General program instructions

1 Preliminary requirements/information

- The following programs are compatible with Windows 10.
- The programs can be used with Python 2 or Python 3 however the code is optimised for Python 3 use in terms of the format of output. Differences between how the two interpreters handle the input will be mentioned when needed.
- To use the files, download them all to the same folder.

2 The create_shelve_sample.py program

2.1 Purpose

This program allows the user to create a shelve of random matrix data, which can be used as a sample in program "ratios.py" to produce expectation value ratios.

2.2 Code description

The main idea is to take a dictionary with matrix values and write this to a shelve. The code for the program is given as follows:

```
 1 #This program generates a shelve of D dimensional square matrices that can be used in
      ratios.py
 2 import shelve
 3 import numpy as np
 4 
 5 while True:
 6     try:
 7         num_of_mat = input("Enter the number of the matrices required in the shelve: ")
 8         num_of_mat = int(num_of_mat)
 9         break
10     except ValueError:
11         print("Please select an integer value for the number of matrices")
12 
13 while True:
14     try:
15         D = input("Enter the dimension of the matrices: ")
16         D = int(D)
17         break
18     except ValueError:
19         print("Please select an integer value for the dimension")
20 
21 mat_dic = {}
22 for number in range(0, num_of_mat):
23     mat_dic["mat" + str(number + 1)] = np.random.rand(D, D)
24 
25 shelve_dict = shelve.open('dim-%d-mat-%d' % (D, num_of_mat),  flag='n')
26 for i in mat_dic.keys():
27     shelve_dict[i] = mat_dic[i]
28 shelve_dict.close()
```

Lines 5-11 and lines 13-19 allow the user the number of matrices they want in the shelve, and the dimension of said matrices respectively. The try statements are present to restrict such choices to integers only. Line 21 opens an empty dictionary called *mat_dic* and lines 22-23 fill it with random matrices by iterating over the number of matrices specified. Here the keys of the dictionary are denoted "matk" with k ranging from one to the number of matrices chosen, and the corresponding values in the dictionary are random matrices of $D \times D$ dimensions. So in the example code, $num\_of\_mat = 10$ and $D = 30$ indicating that after executing the for statement in lines 22-23, the dictionary *mat_dic* will contain 10 randomly generated matrices, each being 30×30 in size and defined by key labels "matk", k ranging from one to ten. The code then finally stores this matrix data into a shelve, a persistent dictionary-like object that can be accessed by other files. This is achieved starting at line 25, by opening the shelve, ready to insert the *mat_dic* stored data (note the flag='n' indicates that a new shelve is being opened which can be read and written to). The name of the shelve for ease of reference, is defined by the number of matrices and dimension. For example if $num\_of\_mat = 10$ and $D = 30$, the shelve name will appear as "dim-30-mat-10". Lines 26-27 then iterate over the keys in *mat_dic* and save these keys as the shelves keys, and likewise, save it's values as the shelves values. The shelve is then closed and the program complete. This has now successfully generated a shelve of matrix data which can be used as input for "ratios.py".

Running this code using a Python 3 interpreter, will produce three files which constitute the created shelve. In keeping with the given example, their names will be: "dim-30-mat-10.bak", "dim-30-mat-10.dat" and "dim-30-mat-10.dir". This shelve of matrix data is now ready to be used in the "ratios.py" program.

Running the code using a Python 2 interpreter on the other hand, will generate a single file named "dim-30-mat-10". The small difference in using a Python 2 or Python 3 generated shelve in the "ratios.py" program, will be addressed in the following section.

3 The ratios.py program

3.1 Purpose

This program receives user input via a shelve of matrix data, and produces the ratio of theoretical and experimental expectation values as output. It also calculates the convergence criteria of the Gaussian model.

3.2 Code description

The ratios.py code can be most effectively described by looking at it, in two separate parts. Part one, executes the extraction of the experimental, linear, quadratic, cubic and quartic matrix expectation values, and part two uses this extracted data to solve for the 13 parameters of the Gaussian model, which are consequently used to evaluate the theoretical cubic and quartic graph values. Finally the ratios between theory and experimental values are computed, in keeping with the methodology of the paper "Permutation Invariant Gaussian Matrix Models" (PIGMM).

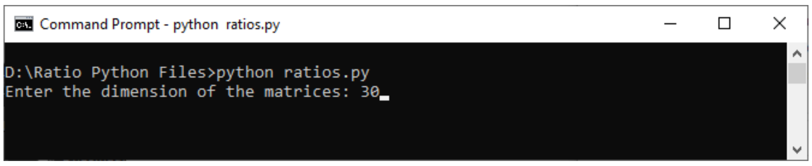
Part one

The code attributed to part one is as follows:

```python
# File Purpose: Extracts the experimental expectation values of linear, quadratic, cubic and quartic graphs, calculates
# the theoretical expectation value equivalent and computes the ratio. It also confirms that the convergence criteria is
# met.

from __future__ import division
import numpy as np
import shelve
import sys
from scipy.optimize import fsolve

# Allowing user inputs: Dimension and File name.
while True:
    try:
        D = input("Enter the dimension of the matrices: ")
        D = int(D)
        break
    except ValueError:
        print("Please select an integer value for the dimension")

fname = input("Enter the name of the shelve file: ")

# Opening the matrix data shelve to read in the data
ts = shelve.open(fname, flag='r', protocol=2)
dic_key = ts.keys()
M = np.zeros((23, len(dic_key)))

# Iterating over all matrices, and calculating the values of the matrix function sums
for widx, k in enumerate(dic_key):
    sys.stdout.write('Dimension: %d, Matrix number: %d      \r' % (D, widx + 1))
    sys.stdout.flush()

    W = ts[k]

    # Linear terms
    M[0, widx] = np.trace(W)  # \sum_{i} W_{ii}
    M[1, widx] = np.sum(W)  # \ sum_{i,j} W_{ij})

    # Quadratic terms

    W1 = W ** 2
    M[2, widx] = np.sum(W1)  # \sum_{i,j} W_{ij}^2
    W2 = W * W.T
    M[3, widx] = np.sum(W2)  # \sum_{i,j} W_{ij} W_{ji}

    Wd = np.diagonal(W)

    W3 = Wd * W.T
    M[4, widx] = np.sum(W3)  # \sum_{i,j} W_{ii} W_{ij}
    W4 = Wd * W
    M[5, widx] = np.sum(W4)  # \sum_{i,j} W_{ii} W_{ji}
    W5 = np.dot(W.T, W)
    M[6, widx] = np.sum(W5)  # \sum_{i,j,k} W_{ij} W_{ik}
    W6 = np.dot(W, W.T)
    M[7, widx] = np.sum(W6)  # \sum_{i,j,k} W_{ij} W_{kj}
    W7 = np.dot(W, W)
    M[8, widx] = np.sum(W7)  # \sum_{i,j,k} W_{ij} W_{jk}

    M[9, widx] = np.sum(W) ** 2  # \sum_{i,j,k,l} W_{ij} W_{kl}

    M[10, widx] = np.trace(W1)  # \sum_{i} W_{ii}^2

    M[11, widx] = np.trace(W) ** 2  # \sum_{i,j,k} W_{ii} W_{jj}

    M[12, widx] = (np.sum(W) * np.sum(Wd))  # \sum_{i,j,k} W_{ii} W_{jk}

    # Higher order matrix sums

    # Cubic terms

    W9 = W ** 3
    M[13, widx] = np.trace(W9)  # \sum_{i} W_{ii}^3  :  1-node case - Graph 1

    M[14, widx] = np.sum(W9)  # \sum_{i,j} W_{ij}^3   :  2-node case - Graph 2
    W10 = np.dot(np.dot(W, W), W)
    M[15, widx] = np.trace(W10)  # \sum_{i,j,k} W_{ij} W_{jk} W_{ki} : 3 node case - Graph
     3

    M[16, widx] = np.sum((W.sum(axis=1)) * (W.sum(axis=0)) * Wd)  # \sum_{i,j,k} M_{ij}M_{
    jj}M_{jk} : 3 node-two case
    # - Graph 4
```

```
 79
 80      M[17, widx] = np.sum(W) * (np.trace(W) ** 2)  # \sum_{i,j,k,l} W_{ij}W_{kk}W_{ll} : 4-
        node case - Graph 5
 81
 82      M[18, widx] = (np.trace(W) * np.sum(W7))  # \sum_{i,j,k,l} M_{ij}M_{jk}M_{ll} : # 4
        node-two case - Graph 6
 83
 84      M[19, widx] = (np.sum(W) ** 2) * (np.trace(W))  # \sum_{i,j,k,l,m} W_{ij}W_{kl}W_{mm}
        : 5 node case - Graph 7
 85
 86      M[20, widx] = np.sum(W) ** 3  # \sum_{i,j,k,l,m,n} W_{ij} W_{kl} W_{mn} :  6-node case
         - Graph 8
 87
 88      # Quartic terms
 89      M[21, widx] = (np.sum(W)) ** 3 * np.trace(W)  # \sum_{i,j,k,l,m,n,o,o}} W_{ij} W_{kl}
        W_{mn} W_{oo} : 7 node case
 90      # - Graph 9
 91
 92      M[22, widx] = np.sum(W) ** 4  # \Sum_{i,j,...,p} W_{ij} W_{kl} W_{mn} W_{op} 8 node
        case - Graph 10
 93
 94  ts.close()
 95
 96  # M1 holds the expectation values corresponding to each matrix sum
 97  M1 = np.mean(M, axis=1)
 98
 99  # Setting these expectation values as a list
100  exp_list = []
101  for k in range(M1.shape[0]):
102      exp_list.append(M1[k])
103
104  print(" \n Experimental expectation values:")
105  print(exp_list)
106
107  # Separating the linear and quadratic exp. vals. (mat_exp) from the cubic and quartic exp.
         vals. (mat_exp_cuqu)
108  mat_exp = exp_list[0:13]
109  mat_exp_cuqu = exp_list[13:24]
```

It begins by importing all of the necessary modules for the program, then includes input statements in lines 5-20 so that a user running the program can define the dimension they are working at, as well as the location of the matrix data shelve. So running this program to analyse the previous example's matrix data shelve, "dim-30-mat-10", the user would be prompted to "Enter the dimension of the matrices: " first, to which they would type the number 30, and press enter, see figure 1. After this, they will be prompted to "Enter the name of the shelve file: " and similarly they would type dim-30-mat-10, and press enter. The code will then execute and results would be produced. Note that the dimensional input is important for Part two, where it is explicitly needed in the 13 parameter computations.



```
D:\Ratio Python Files>python ratios.py
Enter the dimension of the matrices: 30
```

Figure 1: Running the ratios.py program through the terminal. The user is prompted to enter a dimension, here the chosen dimension is 30.

Lines 22-25 open the data matrix shelve and sets its keys as *dic_key* and defines the array M which will hold all of the 23 possible matrix sums, for each data matrix in the set of matrices. The for statement of Line 28 onwards then iterates over each data matrix, computing and storing the matrix sum data. The shelve is then closed and line 97 expresses the mean calculation which produces the required expectation values. To be clear, for each of the 23 matrix sums, the value of these matrix sums for each data matrix is added together and divided by the number of matrices in the dictionary, providing 23 individual expectation values. This data is stored in M1. The final lines of the code, lines 99-109, simply set these expectation values into list format via *exp_list*. This list is consequently split into two new lists: the first containing the linear and quadratic matrix expectation values called *mat_exp*, and the second containing the cubic and quartic matrix expectation values called *mat_exp_cuqu*.

Part two

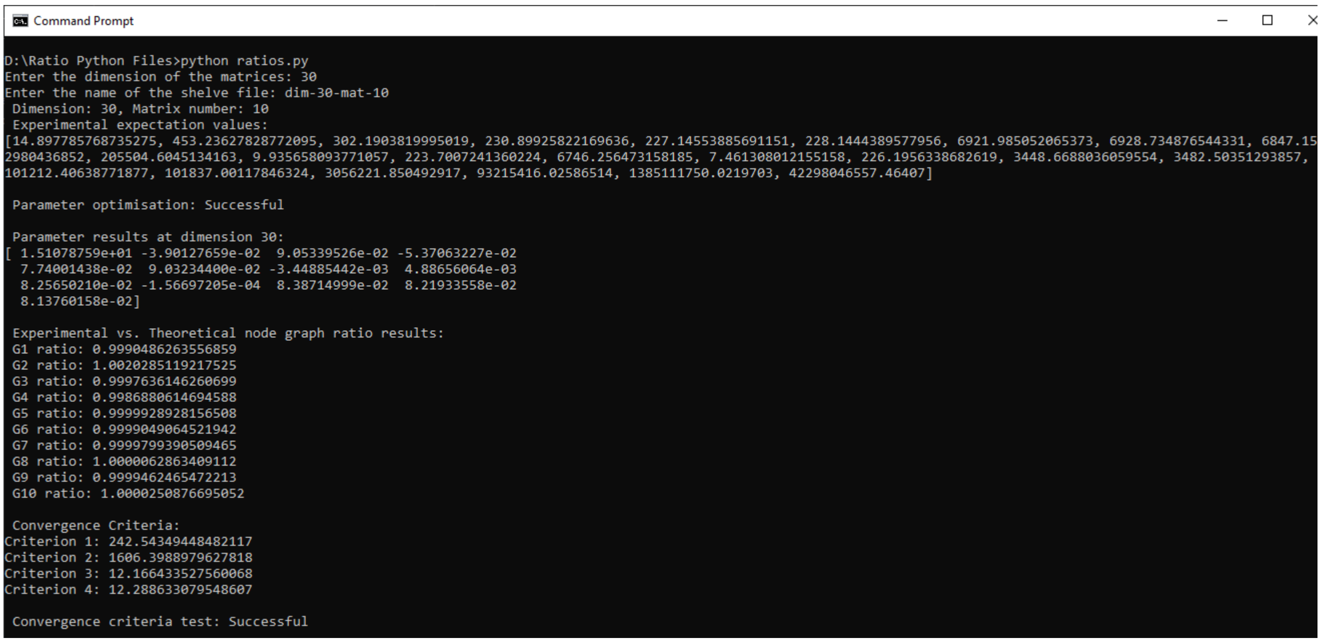
The code that executes part two is as follows:

```python
# Defining a function that contains all 13 equations in the PIGMM paper. By including the experimental expectation
# values, the 13 parameters of the model can be solved for.


def f(y):
    f1 = y[0] + (np.sqrt(D-1))*y[1] - float(mat_exp[0])  # <sum_{i} M_{ii}>
    f2 = D*y[0] - float(mat_exp[1])  # <sum_{i,j} M_{ij}>
    f3 = y[0]**2 + y[1]**2 + y[2] + y[4] + (D-1)*y[8] + (D-1)*y[10] + (D-1)*y[5] + ((D*(D-3))/2)*y[11] +\
        (((D-1)*(D-2))/2)*y[12] - float(mat_exp[2])  # <sum_{i,j} M_ij M_ij>
    f4 = ((D*(D-3))/2)*y[11] - (((D-1)*(D-2))/2)*y[12] + 2*(D-1)*y[6] + (D-1)*y[10] + y[2] + y[4] + y[0]**2 +\
        y[1]**2 - float(mat_exp[3])  # <sum_{i,j} M_ij M_ji>
    f5 = y[2] + np.sqrt(D-1)*y[3] + (D-1)*y[6] + (D-1)*y[8] + (D-1)*(np.sqrt(D-2))*y[9] + y[0]**2 +\
        y[0]*y[1]*(np.sqrt(D-1)) - float(mat_exp[4])  # <sum_{i,j} M_ii M_ij>
    f6 = y[2] + np.sqrt(D-1)*y[3] + (D-1)*y[6] + (D-1)*y[5] + (D-1)*(np.sqrt(D-2))*y[7] + y[0]**2 +\
        y[0]*y[1]*(np.sqrt(D-1)) - float(mat_exp[5])  # <sum_{i,j} M_ii M_ji>
    f7 = D*y[2] + D*(D-1)*y[8] + D*(y[0]**2) - float(mat_exp[6])  # <sum_{i,j,k} M_ij M_ik>
    f8 = D*y[2] + D*(D-1)*y[5] + D*(y[0]**2) - float(mat_exp[7])  # <sum_{i,j,k} M_ij M_kj>
    f9 = D*y[2] + D*(D-1)*y[6] + D*(y[0]**2) - float(mat_exp[8])  # <sum_{i,j,k} M_ij M_jk>
    f10 = (D**2)*y[2] + (D**2)*(y[0]**2) - float(mat_exp[9])  # <sum_{i,j,k,l} M_ij M_kl>
    f11 = (D**-1)*y[2] + ((D-1)/D)*y[4] + 2*((np.sqrt(D-1))/D)*y[3] + ((D-1)/D)*y[5] + ((D-1)/D)*y[8] +\
        ((D-1)/D)*(D-2)*y[10] + 2*((D-1)/D)*y[6] + 2*((D-1)/D)*(np.sqrt(D-2))*y[7] +\
        2*((D-1)/D)*(np.sqrt(D-2))*y[9] + ((y[0]**2)/D) + 2*((np.sqrt(D-1))/D)*y[0]*y[1] +\
        ((D-1)/D)*(y[1]**2) - float(mat_exp[10])  # <sum_{i} M_ii M_ii OR (M_ii)^2>
    f12 = y[2] + (D-1)*y[4] + 2*(np.sqrt(D-1))*y[3] + y[0]**2 + 2*(np.sqrt(D-1))*y[0]*y[1] +\
        (D-1)*(y[1]**2) - float(mat_exp[11])  # <sum_{i,j} M_ii M_jj>
    f13 = D*y[2] + D*(np.sqrt(D-1))*y[3] + D*(y[0]**2) +\
        D*(np.sqrt(D-1))*y[0]*y[1] - float(mat_exp[12])  # <sum_{i,j,k} M_{ii} M_{jk}>

    return [f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13]


x = fsolve(f, [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

# x is an array containing the solved model parameters.
# Each parameter as defined in PIGMM is associated to each x[i] as follows:
# x[0] = tilde(mu)_1
# x[1] = tilde(mu)_2
# x[2] = (LamV0^-1)_11
# x[3] = (LamV0^-1)_12
# x[4] = (LamV0^-1)_22
# x[5] = (LamVH^-1)_11
# x[6] = (LamVH^-1)_12
# x[7] = (LamVH^-1)_13
# x[8] = (LamVH^-1)_22
# x[9] = (LamVH^-1)_23
# x[10] = (LamVH^-1)_33
# x[11] = (LamV2^-1)
# x[12] = (LamV3^-1)

# opt_check describes how well the optimisation using fsolve has worked. If the process fails, the user is notified.
opt_check = np.isclose(f(x), [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
for p in opt_check:
    if not p:
        print("\n Parameter optimisation: Unsuccessful, fsolve procedure has failed.")
        sys.exit()
    else:
        continue
print("\n Parameter optimisation: Successful")

print(" \n Parameter results at dimension " + str(D) + ":")
print(x)

# Theoretical expectation values - the following graph labelling system is used to match that of the GTMDS paper.

# G1: Exp Val for sum_{i} M_{ii}^3
G1 = 3*((x[0]/D) + ((np.sqrt(D-1))/D)*x[1]) * ((x[2]/D) + ((D-1)/D)*x[4] + 2*(np.sqrt(D-1)/D)*x[3] + ((D-1)/D)*x[5] +\
    ((D-1)/D)*x[8] + ((D-1)/D)*(D-2)*x[10] + 2*((D-1)/D)*x[6] + 2*((D-1)/D)*(np.sqrt(D-2))*x[7] +\
    2*((D-1)/D)*(np.sqrt(D-2))*x[9]) + (D**-2)*((x[0] + (np.sqrt(D-1))*x[1])**3)

# G2: Exp Val for sum_{i,j} M_{ij}^3
G2 = (x[0]**3)/D + (3/D)*x[0]*(x[1]**2) + ((D-2)/(D*(np.sqrt(D-1))))*(x[1]**3) + 3*(x[0]/D)*(x[2] + x[4] +\
    (D-1)*x[8] + (D-1)*x[10] + (D-1)*x[5] + ((D*(D-3))/2)*x[11] + (((D-1)*(D-2))/2)*x[12]) +\
    3*x[1]*x[4]*((D-2)/(D*(np.sqrt(D-1)))) + (6*x[1]*x[3])/D + 3*x[1]*x[10]*((D-3)/D)*(np.sqrt(D-1)) +\
    6*x[1]*x[6]*((np.sqrt(D-1))/D) + 6*x[1]*x[7]*((np.sqrt((D-1)*(D-2)))/D) +\
    6*x[1]*x[9]*((np.sqrt((D-1)*(D-2)))/D) + 3*x[1]*x[11]*((-D**2 + 3*D)/(2*D*np.sqrt(D-1))) -\
    (3/2)*x[1]*x[12]*(((D-2)*np.sqrt(D-1))/D)

# G3: Exp Val for sum_{i,j,k} M_{ij} M_{jk} M_{ki}
G3 = x[0]**3 + (x[1]**3)/(np.sqrt((D-1))) + 3*x[0]*(x[2] + (D-1)*x[6]) + 3*(x[1]/(np.sqrt(D-1)))*x[4] +\
    3*x[1]*(np.sqrt(D-1))*x[10] + 3*(x[1])*(np.sqrt(D-1))*x[6] + 3*x[1]*x[11]*((D*(D-3))/(2*(np.sqrt(D-1)))) -\
    3*x[1]*x[12]*(((D-2)*(np.sqrt(D-1)))/2)

# G4: Exp Val for sum_{i,j,k} M_{ij} M_{jj} M_{jk}
G4 = x[0]*x[2] + np.sqrt(D-1)*x[0]*x[3] +(D-1)*x[0]*x[5] + (D-1)*x[0]*x[6] +(D-1)*np.sqrt(D-2)*x[0]*x[7] + x[0]*x[2] +\
     (D-1)*x[0]*x[6] + np.sqrt(D-1)*x[1]*x[2] + ((D-1)**(3/2))*x[1]*x[6] + x[0]*x[2] + np.sqrt(D-1)*x[0]*x[3] +\
     (D-1)*x[0]*x[6] + x[0]*(D-1)*x[8] + (D-1)*np.sqrt(D-2)*x[0]*x[9] + (x[0]**3) + (x[0]**2)*x[1]*np.sqrt(D-1)

# G5: Exp value for sum_{i,j,k,l} M_{ij}M_{kk}M_{ll}
G5 = 2*(D*x[2] + D*(np.sqrt(D-1))*x[3])*(x[0] + np.sqrt(D-1)*x[1]) + (x[2] + (D-1)*x[4] +\
     2*np.sqrt(D-1)*x[3] + ((x[0] + np.sqrt(D-1)*x[1])**2)) * D*x[0]

# G6: Exp value for sum_{i,j,k,l} M_{ij}M_{jk}M_{ll}
G6 = 3*D*x[0]*x[2] + D*np.sqrt(D-1)*x[2]*x[1] + D*(D-1)*x[6]*x[0] + D*((D-1)**(3/2))*x[6]*x[1] +\
     2*D*(np.sqrt(D-1))*x[0]*x[3] + D*(x[0]**3) + D*np.sqrt(D-1)*(x[0]**2)*x[1]

# G7: Exp val for sum_{i,j,k,l,m} M_{ij}M_{kl}M_{mm}
G7 = ((D**2)*x[2])*(x[0] + np.sqrt(D-1)*x[1]) + 2*(D*x[2] + D*np.sqrt(D-1)*x[3])*D*x[0] + ((D*x[0])**2)*(x[0] +\
     np.sqrt(D-1)*x[1])

# G8: Exp Val for sum_{i,j,k,l,m,n} M_{ij} M{kl} M{mn}
G8 = 3*x[0]*(D**3)*x[2] + (x[0]**3)*(D**3)

```

```
103 # G9: Exp Val for sum_{i_(1,2,3,4,5,6,7)} M_{i_(1,2)} M_{i_(3,4)} M_{i_(5,6)}M_{i_(7,7)}
104 G9 = 3*((D**2)*x[2]*(D*x[2] + D*np.sqrt(D-1)*x[3])) + 3*((D**2)*x[2]*(D*x[0])*(x[0] + np.sqrt(D-1)*x[1]) +\
105     (D*x[2] + D*np.sqrt(D-1)*(x[3]))*(D*x[0])**2) + ((D*x[0])**3)*(x[0] + np.sqrt(D-1)*x[1])
106 
107 # G10: Exp Val for sum_{i_(1,2,3,4,5,6,7,8)} M_{i_(1,2)} M_{i_(3,4)} M_{i_(5,6)}M_{i_(7,8)}
108 G10 = 3*(D**4)*(x[2]**2) + 6*(D**4)*x[2]*(x[0]**2) + (D*x[0])**4
109 
110 # The ratios for each graph are then printed.
111 print("\n Experimental vs. Theoretical node graph ratio results:")
112 
113 print(" G1 ratio:", G1/float(mat_exp_cuqu[0]), "\n G2 ratio:", G2/float(mat_exp_cuqu[1]),
114       "\n G3 ratio:", G3/float(mat_exp_cuqu[2]), "\n G4 ratio:", G4/float(mat_exp_cuqu[3]),
115       "\n G5 ratio:", G5/float(mat_exp_cuqu[4]), "\n G6 ratio:", G6/float(mat_exp_cuqu[5]),
116       "\n G7 ratio:", G7/float(mat_exp_cuqu[6]), "\n G8 ratio:", G8/float(mat_exp_cuqu[7]),
117       "\n G9 ratio:", G9/float(mat_exp_cuqu[8]), "\n G10 ratio:", G10/float(mat_exp_cuqu[9]))
118 
119 # Calculation of the convergence criteria.
120 print("\n Convergence Criteria:")
121 
122 
123 v0_power_of_neg_1 = np.array([[x[2], x[3]],
124                               [x[3], x[4]]])
125 
126 V0 = np.linalg.inv(v0_power_of_neg_1)
127 det_V0 = np.linalg.det(V0)
128 print("Criterion 1:", det_V0)
129 
130 vh_power_of_neg_1 = np.array([[x[5], x[6], x[7]],
131                               [x[6], x[8], x[9]],
132                               [x[7], x[9], x[10]]])
133 
134 VH = np.linalg.inv(vh_power_of_neg_1)
135 det_VH = np.linalg.det(VH)
136 print("Criterion 2:", det_VH)
137 
138 Lam_V2 = (x[11]**-1)
139 Lam_V3 = (x[12]**-1)
140 print("Criterion 3:", Lam_V2)
141 print("Criterion 4:", Lam_V3)
142 
143 if det_V0 >= 0 and det_VH >= 0 and Lam_V2 >= 0 and Lam_V3 >= 0:
144     print("\n Convergence criteria test: Successful")
145 else:
146     print("\n Convergence criteria test: Unsuccessful")
```

Lines 5-32 in the part two code block above is where the 13 parameters of the Gaussian matrix model are identified. It takes the 13 theoretically derived equations in Section 3 of “Permutation Invariant Gaussian Matrix Models” and sets them in a Python defined function f(y). The Python module fsolve is then used to solve these 13 equations. Essentially it attempts to set all of f1, f2, ..., f13 to zero by assigning certain values to y[i] with i = 0,...,12. This is solvable since the linear and quadratic experimental expectation values are used from list “mat_exp”, defined in part one. Line 32 produces our output parameters as a (13,) array. The commented out section in lines 34-48 is simply a reference to the labels parameters are given in PIGMM.

Lines 50-60 tells the user how well fsolve performed, by asking whether the functions f1, f2... e.t.c are close to zero. If the functions are suitably close to zero, the array *opt_check* will give assign that function a True value and False if not. For successful optimisation and for the program to finish running, all functions must meet the True criteria. The user is notified if this optimisation has been successful or has failed.



```
D:\Ratio Python Files>python ratios.py
Enter the dimension of the matrices: 30
Enter the name of the shelve file: dim-30-mat-10
 Dimension: 30, Matrix number: 10
 Experimental expectation values:
[14.897785768735275, 453.23627828772095, 302.1903819995019, 230.89925822169636, 227.14553885691151, 228.1444389577956, 6921.985052065373, 6928.734876544331, 6847.15
2980436852, 205504.6045134163, 9.935658093771057, 223.7007241360224, 6746.256473158185, 7.461308012155158, 226.1956338682619, 3448.6688036059554, 3482.50351293857,
101212.40638771877, 101837.00117846324, 3056221.850492917, 93215416.02586514, 1385111750.0219703, 42298046557.46407]

 Parameter optimisation: Successful

 Parameter results at dimension 30:
[ 1.51078759e+01 -3.90127659e-02  9.05339526e-02 -5.37063227e-02
  7.74001438e-02  9.03234400e-02 -3.44885442e-03  4.88656064e-03
  8.25650210e-02 -1.56697205e-04  8.38714999e-02  8.21933558e-02
  8.13760158e-02]

 Experimental vs. Theoretical node graph ratio results:
 G1 ratio: 0.9990486263556859
 G2 ratio: 1.0020285119217525
 G3 ratio: 0.9997636146260699
 G4 ratio: 0.9986880614694588
 G5 ratio: 0.9999928928156508
 G6 ratio: 0.9999049064521942
 G7 ratio: 0.9999799390509465
 G8 ratio: 1.0000062863409112
 G9 ratio: 0.9999462465472213
 G10 ratio: 1.0000250876695052

 Convergence Criteria:
Criterion 1: 242.54349448482117
Criterion 2: 1606.3988979627818
Criterion 3: 12.166433527560068
Criterion 4: 12.288633079548607

 Convergence criteria test: Successful
```

Figure 2: The ratios.py results from using the dim-30-mat-10 shelve matrix data.

Lines 63-108 of the part 2 code contain the cubic and quartic theoretical expectation value equations (denoted by graph number), building these from the 13 parameters identified using fsolve, and the theoretical method set in PIGMM.

Lines 113-117 then take the ratios between theoretical and experimental expectation values as required. To do this, the code uses the list *mat_exp_cuqu* that was defined in part one, which contains the experimental cubic and quartic expectation values.

The final lines 119-146 evaluate the convergence criteria of the Gaussian model. The specifics of these constraints are detailed in section 2 of PIGMM. It requires that the matrices Λ_{V_0} and Λ_{V_H} (populated by the associated parameters) are positive semi-definite and that $\Lambda_{V_3} \geq 0$ and $\Lambda_{V_2} \geq 0$ also. Since the model finds the inverse parameters, note that the inverse matrices are constructed first e.g. $\Lambda_{V_0}^{-1}$ and then inverted to then identify the criteria. If all 4 criteria conditions are met, the user is notified that the test is successful and likewise when the test is unsuccessful if any of the criteria fail.

Figure 2 displays the output of the program when using the example shelve "dim-30-mat-10". The shelve was created using "create_sample_shelve.py" in Python 3.8.5, and "ratios.py" was run using the same version.

3.3 Using shelves in ratios.py

Shelves changed output format between Python 2 and Python 3 (see the create_shelve_sample discussion in section 2.2) hence small adaptations must be made when creating and using them. If a shelve was generated in Python 2, the "ratios.py" program should also be run in the same Python 2 interpreter . The same reasoning applies to Python 3. Having produced a shelve via "create_shelve_sample.py" in Python 2, in order to execute the ratios.py program using this shelve correctly, one change is required: the user must include quotation marks around the input file name. I.e. when the user is given prompt "Enter the name of the shelve file", for some arbitrary shelve file, say test.shelve, the input must be "test.shelve".

4 The SVD-JJ-0700-to-50.shelve file

This file is a shelve of matrix data obtained from the linguistic corpora used to build the dataset in "Lingusitic matrix theory". It has been constructed using singular value decomposition (SVD) to reduce the size and contains matrices associated to adjective words. More specifically, the shelve contains 273 adjective word matrices, all of which are 50×50 in size (having been reduced from 700×700 by the SVD process). This shelve was produced in Python 2.7 and therefore ratios.py must also use Python 2.7 to extract the expectation values.