un modelo (`.ols`), lo estimamos (`.fit`) y obtenemos una tabla resumen de los resultados (`.summary`). Similarmente, a los datos de la tabla `usdata` le hemos calculado sus logaritmo (`log`) y al resultado le hemos calculado su primera diferencia (`.diff`). Para apreciar mejor cuán conciso es este código, comparemos esa línea con el siguiente bloque, que toma 8 líneas de código para hacer las mismas operaciones:

```
usdata['lPCEC'] = log(usdata['PCEC'])
usdata['lGDP'] = log(usdata['GDP'])
usdata['dlPCEC'] = usdata['lPCEC'].diff()
usdata['dlGDP'] = usdata['lGDP'].diff()
model = smf.ols('dlPCEC ~ dlGDP', usdata)
results = model.fit()
table = results.summary()
print(table)
```

Con los resultados del cuadro 3 en mano, podríamos predecir que un aumento del crecimiento del PIB de un punto porcentual estaría asociado a un aumento del crecimiento del consumo de 0,618 puntos porcentuales. Ahora bien, dado que la muestra de datos abarca un período tan extenso (cerca de 70 años de observaciones trimestrales),

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   PCEC   R-squared:                       0.497
Model:                            OLS   Adj. R-squared:                  0.495
Method:                 Least Squares   F-statistic:                     274.8
Date:                Tue, 12 Dec 2017   Prob (F-statistic):           2.15e-43
Time:                        15:52:55   Log-Likelihood:                 998.76
No. Observations:                 280   AIC:                            -1994.
Df Residuals:                     278   BIC:                            -1986.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.0062      0.001      8.743      0.000       0.005       0.008
GDP            0.6180      0.037     16.578      0.000       0.545       0.691
==============================================================================
Omnibus:                      106.686   Durbin-Watson:                   2.536
Prob(Omnibus):                  0.000   Jarque-Bera (JB):             1182.717
Skew:                          -1.195   Prob(JB):                    1.50e-257
Kurtosis:                      12.781   Cond. No.                         91.0
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

Cuadro 3: Estimación de una función de consumo, series en diferencia

es razonable dudar de que los parámetros de este modelo sean constantes, por cuanto pudieron suceder varios cambios estructurales a través del tiempo. Una forma de evaluar esta posibilidad es estimar el modelo con una muestra de datos distinta cada vez. En particular, vamos a estimar este modelo con una ventana de 24 observaciones trimestrales, en cada paso moviendo la muestra un trimestre.

En este caso, como vamos a necesitar los datos de crecimiento muchas veces, es más eficiente calcularlos una sola vez y asignarlos a la variable `growth`. Con la indicación `[1:]` estamos desechando la primera observación, la cual perdemos cuando calculamos la primera diferencia (`.diff`). Además, usamos la propiedad `.shape` de la tabla para determinar cuantas observaciones `T` tenemos, y fijamos la ventana en `h`=24 observaciones:

```
growth = (100*log(usdata).diff())[1:]
T, nvar = growth.shape
h = 24
```

Para facilitar la siguiente parte, definimos la función `window_beta1`, la cual toma como único argumento el número de la última observación a incluir en la estimación, y da como resultado el valor del coeficiente estimado asociado al PIB

```
def window_beta1(k):
    return smf.ols('PCEC~GDP',growth[k-h:k]).fit().params['GDP']
```

Con esto, estamos listos para estimar el modelo muchas veces, agregando los resultados a la tabla `growth` como el "indicador" `beta1`. Graficando los resultados obtenemos la figura 8, donde se observa claramente que el efecto del crecimiento del ingreso sobre el crecimiento del consumo es muy inestable, y por tanto las predicciones hechas a partir de este sencillo modelo podría ser muy pobres.

```
growth.loc[h-1:,'beta1'] = [window_beta1(k) for k in range(h,T+1)]
growth[['beta1']].plot()
```

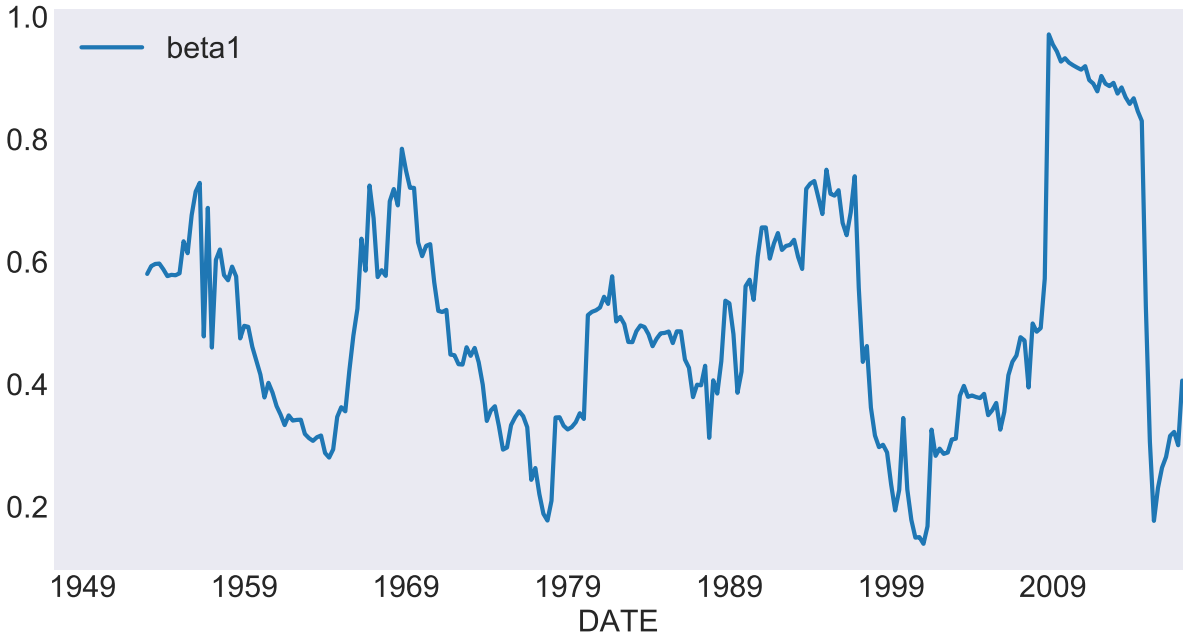

Figura 8: Efecto estimado del ingreso sobre el consumo, ventanas de 24 observaciones

Ejemplo 5: Documentos dinámicos

Para concluir esta nota, hago del conocimiento del lector que este documento mismo es un ejemplo de lo que se conoce como un "documento dinámico", en el sentido de que fue generado intercalando código de LaTeXcon código de Python. Esto permite que si en un futuro fuese necesario actualizar los ejemplos anteriores, en particular para contar con datos más recientes para los cuadros y las figuras, bastaría con ejecutar nuevamente el código que generó este documento. No será necesario usar un navegador de Internet para obtener los datos, ni copiar y pegar las figuras actualizadas en el documento.

Los documentos dinámicos son de gran utilidad, porque permiten ahorros significativos de tiempo en la actualización de informes periódicos. El lector interesado en aprender cómo crear uno de estos documentos necesitará conocer LaTeXy consultar la documentación de pythontex.

Referencias

Judd, Kenneth L. (1998). *Numerical Methods in Economics*. MIT Press. ISBN: 978-0-262-10071-7.

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

Press, William H., Saul A. Teukolsky y William T. Vetterling and Brian P. Flannery (2007). *Numerical Recipes: The Art of Scientific Computing*. 3ª ed. Cambridge University Press. ISBN: 978-0521880688.

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