**Objectives:**

* Identify matrices as operators
* Relate the transformation matrix to a set of new basis vectors
* Formulate code for mappings based on these transformation matrices
* Write code to find an orthonormal basis set computationally

**Python Code for Gram-Schmidt Process**

# GRADED FUNCTION

import numpy as np

import numpy.linalg as la

verySmallNumber = 1e-14 # That's 1×10⁻¹⁴ = 0.00000000000001

# Our first function will perform the Gram-Schmidt procedure for 4 basis vectors.

# We'll take this list of vectors as the columns of a matrix, A.

# We'll then go through the vectors one at a time and set them to be orthogonal

# to all the vectors that came before it. Before normalising.

# Follow the instructions inside the function at each comment.

# You will be told where to add code to complete the function.

def gsBasis4(A) :

B = np.array(A, dtype=np.float\_) # Make B as a copy of A, since we're going to alter it's values.

# The zeroth column is easy, since it has no other vectors to make it normal to.

# All that needs to be done is to normalise it. I.e. divide by its modulus, or norm.

B[:, 0] = B[:, 0] / la.norm(B[:, 0])

# For the first column, we need to subtract any overlap with our new zeroth vector.

B[:, 1] = B[:, 1] - B[:, 1] @ B[:, 0] \* B[:, 0]

# If there's anything left after that subtraction, then B[:, 1] is linearly independant of B[:, 0]

# If this is the case, we can normalise it. Otherwise we'll set that vector to zero.

if la.norm(B[:, 1]) > verySmallNumber :

B[:, 1] = B[:, 1] / la.norm(B[:, 1])

else :

B[:, 1] = np.zeros\_like(B[:, 1])

# Now we need to repeat the process for column 2.

# Insert two lines of code, the first to subtract the overlap with the zeroth vector,

# and the second to subtract the overlap with the first.

B[:, 2] = B[:, 2] - B[:, 2] @ B[:, 0] \* B[:, 0]

B[:, 2] = B[:, 2] - B[:, 2] @ B[:, 1] \* B[:, 1]

# Again we'll need to normalise our new vector.

# Copy and adapt the normalisation fragment from above to column 2.

if la.norm(B[:, 2]) > verySmallNumber :

B[:, 2] = B[:, 2] / la.norm(B[:, 2])

else :

B[:, 2] = np.zeros\_like(B[:, 2])

# Finally, column three:

# Insert code to subtract the overlap with the first three vectors.

B[:, 3] = B[:, 3] - B[:, 3] @ B[:, 0] \* B[:, 0]

B[:, 3] = B[:, 3] - B[:, 3] @ B[:, 1] \* B[:, 1]

B[:, 3] = B[:, 3] - B[:, 3] @ B[:, 2] \* B[:, 2]

# Now normalise if possible

if la.norm(B[:, 3]) > verySmallNumber :

B[:, 3] = B[:, 3] / la.norm(B[:, 3])

else :

B[:, 3] = np.zeros\_like(B[:, 3])

# Finally, we return the result:

return B

# The second part of this exercise will generalise the procedure.

# Previously, we could only have four vectors, and there was a lot of repeating in the code.

# We'll use a for-loop here to iterate the process for each vector.

def gsBasis(A) :

B = np.array(A, dtype=np.float\_) # Make B as a copy of A, since we're going to alter it's values.

# Loop over all vectors, starting with zero, label them with i

for i in range(B.shape[1]) :

# Inside that loop, loop over all previous vectors, j, to subtract.

for j in range(i) :

# Complete the code to subtract the overlap with previous vectors.

# you'll need the current vector B[:, i] and a previous vector B[:, j]

B[:, i] = B[:, i] - B[:, i] @ B[:, j] \* B[:, j]

# Next insert code to do the normalisation test for B[:, i]

if la.norm(B[:, i]) > verySmallNumber :

B[:, i] = B[:, i] / la.norm(B[:, i])

else:

B[:, i] = np.zeros\_like(B[:, i])

# Finally, we return the result:

return B

# This function uses the Gram-schmidt process to calculate the dimension

# spanned by a list of vectors.

# Since each vector is normalised to one, or is zero,

# the sum of all the norms will be the dimension.

def dimensions(A) :

return np.sum(la.norm(gsBasis(A), axis=0))

**Tester Code**

V = np.array([[1,0,2,6],

[0,1,8,2],

[2,8,3,1],

[1,-6,2,3]], dtype=np.float\_)

gsBasis4(V)

# Once you've done Gram-Schmidt once,

# doing it again should give you the same result. Test this:

U = gsBasis4(V)

gsBasis4(U)

# Try the general function too.

gsBasis(V)

# See what happens for non-square matrices

A = np.array([[3,2,3],

[2,5,-1],

[2,4,8],

[12,2,1]], dtype=np.float\_)

gsBasis(A)

dimensions(A)

B = np.array([[6,2,1,7,5],

[2,8,5,-4,1],

[1,-6,3,2,8]], dtype=np.float\_)

gsBasis(B)

dimensions(B)

# Now let's see what happens when we have one vector that is a linear combination of the others.

C = np.array([[1,0,2],

[0,1,-3],

[1,0,2]], dtype=np.float\_)

gsBasis(C)

dimensions(C)

**Python Code for Reflective Matrix over Plane**

# PACKAGE

# Run this cell first once to load the dependancies.

import numpy as np

from numpy.linalg import norm, inv

from numpy import transpose

from readonly.bearNecessities import \*

# GRADED FUNCTION

# You should edit this cell.

# In this function, you will return the transformation matrix T,

# having built it out of an orthonormal basis set E that you create from Bear's Basis

# and a transformation matrix in the mirror's coordinates TE.

def build\_reflection\_matrix(bearBasis) : # The parameter bearBasis is a 2×2 matrix that is passed to the function.

# Use the gsBasis function on bearBasis to get the mirror's orthonormal basis.

E = gsBasis(bearBasis)

# Write a matrix in component form that performs the mirror's reflection in the mirror's basis.

# Recall, the mirror operates by negating the last component of a vector.

# Replace a,b,c,d with appropriate values

TE = np.array([[1, 0],

[0, -1]])

# Combine the matrices E and TE to produce your transformation matrix.

T = E @ TE @ transpose(E)

# Finally, we return the result. There is no need to change this line.

return T

**Tester Code**

# First load Pyplot, a graph plotting library.

%matplotlib inline

import matplotlib.pyplot as plt

# This is the matrix of Bear's basis vectors.

# (When you've done the exercise once, see what happns when you change Bear's basis.)

bearBasis = np.array(

[[1, -1],

[1.5, 2]])

# This line uses your code to build a transformation matrix for us to use.

T = build\_reflection\_matrix(bearBasis)

# Bear is drawn as a set of polygons, the vertices of which are placed as a matrix list of column vectors.

# We have three of these non-square matrix lists: bear\_white\_fur, bear\_black\_fur, and bear\_face.

# We'll make new lists of vertices by applying the T matrix you've calculated.

reflected\_bear\_white\_fur = T @ bear\_white\_fur

reflected\_bear\_black\_fur = T @ bear\_black\_fur

reflected\_bear\_face = T @ bear\_face

# This next line runs a code to set up the graphics environment.

ax = draw\_mirror(bearBasis)

# We'll first plot Bear, his white fur, his black fur, and his face.

ax.fill(bear\_white\_fur[0], bear\_white\_fur[1], color=bear\_white, zorder=1)

ax.fill(bear\_black\_fur[0], bear\_black\_fur[1], color=bear\_black, zorder=2)

ax.plot(bear\_face[0], bear\_face[1], color=bear\_white, zorder=3)

# Next we'll plot Bear's reflection.

ax.fill(reflected\_bear\_white\_fur[0], reflected\_bear\_white\_fur[1], color=bear\_white, zorder=1)

ax.fill(reflected\_bear\_black\_fur[0], reflected\_bear\_black\_fur[1], color=bear\_black, zorder=2)

ax.plot(reflected\_bear\_face[0], reflected\_bear\_face[1], color=bear\_white, zorder=3);











