Teoría de control en Python con SciPy (I): Conceptos básicos

October 10, 2013

Juan Luis Cano

control python python3 scipy scipy.signal

Introducción

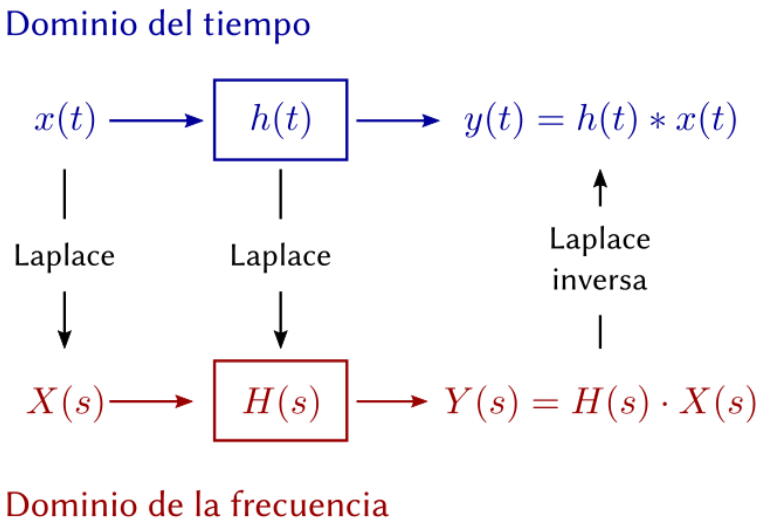
En esta serie de artículos vamos a estudiar **cómo podemos aplicar Python al estudio de la teoría de control**, en este caso utilizando SciPy. La teoría de control se centra en los **sistemas dinámicos** con entradas: sistemas físicos cuyo estado evoluciona con el tiempo en función de la información que reciben del exterior. Como puedes ver, esta definición es enormemente amplia: el control toca aspectos de la ingeniería y de las matemáticas, y tiene aplicaciones también en las ciencias sociales: psicología, sociología, finanzas...

1. **Conceptos básicos**
2. Control PID

En esta primera parte vamos a hacer una breve introducción matemática para centrar el tema y vamos a ver el manejo básico de sistemas LTI.

Cuando uno piensa en estudiar sistemas dinámicos con un ordenador, automáticamente se le viene a la cabeza **MATLAB**, y no sin motivo. Este programa tiene unas capacidades extraordinarias en este campo, y aunque nos duela decirlo *Python no está al mismo nivel*. Sin embargo, queremos mostrar en este artículo que Python tiene el potencial de ser una alternativa real a MATLAB, enseñando los fundamentos del análisis de sistemas dinámicos utilizando el paquete `scipy.signal`. Yo mismo he trabajado un poco en este paquete en los últimos meses, así que he tenido la oportunidad de ver cómo funciona y también de conocer sus carencias; algunas de mis contribuciones han visto la luz en la recién liberada versión 0.13 de SciPy,

pero aún queda mucho por mejorar.



Equivalencia entre los dominios del tiempo y de la frecuencia a través de la transformada de Laplace

Los ejemplos para este artículo los he sacado de [Sedra y Smith, 2004], un excelente libro de electrónica, y de [Messner et al. 2011], unos tutoriales para MATLAB y Simulink. Para la teoría, recomiendo el excelente [Gil y Rubio 2009], un libro editado por la Universidad de Navarra y disponible para visualización, impresión y copia para uso personal sin fines de lucro (¡gracias @Alex__S12!).

En esta entrada se han usado python 3.3.2, numpy 1.8.0, scipy 0.13.0 y matplotlib 1.3.0.

Sistemas lineales invariantes en el tiempo o LTI

Vamos a centrarnos en los **sistemas lineales e invariantes en el tiempo** (en inglés, sistemas *LTI*), que como su propio nombre indica tienen estas propiedades:

- **Linealidad**: el sistema cumple el principio de superposición.
- **Invarianza en el tiempo**: el comportamiento del sistema no varía con el tiempo: una misma entrada en dos instantes de tiempo diferentes siempre producirá la misma salida.

Concretamente, muchos de estos sistemas pueden modelarse como un sistema de ecuaciones diferenciales ordinarias lineales de coeficientes constantes. Restringiendo aún más nuestro objeto de estudio, para **sistemas de una sola entrada y una sola salida** (*SISO* en inglés) tendremos:

$$a_n \frac{d^n y}{dt^n} + a_{n-1} \frac{d^{n-1} y}{dt^{n-1}} + \cdots + y(t) = b_m \frac{d^m x}{dt^m} + b_{m-1} \frac{d^{m-1} x}{dt^{m-1}} + \cdots + x(t)$$

donde $y(t)$ es la respuesta del sistema y $x(t)$ es la entrada, conocida. Aplicando la **transformada de Laplace** a la ecuación tendremos:

$$Y(s) = \frac{b_m s^m + b_{m-1} s^{m-1} + \cdots + 1}{a_n s^n + a_{n-1} s^{n-1} + \cdots + 1} X(s) = H(s) X(s)$$

siendo $Y(s)$ y $X(s)$ las transformadas de laplace de $y(t)$ y $x(t)$ y $H(s)$ la **función de transferencia del sistema**. Vemos que la ecuación resultante es algebraica y por tanto muy fácil de resolver.

Precisamente la función de transferencia es una de las formas que tenemos de definir un sistema LTI en Python. Para ello simplemente tenemos que usar la función `signal.lti`, que acepta como argumento una tupla de dos, tres o cuatro elementos:

- Si damos **dos** elementos, deberán ser dos listas con los coeficientes de los polinomios del numerador y del denominador, **siguiendo la convención antigua de coeficientes decrecientes** (la convención contraria a la usada en nuestro artículo sobre ajuste e interpolación).
- Si damos **tres** elementos, deberán ser los ceros, polos y ganancia del sistema (ver más adelante).
- Si damos **cuatro** elementos, deberán ser las matrices de espacio de estado A, B, C y D (ver más adelante).

En la siguiente sección veremos cómo se utiliza.

Respuesta en frecuencia

Por ejemplo, representemos la respuesta en frecuencia de un **filtro pasabajos** sencillo:

$$H(s) = \frac{K}{(s/omega_0) + 1}$$

usando el diagrama de Bode. Este es el código:

```python
from scipy import signal
import matplotlib.pyplot as plt
```

```
K = 1
w0 = 1e3 # rad / s
sys1 = signal.lti([K], [1 / w0, 1]) # Creamos el sistema
w, mag, phase = signal.bode(sys1) # Diagrama de bode: frecuencias, magnitud y fase
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 6))
ax1.semilogx(w, mag) # Eje x logarítmico
ax2.semilogx(w, phase) # Eje x logarítmico
```

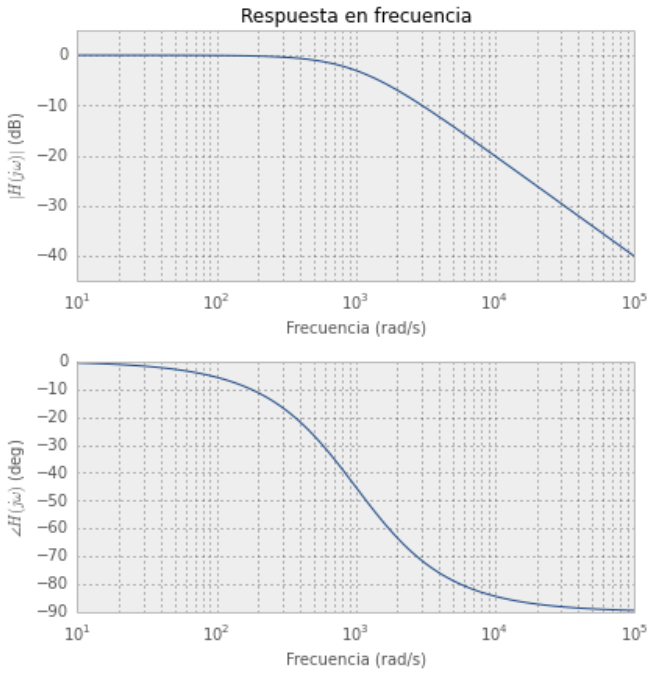


Diagrama de Bode de un filtro pasabajos.

¿Qué sucede si queremos acceder a las otras representaciones de nuestro sistema? Tenemos los atributos (`zeros`, `poles`, `gain`) y (`A`, `B`, `C`, `D`):

python

```
print(sys1.zeros, sys1.poles, sys1.gain) # [] [-1000.] 1000.0
print(sys1.A, sys1.B, sys1.C, sys1.D) # [[-1000.]] [[ 1.]] [[ 1000.]] [ 0.]
```

Otras dos herramientas que nos pueden ser útiles para analizar un sistema LTI son su **diagrama de Nyquist** y su **mapa de ceros-polos**. Para el primero podemos valernos de la función `signal.freqresp`, que devuelve la respuesta compleja en frecuencia de un sistema, y representar la parte imaginaria respecto a la parte real. Para el segundo, lo que hemos visto en el párrafo anterior:

python

```
sys2 = signal.lti([1, 2], [1, 6, 25]) # H(s) = (s + 2) / (s ** 2 + 6 * s + 25)
w, H = signal.freqresp(sys2)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
ax1.plot(H.real, H.imag)
ax1.plot(H.real, -H.imag)
ax2.plot(sys2.zeros.real, sys2.zeros.imag, 'o')
```

```
ax2.plot(sys2.poles.real, sys2.poles.imag, 'x')
```

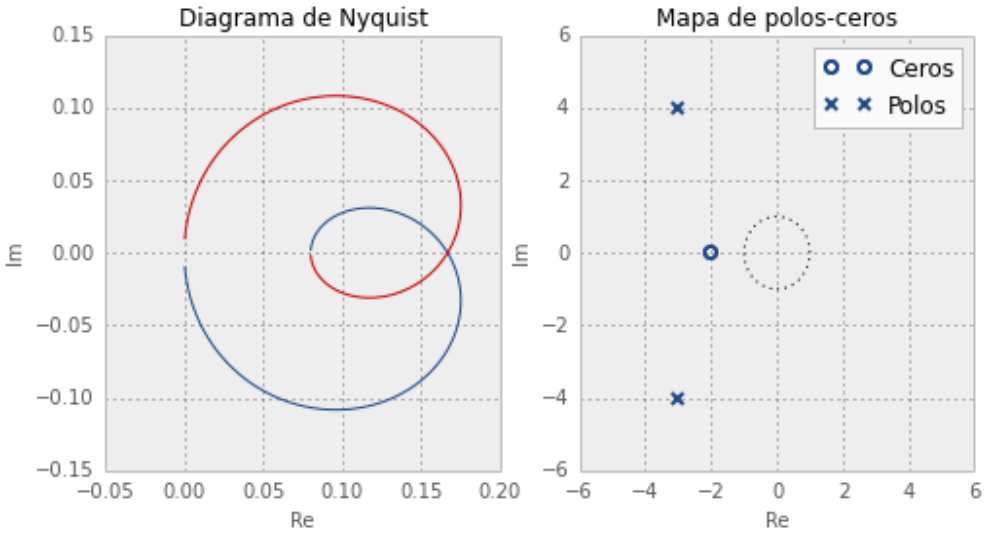


Diagrama de Nyquist y mapa de polos-ceros.

Respuesta temporal

Vamos a trabajar ahora sobre un ejemplo más elaborado: **el control de velocidad de crucero de un coche**. El objetivo es mantener constante la velocidad frente a perturbaciones externas, como por ejemplo cambios en la pendiente o ráfagas de viento. Si asumimos que las fuerzas de resistencia varían linealmente con la velocidad, tendremos:

$$m\frac{d^2x}{dt^2} = F - bv \Rightarrow m\dot{v} + bv = F$$

siendo m la masa del vehículo.

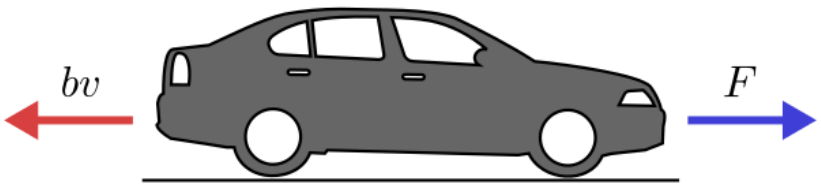


Diagrama de fuerzas sobre nuestro coche

La **entrada** de nuestro sistema será la **fuerza de tracción** aplicada, y la **salida** o variable que queremos controlar será la **velocidad**. La función de transferencia será:

$$H(s) = fracV(s)F(s) = \frac{1}{ms + b}$$

Ya podemos definir el sistema:

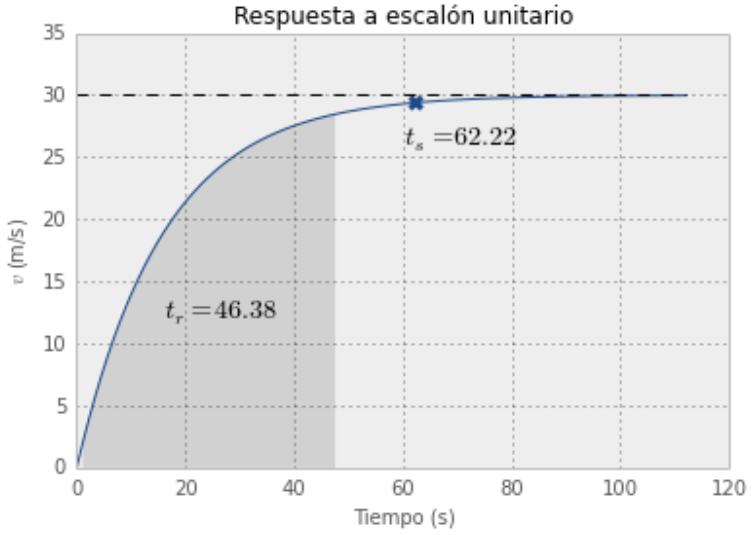
```python
m = 1200  # kg
b = 75  # Ns / m
sys_car = signal.lti(1, [m, b])
```

Con estos datos, necesitaría una fuerza de 2250 N para conseguir una velocidad de 30 m/s. Podemos ver cómo será la respuesta del sistema utilizando la función `signal.step2`.

Nota: La función ~~`signal.step`~~ tiene problemas y se escribió `signal.step2` como reemplazo. En un futuro estas dos funciones se fusionarán bajo el mismo nombre, pero mientras tanto se recomienda usar `signal.step2`.

Esta función **calcula la respuesta a una entrada escalón unidad**. Como el sistema es lineal, podemos multiplicar la salida por el valor de la fuerza y este resultado será igual a la respuesta a un escalón de altura ese valor:

```python
t, y = signal.step2(sys_car) # Respuesta a escalón unitario
plt.plot(t, 2250 * y) # Equivalente a una entrada de altura 2250
```



Respuesta del sistema a una entrada escalón.

Vemos que la velocidad va aumentando, al principio rápidamente y luego más despacio (producto de las fuerzas de resistencia), hasta llegar a un valor límite, que es 30 m/s como habíamos dicho al principio. En la gráfica hemos

incluido también:

- El **tiempo de subida** (*rising time*), definido como el tiempo necesario para pasar de un 5 % a un 95 % de la solución estacionaria, y
- el **tiempo de establecimiento** (*settling time*), definido como el tiempo necesario para que la respuesta se mantenga dentro de un margen del 2 % de la solución estacionaria.

La conclusión que podemos extraer de este gráfico es que **para pasar de 0 a 100 km/h, nuestro coche necesita casi un minuto**. ¡Fatal! ¿Cómo arreglamos esto? Aquí entra la belleza de la teoría de control, pero lo vamos a dejar para la segunda parte

Este artículo ha sido muy difícil de escribir, y el siguiente probablemente también lo será. Por eso os pedimos que **nos contéis en los comentarios** qué os ha parecido, qué partes se podrían mejorar, y puesto que hemos reconocido la debilidad de SciPy en este campo, qué cosas positivas podría tomar de MATLAB.

¡Un saludo!

Referencias

1. SEDRA, Adel S.; SMITH, Kenneth C. *Microelectronic circuits*. Oxford University Press, 2004.
2. MESSNER, Bill et al. *Control Tutorials for MATLAB and Simulink* [en línea]. 2011. Disponible en web: <http://ctms.engin.umich.edu/>. [Consulta: 10 de octubre de 2013]
3. GIL, Jorge Juan; RUBIO, Ángel. *Fundamentos de Control Automático de Sistemas Continuos y Muestreados*. Universidad de Navarra, 2009.

Juan Luis Cano

Estudiante de ingeniería aeronáutica y con pasión por la programación y el software libre. Obsesionado con mejorar los pequeños detalles y con ganas de cambiar el mundo. Divulgando Python en español a través de Pybonacci y la asociación Python España.

More Posts - Website

Follow Me:

← Hoja de ruta para SciPy 1.0

Teoría de control en Python con SciPy (II): Control PID →

17 thoughts on “Teoría de control en Python con SciPy (I): Conceptos básicos”

Schcriher

October 10, 2013 at 8:24 PM

¡Hola Juan! Excelente artículo! me ha gustado mucho y estaré (aún mas) pendiente de Pybonacci por cualquier otro artículo que toque la teoría de control, por un lado porque me gusta y por otro por que me puede resultar de mucha utilidad en mi futuro profesional. Hace como dos años que cursé la materia de Control de Procesos en la facultad (de cuestionable calidad) y sumado a que no lo he vuelto a usar es poco y nada lo que puedo aportar aquí. Nosotros los usábamos para definir el sistema de control (P+I+D) a usar en algún proceso (tanques agitados, con y sin reacción química, con y sin calefacción/refrigeración, y cosas así). Saludos!!!

Reply

Juanlu001

October 10, 2013 at 8:31 PM

¡Hola! Muchas gracias yo he empezado a estudiarla este año y me está encantando (aunque como soy un novato puedo haber cometido algún error en el artículo). Precisamente la segunda parte va a tratar sobre control PID, pero me parecía que el artículo estaba ya muy cargado así que he decidido dividirlo en dos partes.

¡Un saludo!

Reply

Pingback: Teoría de control en Python con SciPy (I...

geopelia

October 11, 2013 at 2:56 PM

Excelente articulo, aunque la verdad excede por muchos mis conocimientos xDDDD

Reply

Juanlu001

October 11, 2013 at 3:07 PM

Admito que este es un poco duro, pero cuando estudias un poco ves lo interesante que es ¡Gracias por el comentario de todos modos!

Reply

cachorrocadi

October 12, 2013 at 3:34 PM

Me ha gustado mucho el post, resulta fácil de leer y de comprender.
La potencia de Matlab para lo que es Control Automático creo que reside principalmente en Simulink.
En la bibliografía, aunque sea a título informativo, echo de menos “Ingeniería de Control Moderna” de K. Ogata (Ed. Pearson)

Reply

Juanlu001

October 12, 2013 at 5:14 PM

Hm, eres la segunda persona que me recomienda el Ogata. El lunes lo voy a pedir y le voy a echar un ojo sin duda. ¡Gracias!

En cuanto a Simulink, supongo que es cuestión de tiempo que alguien

escriba algo parecido pero aún no se ha hecho. De todas formas `signal` es el paquete más antiguo de SciPy y el que necesita más mejoras, así que cuando lleguen tal vez veremos algo.

¡Un saludo!

Reply

Pingback: Teoría de control en Python con SciPy (I...

Pingback: #Developers : Teoría de Control en #Pyth...

Alex

October 26, 2013 at 7:40 PM

Estupendo el artículo!
Waiting for the second part!

Reply

Juanlu001

October 27, 2013 at 1:35 PM

¡Gracias Álex! Caerá más pronto que tarde

Reply

Álex S (@Alex__S12)

February 8, 2014 at 8:01 PM

Estaba yo probando cosillas y me da error al intentar hacer un derivador de la siguietne manera:
derivador = signal.lti([1,0], [1])

ValueError: Improper transfer function. `num` is longer than `den`.

No entiendo, por qué no deja hacer esto. ¿sugerencias?

Reply

Juanlu001

February 8, 2014 at 10:39 PM

Con esos coeficientes estás implementando esta función:

$latex displaystyle T(s) = frac{s}{1}$

No sé si es esto lo que querías, pero esta función de transferencia es impropia porque tiene más ceros que polos. Prueba con esta:

$latex displaystyle T(s) = frac{s}{s + 1}$

que en SciPy sería `lti([1, 0], [1, 1])`
(http://en.wikipedia.org/wiki/Passive_differentiator_circuit#Transfer_function)

Pingback: Teoría de control en Python con SciPy (II): Control PID | Pybonacci

Pingback: Meme Python de Año Nuevo 2014 | Pybonacci

Pingback: Análisis de nuestro primer año #2014pythonmeme - CAChemE

Pingback: Control PID con Python e interfaz web

Leave a Reply

Enter your comment here...