Υπολογιστική Φυσική - 1η Σειρά Ασκήσεων

Αναστάσιος Τζαβέλλας / AM:1110 2016 00255

07/11/2017

1η Άσκηση

Το επίπεδο σώμα εκτείνεται μόνο σε 2 διαστάσεις, στο χωρίο

$$D = \left\{(x, y) : x \in [0, 2], y \in [0, 2], x^2 + y^2 > 1, x + y < 2\right\}$$

Η μάζα και το κέντρο μάζας του σώματος δίνονται από τα ολοκληρώματα (1), (2), (3). Είναι προφανές από τα όρια του χωρίου D, ότι οι συντεταγμένες του κέντρου μάζας θα είναι ίσες.

$$m = \int_D 1 + x^2 + y^2 \mathrm{d}x\mathrm{d}y \tag{1}$$

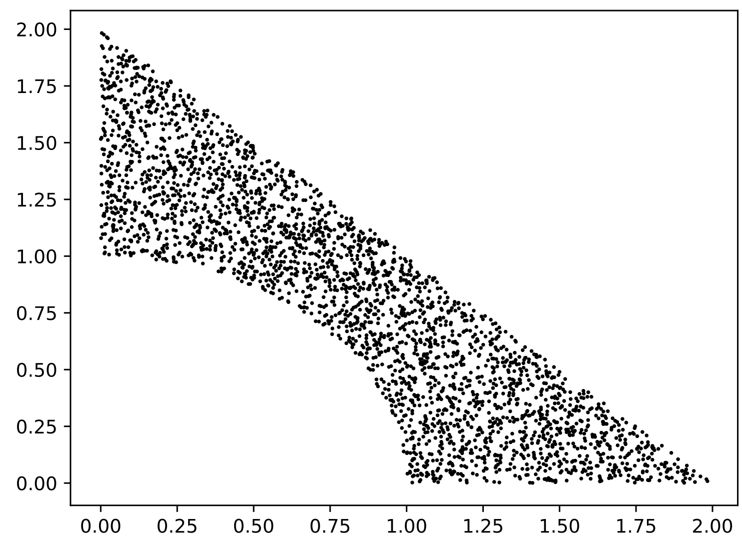
$$x = \frac{1}{m} \int_D x \left(1 + x^2 + y^2\right) \mathrm{d}x\mathrm{d}y \tag{2}$$

$$y = \frac{1}{m} \int_D y \left(1 + x^2 + y^2\right) \mathrm{d}x\mathrm{d}y \tag{3}$$

Από αναλυτικό υπολογισμό των ολοκληρωμάτων, που έγινε με χρήση του πακέτου Mathematica, αναμένεται $m = 3.4885$, $x_{cm} = 0.84084$ και $y_{cm} = 0.84084$. Η μέθοδος της απόρριψης για 10000 δείγματα δίνει τα εξής: $m = 3.53 \pm 0.05$, $x_{cm} = 0.833 \pm 0.013$ και $y_{cm} = 0.846 \pm 0.013$.

Όπως φαίνεται από τη σύγκριση των τιμών, οι υπολογισμοί Monte Carlo με τα απόλυτα σφάλματά τους είναι κοντά στις αναλυτικές λύσεις των ολοκληρωματων. Η μικρή διαφορά στην τιμή των συντεταγμένων του κέντρου μάζας οφείλεται στη δειγματοληψία διαφορετικών γεννητριών τυχαίων αριθμών για τα x και y. Ωστόσο, το σφάλμα των μέσων τιμών δεν αλλάζει καθώς οι γεννήτριες έχουν τα ίδια χαρακτηριστικά.

Κατά την εκτέλεση του προγράμματος, παράχθηκαν τα σημεία που φαίνονται στο Σχήμα 1.



Σχήμα 1: Σημεία εκτέλεσης Monte Carlo

Παρακάτω ακολουθεί ο κώδικας που γράφτηκε σε Python και έγινε χρήση της βιβλιοθήκης Numpy.

mc_rejection.py

```
import numpy as np
import matplotlib.pyplot as plt

# ------------------ Start of the program ------------------

N = 10000       # Number of samples
xInterval = np.array([0, 2])  # Interval of x values
yInterval = np.array([0, 2])  # Interval of y values
```

```python
area = xInterval[1] * yInterval[1]  # Total area of Rejection method

x = np.random.uniform(0, xInterval[1], N)  # Generate N random values in U(0, xInterval[1])
y = np.random.uniform(0, yInterval[1], N)  # Generate N random values in U(0, yInterval[1])

inside = np.where( (x+y<2) & (x*x+y*y>1) )  # inside contains the indices of the x and y samples
                 # that lie within  the boundaries of the object
density = 1+ np.power(x[inside], 2) + np.power(y[inside], 2) # density contains the density values at the above indices

sumMass = np.sum(density)       # The sumMass is the sum of the density values
sumSigmaMass = np.sum(density * density)   # The sumSigmaMass is the sum of the square of the density values

sumXcm = np.sum(x[inside]*density)       # The sumXcm is the sum of the x*density values
sumSigmaXcm = np.sum((x[inside]*density) * (x[inside]*density)) # The sumSigmaXcm is the sum of the square of the x*density values

sumYcm = np.sum(y[inside]*density)     # The sumYcm is the sum of the y*density values
sumSigmaYcm = np.sum((y[inside]*density) * (y[inside]*density)) # The sumSigmaYcm is the sum of the square of the y*density values

mass = area * sumMass / N       # The mass of the object
errorMass = mass * np.sqrt(1-inside[0].size/N) / np.sqrt(inside[0].size) # The error of the calculation

x_cm = (area * sumXcm/N) / mass      # The x coordinate of the center of mass of the object
errorXcm = x_cm * np.sqrt(1-inside[0].size/N) / np.sqrt(inside[0].size) # The error of the calculation
```

```
y_cm = (area * sumYcm/N) / mass     # The y coordinate of the center of mass of the object
errorYcm = y_cm * np.sqrt(1-inside[0].size/N) / np.sqrt(inside[0].size) # The error of the calculation

print("Monte Carlo Rejection Method (", N, " samples)\n")
print('mass = {:.3f} +- {:.3f}'.format(mass, errorMass))
print('x_cm = {:.3f} +- {:.3f}'.format(x_cm, errorXcm))
print('y_cm = {:.3f} +- {:.3f}'.format(y_cm, errorYcm))

print("\nAnalytic Evaluation for Comparison\n")
print('mass = 3.48857')
print('x_cm = 0.840841')
print('y_cm = 0.840841')

plt.plot(x[inside], y[inside], 'k.', markersize=2)
plt.show()
```

2η Άσκηση

Η άσκηση απαιτεί υπολογισμό του ολοκληρώματος 4, κάνοντας χρήση της απλής μεθόδου Monte Carlo. Το αποτέλεσμα του ολοκληρώματος έγινε αριθμητικά με χρήση του πακέτου Mathematica, και προέκυψε ίσο με $I = 1.72 \cdot 10^7$. Η μεγάλη τιμή του ολοκληρώματος είναι αναμενόμενη, καθώς η ολοκληρωτέα συνάρτηση αυξάνεται με μεγάλο ρυθμό στο διάστημα ολοκλήρωσης.

$$I = \int_0^L r^2 \left(1 + e^{r^2}\right) \mathrm{d}r \tag{4}$$

Ο υπολογισμός Monte Carlo παρήγαγε αποτέλεσμα $I = (17.2 \pm 0.2) \cdot 10^6$, που είναι πολύ κοντά στο αριθμητικό αποτέλεσμα. Επιπλέον, μολονότι το απόλυτο σφάλμα φαίνεται μεγάλο, το σχετικό σφάλμα είναι μόνο 1.16%, άρα μικρό.

Αν γίνει ολοκλήρωση σε μέρη, αναμένεται η ίδια τιμή ολοκληρώματος αλλά μικρότερο σφάλμα. Αυτό οφείλεται στο γεγονός ότι το ολοκλήρωμα κάθε μέρους αποτελεί στην ουσία μια μέτρηση στην οποία αντιστοιχεί ένα σφάλμα. Η μέτρηση δίνεται από την εξίσωση (5) και το σφάλμα από την εξίσωση (6).

$$I_i = (x_{i+1} - x_i) \sum_{i=1}^{N_i} f(r_i) \tag{5}$$

$$\delta I_i = \frac{x_{i+1} - x_i}{\sqrt{N_i}} \sqrt{\frac{1}{N_i} \sum_{i=1}^{N_i} f^2(r_i) - \left(\frac{1}{N_i} \sum_{i=1}^{N_i} f(r_i) \right)^2} \tag{6}$$

Αν το τελικό αποτέλεσμα είναι $I = \sum_{i=1}^{M} I_i$, τότε το σφάλμα του είναι $\delta I = \sqrt{\sum_{i=1}^{M} \delta I_i^2}$, σύμφωνα με το νόμο διάδοσης σφαλμάτων. Επομένως, το σφάλμα του ολοκληρώματος θα οφείλεται κυρίως στο μέρος των μετρήσεων που αντιστοιχούν στο πιο μεγάλο δI_i, και όχι ισομερώς σε όλες τις μετρήσεις όπως στην απλή περίπτωση Monte Carlo.

Πράγματι για ολοκλήρωση σε 4 μέρη, προκύπτει $I = (17.20 \pm 0.19) \cdot 10^6$, δηλαδή σχετικό σφάλμα 1.10%. Τέλος, για ολοκλήρωση σε 16 μέρη προκύπτει $I = (17.20 \pm 0.11) \cdot 10^6$ και σχετικό σφάλμα 0.64%. Παρατηρείται ότι, η αύξηση του αριθμού των μερών οδηγεί σε μείωση του σφάλματος της μεθόδου, χωρίς υπολογιστική επιβάρυνση, μιας και εκτελείται περίπου ο ίδιος αριθμός πράξεων κινητής υποδιαστολής.

Παρακάτω ακολουθεί ο κώδικας που γράφτηκε σε Python και έγινε χρήση της βιβλιοθήκης Numpy.

mc_MomentOfInertia.py

```python
import numpy as np

# ------------------ Function Definitions ------------------

def partialSums(Lmin, Lmax, N):
    """
    Calculates the sum values of function r^2 * (1 + exp(r^2))
    as well as the sum of their squares, at interval [Lmin, Lmax
    ]

    Args:
        Lmin: The lower bound of the interval
        Lmax: The upper bound of the interval
        N: The number of random samples
```

```
    Returns:
        A dictionary with the sum of the
        values and the sum of their squares
    """
    r = np.random.uniform(Lmin, Lmax, N)      # Generate N random samples from U(Lmin, Lmax)
    f = np.power(r, 2) * (1 + np.exp(np.power(r, 2)))  # Evaluate integrant at these points
    meanSum = np.sum(f)          # Store their sum in meanSum - Will be used to compute mean value
    sigmaSum = np.sum(f*f)          # Store the sum of their squares in sigmaSum - Will be used to compute variance
    return {'meanSum':meanSum, 'sigmaSum':sigmaSum}

def StratifiedMonteCarlo(Lmin, Lmax, N, strata):
    """
    Performs Stratified Monte Carlo Integration
    or Crude Monte Carlo integration, if strata=1
    at interval [Lmin, Lmax]

    Args:
        Lmin: The lower bound of the interval
        Lmax: The upper bound of the interval
        N: The number of random samples
        strata: The number of strata

    Returns:
        A dictionary with the integral value
        and the error value of the integrals
    """
    r = np.linspace(Lmin, Lmax, strata + 1)    # Split integration interval in strata number of sub-intervals
    Ni = N // strata          # N/strata number of samples will be generated within that interval
    integral = np.zeros(strata)       # Initialize integral and error vectors that store the result of each interval
    error = np.zeros(strata)
    for i in range(strata):
        ret = partialSums(r[i], r[i+1], Ni)    # Calculate
```

```
    meanSum and sigmaSum
        mean = ret['meanSum'] / Ni       # mean holds the mean value of the samples
        integral[i] =  mean * (r[i+1]-r[i])     # integral[i] holds the integral of the sub-interval
        error[i] =  np.sqrt( ret['sigmaSum']/Ni - np.power(mean, 2)) * (r[i+1]-r[i]) / np.sqrt(Ni) # error[i] holds the error of the sub-interval
            # The total integral is the sum of the integrals of every sub-interval I = I1 + ... + Ik
            # The total error is calculated by the error propagation rule DI = sqrt( DI1^2 + .. + DIk^2)
    return {'integral': np.sum(integral), 'error': np.sqrt(np.sum(error * error))}

# ------------------ Start of the program ------------------
interval = np.array([0, 4])  # interval of integration
N = 100000       # Number of Samples

strata = 1
result = StratifiedMonteCarlo(interval[0], interval[1], N, strata)      # Crude Monte Carlo, only 1 stratum
print('\nCrude Monte Carlo ({:d} samples)'.format(N))
print('momentInertia = {:.2e} +- {:4.2e}'.format(result['integral'], result['error']))

strata = 4
result = StratifiedMonteCarlo(interval[0], interval[1], N, strata)      # Stratified Monte Carlo wtih 4 strata
print('\nStratified Monte Carlo ({:d} strata)'.format(strata))
print('momentInertia = {:.2e} +- {:4.2e}'.format(result['integral'], result['error']))

strata = 16
result = StratifiedMonteCarlo(interval[0], interval[1], N, strata)      # Stratified Monte Carlo with 16 strata
print('\nStratified Monte Carlo ({:d} strata)'.format(strata))
print('momentInertia = {:.2e} +- {:4.2e}'.format(result['integral'], result['error']))
```

```
print("\nAnalytic Evaluation for Comparison")        # Analytic
    result for compilation
M = 1.71975e7
print('mass = {:.2e}'.format(M))
```

3η Άσκηση

Περιγραφή Προσομοίωσης

Για την προσομοίωση του πειράματος μέτρησης της επιτάχυνσης της βαρύτητας εφαρμόστηκαν δύο μέθοδοι. Αρχικά ορίστηκε ένα διάνυσμα $\vec{L}$ με τα μήκη της εκφώνησης. Σε κάθε μήκος L_i, γίνεται προσομοίωση μέτρησης της περιόδου.

Συγκεκριμένα, κάθε μέτρηση περιόδου, θεωρείται πως προκύπτει από την πραγματική περίοδο (7), για δοσμένη επιτάχυνση βαρύτητας g (πχ $9.8\frac{m}{s^2}$) και μήκος εκκρεμούς L_i, στην οποία έχει προστεθεί ένας τυχαίος αριθμός. Δηλαδή η μέτρηση είναι μια τυχαία μεταβλητή $T_i = T + \delta T_i$, όπου $\delta T_i \sim N(0,1) \cdot \delta T$. Τελικά προκύπτει ένα νέο διάνυσμα $\vec{T}$.

$$T = 2\pi\sqrt{\frac{L}{g}} \tag{7}$$

Από κάθε ζεύγος τιμών (L_i, T_i) υπολογίζεται έμμεσα η τιμή της επιτάχυνσης της βαρύτητας g_i σύμφωνα με την εξίσωση (8), με σφάλμα που δίνεται από το νόμο διάδοσης σφαλμάτων (9).

$$g_i = 4\pi^2\frac{L_i}{T_i^2}, i = 1,2,3,4 \tag{8}$$

$$\delta g_i = 4\pi^2\sqrt{\left(\frac{\delta L}{T_i^2}\right)^2 + \left(\frac{2L_i\delta T}{T_i^3}\right)^2}, i = 1,2,3,4 \tag{9}$$

Η μέση τιμή των g_i δίνει μια εκτίμηση για την επιτάχυνση της βαρύτητας $\bar{g} = \frac{1}{4}\sum_i g_i$, με σφάλμα $\delta g = \frac{1}{4}\sqrt{\sum_i \delta g_i^2}$.

Αν η εξίσωση (7) λυθεί ως προς T^2, τότε προκύπτει η εξίσωση (10), η οποία είναι μια γραμμική συνάρτηση $T^2(L)$, με τη σταθερά g κρυμμένη μέσα στο συντελεστή διεύθυνσης της ευθείας.

$$T^2 = \frac{4\pi^2}{g} L \tag{10}$$

Τα ζεύγη (L_i, T_i^2) μπορούν να χρησιμοποιηθούν για να χαραχτεί μια ευθεία ελαχίστων τετραγώνων, η οποία θα είναι κοντά στην εξίσωση (10). Από την κλίση της ευθείας ελαχίστων τετραγώνων υπολογίζεται η επιτάχυνση της βαρύτητας σύμφωνα με την εξίσωση (11).

$$g = \frac{4\pi^2}{A} \tag{11}$$

Ο αλγόριθμος ευθείας ελαχίστων τετραγώνων δίνει το σφάλμα της κλίσης δA, αλλά αυτό δεν συμπίπτει με το σφάλμα δg. Για τον προσδιορισμό του δg απαιτείται εφαρμογή του νόμου διάδοσης σφαλμάτων στην εξίσωση(11), δηλαδή η εξίσωση (12).

$$\delta g = \frac{4\pi^2}{A^2} \delta A \tag{12}$$

Το πείραμα προσομοιώνεται δύο φορές, όμως στη δεύτερη εκτέλεση υπάρχει ένα σταθερό συστηματικό σφάλμα στη μέτρηση του μήκους του εκκρεμούς, $\delta L_{sys} = 0.05m$. Για καλύτερη σύγκριση των δύο μεθόδων, δεν επαναλαμβάνονται η μετρήσεις της περιόδου.

Αναμένεται η ευθεία ελαχίστων τετραγώνων να είναι μετατοπισμένη κατά $0.05m$ δεξιά.

Αποτελέσματα Προσομοίωσης

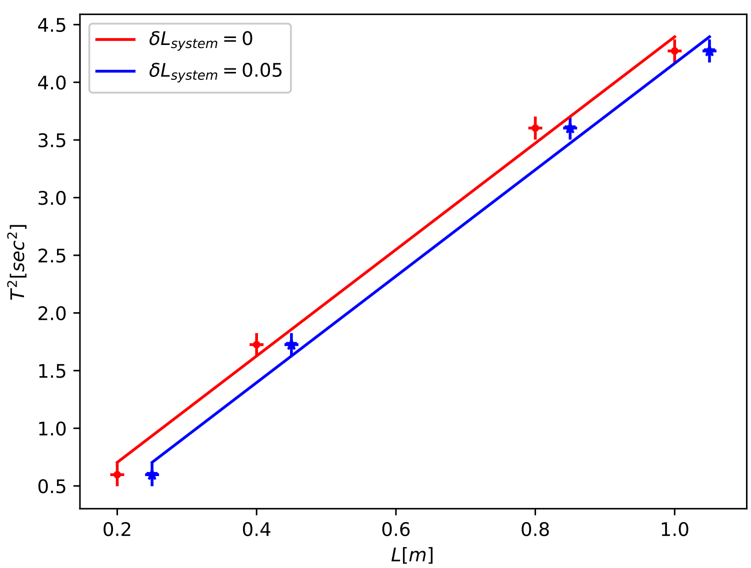
Τα αποτελέσματα της προσομοίωσης είναι συγκεντρωμένα στον Πίνακα 1. Η θεωρητική τιμή της επιτάχυνσης της βαρύτητας είχε τεθεί ως $9.8\frac{m}{s^2}$ αλλά παρατηρείται ότι και οι δύο μέθοδοι έχουν σημαντική απόκλιση της τάξης του 10%. Η απόκλιση μπορεί να δικαιολογηθεί από το σφάλμα μέτρησης του χρόνου της περιόδου που κυμαίνεται από 5% - 13%.

Στο πρώτο πείραμα και οι δύο μέθοδοι βρίσκουν περίπου την ίδια τιμή για την επιτάχυνση της βαρύτητας αλλά το σφάλμα της μεθόδου ελαχίστων τετραγώνων είναι μεγαλύτερο από αυτό της μέσης τιμής. Αυτή η κακή επίδοσης της μεθόδου ελαχίστων τετραγώνων, οφείλεται στην χρήση του κανόνα διάδοσης σφαλμάτων (12).

δL_{syst}	Μέση Τιμή	Ελάχιστα Τετράγωνα
0	10.4 ± 0.9	10.2 ± 1.9
0.05	11.8 ± 1.1	10.2 ± 1.9

Πίνακας 1: Αποτελέσματα Προσομοίωσης

Ωστόσο, μολονότι η μέθοδος ελαχίστων τετραγώνων έχει μεγαλύτερη ανακρίβεια, υπερτερεί της μέσης τιμής γιατί είναι αναίσθητη σε συστηματικά σφάλματα. Όπως φαίνεται από το Σχήμα 2, αλλά και τις μετρήσεις του Πίνακα 1, το συστηματικό σφάλμα δεν μετέβαλλε την τιμή του στατιστικού σφάλματος της ευθείας ελαχίστων τετραγώνων.



Σχήμα 2: Ελάχιστα Τετράγωνα Προσομοίωσης

Πηγαίος Κώδικας

Παρακάτω ακολουθεί ο κώδικας που γράφτηκε σε Python και έγινε χρήση της βιβλιοθήκης Numpy.

mc_experiment.py

```python
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def line(x, a, b):
    """
    Line equation, used
    for curve fitting algorithm

    Args:
        x: x value
         a: line coefficient
         b: line constant term

    Returns:
        The y coordinate of the point
    """
    return a*x + b

def period(L):
    """
    Calculates the theoretical period of a
    pendulum with length L. It assumes that
    the actual value of g is 9.8m/s2

    Args:
        L: The length of the pendulum

    Returns:
        The period of the pendulum in sec
    """
    g = 9.8
    return 2*np.pi*np.sqrt(L/g)

def gError(L, T, dL, dT):
```

```
    """
    Calculates the error of g
    estimation given the length, the
    period and their respective errors
    using the error propagation rule

    Args:
        L: A vector of length values
        T: A vector of period values
        dL: The error in length measurement
        dT: The error in period measurement

    Returns:
        A vector with g error values
    """
    dg = np.power(2*np.pi, 2) * np.sqrt( np.power(dL/(T*T), 2) +
     np.power(2*L*dT/np.power(T, 3), 2) )
    return dg

def experiment(L, T, dL, dT, dLsystm = 0):
    """
    Performs a g-measurement experiment

    Args:
        L: A vector of length measurements of the pendulum
        T: A vector of period measurements of the pendulum
        dL: The error in length measurement
        dT: The error in period measurement
        dLsystm: Systematic error of length measurement, default
     value 0

    Returns:
        A dictionary with the mean values of g,
        the g-error values and the measured period
        values, for each length
    """
    L = L + dLsystm
            # Add systematic error, if it exists
    g = np.power(2*np.pi, 2) * L / np.power(T, 2)
      # Indirect g measurement from length and period
```

```
    dg = gError(L, T, dL, dT)        # g measurement error
    gMean = np.sum(g)/g.size        # Mean value of g measurements
    dgMean = np.sqrt(np.sum(dg*dg))/dg.size        # Error of mean value of g
    return {'g':gMean, 'dg':dgMean}

def fit(experiment, L, T, dLsystm = 0):
    """
    Performs Least Square Fit on the given experiment

    Args:
        experiment: The experiment to perform LSF

    Returns:
        A dictionary with the LSF value of g, the
        LSF coefficients, and the values used for the fit
    """
    y = np.power(T, 2)
    x = L + dLsystm
    result =curve_fit(line, x, y)
    A = result[0][0]
    B = result[0][1]
    dA = np.sqrt(np.diag(result[1]))
    g = np.power(2*np.pi, 2)/A              # Coefficient A gives g: A = (2*pi)^2 / g
    dg = np.power(2*np.pi, 2)*dA[0]/(A*A)         # Error of g is using error propagation rule
    return {'g':g, 'dg':dg, 'A':A, 'B':B, 'x':x, 'y':y}



gTheory = 9.8
L = np.array([0.2, 0.4, 0.8, 1.0])
dL = 0.01
dT = 0.1
T = period(L) + np.random.standard_normal(L.size) * dT      # Period measurements with dt=0.1s accuracy

```

```
print("Experiment without systematic error")
experiment1 = experiment(L, T, dL, dT)    # Perform experiment 1
print("Mean Value Method")
print("-----------------")
print("g = {:.2f} +- {:.2f}".format(experiment1['g'], experiment1['dg']))
print("\nLeast Squares Fit Method")
print("------------------------")
lsq1 = fit(experiment1, L, T)                              # Perform LSF for experiment 1
print("g = {:.2f} +- {:.2f}".format(lsq1['g'], lsq1['dg']))
xn = lsq1['x']
yn = np.polyval([lsq1['A'], lsq1['B']], xn)
plt.plot(xn, yn, 'r', label='$\delta L_{system} = 0$')# Plot least square line
plt.errorbar(lsq1['x'], lsq1['y'], xerr=dL, yerr=dT, fmt='r.')            # Plot measurements
plt.ylabel('$T^2[sec^2]$')
plt.xlabel('$L[m]$')

dLsystm = 0.05
print("\n\nExperiment with systematic error={:.2f}".format(dLsystm))
experiment2 = experiment(L, T, dL, dT, dLsystm)   # Perform experiment 2 with dL = 0.05
print("Mean Value Method")
print("-----------------")
print("g = {:.2f} +- {:.2f}".format(experiment2['g'], experiment2['dg']))
print("\nLeast Squares Fit Method")
print("-------------------------")
lsq2 = fit(experiment2, L, T, dLsystm)
print("g = {:.2f} +- {:.2f}".format(lsq2['g'], lsq2['dg']))
xn = lsq2['x']
yn = np.polyval([lsq2['A'], lsq2['B']], xn)
plt.plot(xn, yn, 'b', label='$\delta L_{system} = 0.05$')     # Plot least square line
plt.errorbar(lsq2['x'], lsq2['y'], xerr=dL, yerr=dT, fmt='b*')                    # Plot measurements
plt.legend()
```

```
plt.show()
```

Ο κώδικας των ασκήσεων είναι διαθέσιμος online για εύκολη εκτέλεση στη σελίδα URL: `https://github.com/tzavellas/ComputationalPhysics.git`.