Stochastic Systems --- Discrete Time Systems

Ejercicios --- Conjunto final

13/12/2021

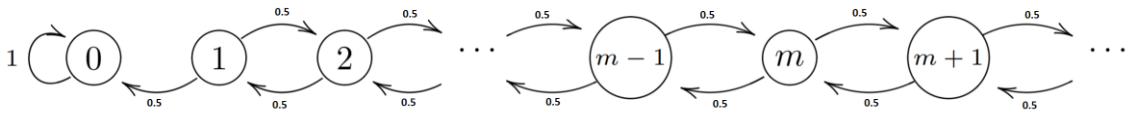
Adrián Rubio Pintado

PARTE I

1 . En los apuntes hay el ejemplo del gambler's ruin como cadena de Markov.

a) **Hacer un gráfico del número medio de jugadas que el jugador puede hacer antes de arruinarse en función del dinero inicial. En cada jugada se juega 1 Euro, y el juego es ecuo (el jugador tiene una probabilidad 1/2 de ganar).**
b) **Estimar media y varianza (usar unas 20 ejecuciones... deberían ser mas que suficientes) considerando un dinero inicial de 1, . . . , 50 Euros. Indicar media y varianza. ¿cómo varían la media y la varianza cuando aumenta el dinero inicial? Dar una estimación de la función T(e) que da el tiempo medio necesario para arruinarse en función de la cantidad inicial de dinero, e. Según esta función, ¿cuánto tarda el jugador en arruinarse si empiezas a jugar con 200 Euros?**

a) La cadena de Markov con el número medio de jugadas que puede hacer antes del arruinarse el jugador es el siguiente:



Dado $k_n = E[H^A]$, es decir, el tiempo medio(número de jugadas medio) para alcanzar el estado 0 partiendo desde el estado n.

Las probabilidades que buscamos es la solución mínima no negativa al sistema recurrente:

$$k_0 = 0$$

$$k_m = 1 + 0.5\, k_{m-1} + 0.5 k_{m+1} \quad \text{con } m = 1,2,3,\ldots$$

La resolución de dicha ecuación en recurrencia nos lleva a una solución de la forma:

$$K_m = -m^2 + Am + m$$

Para todo m positivo. Es por ello que la única conclusión que podemos sacar es que no arruinaremos en $k_m = \infty$. Sabemos que nos arruinaremos, en un tiempo infinito, sin importar la cantidad de dinero que apostemos o lleguemos a ganar.

b) Simulación del tiempo medio y varianza del tiempo medio en arruinarse.

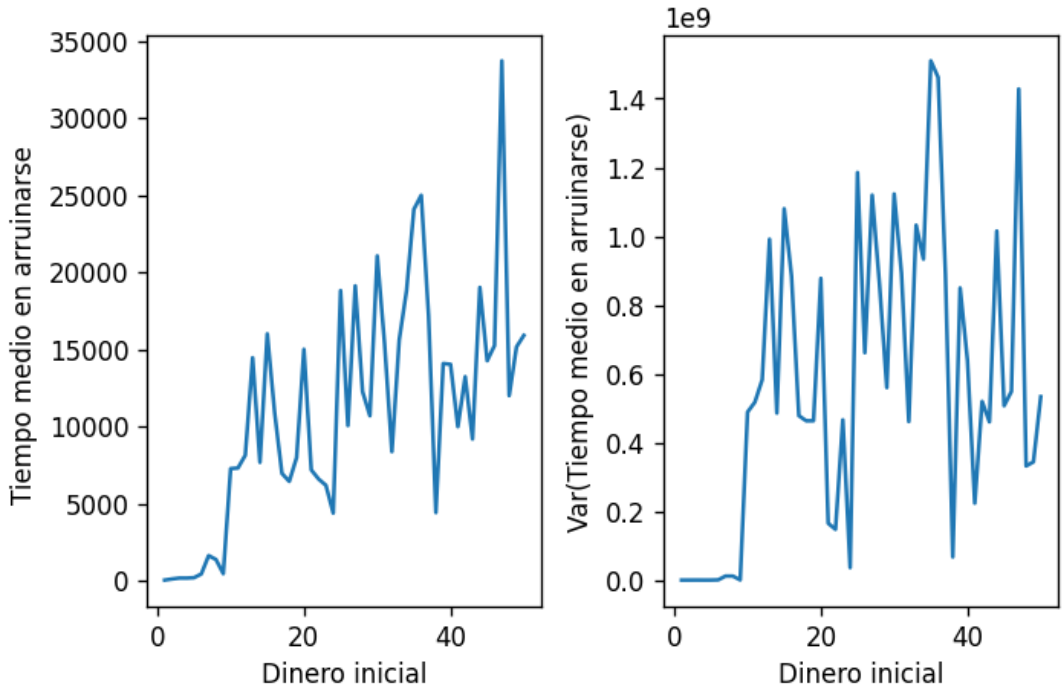
Dado que es imposible simular hasta un tiempo infinito, tomamos un suficientemente grande como infinito. Si el jugador no se ha arruinado llegado ese tiempo, debemos interpretar esa medida como infinito. El valor elegido es t_infinito = 10^5.

Escribimos un programa Python que simule esta cadena de Markov con infinitos estados posibles, con un estado absorbente m=0, tal que:

```
[23] from scipy.stats import uniform
     import numpy as np
     import matplotlib.pyplot as plt
     %matplotlib inline

     #Ruina del Jugador
     class Jugador:

       def __init__(self,m_inicial, p_win):
         #probabilidad de ganar p_win = p
         #progabilidad de perder: q = 1-p
         self.p = p_win
         self.m_inicial = m_inicial #dinero inicial
         self.m = m_inicial #dinero acutal

       def run(self, max_steps):

         if(self.m_inicial == 0 ):
           return 0

         for t in range(1,max_steps+1):
           u=uniform.rvs()
           if(u <= self.p):#win
             self.m = self.m +1
           else:#lose
             self.m = self.m -1
           #Check if player is ruined
           if(self.m ==0):
             return t

         return max_steps #player is not ruined(t = inf)

       def run_n_simulations(self,max_steps, n_simul):
         tiempos_ruina = []
         for i in range(n_simul):
           self.m = self.m_inicial #dinero acutal
           tiempos_ruina.append(self.run(max_steps))
         return tiempos_ruina
```

```python
#Tiempo medio en arruinarse según el dinero inicial
p_win = 0.5
n_simul = 20
dineros_iniciales = [i for i in range(1,51)]
max_steps = 10**5
medias = []
varianzas = []
for m in dineros_iniciales:
  print('Simul. para dinero inicial: ', m)
  jugador = Jugador(m,p_win)
  tiempos = jugador.run_n_simulations(max_steps, n_simul)
  media = np.mean(tiempos)
  var = np.var(tiempos)
  medias.append(media)
  varianzas.append(var)
```

Pintamos los resultados de la simulación para un dinero inicial [1,50], obteniendo los siguientes resultados para el tiempo medio en arruinarse y la varianza del mismo:



Tener una esperanza teórica de infinito, no introduce mucha varianza, lo cual podemos ver en la gráfica de la derecha. Tener una varianza tan inestable, nos deja un número medio de tiempo medio en arruinarse con muchos picos. Lo que sí que podemos ver es que el tiempo medio en arruinarse crece según el dinero inicial, es decir, cuanto más dinero inicial tengamos, más tardamos en arruinarnos de media. Además, cuanto más dinero inicial tengamos, la varianza fluctúa con picos más grandes.

Para poder dar una estimación T(e) como el tiempo medio necesario en arruinarse en función de la cantidad inicial de dinero e, entrenamos un modelo de regresión lineal con sklearn para los tiempos medios obtenidos en la simulación:

```
from sklearn import  linear_model


# Create linear regression object
regr = linear_model.LinearRegression()

x = np.array(dineros_iniciales).reshape(-1, 1)
y = np.array(medias).reshape(-1, 1)
# Train the model using the training sets
regr.fit(x,y)
# The coefficients
print("Coefficients: \n", regr.coef_)
print("Intercept: \n", regr.intercept_)

#predecimos para un dinero inicial de 200
t_medio_arruinarse_con_200e = regr.predict(np.array(200).reshape(1, -1))
print('t_medio_arruinarse_con_200e: ',t_medio_arruinarse_con_200e)
```

```
Coefficients:
 [[353.86092197]]
Intercept:
 [1844.9644898]
t_medio_arruinarse_con_200e:  [[72617.14888355]]
```

Obtenemos una estimación por regresión lineal de T(e) tal que

T(e)_estimador = `1844.9644898 + e * 353.86092197`

Si queremos predecir el tiempo medio en arruinarse empezando una cantidad de dinero e = 200, obtenemos un tiempo medio aproximado de **72617** pasos.

Es decir, el jugador tardaría una media de **72617** tiradas en arruinarse.

PARTE III

a. Crear cuatro sistemas dinámicos del tipo

$$x_{t+1} = \mathbf{A}x_t + \mathbf{B}u_t + w_t z_t \quad = \mathbf{C}x_t + v_t \tag{1}$$

con $x_t \in \mathbb{R}^4$, $u_t \in \mathbb{R}$, $z_t \in \mathbb{R}^2$, y

$$\begin{aligned} \mathbb{B} &= [1, 1, 1, 1]' \\ \mathbb{C} &= \begin{bmatrix} 1, 0, 0, 0 \\ 0, 1, 0, 0 \end{bmatrix} \end{aligned} \tag{2}$$

Los sistema se distinguen por la matrix $\mathbf{A}$, que es, por cada sistema una de las cuatro matrices generadas usando la función mk_mat (en el fichero kalman_aux) a partir de las siguientes listas de autovalores:

$$\begin{aligned} \Lambda_1 &= [0.2, 0.1, 0.0, -0.1] \\ \Lambda_2 &= [0.99, 0.1, 0.0, -0.1] \\ \Lambda_3 &= [1, 0.1, 0.0, -0.1] \\ \Lambda_4 &= [0.2, 0.1, 0.0, -1] \end{aligned} \tag{3}$$

Los ruidos $w_t \in \mathbb{R}^4$ y $v_t \in \mathbb{R}^2$ son gausiano y tienen matrices de covarianza

$$\begin{aligned} \mathbf{Q} &= \sigma_w^2 \mathbf{I}_4 \\ \mathbf{R} &= \sigma_v^2 \mathbf{I}_2 \end{aligned} \tag{4}$$

donde $\mathbf{I}_n$ es la matriz identidad de órden n.

El fichero kalman_aux.py define las matrices $\mathbb{B}$, $\mathbb{C}$, $\mathbb{Q}$ y $\mathbb{R}$, y proporciona la función mk_mat para crear la matriz $\mathbb{A}$ dada la lista de autovalores[1]

Utilizando el código auxiliar adjunto a la práctica kalman_aux.py, definimos los 4 sistemas:

```
eigens1 = [0.2, 0.1, 0.0, -0.1]
eigens2= [0.99, 0.1, 0.0, -0.1]
eigens3 = [1, 0.1, 0.0, -0.1]
eigens4 = [1, 0.1, 0.0, -0.1]

A1 =  mk_mat(eigens1)
A2 = mk_mat(eigens2)
A3 = mk_mat(eigens3)
A4 = mk_mat(eigens4)
```

Obteniendo las siguientes matrices A que los determinan:

```
A1:
 [[ 0.09830046 -0.02112914 -0.02045151  0.07385466]
 [-0.02112914 -0.00881251  0.09324886 -0.07499103]
 [-0.02045151  0.09324886  0.09074968 -0.01335784]
 [ 0.07385466 -0.07499103 -0.01335784  0.01976237]]
A2:
 [[ 0.33268965  0.18627666 -0.05618328 -0.43219027]
 [ 0.18627666  0.09338021 -0.01837851 -0.30747963]
 [-0.05618328 -0.01837851  0.07909591 -0.01277468]
 [-0.43219027 -0.30747963 -0.01277468  0.48483423]]
A3:
 [[ 0.28834509 -0.37543475 -0.1385554  -0.01990967]
 [-0.37543475  0.69012454  0.18997508  0.17202943]
 [-0.1385554   0.18997508  0.0657714   0.03035032]
 [-0.01990967  0.17202943  0.03035032 -0.04424103]]
A4:
 [[ 1.40056550e-01  5.22614441e-02  1.26515422e-01  2.79486137e-02]
 [ 5.22614441e-02  1.44337085e-01 -2.01492374e-02 -4.39458688e-04]
 [ 1.26515422e-01 -2.01492374e-02 -9.59388315e-01 -1.55965257e-01]
 [ 2.79486137e-02 -4.39458688e-04 -1.55965257e-01 -2.50053196e-02]]
```

b. Por cada uno de los sistemas generados, crear un filtro de Kalman para estimar el estado de este sistema utilizando como función de input la función

$$u(t) = \begin{cases} 0 & t \leq 50 \\ 1 & t > 50 \end{cases}$$

(el fichero también contiene una función auxiliaria u_f(t) que define la función). Se haga la simulación con $t = 0, \ldots, 99$ y se dibuje un gráfico del error relativo

$$e(t) = \sum_{k=1}^{t} \frac{\|\hat{x}_t - x_t\|^2}{\|x_t\|^2} \tag{5}$$

Definimos una función que nos permita calcular la simulación Kalman para t pasos. Para la implementación se basa en el pseudocódigo dado en el enunciado:

Compute the Kalman Gain	$K_t = \mathbf{P}_t\mathbf{C}'(\mathbf{C}\mathbf{P}_t\mathbf{C}' + \mathbf{R})^{-1}$
Update the estimate	$\hat{x}_t = \bar{x}_t + K_t(z_t - \mathbf{C}\bar{x}_t)$
Update the covariance	$\mathbf{P}_t = (\mathbf{I} - K_t\mathbf{C})\mathbf{P}_t$
Compute the priors	$\bar{x}_{t+1} = \mathbf{A}\bar{x}_t + \mathbf{B}u_t$
	$\mathbf{P}_{t+1} = \mathbf{A}\mathbf{P}_t\mathbf{A}' + \mathbf{Q}$

Para poder hacer una simulación, necesitamos partir de un priox x y una matriz de covarianzas P. Dado que no disponemos de ella, y esperamos que tras un número considerable de pasos(tiempo t), el problema corrija los errores aproximándose lo mayor posible al estado real del sistema, **inicializamos** los mismos de la siguiente manera:

- Los **priors x,** los inicializamos aleatoriamente (con distribución uniforme) ya que no conocemos nada a priori sobre el estado inicial del sistema. Por decisión propia de diseño he elegido un rango de la distribución uniforme de [0,9999] por considerarlo suficientemente amplio para el ámbito del ejercicio.
- La **matriz de covarianzas P**, entre los priors y el valor real del sistema, como una matriz diagonal con valores muy grandes, significando esto una completa incorrelación entre las medidas, dado que acabamos de elegir los priors de manera arbitraria.

Para ello se ha implementado una función para que funcione para tamaños de matriz genéricos, y no sujetos al tamaño n=4 del ejercicio.La función es la siguiente:

```
def filtro_kalman_simulacion(t_steps,x , u_f, A, B, C, Q, R):
  ''' Run [t_ini, t_fin] Kalman filter similation steps the fun, params:

    Args:
      t_ini(int): time steps to simulate
      x(np array):  real system state values dim:x
      u_f(fun): funtion u_f(t) tha returns the user input  given a time step t
      A(np array):  dim: n*n
      B(np array):
      C(np array):
      Q(np array):
      R(np array):

    Returns:
      err, cum_err: List of Relatives and Cumulated relatives  errors caculated between real system value and predicted one
       over  each step(time t) of the simulation
  '''
  #max random num
  max_num = 9999
  # x_bar: random init of prior estimation
  x_bar = np.random.randint(low =0 , high =max_num , size=x.shape[0])
  #Cov. matriz:  diagonal with very large values and all the off diagonal terms to zero
  #meaning completely um-corelated measurements: range choosed:(-max_num,max_num).
  P_bar = np.diag(uniform.rvs(loc=-max_num, scale = max_num*2, size = x.shape[0]))

  I = np.identity(x.shape[0])

  # v = observation noise, we use gaussian distrib with mean 0 and cov. matrix R
  v = np.random.multivariate_normal(mean = [0,0], cov = R)
  #z = Cx + v
  z = C @ x + v
  #print('z.shape', z.shape)
  #Relative err cal
  x_sqabs=np.linalg.norm(x)**2
  err_t = []
```

```
  for t in range(t_steps):
    #Compute the Kalman Gain
    op1 = P_bar @ C.T
    op2 = (C @ P_bar @ C.T) + R
    op2_inversa = np.linalg.inv(op2)
    K = op1 @ op2_inversa

    #Update the estimate: x_hat
    x_hat =  x_bar + (K @(z - (C @ x_bar) ))
    #Calculte the relative error at time t
    err = np.linalg.norm(x - x_hat)**2 / x_sqabs
    err_t.append( err)

    #Update the covariance: P
    P = (I   -  (K @ C)  )@ P_bar

    #Compute the priors(for next step): x_bar, P_bar
    x_bar = ( A @ x_bar) + (B * u_f(t))
    P_bar = (A @ P @ A.T) + Q


  return err_t, np.cumsum(err_t)
```

Siendo el argumento u_f(t) una función que nos de para un instante t, una entrada tal que:

```
def u_f(t):
    return  0.0 if t < Tstep else 1.0
```

Devolviendo el error relativo de cada paso Kalman, y el error relativo acumulado. El error relativo acumulado lo calculamos tal que:

$$e(t) = \sum_{k=1}^{t} \frac{\|\hat{x}_t - x_t\|^2}{\|x_t\|^2}$$

Escribimos una función para la graficación del error relativo y el error relativo acumulado:

```python
def plot_relative_err_evoltion_overtime(t, err_list,cumulative_err, title = ''):
  ''' Plots the simulation error( over time and cumulative) evolution over time
      Args:
        t(list): simulation times list
        err_list(list): list of errors at time t
        cumulative_err(list): Cumulative errors from simulation over time
  '''
  fig, (ax1, ax2) = plt.subplots(1, 2)
  fig.suptitle(title + ': Evolución del error durante la simulación Kalman', fontsize=9)

  ax1.plot(t, err_list)
  ax1.set_xlabel('Tiempo t')
  ax1.set_ylabel('Error relativo')

  ax2.plot(t, cumulative_err)
  ax2.set_xlabel('Tiempo t')
  ax2.set_ylabel('Error relativo acumulado')


  plt.tight_layout()
  fig.set_dpi(100)
```

Y graficamos para los 4 sistemas:

```python
[63] t_steps= 99
     #ejemplo de sistema en estado x
     x = np.array([10,4,5,6])
     t_list = [i for i in range(t_steps)]
```

```python
[64] err_A1, cum_err_A1 = filtro_kalman_simulacion(t_steps,x , u_f, A1, B, C, Q, R)
     plot_relative_err_evoltion_overtime(t_list,err_A1, cum_err_A1, title = 'Matrix A1')
```

Obteniendo las siguientes evoluciones para el error relativo:

Ilustración 1 A1:Sistema 1

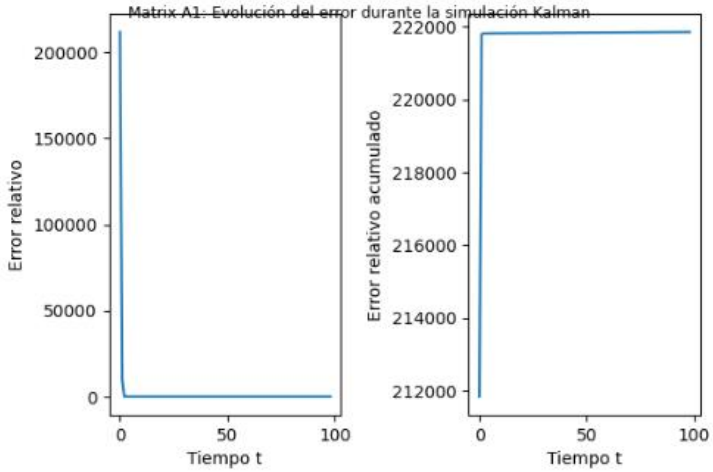


Ilustración 2 A2:Sistema 2

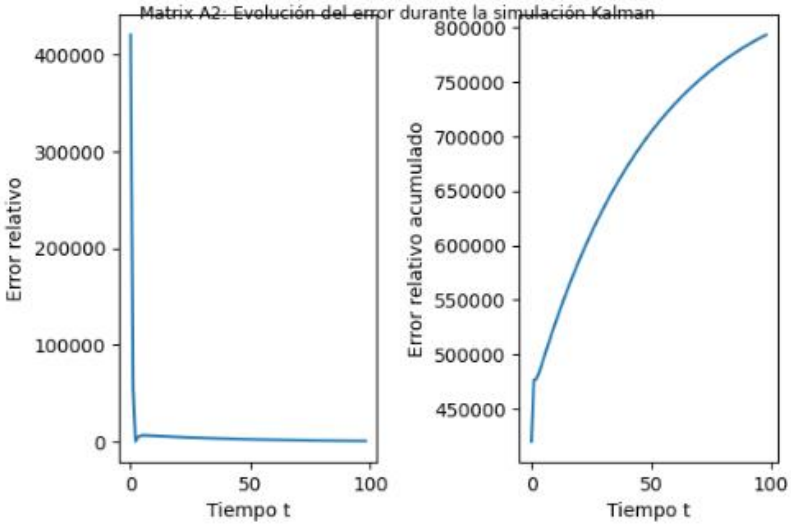


Ilustración 3 A3:Sistema 3

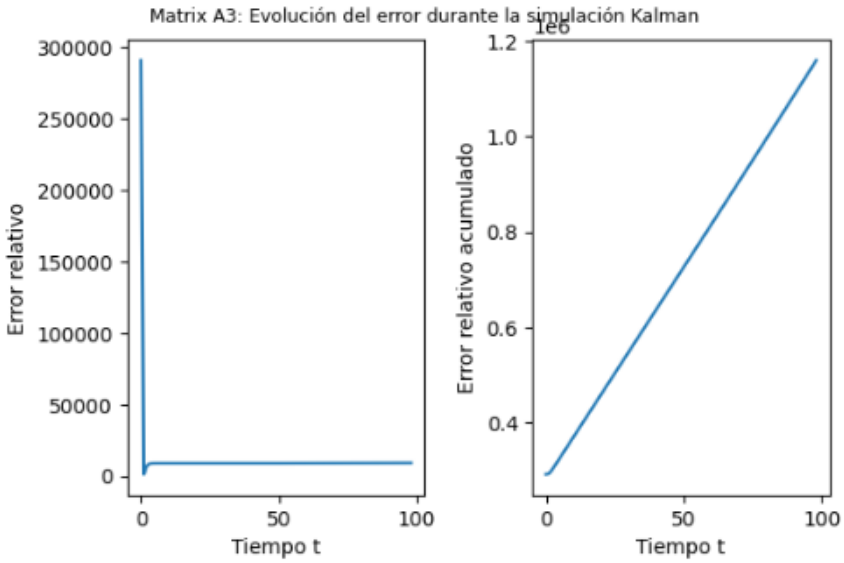
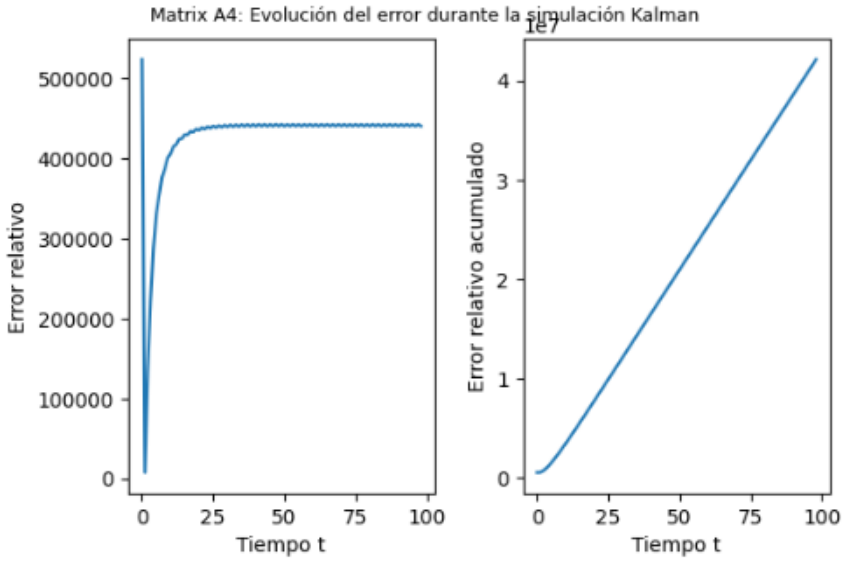


Ilustración 4 A4:Sistema 4



c. Discutir como los autovalores afectan el error.

Vemos como para el sistema 1, con autovalores próximos a 0, el error converge muy rápido, casi instantáneamente (en menos de t=10) prácticamente a 0. Es por ello que el error acumulado se estabiliza en el valor que tiene inicialmente, de manera casi inmediata. Este error es tan alto, debido a la inicialización de la matriz de covarianzas P que hemos elegido inicialmente y a al gran rango de valores arbitrarios que le hemos dado a los priores x.

En el sistema 2, que contiene los mismos autovalores que el 1, solo que uno de ellos tiene un valor muy próximo a 1, vemos como afecta mucho a la evolución del error. Vemos como la estabilización del error relativo acumulado ahora tiene un comportamiento logarítmico.

Para el sistema 3, vemos como cambiamos el autovalor de 0.99 a 1 directamente, manteniendo el resto igual que los 2 anteriores sistemas. Vemos como este ligero cambio convierte el crecimiento del error de logarítmico a lineal. Es decir, ralentiza la estabilización del error relativo acumulado.

Para el sistema 4 elegimos los mismos autovalores que el sistema 1 de nuevo, solo que el último de ellos lo pones ahora próximo a menos 1, para ver como el signo cambia la evolución del error relativo. Observamos un comportamiento también lineal del error relativo acumulado como en el caso anterior.

Es decir, podemos concluir que unos valores próximos a cero aceleran la convergencia del error a 0(es decir, conseguimos aproximar el valor estimado del sistema muy rápidamente al valor real con pocos pasos en la simulación). Mientras que valores cada vez más altos de los autovalores ralentizan dicha convergencia del error (primero se convierte en una estabilización logarítmica y a medida que llegamos a la unidad, en lineal). Observamos cómo es el valor absoluto de los autovalores el que modifica la convergencia del error, y no su signo.