Gram-Schmidt process

Junghanss, Juan Cruz & Marco, Tomás Guido

Métodos Cuantitativos para Ciencias Sociales y Negocios - Prof. Sergio Pernice

March 30, 2021

1 Homework - Python and GeoGebra

[29]:
```
import numpy as np # Primero que nada, importamos la librería 'numpy', y lo␣
 ↪denominamos 'np' para cuando queramos referenciarla
from numpy import linalg as LA  # Y también importamos el paquete 'linalg' de la␣
 ↪librería 'numpy', y denominamos a ese
 # paquete 'LA', haciendo referencia a 'Linear Algebra', de modo tal que podamos␣
 ↪referenciarlo fácilmente cuando lo necesitemos.
```

[30]:
```
# 1°: Generación de un vector 3x1 con elementos aleatorios entre -1 y 1

# a) Generamos un vector columna de tres dimensiones, es decir, tres filas: es␣
 ↪un vector de 3x1. Y queremos que los elementos
# sean números aleatorios entre 0 y 1. Para esto, usamos el comando 'np.random.
 ↪rand(3,1)', tal que: i) "np" refiere la librería
# 'np'; ii) "random.rand" es el comando para que genere un vector con elementos␣
 ↪aleatorios entre 0 y 1; iii) El paréntesis
# "(3,1)" refiere a que queremos que sea un vector de una columna y tres filas.

# b) Querremos que el vector esté centrado, para lo cual restamos 0.5 a todo el␣
 ↪vector ("-0.5") y multiplicamos todo por 2 ("2*")
# Eso de restar un escalar (0.5) a un vector no tiene sentido analíticamente,␣
 ↪pero se puede hacer porque Python lo interpreta
# como un vector de la misma dimensionalidad (un vector de 3x1 cuyos tres␣
 ↪elementos son -0.5); esto se llama "broadcasting".
# Al restar 0,5 y multiplicar por 2 lo que hacemos es transformar al vector de␣
 ↪modo tal que el centro sea 0, ya que ahora, en
# vez de ir entre 0 y 1 (como lo hace el comando inicial básico), va entre -1 y␣
 ↪1 (si solo hubiéramos restado 0.5 y no
# multiplicado por 2, los elementos del vector irían entre -0.5 y 0.5).

# c) Denominaremos al vector 'c1'
```

```
# d) Para poder verlo, le pedimos a Python que nos imprima el vector, usando el␣
↪comando 'print(c1)'

# e) Le pedimos a Python que nos diga la shape del vector. Debería devolvernos␣
↪que es un vector de 3x1.

# Es decir:
```

[31]:
```
c1 = 2*(np.random.rand(3,1)-0.5)
print(c1)
c1.shape
```

```
[[ 0.14838055]
 [-0.77272532]
 [-0.64195945]]
```

[31]: (3, 1)

[32]:
```
## PROCESO DE GRAM-SCHMIDT
# Trataremos de usar Python para entender el proceso de Gram-Schmidt, y ver que␣
↪funciona para lo que lo queremos: transformar
# un conjunto cualquiera de vectores linealmente independientes en otro conjunto␣
↪de vectores ortogonales que spanean el mismo
# subespacio.
# Veremos el caso de 3 vectores, para no hacer tan extenso el desarrollo, pero␣
↪es extensive a nD (cualquier dimensión).
```

[33]:
```
# 2°: Generación de tres vectores aleatorios

# Usando la misma estructura que explicamos en 1°, generamos tres vectores␣
↪centrados con elementos aleatorios entre -1 y 1.

# Estos son los vectores del conjunto J, es decir, los vectores que queremos␣
↪transformar. Pueden ser cualquier tipo de vectores;
# lo único que deben cumplir es que deben ser linealmente independientes. Como␣
↪los generaremos al azar, la probabilidad de que
# sean linealmente dependientes es infinitesimla, pequeñísimo, prácticamente␣
↪cero, sería demasiada casualidad; igualmente, en 3°
# lo chequearemos.

# Llamaremos a estos tres vectores 'c1', 'c2' y 'c3'. Usamos la estructura␣
↪explicada en 1°.

# Para que nos los muestre (imprima), usamos el comando print, y en vez de solo␣
↪escribir el nombre del vector entre paréntesis,
```

```
# podemos escribir además "cn =" para que al imprimir nos salga más prolijo␣
↪indicando cuál es cada vector.

# Es decir:
```

[34]:
```
c1 = 2*(np.random.rand(3,1)-0.5) # c1, c2 y c3 son dos vectores 3x1 con␣
↪elementos aleatorios en el
c2 = 2*(np.random.rand(3,1)-0.5) # rango (-1,1). Pregunta: por que puedo sumar␣
↪un vector
c3 = 2*(np.random.rand(3,1)-0.5) # y un escalar? Buscar "numpy broadcasting"

print("c1 =", c1)
print("c2 =", c2)
print("c3 =", c3)
```

```
c1 = [[-0.49078341]
 [-0.10362243]
 [ 0.20226701]]
c2 = [[0.70668634]
 [0.05532775]
 [0.59915093]]
c3 = [[-0.27644656]
 [-0.51922044]
 [ 0.98632303]]
```

[35]:
```
# 3°: Chequeamos que los vectores del conjunto J generados aleatoriamente (c1,␣
↪c2 y c3) son linealmente independientes.

# Lo demostraremos con dos vectores (c1 y c2), pero para estar del todo seguro␣
↪debería hacerse entre los tres.

# EL producto escalar entre c1 y c2 es c1.c2 = |c1||c2|cos(theta) (donde |cn|␣
↪refiere al módulo del vector cn y theta es
# el ángulo que conforman entre los vectores c1 y c2). Entonces: c1.c2/
↪(|c1||c2|) = cos(theta), por lo que si este numero (es
# decir, c1.c2/(|c1||c2|)) NO es 1 ni -1, sabremos que los # vectores c1 y c2 no␣
↪son linealmente dependientes (es decir, son
# linealmente independientes), ya que esto ocurre para los ángulos de 0 y 180:␣
↪cos(0)=1 y cos(180)=-1. Por ende, si el resultado
# de c1.c2/(|c1||c2|) no da ni 1 ni -1, entonces sabremos que los vectores no␣
↪son linealemnte dependientes; son linealmente
# independientes.

# Veamos como escribir "c1.c2/(|c1||c2|)" en Python.
# Para multiplicar vectores (escalarmente) (en este caso, c1.c2) usamos el␣
↪comando 'np.dot1'
```

```
# Para dividir, usamos '/'.
# Para multiplicar escalares (en este caso, |c1||c2|), usamos '*'.

# Para que pueda escalarmente dos vectores (c1 y c2) de 3x1, debemos transponer␣
↪al primer vector (c1). Y así, sí podremos
# multiplicar: c1 transpuesto por c2; el primero de 1x3 y el segundo de 3x1 ==>␣
↪se puede multiplicar sin problemas, y el
# resultado será un escalar (1x1).

# Para transponer un vector, en este caso c1, usamos el comando 'np.
↪transpose(c1)'.

# La norma de un vector cN se calcula usando el comando 'LA.norm(cN)'

# Una manera más 'manual' de calcular la norma sería usando su definición: la␣
↪raíz cuadrada del producto escalar del vector con
# sí mismo (para lo cual hay que transponer el primero): np.sqrt(np.dot(np.
↪transpose(c1),c1))

# Directamente metemos lo escrito en print, para que de una nos imprima el␣
↪resultado, ya que solo nos interesa ver que no da ni
# -1 ni 1.
```

[36]:
```
print(np.dot(np.transpose(c1),c2)/(LA.norm(c1)*LA.norm(c2)))
```

```
[[-0.46091929]]
```

[37]:
```
# 4°: Vector unitario en la dirección de c1: vector n1

# Denominaremos al vector unitario en la dirección de c1 como 'n1' (es lo que en␣
↪las notas de clase se llama 'e1').

# Recordemos que un vector unitario es simplemente la división entre el vector,␣
↪c1, y su norma, |c1|, que en Python se escribe
# 'LA,norm(c1)'. Por ende: n1 = c1 / LA.norm(c1)

# Verificamos que su norma es efectivamente 1 multiplicándolo escalarmente por␣
↪sí mismo, transponiendo al primero para que sea
# posible multiplicar (np.dot(np.transpose(n1),n1)) (estrictamente por la␣
↪defínción de norma, deberíamos además aplicarle
# raíz cuadrada a eso, pero   como da 1, da lo mismo aplicarle o no la raíz␣
↪cuadrada). Esto lo calculamos directamente dentro
# dentro del comando 'print', porque lo que nos interesa es ver si da o no 1 (si␣
↪da 1, significa que hemos hecho bien a n1).
```

```
# Lo vemos:
```

[38]:
```
n1 = c1/LA.norm(c1)

print(n1)

print(np.dot(np.transpose(n1),n1))
```

```
[[-0.90743107]
 [-0.19159207]
 [ 0.37398039]]
[[1.]]
```

[39]:
```
# 5º: Vector unitario en la dirección de c2: vector n2

# Ya tenemos el vector unitario n1, que es proporcional a c1 (apunta en la misma␣
 ↪dirección), pero tiene tamaño/magnitud/norma 1.

# Ahora queremos construir el vector unitario n2, que apunta en la dirección de␣
 ↪c2 y tiene tamaño/magnitud/norma 1.

# Recordemos que n2 tiene que ser ortogonal a n1, y tal que el span del conjunto␣
 ↪de vectores unitarios K={n1,n2} sea el mismo
# que el del conjunto de vectores linealmente independientes J={c1, c2}.

# El mecanismo para hallar n2 no es tan simple como n1.

# El mecanismo para hallar n2 es:

# a) Calculamos la proyección de c2 en la dirección de n1 (componente de c2␣
 ↪paralela a n1), que llamaremos 'P_c2_n1'. Recordemos
# que, por definición, el cálculo de la pryección de un vector ci en la␣
 ↪dirección de nj es: P_ci_nj = (nj.ci)*nj. Por ende, en
# nuestro caso: P_c2_n1 = (n1.c2)*n1. El producto entre paréntesis es un␣
 ↪producto escalar ('np.dot') entre dos vectores, de
# modo que es necesario transponer el primer vector (n1), y dará un escalar, de␣
 ↪modo que el producto entre el paréntesis y n1
# es un producto normal (*). La intuición de P_c2_n1 = (n1.c2)*n1 es que 'c2.n1'␣
 ↪da la magnitud de la proyección de c2 sobre
# n1, mientras que, al multiplicarlo por n1, el vector resultante (P_c2_n1)␣
 ↪apunta en la dirección de n1. Laformula general
# incluye una división por (n1.n1), pero en en este caso no es necesario porque␣
 ↪(n1.n1) = 1 (ya que n1 es unitario).

# b) Después, debemos calcular la proyección de c2 en la dirección de n2␣
 ↪(componente de c2 paralela a n2), que, como n2 es
```

```
# ortogonal n1, es igual al componente de c2 perpendicular a n1, y por eso lo␣
↪denominaremos 'P_c2_perp_n1'. Por definición,
# P_c2_perp_n1 es igual a la resta entre el vector c2 y la componente de c2␣
↪paralela a n1 (esto es, la proyección de c2 en la
# dirección de n1):  P_c2_perp_n1 = c2 - P_c2_n1.

# c) Chequeamos que c2 =  P_c2_n1 + P_c2_perp_n1. Lo hacemos de dos maneras:␣
↪Forma 1: observando numéricamente si coinciden
# 'c2' y 'P_c2_n1 + P_c2_perp_n1'; Forma 2: empleando operadores lógicos:␣
↪preguntándole a Python con "=="; si da todo 'True',
# es que se verifica. Con esta segunda forma porque puede llegar a dar False por␣
↪errores que la compu puede cometer errores
# cuando trabaja con floating  points como estos (no obstante, suele ocurrir con␣
↪cáculos mas complejos, no los que hacemos
# nosotros))

# d) Finalmente, calculamos n2, que es el vector unitario que apunta en la␣
↪dirección de c2.  Para calcular n2, por definición,
# simplemente debemos dividir P_c2_perp_n1 (= P_c2_n2) por su norma:  n2 =␣
↪P_c2_n2 / |P_c2_n2| = P_c2_perp_n1 / |P_c2_perp_n1|.
# Como lo denominamos como 'P_c2_perp_n1', lo calcularemos  con ese nombre. El␣
↪código
# es  n2 = P_c2_perp_n1 / LA.norm(P_c2_perp_n1)

# 5e) Verificamos que el conjunto {n1, n2} es ortonormal. El conjunto {n1, n2}␣
↪es ortonormal si el producto escalar entre n1
# y n2 da 0 (o, por lo errores de cálculo de Python, prácticamente 0):
```

[40]:
```
# 5a):

# Calculamos la proyección de c2 en la dirección de n1 (componente de c2␣
↪paralela a n1), que llamaremos 'P_c2_n1'. Recordemos
# que, por definición, el cálculo de la pryección de un vector ci en la␣
↪dirección de nj es: P_ci_nj = (nj.ci)*nj. Por ende, en
# nuestro caso: P_c2_n1 = (n1.c2)*n1. El producto entre paréntesis es un␣
↪producto escalar ('np.dot') entre dos vectores, de
# modo que es necesario transponer el primer vector (n1), y dará un escalar, de␣
↪modo que el producto entre el paréntesis y n1
# es un producto normal (*). La intuición de P_c2_n1 = (n1.c2)*n1 es que 'c2.n1'␣
↪da la magnitud de la proyección de c2 sobre
# n1, mientras que, al multiplicarlo por n1, el vector resultante (P_c2_n1)␣
↪apunta en la dirección de n1. Laformula general
# incluye una división por (n1.n1), pero en en este caso no es necesario porque␣
↪(n1.n1) = 1 (ya que n1 es unitario).
```

[41]:

```
P_c2_n1 = np.dot(np.transpose(n1),c2)*n1

print(P_c2_n1)
```

```
[[ 0.38819793]
 [ 0.08196286]
 [-0.15998836]]
```

[44]:

```
# 5b):

# Después, debemos calcular la proyección de c2 en la dirección de n2␣
↪(componente de c2 paralela a n2), que, como n2 es
# ortogonal n1, es igual al componente de c2 perpendicular a n1, y por eso lo␣
↪denominaremos 'P_c2_perp_n1'. Por definición,
# P_c2_perp_n1 es igual a la resta entre el vector c2 y la componente de c2␣
↪paralela a n1 (esto es, la proyección de c2 en la
# dirección de n1):  P_c2_perp_n1 = c2 - P_c2_n1.

# Para chequear la perpendicularidad de P_c2_perp_n1 y P_c2_n1 calculamos su␣
↪producto escalar: debe dar cero. Como la compu
# comete algunos pequeños errores con los decimales, si nos da un número muy␣
↪cercano a cero, lo tomamos como 0. En este caso
# nos da '6,93*e^{-17}', que, a propósitos prácticos, es 0.

# En resumen: teníamos los vectores c1 y c2. Primero calculamos el vector n1, y␣
↪depués la componente de c2 (proyección de
# c2 en la dirección de n1). Esto último se lo restamos a c2, de modo que me␣
↪queda un vector a 90 grados, que chequeamos
# que efectivamente lo es, porque su coseno da 0 (6,93*e^{-17}).
```

[45]:

```
P_c2_perp_n1 = c2 - P_c2_n1

print(P_c2_perp_n1)
print(np.dot(np.transpose(P_c2_perp_n1),P_c2_n1))
```

```
[[ 0.31848841]
 [-0.02663511]
 [ 0.75913929]]
[[6.9388939e-17]]
```

[46]:

```
# 5c) Chequeamos que c2 =  P_c2_n1 + P_c2_perp_n1.

# Lo hacemos de dos maneras:
```

```
# Forma 1: observando numéricamente si coinciden 'c2' y 'P_c2_n1 + P_c2_perp_n1'.
↪

# Forma 2: empleando operadores lógicos: preguntándole a Python con "=="; si da␣
↪todo 'True', es que se verifica. Con esta
# segunda forma porque puede llegar a dar False por errores que la compu puede␣
↪cometer errores cuando trabaja con floating
# points como estos (no obstante, suele ocurrir con cáculos mas complejos, no␣
↪los que hacemos nosotros))
```

[47]:
```
# Forma 1 de chequeo:
print(c2)
print(P_c2_n1 + P_c2_perp_n1)
```

```
[[0.70668634]
 [0.05532775]
 [0.59915093]]
[[0.70668634]
 [0.05532775]
 [0.59915093]]
```

[48]:
```
# Forma 2 de chequeo:
c2 == P_c2_n1 + P_c2_perp_n1
```

[48]:
```
array([[ True],
       [ True],
       [ True]])
```

[49]:
```
# 5d) Finalmente, calculamos n2, que es el vector unitario que apunta en la␣
↪dirección de c2.

#  Para calcular n2, por definición, simplemente debemos dividir P_c2_perp_n1 (=␣
↪P_c2_n2) por su norma:
# n2 = P_c2_n2 / |P_c2_n2| = P_c2_perp_n1 / |P_c2_perp_n1|. Como lo denominamos␣
↪como 'P_c2_perp_n1', lo calcularemos
# con ese nombre. El código es  n2 = P_c2_perp_n1 / LA.norm(P_c2_perp_n1)

# Además, verificamos que su norma es efectivamente 1 multiplicándolo␣
↪escalarmente por sí mismo, transponiendo al primero
# para que sea posible multiplicar (np.dot(np.transpose(n2),n2)) (estrictamente␣
↪por la defición de norma, deberíamos además
# aplicarle raíz cuadrada a eso, pero   como da 1, da lo mismo aplicarle o no la␣
↪raíz cuadrada). Esto lo calculamos directamente
# dentro dentro del comando 'print', porque lo que nos interesa es ver si da o␣
↪no 1 (si da 1, significa que hemos hecho bien
# a n2).
```

```
# Lo vemos:
```

[50]:
```
n2 = P_c2_perp_n1/LA.norm(P_c2_perp_n1)
print(n2)

print(np.dot(np.transpose(n2),n2))
```

```
[[ 0.38666864]
 [-0.032337  ]
 [ 0.92165161]]
[[1.]]
```

[51]:
```
# 5e) Verificamos que el conjunto {n1, n2} es ortonormal

# El conjunto {n1, n2} es ortonormal si el producto escalar entre n1 y n2 da 0␣
↪(o, por lo errores de cálculo de
# Python, prácticamente 0: en este caso 5,5*e^{-17}):
```

[28]:
```
print(np.dot(np.transpose(n1),n2))
```

```
[[5.55111512e-17]]
```

[52]:
```
# 6°: Vector unitario en la dirección de c3: vector n3

# Ya tenemos el vector unitario n1, que es proporcional a c1 (apunta en la misma␣
↪dirección), pero tiene tamaño/magnitud/norma 1,
# y el vector unitario n2, que es proporcional a c2 (apunta en la misma␣
↪dirección), pero tiene tamaño/magnitud/norma 1

# Ahora queremos construir el vector unitario n3, que apunta en la dirección de␣
↪c3 y tiene tamaño/magnitud/norma 1.

# Ya tenemos los vectores c1, c2 y c3 (que son los que generamos aleatoriamente;␣
↪son dato), y los vectores n1 y n2 (que los
# calculamos recién). Nos falta n3

# El mecanismo para hallar n2 no es tan simple como n1. Es el siguiente:

# a) Calculamos la proyección de c3 en la dirección de n1 (componente de c3␣
↪paralela a n1): P_c3_n1

# b) Calculamos la proyección de c3 en la dirección de n2 (componente de c3␣
↪paralela a n2): P_c3_n2

# c) Calculamos la componente de c3 perpendicular a n1 y n2 (P_c3_perp_n1yn2):␣
↪P_c3_perp_n1yn2 = c2 - P_c3_n1 - P_c3_n2
```

```
# d) Finalmente, calculamos n3, que es el vector unitario que estábamos buscando␣
↪(vector unitario que apunta en la dirección
# de c3), y se calcula simplemente como el vector P_c3_perp_n1yn2 normalizado:␣
↪n3 = P_c3_perp_n1yn2 / |P_c3_perp_n1yn2|
```

[53]:
```
# 6a) Proyección de c3 en la dirección de n1 (componente de c3 paralela a n1):␣
↪P_c3_n1

P_c3_n1 = np.dot(np.transpose(n1),c3)*n1

print(P_c3_n1)
```

```
[[-0.6526246 ]
 [-0.13779305]
 [ 0.26896676]]
```

[54]:
```
# 6b) Proyección de c3 en la dirección de n2 (componente de c3 paralela a n2)
```

[55]:
```
P_c3_n2 = np.dot(np.transpose(n2),c3)*n2

print(P_c3_n2)
```

```
[[ 0.31665958]
 [-0.02648216]
 [ 0.75478015]]
```

[56]:
```
# 6c) Componente de c3 perpendicular a n1 y n2:

# Por definición, el componente de c3 perpendicular a n1 y n2 (P_c3_perp_n1yn2)␣
↪es P_c3_perp_n1yn2 = c2 - P_c3_n1 - P_c3_n2, es
# decir, restamos c2 los dos componentes que recién calculamos.

# Para chequear la perpendicularidad/ortogonalidad de P_c3_perp_n1yn2 a n1 y n2␣
↪calculamos el producto escalar de aquel con
# cada uno de éstos, y deben dar cero. Como la compu comete algunos pequeños␣
↪errores con los decimales, si nos da un número
# muy cercano a cero, lo tomamos como 0.
```

[58]:
```
P_c3_perp_n1yn2 = c3 - P_c3_n1 - P_c3_n2 # Extraemos la componente de c3␣
↪perpendicular al plano
                                          # spanned por n1 y n2; perpendicular a␣
↪n1  y n2
print(P_c3_perp_n1yn2)

print(np.dot(np.transpose(n2),P_c3_perp_n1yn2))
print(np.dot(np.transpose(n1),P_c3_perp_n1yn2))
```

```
[[ 0.05951845]
 [-0.35494522]
 [-0.03742388]]
[[7.63278329e-17]]
[[1.70002901e-16]]
```

[59]:
```
# 6d) Finalmente, calculamos n3, que es el vector unitario que estábamos␣
 ↪buscando (vector unitario que apunta en la dirección
# de c3), y se calcula simplemente como el vector P_c3_perp_n1yn2 normalizado:␣
 ↪n3 = P_c3_perp_n1yn2 / |P_c3_perp_n1yn2|

# Constatamos que tiene tamaño 1 multiplicándolo escalarmente por sí mismo.
```

[60]:
```
# Calculemos entonces n3, que es igual P_c3_perp_n1yn2 a 1:
n3 = P_c3_perp_n1yn2/LA.norm(P_c3_perp_n1yn2)
print(n3)

# Constatemos que esta normalizado a 1
print(np.dot(np.transpose(n3),n3))
```

```
[[ 0.16448773]
 [-0.98094179]
 [-0.10342625]]
[[1.]]
```

[61]:
```
# Así, terminamos el proceso de Gram-Schmidt: ya terminamos de construir la base␣
 ↪ortonormal de R3: E =  {n1, n2, n3}:
```

[62]:
```
print("n1 =", n1)
print("n2 =", n2)
print("n3 =", n3)
```

```
n1 = [[-0.90743107]
 [-0.19159207]
 [ 0.37398039]]
n2 = [[ 0.38666864]
 [-0.032337  ]
 [ 0.92165161]]
n3 = [[ 0.16448773]
 [-0.98094179]
 [-0.10342625]]
```

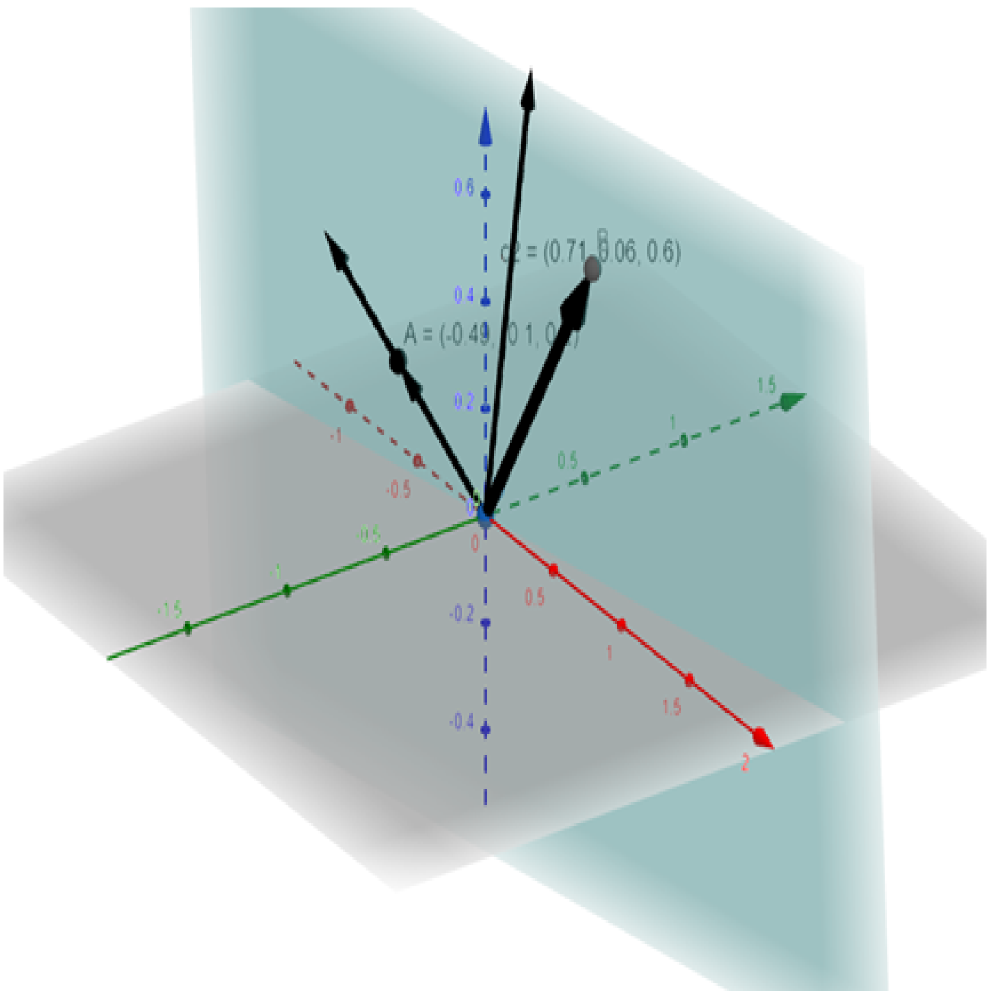
In summary, with the Gram-Schmidt process we managed to transform a set $\mathcal{J} = \{c_1, c_2, c_3\}$ of linearly independent vectors, into another set $\mathcal{K} = \{n_1, n_2, n_3\}$ of orthonormal vectors that span the same subspace as $\mathcal{J}$.

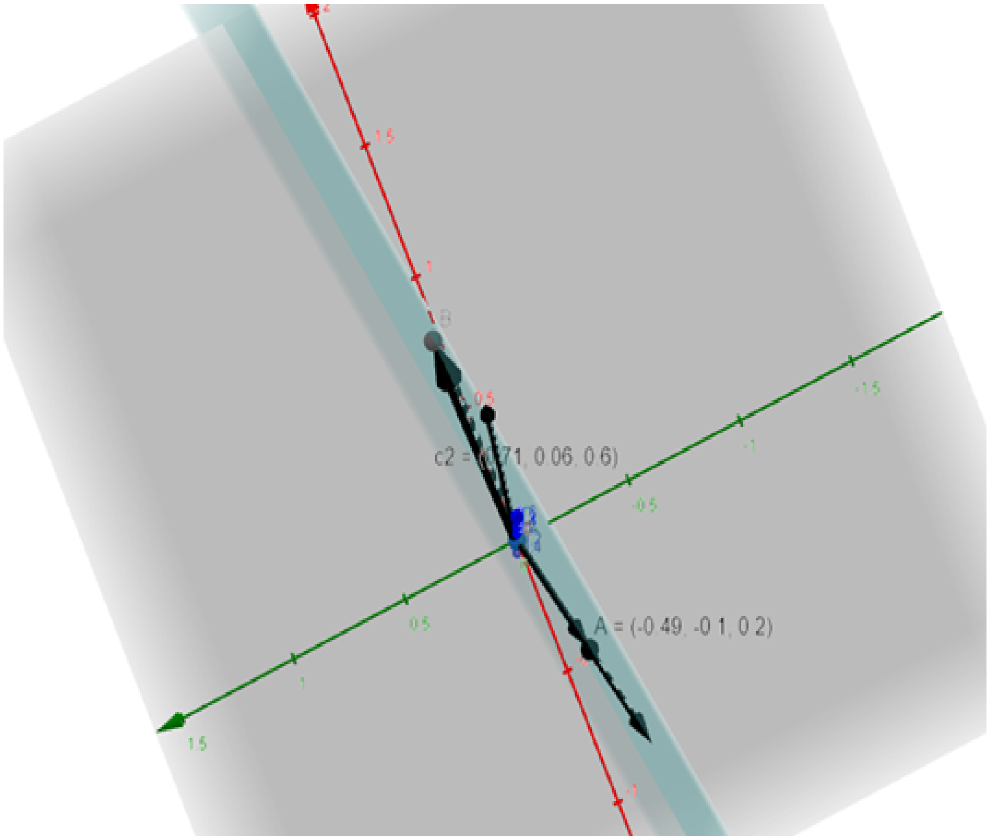
We can demonstrate this graphically using GeoGebra: taking two vectors from the set $\mathcal{J}$ and seeing the plane they constitute (that is, their span), we should observe that the two associated unit vectors of the set $\mathcal{K}$ belong to exactly the same subspace.

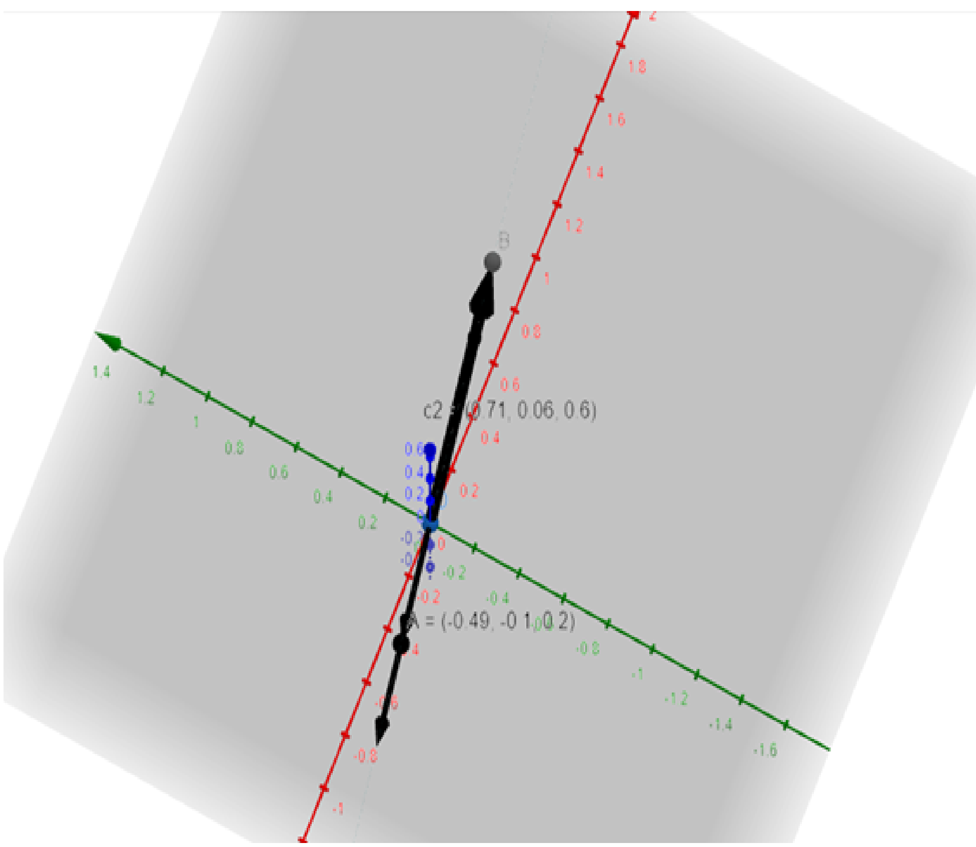
To prove it, we take c_1 and c_2 from the set $\mathcal{J}$, and the associated unit vectores, n_1 and n_2, from the set $\mathcal{K}$ of orthonormal vectors.

The procedure in GeoGebra was as follows. First, we defined the vectors of the set $\mathcal{J}$: $c_1 = (-0.49078341, -0.10362243, 0.20226701)$ y $c_2 = (0.70668634, 0.05532775, 0.59915093)$. Then, we defined the points $A = c_1$, $B = c_2$ y $D = (0, 0, 0)$, with which we constituted the plane: $Plane(A, B, D)$, which in the image looks light blue.

Finally, we defined the associated orthonormal unit vectors of the set $\mathcal{K}$: $n_1 = (-0.90743107, -0.19159207, 0.37398039)$ y $n_2 = (0.38666864, -0.032337, 0.92165161)$, and we see that indeed span the subspace that the vectors c_1 and c_2 span. This is clearly seen in the third image.




B
c2 = (0.71, 0.06, 0.6)
A = (-0.49, -0.1, 0.2)

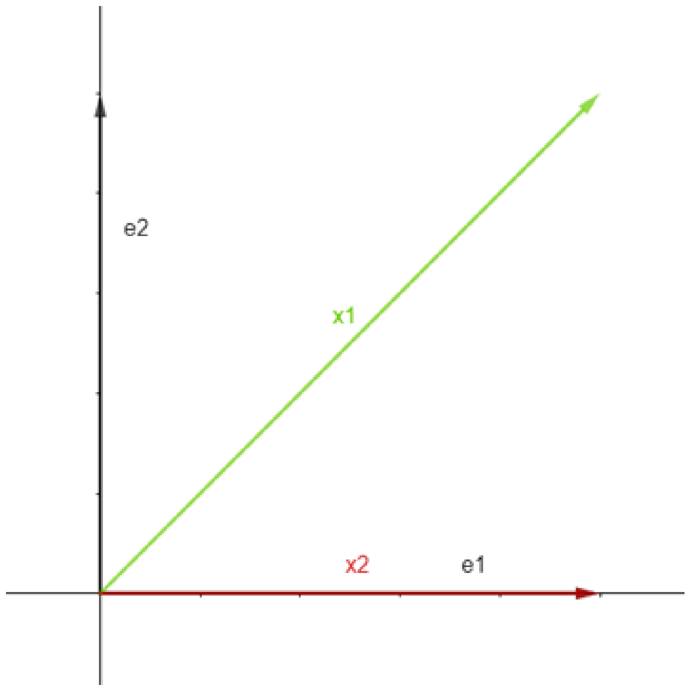

B
c2 = (0.71, 0.06, 0.6)
A = (-0.49, -0.1, 0.2)

2 Homework - Working Paper N°689 [1]

2.1 HW 5.14:

In $\mathcal{R}^2$ suppose that you have the basis $\{x^1, x^2\}$ given by $x^1 = \hat{e}^1 + \hat{e}^2, x^2 = \hat{e}^1$, where $\hat{e}^1$ and $\hat{e}^2$ are orthonormal. a) Draw x^1 and x^2 in a plane with the horizontal axis pointing in the direction of $\hat{e}^1$ and the vertical axis pointing in the direction of $\hat{e}^2$. b) Define $v^1 = x^1$ (see (5.36)) and compute v^2 as in(5.38). Verify that v^2 is orthogonal to v^1. c)Draw v^1 and v^2 in plane with the horizontal axis pointing in the direction of $\hat{e}^1$ and the vertical axis pointing in the direction of $\hat{e}^2$. d)Normalize v^1 and v^2 to obtain $\hat{e}^1$ and $\hat{e}^2$. e)Assuming that the column vector representation of $\hat{e}^1 = (1;0)^T$ and $\hat{e}^2 = (0;1)^T$, give the column vector representation of v^1 and v^2 and of $\hat{e}^1$ and $\hat{e}^2$.

A - If we draw x^1 and x^2 in a plane, with the above mentioned directions for both axis, we can observe:



B - Following the Gram-Schmidt process so as to obtain an orthonormal span of vectors to $\{x^1, x^2\}$: We formalize the definition of v^1 as an identity of x^1 vector:

$$v^1 \equiv x^1$$

The initial step consist of normalizing v^1 and that will give us the first orthonormal vector:

$$\hat{e}^1 = \frac{v^1}{||v^1||}$$

To compute v^2, we need to find an orthogonal vector to v^1 or $\hat{e}^1$, since $\hat{e}^1$ is the unit vector of v^1. For the sake of the exercise, we will be computing it in respect to v^1. We define v^2 as follows:

$$v^2 = x^2 - Proj_{v1}\left(x^2\right) = x^2 - \left(\frac{v^1 \cdot x^2}{v^1 \cdot v^1}\right) \cdot v^1$$

Now we can verify that v^2 is orthogonal to v^1 (remember that we have not normalized v^2 yet). Considering that $\hat{e}^1 = (1,0)$; $\hat{e}^2 = (0,1)$ and therefore $x^1 = (1,1); x^2 = (1,0)$, we could see that:

$$v^1 = (1,1)$$

$$v^2 = (1,0) - \left(\frac{(1,1)\cdot(1,0)^{\mathbf{T}}}{(1,1)\cdot(1,1)^{\mathbf{T}}}\right)\cdot(1,1) = (1,0) - \frac{1}{2}\cdot(1,1) = (1,0) - \left(\frac{1}{2},\frac{1}{2}\right)$$

$$v^2 = \left(\frac{1}{2},-\frac{1}{2}\right)$$

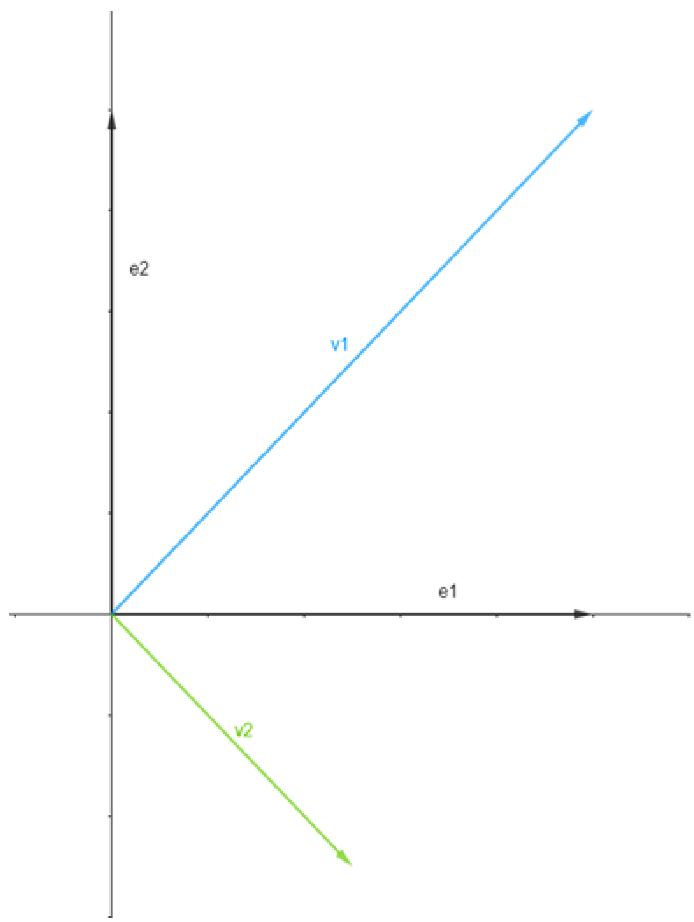
Recall that orthogonality is given by $v^1 \cdot v^2 = 0$. Let's prove it by taking the dot product:

$$v^1 \cdot v^2 = (1,1)\cdot(\frac{1}{2},-\frac{1}{2}) = 0.5 - 0.5 = 0$$

Otherwise, we can work the complete equation $v^1 \cdot v^2 = ||v^1||\cdot||v^2||\cdot\cos\theta = 0$

$$\cos^{-1}\left(\frac{v^1\cdot v^2}{||v^1||\cdot||v^2||}\right) = \cos^{-1}(0) = \frac{\pi}{2}$$

C - The plane with v^1 and v^2 in it looks as following:



D - As we already did in exercise B for v^1, we can normalize both vectors. Recall that $\hat{e}^1$ implies:

$$\hat{e}^1 = \frac{v^1}{||v^1||} \Longrightarrow \hat{e}^1 = \frac{(1,1)}{\sqrt{2}} = \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$$

We can take a step further to rationalize $\hat{e}^1$:

$$\hat{e}^1 = \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right)$$

Computing now normalization for v^2:

$$\hat{e}^2 = \frac{v^2}{||v^2||} \Longrightarrow \hat{e}^2 = \frac{(\frac{1}{2}, -\frac{1}{2})}{\sqrt{\frac{1}{2}}} = \left(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right)$$

E - As our numeric example showed, the column vector representation stands as follow:

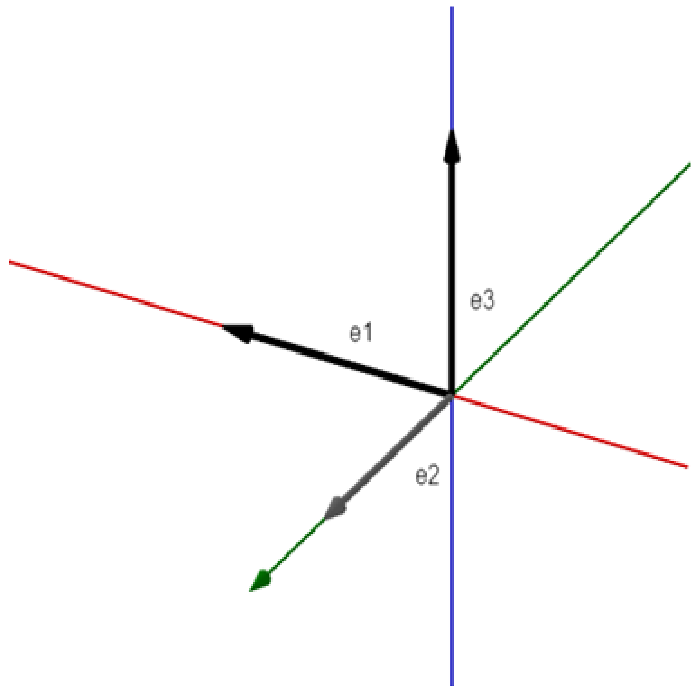
$$v^1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}; v^2 = \begin{bmatrix} 0.5 \\ -0.5 \end{bmatrix};$$

$$\hat{e}^1 = \begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix}; \hat{e}^2 = \begin{bmatrix} \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} \end{bmatrix};$$

2.2 HW 5.15:

Do the same in $\mathcal{R}^3$. Assume that you have the base $\{x^1, x^2, x^3\}$ given by $x^1 = \hat{e}^1 + \hat{e}^2 + \hat{e}^3, x^2 = \hat{e}^1, x^3 = \hat{e}^3$, where $\hat{e}^1$, $\hat{e}^2$ and $\hat{e}^3$ are orthonormal. Assume first $v^1 = x^1$. Do the same assuming $v^1 = x^2$. Explain why you obtain two different bases. Prove that these two different bases span the same vector space.

A - Considering the previous exercise (2.1) we can do the same for the base $\{x^1, x^2, x^3\}$ given by $x^1 = \hat{e}^1 + \hat{e}^2 + \hat{e}^3, x^2 = \hat{e}^1, x^3 = \hat{e}^3$. Drawing the base $\{x^1, x^2, x^3\}$ in a plane we can simply observe that $\hat{e}^1$, $\hat{e}^2$, $\hat{e}^3$ are orthonormal:



B - We will first proceed with Gram-Schmidt assuming $v^1 \equiv x^1 = (1,1,1)$. Normalizing v^1 in order to obtain our first orthonormal vector of the basis:

$$\hat{e}^1 = \frac{v^1}{||v^1||} \Longrightarrow \hat{e}^1 = \frac{(1,1,1)}{\sqrt{3}} = \left(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$$

$$\hat{e}^1 = \left(\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}\right)$$

Now, we define:

$$v^2 = x^2 - Proj_{v1}\left(x^2\right) = x^2 - \left(\frac{v^1 \cdot x^2}{v^1 \cdot v^1}\right) \cdot v^1$$

$$v^2 = (1,0,0) - \left(\frac{(1,1,1) \cdot (1,0,0)^{\mathbf{T}}}{(1,1,1) \cdot (1,1,1)^{\mathbf{T}}}\right) \cdot (1,1,1) = (1,0,0) - \frac{1}{3} \cdot (1,1,1) = \left(\frac{2}{3}, -\frac{1}{3}, -\frac{1}{3}\right)$$

Normalizing v^2:

$$\hat{e}^2 = \frac{v^2}{||v^2||} \Longrightarrow \hat{e}^2 = \frac{\left(\frac{2}{3}, -\frac{1}{3}, -\frac{1}{3}\right)}{\left(\frac{\sqrt{6}}{3}\right)} = \left(\frac{\sqrt{6}}{3}, -\frac{\sqrt{6}}{3}, -\frac{\sqrt{6}}{3}\right)$$

Lastly, for v^3:

$$v^3 = x^3 - Proj_{v1}\left(x^3\right) - Proj_{v2}\left(x^3\right) = x^3 - \left(\frac{v^1 \cdot x^3}{v^1 \cdot v^1}\right) \cdot v^1 - \left(\frac{v^2 \cdot x^3}{v^2 \cdot v^2}\right) \cdot v^2$$

$$v^3 = (0,1,0) - \left(\frac{(1,1,1) \cdot (0,1,0)^{\mathbf{T}}}{(1,1,1) \cdot (1,1,1)^{\mathbf{T}}}\right) \cdot (1,1,1) - \left(\frac{\left(\frac{2}{3}, -\frac{1}{3}, -\frac{1}{3}\right) \cdot (0,1,0)^{\mathbf{T}}}{\left(\frac{2}{3}, -\frac{1}{3}, -\frac{1}{3}\right) \cdot \left(\frac{2}{3}, -\frac{1}{3}, -\frac{1}{3}\right)^{\mathbf{T}}}\right) \cdot \left(\frac{2}{3}, -\frac{1}{3}, -\frac{1}{3}\right)$$

$$v^3 = (0,1,0) - (\frac{1}{3}) \cdot (1,1,1) - \left(-\frac{1}{2}\right) \cdot \left(\frac{2}{3}, -\frac{1}{3}, -\frac{1}{3}\right) = (0,1,0) - \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right) - \left(-\frac{1}{3}, \frac{1}{6}, \frac{1}{6}\right)$$

$$v^3 = \left(0, \frac{1}{2}, -\frac{1}{2}\right)$$

Normalizing v^3:

$$\hat{e}^3 = \frac{v^3}{||v^3||} \Longrightarrow \hat{e}^3 = \frac{\left(0, \frac{1}{2}, -\frac{1}{2}\right)}{\left(\frac{\sqrt{2}}{2}\right)} = \left(0, \frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}\right)$$

Hence, the basis $\mathcal{B}^1 = \left\{\hat{e}^1, \hat{e}^2, \hat{e}^3\right\}$ is orthonormal to the vector space $\left\{x^1, x^2, x^3\right\}$.

$$\mathcal{B}^1 = \left\{\left(\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}\right); \left(\frac{\sqrt{6}}{3}, -\frac{\sqrt{6}}{3}, -\frac{\sqrt{6}}{3}\right); \left(0, \frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}\right)\right\}$$

C - Now, we can do the same process by assuming $v^1 \equiv x^2 = (1,0,0)$ and therefore v^2 depends on x^3 and v^3 of x^1.

$$\hat{e}^1 = \frac{v^1}{||v^1||} \Longrightarrow \hat{e}^1 = \frac{(1,0,0)}{\sqrt{1}} = (1,0,0)$$

Now, we define:

$$v^2 = x^3 - Proj_{v1}\left(x^3\right) = x^3 - \left(\frac{v^1 \cdot x^3}{v^1 \cdot v^1}\right) \cdot v^1$$

$$v^2 = (0,1,0) - \left(\frac{(1,0,0) \cdot (0,1,0)^{\mathbf{T}}}{(1,0,0) \cdot (1,0,0)^{\mathbf{T}}}\right) \cdot (1,0,0) = (0,1,0) - 0 \cdot (1,0,0) = (0,1,0)$$

Normalizing v^2:

$$\hat{e}^2 = \frac{v^2}{||v^2||} \Longrightarrow \hat{e}^2 = \frac{(0,1,0)}{\sqrt{1}} = (0,1,0)$$

Lastly, for v^3:

$$v^3 = x^1 - Proj_{v1}\left(x^1\right) - Proj_{v2}\left(x^1\right) = x^1 - \left(\frac{v^1 \cdot x^1}{v^1 \cdot v^1}\right) \cdot v^1 - \left(\frac{v^2 \cdot x^1}{v^2 \cdot v^2}\right) \cdot v^2$$

$$v^3 = (1,1,1) - \left(\frac{(1,0,0)\cdot(1,1,1)^{\mathbf{T}}}{(1,0,0)\cdot(1,0,0)^{\mathbf{T}}}\right)\cdot(1,0,0) - \left(\frac{(0,1,0)\cdot(1,1,1)^{\mathbf{T}}}{(0,1,0)\cdot(0,1,0)^{\mathbf{T}}}\right)\cdot(0,1,0)$$

$$v^3 = (1,1,1) - 1\cdot(1,0,0) - 1\cdot(0,1,0) = (0,0,1)$$

And normalizing v^3:

$$\hat{e}^3 = \frac{v^3}{||v^3||} \Longrightarrow \hat{e}^3 = \frac{(0,0,1)}{(\sqrt{1})} = (0,0,1)$$

The basis $\mathcal{B}^2 = \{\hat{e}^1, \hat{e}^2, \hat{e}^3\}$ is orthonormal to the vector space $\{x^1, x^2, x^3\}$ and it turns to be a canonical base for $\mathcal{R}^3$:

$$\mathcal{B}^2 = \{(1,0,0)\,;(0,1,0)\,;(0,0,1)\}$$

D - We can explain why we obtain two different bases by recalling the following theorem from the working paper [1], that shows us how our operations for $v^2; v^3$ will vary considering that the vectors operating inside the equation are a unique expansion in terms of x^i. Hence, results will numerically vary.

Theorem 1 *For every orthogonal collection of vectors $v^i, i = 1, \ldots, j$, spanning a subspace $\mathcal{W}$ of the dot product state $\mathcal{V}$, every vector $w \in \mathcal{W}$ has a (unique) expansion in terms of v^i*

$$w = \sum_{i=1}^{j}\left(\frac{w\cdot v^i}{v^i\cdot v^i}\right)\cdot v^i$$

References

[1] Sergio A. Pernice, 2019. "Intuitive Mathematical Economics Series. Linear Structures I. Linear Manifolds, Vector Spaces and Scalar Products," CEMA Working Papers: Serie Documentos de Trabajo. 689, Universidad del CEMA.