Υπολογιστική Φυσική - 4η Σειρά Ασκήσεων

Αναστάσιος Τζαβέλλας / AM:1110 2016 00255

04/01/2017

1η Άσκηση

Ζητείται η ροπή αδράνειας της ράβδου με πυκνότητα $\rho(x) = 1 + x^3$ με $x \in (0, 3)$ ως προς άξονα που περνάει από το $x = 0$. Αναλυτικά, η ροπή αδράνειας υπολογίζεται από την εξίσωση (1).

$$\begin{aligned} I &= \int_0^3 x^2 \rho(x) dx \rightarrow \\ &= \int_0^3 x^2 + x^5 dx \rightarrow \\ &= 130.5 \end{aligned} \tag{1}$$

Το ολοκλήρωμα υπολογίστηκε με τη μέθοδο του τραπεζίου καθώς και με τη μέθοδο Simpson. Η μέθοδος του τραπεζίου έδωσε $I_{tr} = 130.5012$ και η μέθοδος Simpson $I_{Sim} = 130.5002$. Η απόκλιση της πρώτης είναι $\delta I_{tr} \approx 0.0012$ και της δεύτερης είναι $\delta I_{Sim} \approx 0.0002$.

Και οι δύο μέθοδοι έδωσαν αποτέλεσμα με τη ζητούμενη ακρίβεια (10^{-2}), ωστόσο, στη μέθοδο τραπεζίου απαιτήθηκαν $n = 10$ επαναλήψεις, ενώ στην Simpson $n = 6$, δηλαδή σχεδόν οι μισές. Επιπλέον, η μέθοδος Simpson είχε μικρότερη απόκλιση από το πραγματικό αποτέλεσμα.

Στη μέθοδο του τραπεζίου, σε κάθε νέα επανάληψη, ο αριθμός των τραπεζίων διπλασιάζεται και επομένως κάποια από τα σημεία της προηγούμενης επανάληψης είναι κοινά. Τελικά, το ολοκλήρωμα υπολογίζεται από τον τύπο (2) και επομένως για $n = 10$ επαναλήψεις απαιτούνται $N = 2 + n - 1 = 11$ υπολογισμοί της $\rho(x)$.

Σε μια έξυπνη και αποδοτική υλοποίηση του αλγορίθμου, οι τιμές της $\rho(x)$ μιας επανάληψης, αποθηκεύονται για χρήση στην επόμενη επανάληψη. Στη συγκεκριμένη όμως υλοποίηση, δεν ήταν απαραίτητη αυτή η βελτιστοποίηση και έτσι στην $i+1$ επανάληψη απαιτούνται $N_{i+1} = 2N_i + 1$ υπολογισμοί. Συνολικά, απαιτήθηκαν $N_{tot} = 1024$ υπολογισμοί.

$$I \approxeq \frac{h}{2}\left[f_0 + 2\sum_{i=1}^{n-1} f_i + f_n\right] \tag{2}$$

Για τη μέθοδο Simpson, σε κάθε υποδιάστημα χρησιμοποιούνται 3 σημεία και όμοια σε κάθε επανάληψη ο αριθμός των υποδιαστημάτων διπλασιάζεται. Το ολοκλήρωμα δίνεται από τον τύπο (3) και επομένως για $n = 6$ επαναλήψεις απαιτούνται $N = 2 + \frac{n}{2} - 1 + \frac{n}{2} = 7$ υπολογισμοί της $\rho(x)$.

Όπως και στην περίπτωση του τραπεζίου, η υλοποίηση του αλγόριθμου Simpson δεν είναι η βέλτιστη για τους ίδιους λόγους και τελικά απαιτήθηκαν $N_{tot} = 64$ υπολογισμοί, δηλαδή ακριβώς οι μισοί σε σχέση με τη μέθοδο του τραπεζίου.

$$I \approxeq \frac{h}{3}\left[f_0 + 2\sum_{i=1}^{\frac{n}{2}-1} f_{2i} + 4\sum_{i=1}^{n/2} f_{2i-1} + f_n\right] \tag{3}$$

Συμπερασματικά, προκύπτει ότι η μέθοδος Simpson είναι λίγο πιο σύνθετη στην υλοποίηση, αλλά υπολογιστικά πιο 'φθηνή' από τη μέθοδο τραπεζίου, για την ίδια ακρίβεια.

Παρακάτω ακολουθεί ο κώδικας που γράφτηκε σε Python και έγινε χρήση της βιβλιοθήκης Numpy. Οι λεπτομέρειες υλοποίησης των μεθόδων τραπεζίου και Simpson δίνονται στο Παράρτημα.

ex1.py

```
1 import numpy as np
  import integration


  def f(x):
6     return np.power(x, 2) + np.power(x, 5)
```

```
IAnalytic = np.power(3, 3) / 3 + np.power(3, 6) / 6
print('Analytic')
print('Integral:', IAnalytic)

a = 0.
b = 3.
epsilon = 1e-2
trapezoid = integration.trapezoid(f, a, b, epsilon)
ITrapezoid = trapezoid['integral']
itTrapezoid = trapezoid['iterations']
print('\nTrapezoid')
print(trapezoid)
print('Deviation:', np.abs(IAnalytic - ITrapezoid))

simpson = integration.simpson(f, a, b, epsilon)
ISimpson = simpson['integral']
itSimpson = simpson['iterations']
print('\nSimpson 1/3')
print(simpson)
print('Deviation:', np.abs(IAnalytic - ISimpson))
```

2η Άσκηση

Το πρόβλημα ρίψης σφαίρας από το έδαφος με αρχική ταχύτητα, λαμβάνοντας υπόψιν μόνο την δύναμη της βαρύτητας, μπορεί να διατυπωθεί μαθηματικά εφαρμόζοντας το δεύτερο νόμο του Newton, από τον οποίον προκύπτει το σύστημα διαφορικών εξισώσεων με αρχικές συνθήκες (4).

$$\begin{cases} m\frac{d^2x}{dt^2} = & 0 \\ m\frac{d^2y}{dt^2} = & -mg \\ x(0) = & 0 \\ y(0) = & 0 \\ x'(0) = & v_0 \cos\frac{\pi}{4} \\ y'(0) = & v_0 \sin\frac{\pi}{4} \end{cases} \tag{4}$$

Για να λυθεί το σύστημα με αριθμητικές μεθόδους, πρέπει να έρθει σε κα-

τάλληλη μορφή $u' = f(t, u)$, όπου u διάνυσμα με άγνωστες συναρτήσεις, t η ανεξάρτητη μεταβλητή και f κατάλληλη συνάρτηση. Ορίζεται $u_0 = x$, $u_1 = x'$, $u_2 = y$ και $u_3 = y'$, οπότε οι εξισώσεις (4) μετασχηματίζονται σε μορφή πινάκων (5)

$$\underbrace{\frac{d}{dt}\begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix}}_{u'} = \underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix}}_{u} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ -g \end{bmatrix}}_{b}, \underbrace{\begin{bmatrix} 0 \\ v_0 \cos\frac{\pi}{4} \\ 0 \\ v_0 \sin\frac{\pi}{4} \end{bmatrix}}_{u(0)} \tag{5}$$

ή συνοπτικά (6), όπου αντιστοιχεί $f(t, u) = Au + b$. Σε αυτή τη μορφή το σύστημα διαφορικών εξισώσεων είναι επιλύσιμο με το σχήμα Euler ή Runge-Kutta 4ης τάξης.

$$u' = Au + b, u_0 = u(0) \tag{6}$$

Συγκεκριμένα, για το σχήμα Euler, εφαρμόζεται η υπολογιστική διαδικασία (7),

$$u_{i+1} = u_i + hf(t_i, u_i), u_0 = u(0) \tag{7}$$

ενώ στο σχήμα Runge-Kutta η (8).

$$\begin{aligned} k_1 =& f(t_i, u_i) \\ k_2 =& f(t_i + \frac{h}{2}, u_i + \frac{h}{2}k_1) \\ k_2 =& f(t_i + \frac{h}{2}, u_i + \frac{h}{2}k_2) \\ k_3 =& f(t_i + h, u_i + hk_3) \\ u_{i+1} =& u_i + \frac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right), u_0 = u(0) \end{aligned} \tag{8}$$

Η επίλυση του συστήματος διαφορικών εξισώσεων με τη μέθοδο Euler για διάφορα μεγέθη χρονικού βήματος παράγει τα Σχήματα 1 και 2. Μαζί με την

αριθμητική λύση δίνεται και η αναλυτική που δίνεται από τις εξισώσεις 9.

$$\begin{cases} x(t) = & v_0 \cos(\phi) t \\ y(t) = & v_0 \sin(\phi) t - \frac{1}{2} g t^2 \end{cases} \tag{9}$$

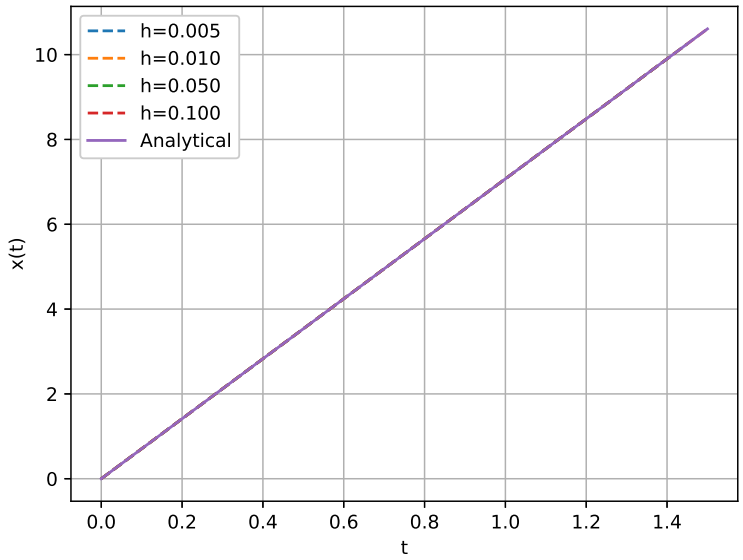
Η σύγκριση αποκαλύπτει ότι η αριθμητική λύση της $x(t)$ συμπίπτει με την αναλυτική για κάθε μέγεθος βήματος, που είναι αναμενόμενο γιατί η λύση είναι γραμμική συνάρτηση και συμπίπτει με την προσέγγιση πρώτης τάξης της μεθόδου Euler. Ωστόσο, για την $y(t)$ που είναι πολυώνυμο 2ου βαθμού, απαιτείται αρκετά μικρό βήμα $h < 0.01$ προκειμένου να προσεγγιστεί ικανοποιητικά η αναλυτική. Μικρότερο βήμα συνεπάγεται μεγαλύτερο αριθμό βημάτων, δηλαδή 142 (και 283 για $h = 0.005$).

Η εφαρμογή της μεθόδου Runge-Kutta 4ης τάξης, παράγει καλύτερα αποτελέσματα. Και πάλι η αριθμητική λύση της $x(t)$ συμπίπτει με την αναλυτική, αλλά επιπλέον η αριθμητική λύση της $y(t)$ είναι πολύ κοντά στην αναλυτική ακόμα και για $h = 0.1$, δηλαδή μεγαλύτερο κατά μία τάξη μεγέθους σε σχέση με την Euler. Τα αποτελέσματα δίνονται στα Σχήματα 3 και 4. Ο αριθμός των βημάτων για κάθε μέγεθος βήματος είναι ίδιος, αλλά εφόσον η Runge-Kutta αποκλίνει πολύ λίγο από τη αναλυτική λύση για μεγάλο βήμα, αποδεικνύεται και η υπεροχή της ως προς την Euler.

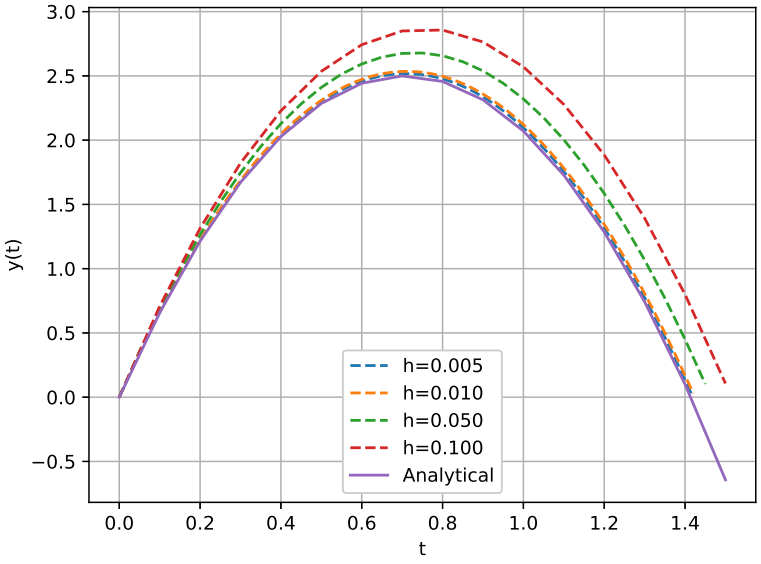
Παρακάτω ακολουθεί ο κώδικας που γράφτηκε σε Python και έγινε χρήση της βιβλιοθήκης Numpy. Οι αλγόριθμοι Euler και Runge-Kutta 4ης τάξης δίνονται στο Παράρτημα.

ex2.py

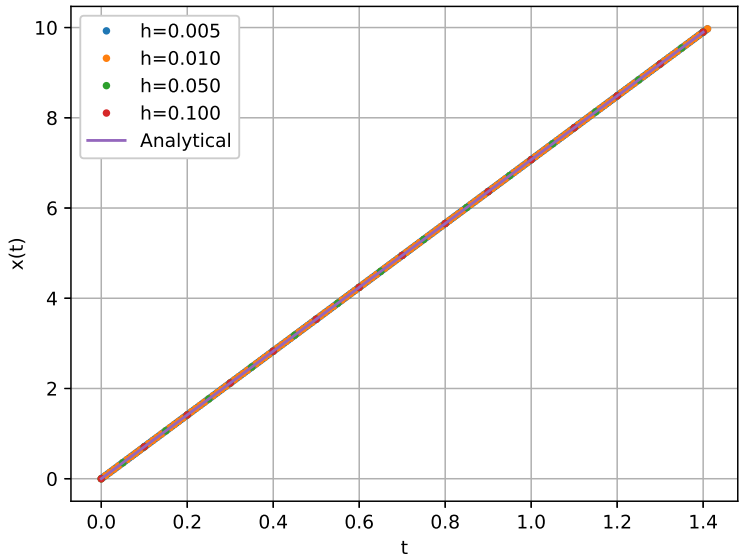
```
import numpy as np
import matplotlib.pyplot as plt
import diffEquation as de


def f(t, y):
    g = 10.
    A = np.array([[0, 1, 0, 0],
                  [0, 0, 0, 0],
                  [0, 0, 0, 1],
                  [0, 0, 0, 0]])
```



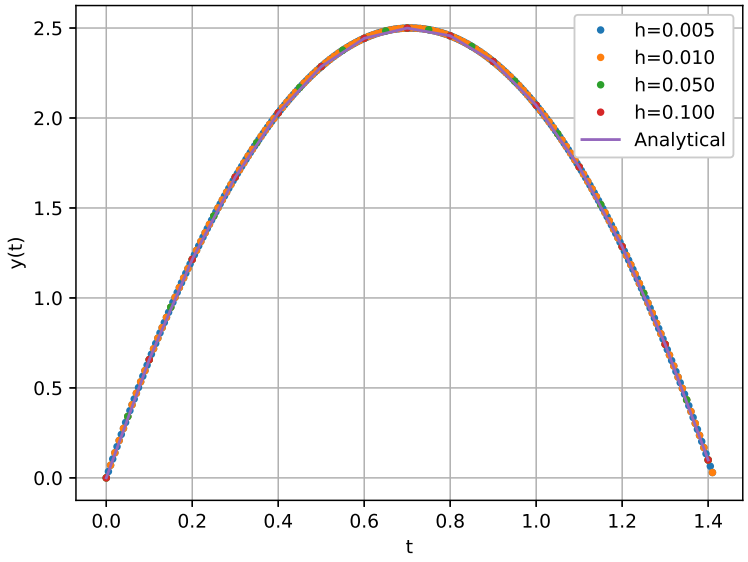
Σχήμα 1: Μέθοδος Euler - $x(t)$



Σχήμα 2: Μέθοδος Euler - $y(t)$



Σχήμα 3: Μέθοδος Runge-Kutta 4ης - $x(t)$



Σχήμα 4: Μέθοδος Runge-Kutta 4ης - $y(t)$

```
    b = np.array([0, 0, 0, -g])
    return np.dot(A, y) + b


def run(Y0, f, method, h, figNum):
    t = np.array([])
    for hi in h:
        solution = de.solve(Y0, f, method, hi)
        print("h=%.3f, iterations=%d"% (hi,
                                        solution['iterations']))
        x = solution['x']
        y = solution['y']
        t = solution['t']
        plt.figure(figNum)
        plt.plot(t, x, '--', label="h=%.3f" % hi)
        plt.figure(figNum + 1)
        plt.plot(t, y, '--', label="h=%.3f" % hi)
    g = 10.
    v0x = Y0[1]
    v0y = Y0[3]
    x = v0x * t
    y = v0y * t - 1/2 * g * np.power(t, 2.)
    plt.figure(figNum)
    plt.plot(t, x, label="Analytical")
    plt.grid()
    plt.legend()
    plt.xlabel('t')
    plt.ylabel('x(t)')
    plt.figure(figNum+1)
    plt.plot(t, y, label="Analytical")
    plt.grid()
    plt.legend()
    plt.xlabel('t')
    plt.ylabel('y(t)')


v0 = 10.  # Initial conditions
phi = np.pi / 4
v0x = v0 * np.cos(phi)
v0y = v0 * np.sin(phi)
```

```
Y0 = np.array([0, v0x, 0, v0y])  # initial vector
h = np.array([0.005, 0.01, 0.05, 0.1])  # time steps
t = np.array([])
plt.close('all')

print('Euler')
run(Y0, f, de.eulerStep, h, 1)  # run with Euler
print('\nRunge-Kutta 4th')
run(Y0, f, de.rungeKutta4, h, 3)  # run with Runge-Kutta 4th
```

3η Άσκηση

Η εύρεση της μέγιστης εμβέλειας με υπολογιστικό τρόπο απαιτεί καταρχάς την επίλυση του προβλήματος με αριθμητικούς τρόπους. Κατόπιν, εφόσον η λύση είναι αριθμητική, απαιτείται επίλυση του προβλήματος για ένα εύρος τιμών γωνίας ρίψης και εύρεση της συνάρτησης εμβέλειας για κάθε γωνία, $x(\phi)$.

Το σύστημα διαφορικών εξισώσεων που περιγράφει τη βολή σφαίρας σε ομογενές πεδίο βαρύτητας με αεροδυναμική τριβή ανάλογη της ταχύτητας, δίνεται από τις εξισώσεις (10), όπου δίνεται $\frac{C_D}{m} = \frac{g}{4}$.

$$\begin{cases} x'' = & \frac{C_D}{m} x' \\ y'' = & \frac{C_D}{m} y' - g \\ x(0) = & 0 \\ y(0) = & 0 \\ x'(0) = & v_0 \cos\phi \\ y'(0) = & v_0 \sin\phi \end{cases} \tag{10}$$

Ορίζεται $u_0 = x$, $u_1 = x'$, $u_2 = y$ και $u_3 = y'$, οπότε οι εξισώσεις (10)

μετασχηματίζονται σε μορφή πινάκων (11)

$$\underbrace{\frac{d}{dt}\begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix}}_{u'} = \underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{C_D}{m} & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\frac{C_D}{m} & 0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix}}_{u} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ -g \end{bmatrix}}_{b}, \underbrace{\begin{bmatrix} 0 \\ v_0 \cos\phi \\ 0 \\ v_0 \sin\phi \end{bmatrix}}_{u(0)} \tag{11}$$

ή συνοπτικά (12), όπου αντιστοιχεί $f(t,u) = Au + b$. Σε αυτή τη μορφή το σύστημα διαφορικών εξισώσεων είναι επιλύσιμο με το σχήμα Runge-Kutta 4ης τάξης.

$$u' = Au + b, u_0 = u(0) \tag{12}$$

Όπως προαναφέρθηκε, η λύση του προβλήματος ρίψης μπορεί να βρεθεί υπολογιστικά για διαφορετικές γωνίες $\phi \in [0, \frac{\pi}{4}]$. Μια απλή μέθοδος εύρεσης της μέγιστης εμβέλειας είναι η επίλυση του παραπάνω προβλήματος για διακριτές τιμές γωνίας και η εύρεση της μέγιστης εμβέλειας από τις τιμές που προέκυψαν.

Μια άλλη μέθοδος και πιο ακριβής είναι η θεώρηση ότι η συνάρτηση $R(\phi)$ είναι μίας μεταβλητής, συνεχής στο $[0, \frac{\pi}{4}]$. Από θεώρημα της Ανάλυσης αυτή θα λαμβάνει μια μέγιστη και μια ελάχιστη τιμή σε αυτό το κλειστό διάστημα. Το μέγιστο της συνάρτησης μπορεί να υπολογιστεί και πάλι αριθμητικά χρησιμοποιώντας κάποιον αλγόριθμο αριθμητικής ανάλυσης, όπως ο αλγόριθμος του Brent.

Ο λόγος που η δεύτερη μέθοδος είναι πιο ακριβής, είναι ότι ο κανόνας του Brent έχει σαν κριτήριο τερματισμού την επίτευξης μιας συγκεκριμένης ακρίβειας (πχ 10^{-3}). Επιπλέον, ο απλός δειγματολειπτικός αλγόριθμος θα επιλύσει το σύστημα διαφορικών εξισώσεων για διάφορες τιμές του ϕ, οι περισσότερες από τις οποίες είναι πολύ μακριά από το ϕ_{max} και επομένως δεν είναι πολύ αποδοτικός.

Για λόγους σύγκρισης υλοποιήθηκαν και οι δύο τρόποι και έδωσαν αποτελέσματα $\phi_{max}^{Br} = 27.61°$ με $R(\phi_{max}^{Br}) = 3.0647$ και $\phi_{max}^{Smpl} = 25.78°$ με $R(\phi_{max}^{Smpl}) = 3.0631$. Όπως προαναφέρθηκε, ο αλγόριθμος του Brent δίνει πιο ακριβές αποτέλεσμα.

Τυπικά, ο αλγόριθμος του Brent υπολογίζει το ελάχιστο μιας συνάρτησης

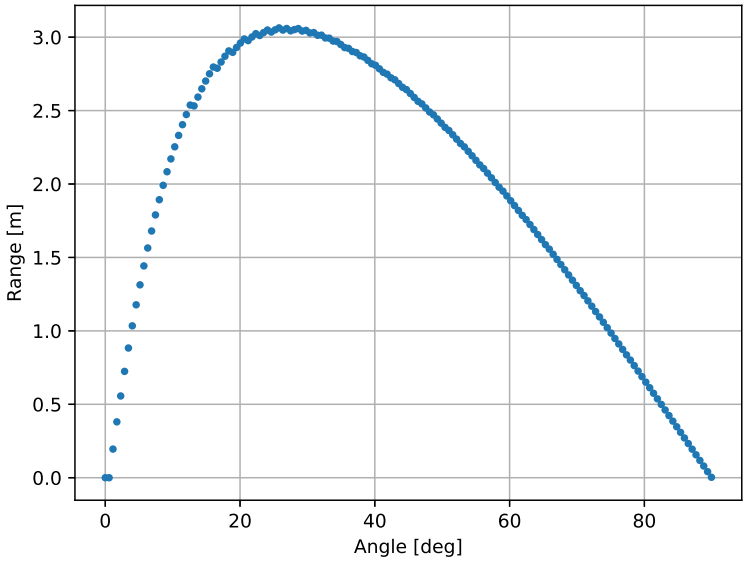
f, αλλά το ελάχιστο της f είναι το μέγιστο της $-f$ και αντίστροφα. Η επίλυση του συστήματος διαφορικών εξισώσεων έγινε με χρονικό βήμα $h = 0.02$, που είναι μια καλή τιμή για επίτευξη ομαλούς λύσης και ταχύτητας.

Στο Σχήμα 5 δίνεται η συνάρτηση της εμβέλειας που υπολογίστηκε αριθμητικά. Στο Σχήμα 6 δίνεται η τροχιά του βλήματος για την γωνία μέγιστης εμβέλειας.

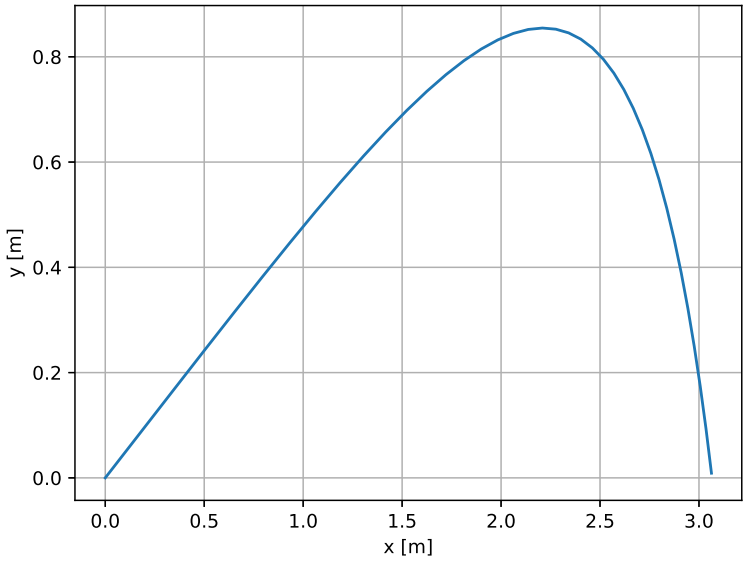
Παρακάτω ακολουθεί ο κώδικας που γράφτηκε σε Python και έγινε χρήση της βιβλιοθήκης Numpy. Η υλοποίηση του αλγόριθμου Runge-Kutta 4ης τάξης δίνεται στο Παράρτημα.

ex3.py

```
import numpy as np
from scipy.optimize import minimize_scalar
import matplotlib.pyplot as plt
import diffEquation as de


def f(t, y):
    g = 10
    c = g/4
    A = np.array([[0, 1, 0, 0],
                  [0, -c, 0, 0],
                  [0, 0, 0, 1],
                  [0, 0, -c, 0]])
    b = np.array([0, 0, 0, -g])
    return np.dot(A, y) + b


def RangeFun(phi, V0, h, f, solver):
    Vx = V0 * np.cos(phi)
    Vy = V0 * np.sin(phi)
    Y0 = np.array([0., Vx, 0., Vy])
    solution = de.solve(Y0, h, f, solver)
    x = solution['x']
    return x[x.size - 1]


plt.close('all')
```



Σχήμα 5: Συνάρτηση $R(\phi)$



Σχήμα 6: Τροχιά βλήματος για $\phi = \phi_{max}$

```
V0 = 10.  # Initial velocity
h = 0.02

# Find maximum using Brent's Algorithm
g = minimize_scalar(lambda phi: -RangeFun(phi,
                                          V0,
                                          f,
                                          de.rungeKutta4,
                                          h),
                    bracket=(0, np.pi/2))
print("Max Range @%.2f deg: %.4f" % (np.rad2deg(g['x']),
                                     -g['fun']))
plt.figure(1)
phi_ = np.arange(0.,
                 np.pi/2,
                 0.01)  # range of angles to find range
Ranges = list()
i = 0
for phi in phi_:
    Ranges.insert(i, RangeFun(phi,
                              V0,
                              f,
                              de.rungeKutta4,
                              h))
    i = i + 1
plt.plot(np.rad2deg(phi_),
         np.array(Ranges), '.')  # Plot range vs angle
plt.ylabel('Range [m]')
plt.xlabel('Angle [deg]')
plt.grid()

# Exhaustive Search - Assumes range functions has one maximum
Range = 0.
Ranges = list()
Y = list()
for i, phi in enumerate(phi_):
    Vx = V0 * np.cos(phi)
    Vy = V0 * np.sin(phi)
    Y0 = np.array([0., Vx, 0., Vy])
    solution = de.solve(Y0,
```

```
                        f,
                        de.rungeKutta4,
                        h)
    Y.insert(i, solution)
    x = solution['x']
    y = solution['y']
    Ranges.insert(i, x[x.size - 1])

maxRange = max(Ranges)  # Find max element of calculated values
i = Ranges.index(maxRange)  # Find phi corresponding to max
    element
maxPhi = np.rad2deg(phi_[i])

print("Max Range @%.2f deg: %.4f" % (maxPhi,
                                     maxRange))
plt.figure(2)
plt.plot(Y[i]['x'], Y[i]['y'])
plt.ylabel('y [m]')
plt.xlabel('x [m]')
plt.grid()
```

Παραρτήματα

Κώδικας Ολοκληρωμάτων

integration.py

```

import numpy as np


def trapezoid(f, a, b, epsilon=1e-3):
    n = 1
    err = 1.
    integralOld = 0.
    fa = f(a)
    fb = f(b)
    evaluations = 2
    while(err > epsilon):
```

```
        h = (b - a)/n
        sumf = 0.
        if n != 1:
            xValues = np.arange(a, b, h)
            fValues = f(xValues)
            evaluations = evaluations + fValues.size
            for i in range(1, n):
                sumf = sumf + fValues[i]
        integralNew = h/2 * (fa + 2*sumf + fb)
        if n != 1:
            err = np.abs(integralNew - integralOld)
        integralOld = integralNew
        n = 2 * n
    return {'integral': integralNew,
            'iterations': np.log2(n).astype(int),
            'evaluations': evaluations}


def simpson(f, a, b, epsilon=1e-3):
    n = 2
    err = 1.
    integralOld = 0.
    evaluations = 2
    fa = f(a)
    fb = f(b)
    while(err > epsilon):
        h = (b - a)/n
        sumfEven = 0.
        sumfOdd = 0.
        xValues = np.arange(a, b, h)
        fValues = f(xValues)
        evaluations = evaluations + fValues.size
        for i in range(1, n):
            if i % 2 == 0:
                sumfEven = sumfEven + fValues[i]
            else:
                sumfOdd = sumfOdd + fValues[i]
        integralNew = h/3 * (fa + 2*sumfEven + 4*sumfOdd + fb)
        if n != 2:
            err = np.abs(integralNew - integralOld)
```

```
        integralOld = integralNew
        n = 2 * n
    return {'integral': integralNew,
            'iterations': np.log2(n).astype(int),
            'evaluations': evaluations}
```

Κώδικας Διαφορικών Εξισώσεων

diffEquation.py

```
def eulerStep(ti, yi, f, h=0.1):
    return yi + h * f(ti, yi)


def rungeKutta4(ti, yi, f, h=0.1):
    k1 = f(ti, yi)
    k2 = f(ti + h/2, yi + h/2 * k1)
    k3 = f(ti + h/2, yi + h/2 * k2)
    k4 = f(ti + h, yi + h * k3)
    return yi + h/6 * (k1 + 2*k2 + 2*k3 + k4)


def solve(Y0, f, method, h=0.1, tmin=0, tmax=None, N=None):
    t = np.array([])  # store time
    x = np.array([])  # store x coord
    y = np.array([])  # store y coord
    vx = np.array([])  # store x velocity
    vy = np.array([])  # store y velocity
    i = 0
    Y = Y0
    while 1:
        x = np.insert(x, i, Y[0])  # store all previous step
    values
        y = np.insert(y, i, Y[2])
        vx = np.insert(vx, i, Y[1])
        vy = np.insert(vy, i, Y[3])
        t = np.insert(t, i, i * h)
        Y = method(t[i], Y, f, h)  # Next step
```

```
        if (Y[2] < 0.) or (i == N):  # break if max iterations
            break  # or if y<.0
        i = i + 1
    return {'x': x,
            'y': y,
            't': t,
            'vx': vx,
            'vy': vy,
            'iterations': i}
```

Εκτέλεση Προγραμμάτων

```
In [8]: runfile('C:/Users/tzave/ComputationalPhysics/Assignment4/ex1.py', wdir='C:/Users/
tzave/ComputationalPhysics/Assignment4')
Reloaded modules: diffEquation
Analytic
Integral: 130.5

Trapezoid
{'integral': 130.50117587954807, 'iterations': 10, 'evaluations': 1024}
Deviation: 0.00117587954807

Simpson 1/3
{'integral': 130.50023174285889, 'iterations': 6, 'evaluations': 64}
Deviation: 0.000231742858887

In [9]: runfile('C:/Users/tzave/ComputationalPhysics/Assignment4/ex2.py', wdir='C:/Users/
tzave/ComputationalPhysics/Assignment4')
Reloaded modules: integration
Euler
h=0.005, iterations=283
h=0.010, iterations=142
h=0.050, iterations=29
h=0.100, iterations=15

Runge-Kutta 4th
h=0.005, iterations=282
h=0.010, iterations=141
h=0.050, iterations=28
h=0.100, iterations=14

In [10]: runfile('C:/Users/tzave/ComputationalPhysics/Assignment4/ex3.py', wdir='C:/
Users/tzave/ComputationalPhysics/Assignment4')
Reloaded modules: diffEquation
Max Range @27.61 deg: 3.0647
Max Range @25.78 deg: 3.0631

In [11]:
```