kalman_aux

December 13, 2021

[6]:
```python
#
#  Project: Auxiliary code for the Stochastic Systems class
#  File:    kalman-aux.py
#  Vers.    1.0
#  Date:    11/26/2021
#
#
#  Utility function that builds a matrix with given eigenvalues. This
#  is to be used to obtain the matrix A of the dynamic system that we
#  shall try to control with a Kalman filter.
#
#  The dimensionality of tha matrix will be congruent with the number
#  of eigenvalues that are desired. Note that there is a plethora
#  (viz., an infinite number) of matrices with the given eigenvalues,
#  depending on whet the eigenvectors are. This program will just
#  build one.
#
#  And... yes... I know... I could have done this in an easier way
#  using numpy. I didn't use it for two reasons:
#
#  i) I don't like to create dependencies on libraries just to do
#     something simple. A dependency on a library represents a cost
#     for that program, and a loss of self-standinghood (the word
#     doesn't exist, but it should). It is a price that I am not
#     wiling to pay to do something as simple as a few vector
#     multiplications.
#
#  ii) If I do something myself, I understand it better than having a
#      black box do it for me. Since this is not a production system,
#      I put a premium on understaning.
#
#  (C) Simone Santini, 2021
#

import math
import random
```

```
########################################################################
#
#   A U X I L I A R Y    F U N C T I O N S
#
#



#
# Cross product of two vectors. No check is made to ensure that the
# two have the same dimension so, whach out!
#
def cross(x, y):
    return sum([a*b for (a,b) in zip(x, y)])

#
# Normalizes a vector. Returns a copy of teh vector with norm equal to
# one.
#
def norm(x):
    q = math.sqrt(sum([a*a for a in x]))
    return [a/q for a in x]

#
# Auxiliary function for the Grahm-Schmidt method: takes two vectors
# and returns a vector that is equal to the first minus the second
# multiplied by the cross product of the two
#
def minus_proj(x, y):
    q = sum([a*b for (a,b) in zip(x, y)])
    w = [a-q*b for (a,b) in zip(x, y)]
    return w

#
# Multiplies two matrices. Dimensions are checked for consistency (Why
# do I do it here and not in the scalar product of two vectors?
# Well... because it suited me to do it that way. Any problem? Do you
# want a piece of me? :D)
#
#
def mulmat(A, B):
    if len(A[0]) != len(B):
        print("Matrix multiplication error")
        sys.exit(1)
    n = len(A)
    u = len(A[0])
    m = len(B[0])
    R = [[0.0 for _ in range(m)] for _ in range(n)]
    for i in range(n):
        for j in range(m):
            for k in range(u):
                R[i][j] = R[i][j] + A[i][k]*B[k][j]
    return R


#
# Multiply a matrix by a vector. In principle, of course, this coule
# be done using mulmat, but this function is more convenient. This
# function takes the vector as a list, while in order to use mulmat
# one should consider it a matrix. That is, if A is a 3x3 matrix, you
# can vall mulvec as
#
#   q = mulvec(A, [v1, v2, v3])
#
# while you should call mulmat as
#
#   q = mulmat(A, [[v1], [v2], [v3]])
#
# which is kin of a pain in the neck
#
def mulvec(A, v):
    if len(A[0]) != len(v):
        print ("Vector multiplication error")
        sys.exit(1)
    n = len(A)
    u = len(A[0])
    R = [0.0 for _ in range(n)]
    for i in range(n):
        for k in range(u):
            R[i] = R[i] + A[i][k]*v[k]
    return R


#
#  Builds an unitary matrix of given dimension. This is a bit
#  tricky. The idea is to start with a generic matrix, considering its
#  columns as vectors, and apply normalization and Grahm-Schmidt to
#  obtain a matrix with normalized, orthogonal vectors in the
#  columns. This, of course, can be done as long as the original
#  matrix is non-singular. But, how can we guarantee that?
#
#  Since this is a didactic program and no lives will be lost if it
#  fails, I decided to adopt the simpler and most obvious solution: I
#  generate a random matrix and hope for the best. Singular matrices
#  in low dimensions (we are not going to use this to create huge
#  matrices) are relatively rare so, crossing our fingers, things
#  should work out quite fine.
#
#  Another minor point:the matrix that we are to produce has to have
#  vectors in the columns, but the matrix with the initial vectors has
#  vectors in the rows, as this is simpler to manage in python. The
#  final orthogonal matrix will then be transposed to obtain the
#  vectors on the columns, as per standard.
#
#
#   Returns:
#
#   (U, UT)
#
#   where U is the unitary matrix with the vectors in the columns, and
#   UT its transpose (we need them both to compute the final matrix)
#
def mk_ortho(n):
    V0 = [ [random.uniform(-2*n,2*n) for _ in range(n)] for _ in range(n)]

    UT = [norm(V0[0])]

    for k in range(1,n):
        v = norm(V0[k])
        for i in range(k):
            v = minus_proj(v, UT[i])
        v = norm(v)
        UT = UT + [v]

    U = [[UT[j][i] for j in range(n)] for i in range(n)]
    return (U, UT)


########################################################################
#
#   P U B L I C    F U N C T I O N S    A N D    C O N S T A N T S
#
#


#
# Given a list of n values, creates a nxn matrix (non-trivial: it will
# not be a diagonal matrix) with those values as eigenvalues.
#
def mk_mat(lb):
    n = len(lb)
    L = [ [0.0 for _ in range(n)] for _ in range(n)]
    for k in range(n):
        L[k][k] = lb[k]
    (U, UT) = mk_ortho(n)
    A = mulmat(U, mulmat(L, UT))
    return A

Tstep = 50

#
# Just in case you need it, this is a function that defines the input
# u
#
def u_f(t):
    return 0.0 if t < Tstep else 1.0



#
# The matrices B and C are fixed, the matrix A must be defined using
# the function mk_mat.
#
#
# Note that this is not really the atrix B, it is the transpose of B.
#
# I write it in this way because it was easier for me to implement the
# system considering that all the vectors were rows, so as to avoid
# awkward lists of lists definition. If it is more convenient for your
# implementation to define it as a column, you can define it as
#
#B = [[1.0],
#     [1.0],
#     [1.0],
#     [1.0]
#     ]

B = [1.0, 1.0, 1.0, 1.0]

C = [ [1.0, 0.0, 0.0, 0.0],
      [0.0, 1.0, 0.0, 0.0]
    ]

#
# This is an example of matrix A. You will have to generate your own
# using the eigenvalues specified in the text

eigens = [0.6, 0.3, 0.2, -0.2]   # This is just an example, NOT one of the␣
 ↪lists of eigenvectors of the assignment

A = mk_mat(eigens)
```

```python
#
#  Variance of the input noise, and covariance matrix
#
sigma_w = 0.1

Q = [ [(sigma_w if i == j else 0.0) for i in range(4)] for j in range(4)]


#
#  Variance of the output noise, and covariance matrix
#
sigma_v = 0.1

R = [ [(sigma_w if i == j else 0.0) for i in range(2)] for j in range(2)]
```

[7]:
```python
print("\nExample eigens:", eigens, "\nA:\n", A)
print("\nVariance of the input noise, and covariance matrix: \n", Q)
print("\nVariance of the output noise, and covariance matrix: \n", R)
```

```
Example eigens: [0.6, 0.3, 0.2, -0.2]
A:
 [[0.33275804954828747, 0.022085985590677773, 0.09615666272212839,
-0.2750776115399824], [0.022085985590677717, 0.18648299063600673,
0.13961035168229888, -0.20327370293905955], [0.09615666272212839,
0.1396103516822989, 0.21091840514857016, 0.09679389439827554],
[-0.27507761153998234, -0.20327370293905955, 0.09679389439827553,
0.1698405546671357]]

Variance of the input noise, and covariance matrix:
 [[0.1, 0.0, 0.0, 0.0], [0.0, 0.1, 0.0, 0.0], [0.0, 0.0, 0.1, 0.0], [0.0, 0.0,
0.0, 0.1]]

Variance of the output noise, and covariance matrix:
 [[0.1, 0.0], [0.0, 0.1]]
```

0.1 Apartado a)

Crear 4 Sistemas dinámicos

[224]:
```python
# imports
import numpy as np
from scipy.stats import uniform
import matplotlib.pyplot as plt
from tqdm import tqdm
from scipy.stats import multivariate_normal
```

[225]:
```
################################################################################################
########### Exercise 3
################################################################################################



######################## Variables Definition
# Simulation time
t = 99+1

# U
ut = np.zeros(t)

# Alfas/Eigens for A Matrices
Alfa1 = [0.2, 0.1, 0.0, -0.1]
Alfa2 = [0.99, 0.1, 0.0, -0.1]
Alfa3 = [1, 0.1, 0.0, -0.1]
Alfa4 = [0.2, 0.1, 0.0, -0.1]

# Generate A Matrices
A1 = mk_mat(Alfa1)
A2 = mk_mat(Alfa2)
A3 = mk_mat(Alfa3)
A4 = mk_mat(Alfa4)

# Transform Lists to matrizs to have a better np implementation:
# Sorry Reacher :(, I would love to understand this better but I have squeez my␣
 ↪brain
# until its looking like a nut... and I contunie dizzy.
# May the force and the number 42 be with me :D.
A1 = (np.array(A1))
A2 = (np.array(A2))
A3 = (np.array(A3))
A4 = (np.array(A4))

B = (np.array(B))
C = (np.array(C))

Q = (np.array(Q))
R = (np.array(R))

######################## 3.A) Xt+1 & Zt

# Traspose B
Bt = [[1],[1],[1],[1]]
Bt = (np.array(Bt))

### Generate Random Initial Values
```

```python
# xt & save the value
xr = np.random.rand(4)
xti1 = np.copy(xr)
xti2 = np.copy(xr)
xti3 = np.copy(xr)
xti4 = np.copy(xr)



#P_
P_t1 = np.copy(Q)
P_t2 = np.copy(Q)
P_t3 = np.copy(Q)
P_t4 = np.copy(Q)

# Generate Noise Distribution
wd = multivariate_normal(None, Q, allow_singular = True)
vd = multivariate_normal(None, R, allow_singular = True)

### Create List to sabe xta results
#1
xt1 = []
zt1 = []
#2
xt2 = []
zt2 = []
#3
xt3 = []
zt3 = []
#4
xt4 = []
zt4 = []

### Loop to calculate Xt+1 and Zt value
for i in range(t):# loop

    # initialize u Value
    ut[i] = np.array(u_f(i))
    # Generate Nois
    wt = wd.rvs()
    vt = vd.rvs()

    # Calculate Values with the given Formula
    ## 1
    xti1 = A1 @ xti1 + Bt @ [ut[i]] + wt
    zti1 = C @ xti1 + vt
    # Save Values
    xt1.append(xti1)
    zt1.append(zti1)

    ## 2
    xti2 = A2 @ xti2 + Bt @ [ut[i]] + wt
    zti2 = C @ xti2 + vt
    # Save Values
    xt2.append(xti2)
    zt2.append(zti2)

    ## 3
    xti3 = A3 @ xti3 + Bt @ [ut[i]] + wt
    zti3 = C @ xti3 + vt
    # Save Values
    xt3.append(xti3)
    zt3.append(zti3)

    ## 4
    xti4 = A4 @ xti4 + Bt @ [ut[i]] + wt
    zti4 = C @ xti4 + vt
    # Save Values
    xt4.append(xti4)
    zt4.append(zti4)

#####
#print("d", np.shape(xt4))
#print("d", np.shape(zt4))
```

0.2 Apartado B

Generar Filtros de Kalman para los sistemas para estimar el estado con input = u_f(t)

Simulación de 0 a 99 dibujar gráfico del error relativo

[226]:
```
##############################################################################################
### Functions ######
##############################################################################################

### Compute the Kalman Gain
def kalman(C, P_t, R):
    # Calculate C Transpose
    Ct = np.transpose(C)
    # Calculate inverse of CP_tC'+R
    inv = np.linalg.inv(C@ P_t @ Ct + R)

    # Formula
    kt = P_t @ Ct @ inv
    return kt
```

```
# Compute Kalman
    #kt = kalman(P_t, C, R)

### Update Step x^t & Pt
def update(x_t, kt, zt, C, P_t):
    # Identiy Matrix
    I = np.identity(C.shape[1])

    # Formulas
    x_cap = x_t + kt@(zt - C@x_t)
    Pt = (I - kt @ C)@P_t
    return x_cap, Pt

# Calculate Priors x_t+1 & P_t+1
def prior(A, x_t, B, ut, Pt, Q):
    # Traspose A
    At = np.transpose(A)

    # Formulas
    Ax = A @ x_t
    #print("Ax", Ax)
    Bx = B @ [ut]
    #print("Bx", Bx)
    #print("ut", [ut])
    x_t1 = Ax + Bx

    x_t1 = A @ x_t + B @ [ut]
    P_t1 = A @ Pt @ At + Q
    return x_t1, P_t1

def alg_compt():
    kt = kalman(C, P_t, R)
    x_cap, Pt = update(x_t, kt, zt, C, P_t)
    x_t1, P_t1 = prior(A, x_t, B, ut, Pt, Q)
```

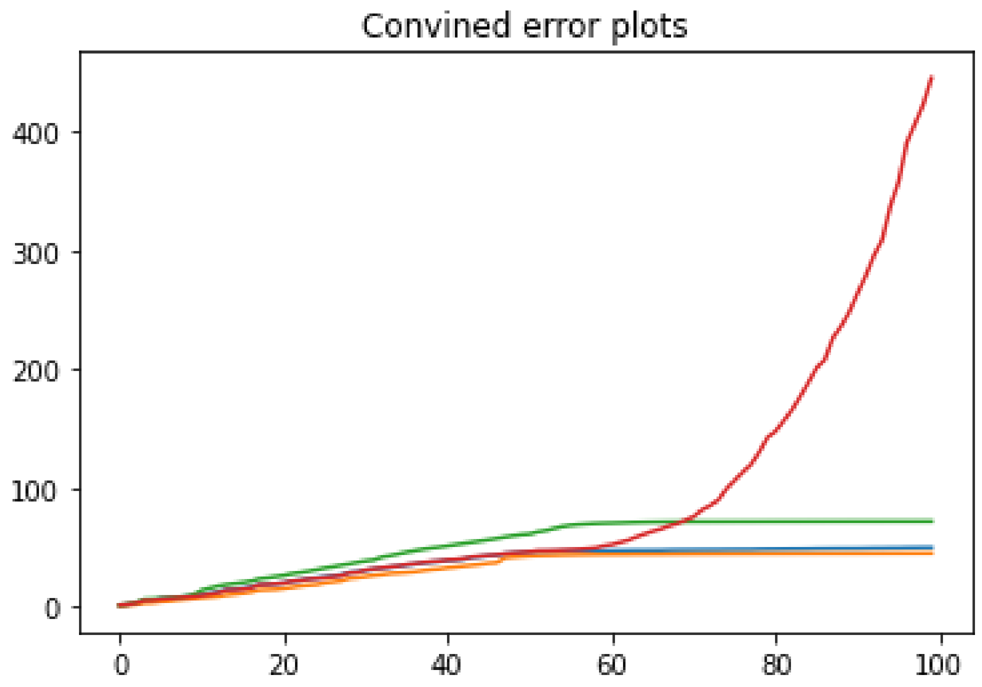
[227]:
```
####################### 3.B) Algorithm
# Upper Cell: Function Definitions
#############
### Definitions
x_t1 = np.copy(xr)
x_t2 = np.copy(xr)
x_t3 = np.copy(xr)
x_t4 = np.copy(xr)

# Variables to save X_caps
```

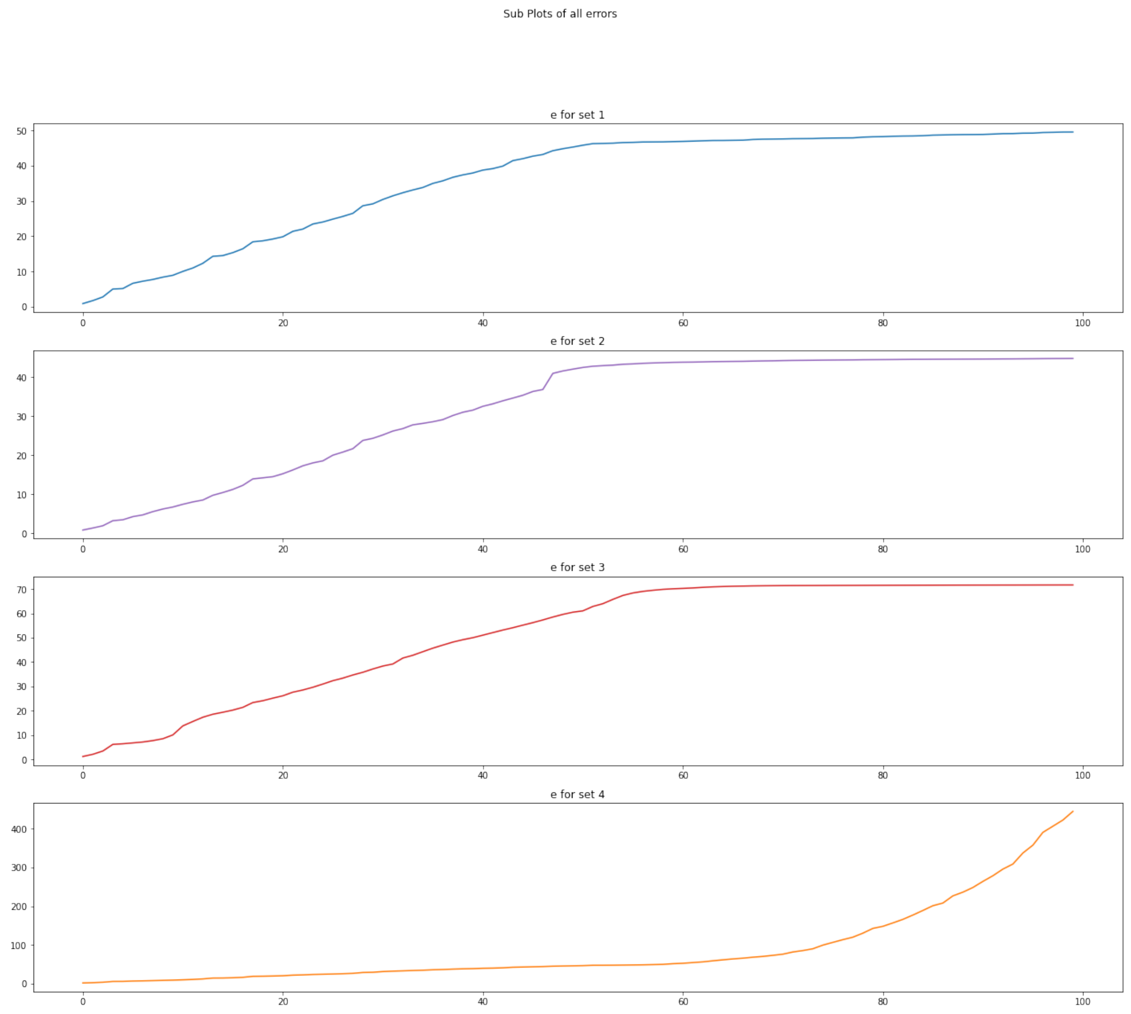
```
x_cap1 = []
x_cap2 = []
x_cap3 = []
x_cap4 = []

#################### Iterate ####################
for i in range(t):
    # Initialize u value:
    u_t[i] = u_f(i)

    ### Compute Algorithm Solutions
    # 1
    # Compute Kalman
    kt1 = kalman(C, P_t1, R)
    # Update estimate & covariance
    xcap1, Pt1 = update(x_t1, kt, zt1[i], C, P_t1)
    # Compute Priors
    x_t1, P_t1 = prior(A1, x_t1, Bt, ut[i], Pt1, Q)
    # Save Values
    x_cap1.append(xcap1)

    # 2
    kt2 = kalman(C, P_t2, R)
    xcap2, Pt2 = update(x_t2, kt, zt2[i], C, P_t2)
    x_t2, P_t2 = prior(A2, x_t2, Bt, ut[i], Pt2, Q)
    x_cap2.append(xcap2)

    # 3
    kt3 = kalman(C, P_t3, R)
    xcap3, Pt3 = update(x_t3, kt, zt3[i], C, P_t3)
    x_t3, P_t3 = prior(A3, x_t3, Bt, ut[i], Pt3, Q)
    x_cap3.append(xcap3)

    # 4
    kt4 = kalman(C, P_t4, R)
    xcap4, Pt4 = update(x_t4, kt, zt4[i], C, P_t4)
    x_t4, P_t4 = prior(A2, x_t4, Bt, ut[i], Pt4, Q)
    x_cap4.append(xcap4)

################################################

#print("d", np.shape(x_cap1))
#print("d", np.shape(x_cap2))
#print("d", np.shape(x_cap3))
#print("d", np.shape(x_cap4))
```

[291]:
```
############ Calculated Error
##### Variable Definition
e1t = []
e2t = []
e3t = []
e4t = []

# Loop
for i in range(t):

    #1
    e1t.append((np.linalg.norm(x_cap1[i] - xt1[i])**2)/(np.linalg.
 ↪norm(xt1[i])**2))

    #2
    e2t.append((np.linalg.norm(x_cap2[i] - xt2[i])**2)/(np.linalg.
 ↪norm(xt2[i])**2))

    #3
    e3t.append((np.linalg.norm(x_cap3[i] - xt3[i])**2)/(np.linalg.
 ↪norm(xt3[i])**2))

    #4
    e4t.append((np.linalg.norm(x_cap4[i] - xt4[i])**2)/(np.linalg.
 ↪norm(xt4[i])**2))

sum_e1 = np.cumsum(e1t)
sum_e2 = np.cumsum(e2t)
sum_e3 = np.cumsum(e3t)
sum_e4 = np.cumsum(e4t)

#print("\ne1t: \n", e1t)
#print("shape(e1t): \n", np.shape(e1t))
#print("\ne2t: \n", e2t)
#print("shape(e2t): \n", np.shape(e2t))
#print("\ne3t: \n", e3t)
#print("shape(e3t): \n", np.shape(e3t))
#print("\ne4t: \n", e4t)
#print("shape(e4t): \n", np.shape(e4t))

plt.title('Convined error plots')
plt.plot(sum_e1)
plt.plot(sum_e2)
plt.plot(sum_e3)
plt.plot(sum_e4)

fig, (ax1, ax2, ax3, ax4) = plt.subplots(4,figsize=(22,18))
```

```
fig.suptitle('Sub Plots of all errors')
ax1.set_title('e for set 1')
ax1.plot(sum_e1)
ax2.set_title('e for set 2')
ax2.plot(sum_e2, 'tab:purple')
ax3.set_title('e for set 3')
ax3.plot(sum_e3, 'tab:red')
ax4.set_title('e for set 4')
ax4.plot(sum_e4, 'tab:orange')
```

```
[291]: [<matplotlib.lines.Line2D at 0x7f1ae220d700>]
```





0.3 Apartado C

Discutir como los autovalores afectan al error

Se peude observar como si los autovalores son mas pequeñi¡os, empieza a converger mas tarde. Por lo general, menos en el último grafo, se ve como se alcanza la convergencia al rededor del valor t = 50, justo cuando u empieza a ser 1.

Se ha encontrado este parrafo en el pdf de teoría el cual parece ser revelador, se cita a continuación: (Creado por Santini): ”” Equation (343) shows how the eigenvalues determine the behavior of the various components of Wkx as k $\rightarrow \infty$. If | i| > 1, the component in the direction D1 2 vi will grow without bounds; if | i| < 1, the corresponding component will shrink to zero; if i = −1, the corresponding component will oscillate back and forth wothout settling to any value; finally, if i = 1 the corresponding component will remain constant. The probabilistic interpretation requires that no component grow without bounds so, if we had | k| > 1 for some k, we would be in trouble. Fortunately, as we shall see shortly, this is not the case. ””

Se encuentra en la página 16/25 ó 70(segun el papel) de los Apuntes sobre modelos ARMA y filtro de Kalman.

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```