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

November 6, 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...

En esta segunda parte, una vez vistos algunos conceptos básicos en la primera, vamos a retomar el problema del control de crucero del coche exactamente donde lo dejamos:

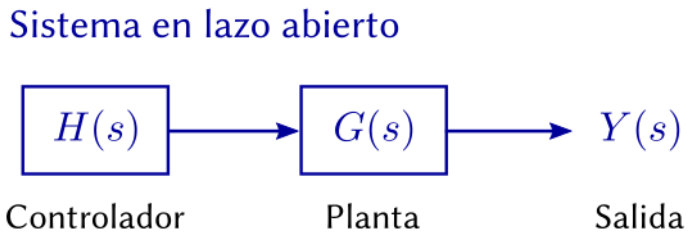
> *La conclusión que podemos extraer de este gráfico es que **para pasar de 0 a 100 km/h, nuestro coche necesita casi un minuto**. ¡Fatal! ¿Cómo arreglamos esto? Aquí entra la belleza de la teoría de control, pero lo vamos a dejar para la segunda parte* 😉

A las referencias que ya recomendé en la primera parte voy a añadir [Ogata, 2010], un excelente libro sobre ingeniería de control traducido al español por profesores de España y Argentina. De ahí he estudiado la parte matemática del control PID y me he inspirado para las figuras. Cualquier aspecto teórico se puede consultar en este libro.

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

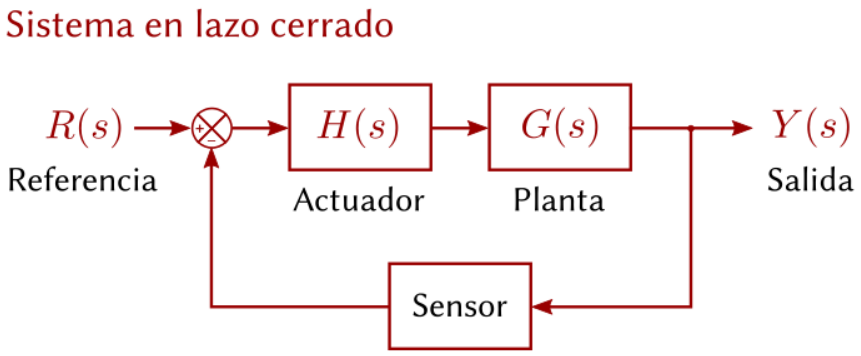
Concepto de realimentación

Si recordamos el modelo de nuestro coche, teníamos un sistema con una entrada (la fuerza de tracción que genera el motor) y una salida o variable a controlar (la velocidad del coche). Este tipo de sistemas se denominan **en lazo abierto** y se pueden esquematizar con un diagrama de bloques de este estilo:



En este caso la **planta** sería el motor y el **controlador** un sistema que consiguiese esa tracción constante que consideramos en la primera parte. Este tipo de sistemas son poco útiles porque no podemos disponer de la información de la salida para controlar la entrada. No les prestaremos más atención.

Si queremos tener en cuenta las posibles perturbaciones de la salida deberíamos medirla continuamente para comprobar que cumple con nuestros requisitos. A esto se le denomina **control en lazo cerrado** y se puede esquematizar de la siguiente manera:



Ya vemos que se van complicando un poco las cosas. En este caso, la entrada de la planta la proporciona el **actuador**, y la entrada del actuador es **la diferencia entre la señal de referencia y la salida**. Esta diferencia se denomina señal **error** por razones obvias. Con este cambio de esquema y de filosofía tenemos en cada instante información sobre la salida del sistema y podemos por tanto ajustar el control del mismo. Veamos un método para llevar a cabo este control.

Comprobábamos con nuestro ejemplo del coche que el tiempo que necesitaba para pasar de 0 a 100 km/h era muy grande. Este tiempo se define como **tiempo de subida**, t_r (*rising time* en inglés): el tiempo requerido para que la respuesta pase del 0 % al 100 % de su valor final. Otros márgenes utilizados son 5-95 % y 10-90 %. Más adelante, cuando empecemos a introducir oscilaciones, veremos otras definiciones similares.

Control PID

El control PID combina una acción **proporcional**, una **integral** y otra **derivativa** sobre la señal de error para dar la entrada de la planta. La ecuación de un controlador de este tipo es:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt}$$

y su función de transferencia, por tanto:

$$H(s) = \frac{U(s)}{E(s)} = K_p + \frac{K_i}{s} + K_d s$$

Podemos escribir también

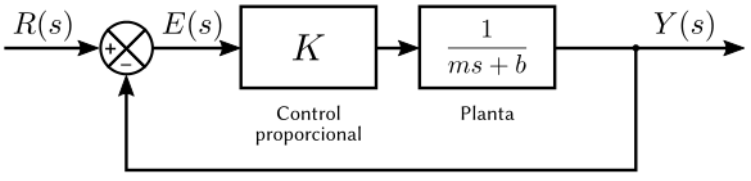
$$K_p + \frac{K_i}{s} + K_d s = K_p(1 + \frac{1}{T_i s} + T_d s)$$

donde T_i se denomina *tiempo integral* y T_d es el *tiempo derivativo*.

Cada uno de estos términos contribuye de una forma distinta al sistema de control, y la correcta ponderación de cada una de estas contribuciones es lo que tendremos que buscar para conseguir una solución satisfactoria. Este proceso se llama **sintonía de controladores PID** (*PID tuning*). Vamos a estudiarlos uno por uno.

Proporcional

Lo primero que se nos puede ocurrir es hacer que la señal de control $u(t)$ (la salida del actuador y la entrada de la planta) sea **proporcional** al error entre la salida y el valor de referencia. Recuperando la función de transferencia de nuestro control de crucero, tendríamos un esquema como el siguiente:



Sistema de control proporcional

Nótese que la naturaleza de nuestro sistema ha cambiado completamente: antes controlábamos directamente la fuerza del motor, ahora indicamos una velocidad de referencia y el sistema ajusta dicha fuerza. La nueva función de transferencia será:

$$\frac{Y(s)}{R(s)} = \frac{Y(s)}{Y(s) + E(s)} = \cdots = \frac{K}{ms + b + K}$$

SciPy no proporciona un modo directo de operar con bloques, de modo que vamos a escribir nuestras propias funciones a tal efecto. Para ello, operaremos las funciones de transferencia de los sistemas como fracciones, extrayendo por separado el numerador y el denominador y utilizando las funciones de NumPy `np.polymul` y `np.polyadd`. Este será el código:

python

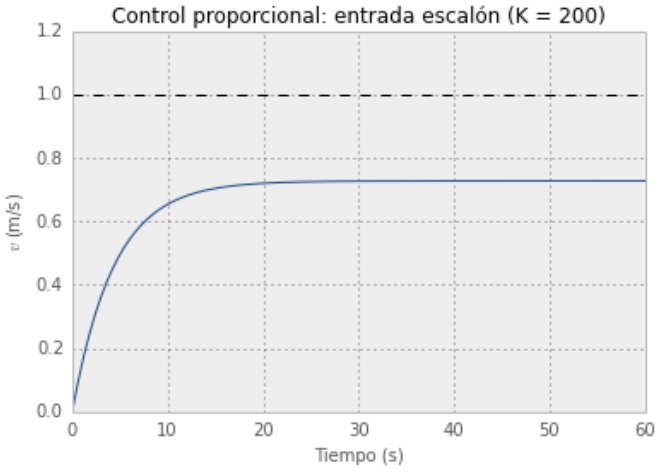
```
def series(sys1, sys2):
    """Series connection of two systems.
    """
    if not isinstance(sys1, signal.lti):
        sys1 = signal.lti(*sys1)
    if not isinstance(sys2, signal.lti):
        sys2 = signal.lti(*sys2)
    num = np.polymul(sys1.num, sys2.num)
    den = np.polymul(sys1.den, sys2.den)
    sys = signal.lti(num, den)
    return sys
def feedback(plant, sensor=None):
    """Negative feedback connection of plant and sensor.
    If sensor is None, then it is assumed to be 1.
    """
    if not isinstance(plant, signal.lti):
        plant = signal.lti(*plant)
    if sensor is None:
        sensor = signal.lti([1], [1])
    elif not isinstance(sensor, signal.lti):
        sensor = signal.lti(*sensor)
    num = np.polymul(plant.num, sensor.den)
    den = np.polyadd(np.polymul(plant.den, sensor.den),
                     np.polymul(plant.num, sensor.num))
    sys = signal.lti(num, den)
    return sys
```

Echemos ahora un vistazo a la respuesta unitaria del sistema, seleccionando por ejemplo un valor de 200 para la ganancia, y veremos que tenemos ciertos problemas:

python

```
K = 200
```

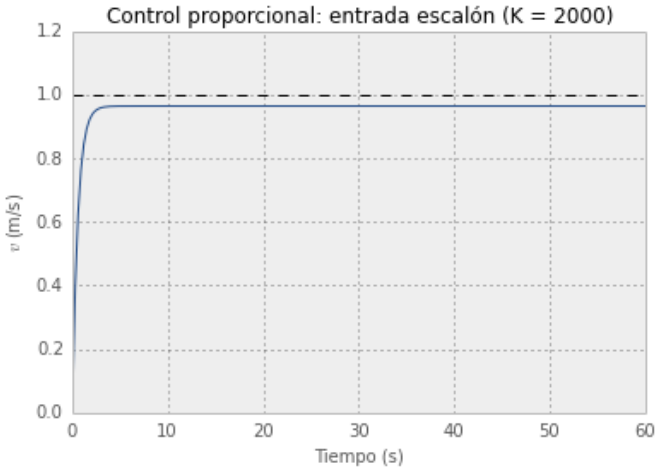
```
# Sistema controlador-planta
sys_pc = series(([K], [1]), sys_car)
# Sistema realimentado
sys_prop = feedback(sys_pc)
# Respuesta a entrada escalón
t = np.linspace(0, 60, num=200)
t, y = signal.step2(sys_prop, T=t)
plt.plot(t, y)
plt.plot([0, t[-1]], [1] * 2, 'k--')
```



De acuerdo, ahora el tiempo de subida es de unos 20 segundos (en vez del minuto de antes) pero ¡la salida **no llega al nivel que queremos**! Esto es así porque los controladores proporcionales introducen un cierto **error en estado estacionario** e_{ss}. Para nuestro sistema, el valor de este error será:

$$e_{ss} = \lim_{t \to \infty} e(t) = \frac{1}{1 + K/b}$$

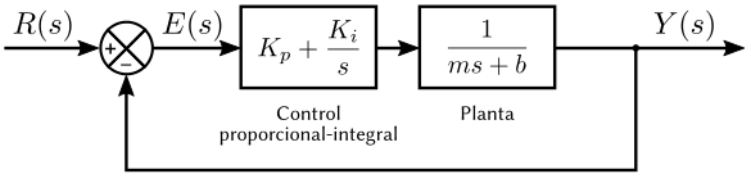
Parece lógico pensar que entonces debemos aumentar la K para reducir el error. Si hacemos esto, tendríamos la siguiente respuesta:



Ahora el tiempo de subida es de menos de 5 segundos y el error de estado estacionario es $e_{ss} \simeq 3.6$. ¿Hemos conseguido ya lo que queríamos? Sobre el papel sí, pero ¿te has fijado en la pendiente de la curva en el punto inicial? **No podemos** aumentar indefinidamente la ganancia proporcional porque eso implica **aumentar indefinidamente la fuerza** del motor. Tenemos que buscar otros métodos.

Integral

La desventaja del control proporcional es que, si la señal de error tiende a cero, la señal de control también. Con el término integral podemos añadir una contribución que depende del área encerrada bajo la curva de la señal error, y por tanto **eliminamos el error en estado estacionario**. Ahora nuestro esquema quedaría de la siguiente forma:



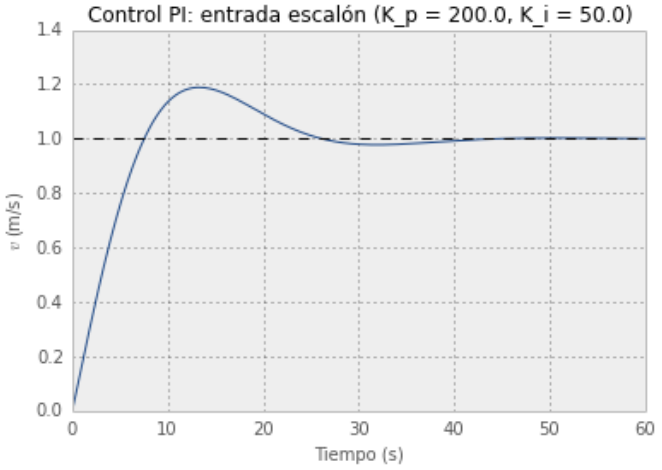
Sistema de control proporcional-integral (PI)

Y la función de transferencia será:

$$\frac{Y(s)}{R(s)} = \frac{K_p s + K_i}{ms^2 + (b + K_p)s + K_i}$$

Exacto: hemos convertido el sistema **en uno de segundo orden**. Esto tendrá algunos efectos nuevos, como se puede ver en la gráfica de la respuesta a escalón unitario del sistema para $K_p = 200$ y $K_i = 50$:

```python
K_p = 200.0
K_i = 50.0
# Sistema controlador-planta
sys_pc = series(([K_p, K_i], [1, 0]), sys_car)
# Sistema realimentado
sys_prop = feedback(sys_pc)
# Respuesta a entrada escalón
t = np.linspace(0, 60, num=200)
t, y = signal.step2(sys_prop, T=t)
plt.plot(t, y)
plt.plot([0, t[-1]], [1] * 2, 'k--')
```



Tenemos un tiempo de subida menor a 10 segundos, pero por contra **hemos introducido una oscilación en el sistema**. En algunos problemas puede ser inadmisible, pero en este caso nos lo podemos permitir. Lo único que tenemos que hacer es controlar la oscilación; para ello tenemos otras dos magnitudes interesantes:

- El **tiempo de pico** t_p (*peak time*) definido como el tiempo requerido para que la respuesta llegue al primer máximo.
- La **sobreelongación máxima** M_p (*overshoot*) definida como el valor máximo de la respuesta, por encima del resultado estacionario.

En general, querremos ajustar nuestro sistema de control para que tenga:

- Tiempo de subida pequeño,
- sobreelongación pequeña, y
- ausencia de error en estado estacionario.

Una cuestión que hay que tener muy en cuenta es que **no podemos optimizar estos tres requisitos a la vez**. Si reducimos el tiempo de subida, la sobreelongación máxima aumentará, y viceversa [Ogata, pp. 171].

Podemos marcar unos requisitos para nuestro caso concreto:

- Tiempo de subida < 5 s
- Máxima sobreelongación < 10 %
- Error en estado estacionario < 2 %

Vamos a escribir un par de pequeñas funciones que nos calculen estas magnitudes:

python

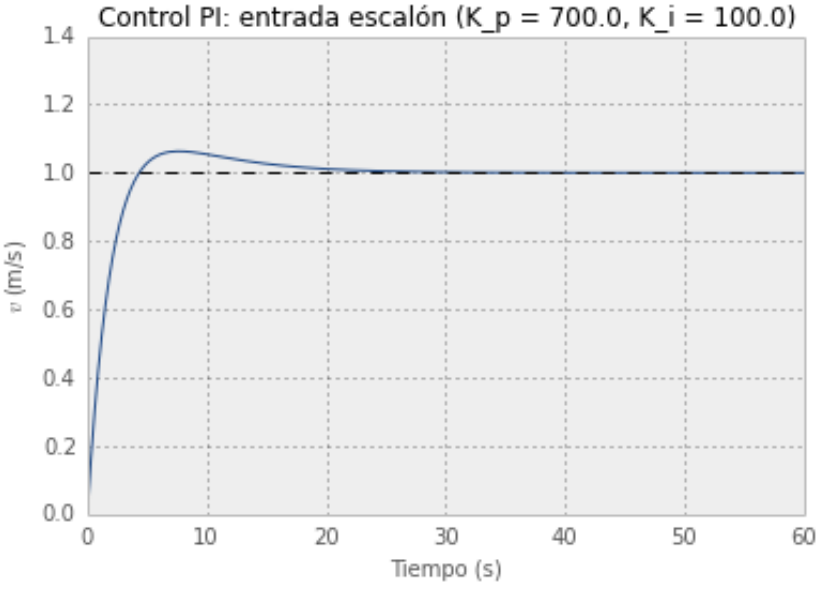
```
def tr(t, y, ys=None, margins=(0.0, 1.0)):
    """Rise time.
    Other possible margins: (0.05, 0.95), (0.1, 0.9). If no ys is given,
    then last value of y is assumed as stationary.
    """
    if ys is None:
        ys = y[-1]
    # Values between margins[0] * ys and margins[1] * ys
    mask = (y > margins[0] * ys) &amp;amp; (y < margins[1] * ys)
    # If response oscillates, only interested in limits of first region
    idx_change = np.nonzero(np.diff(mask))[0]
    # Initial and final indexes
    idx = idx_change[0], idx_change[1]
    # Time difference
    return t[idx[1]] - t[idx[0]]
def Ms(y, ys=None):
    """Maximum overshoot.
    Other possible margins: (0.05, 0.95), (0.1, 0.9). If no ys is given,
    then last value of y is assumed as stationary.
    """
    if ys is None:
```

```
        ys = y[-1]
    ymax = np.max(y)
    Ms = (ymax - ys) / ys
    return Ms
```

Se comprueba que para $K_p = 700$ y $K_i = 100$ cumplimos:

python

```
K_p = 700.0
K_i = 100.0
# Sistema controlador-planta
sys_pc = series(([K_p, K_i], [1, 0]), sys_car)
# Sistema realimentado
sys_prop = feedback(sys_pc)
# Respuesta a entrada escalón
t = np.linspace(0, 60, num=200)
t, y = signal.step2(sys_prop, T=t)
print("Tiempo de subida: {:.2f} s".format(tr(t, y)))
print("Máxima sobreelongación: {:.1f} %".format(Ms(y) * 100))
# Tiempo de subida: 4.22 s
# Máxima sobreelongación: 6.3 %
```



¡Genial!

Derivativo

Ya hemos cumplido nuestros requisitos así que no tendríamos porqué añadir un término derivativo, y por tanto en este artículo no lo vamos a estudiar en detalle. Este tendrá como efecto **suavizar** la respuesta, aunque tiene una contrapartida importante: en presencia de ruido

puede desestabilizar el sistema. El artículo ya es demasiado largo y el análisis es idéntico al efectuado anteriormente, así que se deja como ejercicio al lector

Otros métodos

El control PID es ampliamente utilizado, especialmente para sistemas de los que desconocemos su funcionamiento interno («cajas negras») y es relativamente simple. Sin embargo, sintonizar controladores PID puede ser más complicado de lo que parece y en general no se consigue un control óptimo. Existen otros métodos como el del lugar de las raíces o el estudio de la respuesta en frecuencia que son también útiles para diseñar sistemas de control. Pero de ellos ya hablaremos en entregas sucesivas, si el público lo reclama

Espero que nos hagáis llegar vuestros comentarios y sugerencias a través del formulario más abajo y que os haya gustado el artículo.

¡Un saludo!

Apéndice: Estado del arte

Aquí hemos visto que con Python hemos tenido que trabajar un poco más de lo que tendríamos que haber trabajado con MATLAB. Por un lado, hemos tenido que escribir nuestras propias funciones para operar con bloques, calcular tiempos de subida y sobreelongaciones... Y por otro lado, se echa en falta una interfaz gráfica con la que trabajar de una manera mucho más intuitiva (los diagramas del artículo están hechos con Inkscape, y ha sido un poco laborioso). En este área SciPy tiene aún mucho que mejorar, y de hecho ya hay algunas ideas.

Implementar operaciones con sistemas LTI o crear una interfaz gráfica son tareas relativamente sencillas, pero requieren que alguien se ponga con ellas. Si alguno de nuestros lectores quiere dar un paso adelante, le animamos a que contacte con nosotros, haga un fork de SciPy y se ponga a trabajar. El inglés no es un problema: podemos

guiarle con el proceso en nuestro propio fork, como ya anunciamos hace unos meses.

Como el título del artículo deja bien claro que íbamos a usar SciPy, he descartado la biblioteca python-control. En muchos sentidos está más avanzada que el paquete signal de SciPy **y proporciona compatibilidad con MATLAB**, pero no me queda claro si la están manteniendo activamente o no, y por tanto no me he decidido a utilizarla. Por si alguien tiene curiosidad, aquí hay un ejemplo analizando el sistema de despegue y aterrizaje de un avión.

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.
4. OGATA, Katsuhiko. *Ingeniería de control moderna*. 5ª ed. Pearson, 2010.

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:

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

Operador dos puntos (:) de MATLAB ¿en Python? →

13 thoughts on “Teoría de control en Python con SciPy (II): Control PID”

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

Schcriher

November 11, 2013 at 3:08 AM

¡¡¡Que buen artículo!!! Muchas gracias por tomarte el tiempo de hacerlo y compartirlo. Por ahora no creo poder aportar nada, pero si leerlo! Saludos!

Reply

Juanlu001

November 11, 2013 at 12:08 PM

¡Muchas gracias Cristian! Si un día pones esto en práctica no tardes en mandarnos los resultados 😀 ¡Un saludo!

Reply

Aberto

November 11, 2013 at 2:32 PM

¡Muy interesante! Muy bien redactado y explicado. Y todos sabemos

lo que cuesta hacer cualquier artículo similar,así que gracias por emplear tu tiempo en ello.Y animarte por supuesto a que sigas con tu tarea.¡Saludos!

Reply

Juanlu001

November 11, 2013 at 7:24 PM

Me alegro de que te haya gustado Alberto, se agradece el comentario porque sí que costó escribir el artículo Sobre todo, y aunque parezca una tontería, ¡preparar las figuras!

¡Un saludo!

Reply

Alex

November 11, 2013 at 6:31 PM

Maravilloso!! Claro que al público le interesa, es muy didáctico, es control y es Python! MOOLA

Reply

Juanlu001

November 11, 2013 at 7:23 PM

Jaja ¡muchas gracias Álex! Me alegro de que te haya gustado, nos vemos luego

Reply

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

Pingback: Control PID con Python e interfaz web

thesuisse

April 24, 2014 at 4:27 PM

Hola Pybonacci muy buena esta entrada, con respecto a las rutinas de control más avanzadas que no son incluidas en scipy tengo entendido que era por un tema de licencia de el package(Fortran) más popular y grande que se llama SLICOT(http://slicot.org/). Pero creo que ahora el licenciamiento es gpl3 y se podrían incluir. Ya existe un wrapper de esa libreria(https://github.com/avventi/Slycot)

Reply

Juanlu001

April 24, 2014 at 4:33 PM

¡Muchas gracias por la información! Tienes razón, con esa licencia no se pueden incluir en SciPy. Me alegra ver que hay un wrapper en Python de todos modos. ¡Un saludo!

Reply

thesuisse

April 24, 2014 at 5:51 PM

Es porque gpl3 no permite derivados comerciales verdad?¿.
Te dejo un articulo que leí el otro día muy interesante en el que estoy 100% de acuerdo

http://www.soa-world.de/echelon/2014/02/stop-teaching-matlab.html

Juanlu001

April 26, 2014 at 10:04 AM

No solo por eso, es porque si incluyes código GPL, entonces todo tu código debe ser GPL y a menos que quieras cambiar tu licencia esto no es posible. Yo también estoy de acuerdo con el artículo sin duda 😉 ¡Saludos!

Leave a Reply

Enter your comment here...