Υπολογιστική Φυσική - 4η Σειρά Ασκήσεων

Αναστάσιος Τζαβέλλας / ΑΜ: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 την εξίσωση (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 την εξίσωση (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

```
import numpy as np
import integration


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

```python
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 πολυώνυμο παρεμβολής Newton μπορεί να βρεθεί με χρήση του αλγορίθμου διηρημένων διαφορών. Συγκεκριμένα το πολυώνυμο που προκύπτει είναι της μορφής (4), όπου οι συντελεστές a_i δίνονται από τις διηρημένες διαφορές (5) και $x_i, i = 0, 1, \ldots n$ τα σημεία παρεμβολής.

$$P_n(x) = a_0 + a_1(x - x_0) + \cdots + a_n(x - x_0)(x - x_1)\cdots(x - x_n) \tag{4}$$

$$a_i = f[x_0, x_1, \ldots, x_i] \tag{5}$$

Το αποτέλεσμα της παρεμβολής δίνεται στο Σχήμα 1. Σχετικά με το σφάλμα προσέγγισης $f(x) - P(x)$ στο $x \in [0, \frac{\pi}{2}]$, αποδεικνύεται ότι δίνεται από την εξίσωση (6), με $\xi \in (a, b)$, όπου $a = \min[x_0, \ldots x_n]$ και $b = \max[x_0, \ldots x_n]$.

$$f(x) - p_n(x) = \frac{(x - x_0) \cdots (x - x_n)}{(n+1)!} f^{(n+1)}(\xi) \tag{6}$$

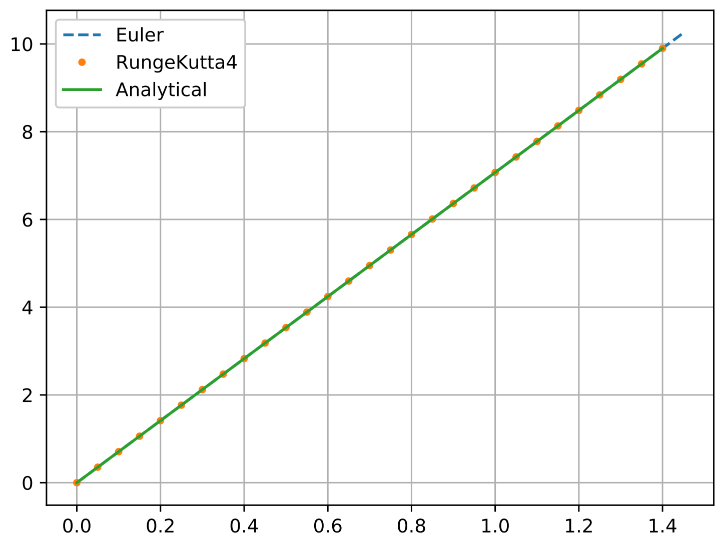
Λαμβάνοντας την απόλυτη τιμή της (6) και για την δοσμένη $f(x)$ προκύπτει η (7). Στο Σχήμα 2, δίνεται η γραφική παράσταση του σφάλματος στο υπό εξέταση διάστημα.

$$\begin{aligned} |f(x) - p_n(x)| &= \left| \frac{x(x - \frac{1}{2})(x-1)}{4!} \sin \xi \right| \rightarrow \\ &\leq \frac{|x||x - \frac{1}{2}||x-1|}{4!} \rightarrow \\ &\leq \frac{\frac{\pi}{2}(\frac{\pi}{2} - \frac{1}{2})(\frac{\pi}{2} - 1)}{4!} \rightarrow \\ &= \frac{\pi^3 - 3\pi^2 + 2\pi}{192} \approx 0.04 \end{aligned} \tag{7}$$

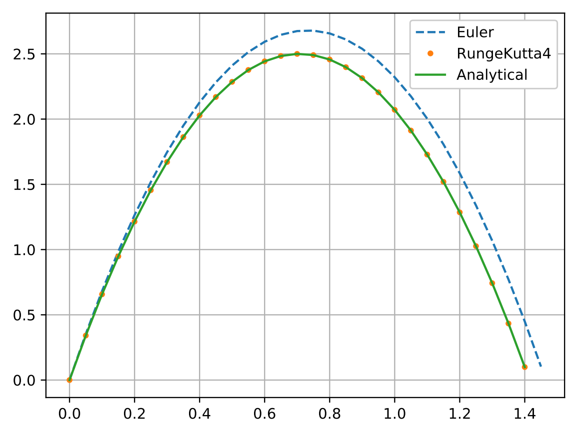
Παρακάτω ακολουθεί ο κώδικας που γράφτηκε σε Python και έγινε χρήση της βιβλιοθήκης Numpy. Ο αλγόριθμος διηρημένων διαφορών και φωλιασμένου υπολογισμού της τιμής του πολυωνύμου δίνεται στο Παράρτημα.

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]])
    b = np.array([0, 0, 0, -g])
```



Σχήμα 1: Παρεμβολή με πολυώνυμο Newton



Σχήμα 2: Παρεμβολή με πολυώνυμο Lagrange

```
    return np.dot(A, y) + b


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])
h = 0.05

eulerSolution = de.solve(Y0, h, f, de.eulerStep)
rK4Solution = de.solve(Y0, h, f, de.rungeKutta4)
x1 = eulerSolution['x']
y1 = eulerSolution['y']
t1 = eulerSolution['t']
x2 = rK4Solution['x']
y2 = rK4Solution['y']
t2 = rK4Solution['t']

g = 10
x3 = v0x * t2
y3 = v0y * t2 - 1/2 * g * np.power(t2, 2.)

plt.close('all')
plt.figure(1)
plt.plot(t1, x1, '--', label='Euler')
plt.plot(t2, x2, '.', label='RungeKutta4')
plt.plot(t2, x3, label='Analytical')
plt.grid()
plt.legend()

plt.figure(2)
plt.plot(t1, y1, '--', label='Euler')
plt.plot(t2, y2, '.', label='RungeKutta4')
plt.plot(t2, y3, label='Analytical')
plt.grid()
plt.legend()
```

3η Άσκηση

Παρατηρείται ότι το πολυώνυμο δίνεται σε φωλιασμένη μορφή άρα είναι πολυώνυμο Newton. Επομένως, μπορεί πολύ εύκολα να υπολογιστεί ένα νέο πολυώνυμο που να παρεμβάλει και το πέμπτο σημείο.

Συγκεκριμένα, αν $p_3 = 2 - (x+1) + x(x+1) - 2x(x+1)(x-1)$ τρίτου βαθμού πολυώνυμο, τότε το νέο πολυώνυμο $p_4(x)$ θα είναι της μορφής (8) και θα είναι τέταρτου βαθμού.

$$p_4(x) = p_3(x) + cx(x+1)(x-1)(x-2) \tag{8}$$

Ο προσδιορισμός της σταθεράς c γίνεται μέσω της τιμής του νέου πολυωνύμου στο επιπλέον σημείο (9).

Το υπολογιστικό φορτίο που απαιτείται για να προστεθεί ένας επιπλέον όρος είναι μικρό: 8 πράξεις κινητής υποδιαστολής (4 προσθαφαιρέσεις, 3 πολλαπλασιασμοί και 1 διαίρεση). Εδώ φαίνεται η υπεροχή της χρήσης των πολυωνύμων Newton σε σχέση με τα πολυώνυμα Lagrange, για τα οποία θα χρειαζόταν ο υπολογισμός όλου του πολυωνύμου εξαρχής.

$$\begin{aligned} p_4(x_4) = 10 \rightarrow & \\ y_4 &= p_3(x_4) + cx_4(x_4+1)(x_4-1)(x_4-2) \rightarrow \\ c &= \frac{y_4 - p_3(x_4)}{x_4(x_4+1)(x_4-1)(x_4-2)} \end{aligned} \tag{9}$$

Στο Σχήμα 4 δίνονται τα $p_3(x)$ και $p_4(x)$. Όπως είναι αναμενόμενο, τα δύο πολυώνυμα είναι σχετικά κοντά μέχρι το $(2, -7)$ και αποκλίνουν σημαντικά μετά από αυτό.

Παρακάτω ακολουθεί ο κώδικας που γράφτηκε σε Python και έγινε χρήση της βιβλιοθήκης Numpy. Η υλοποίηση των αλγορίθμων προσθήκης νέου όρου στο πολυώνυμο Newton και φωλιασμένου υπολογισμού της τιμής του πολυωνύμου δίνονται στο Παράρτημα.

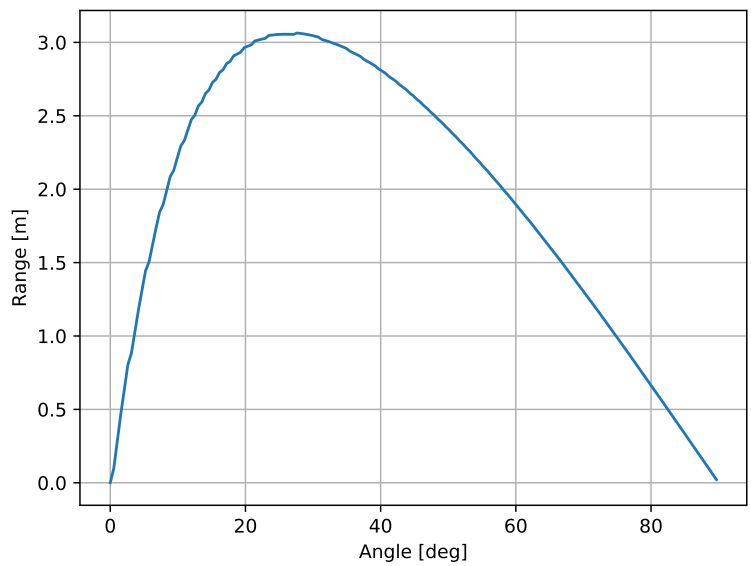
ex3.py

```
import numpy as np
```

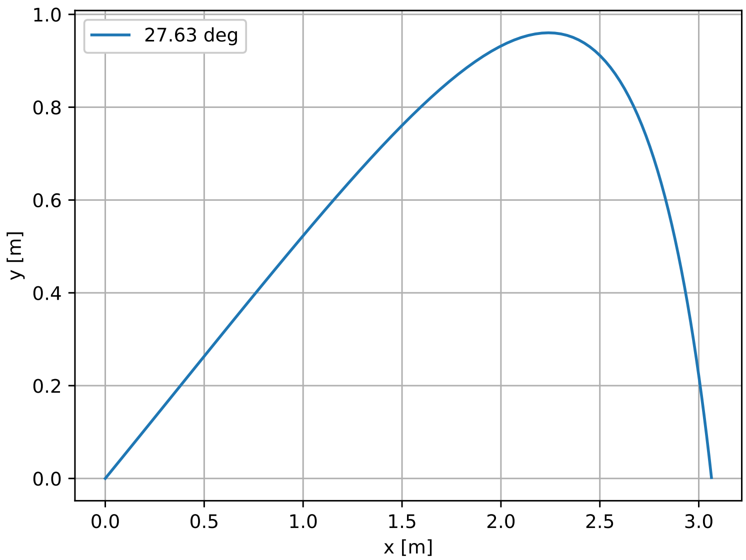
```
from scipy.optimize import minimize_scalar
import matplotlib.pyplot as plt
import diffEquation as de


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')
V0 = 10.  # Initial velocity
h = 0.01

# Find maximum using Brent's Algorithm
g = minimize_scalar(lambda phi: -RangeFun(phi,
                                          V0,
                                          h,
                                          f,
                                          de.rungeKutta4),
                    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,
                 0.0091)  # range of angles to find range
Ranges = list()
i = 0
for phi in phi_:
    Ranges.insert(i, RangeFun(phi,
                              V0,
                              h,
                              f,
                              de.rungeKutta4))
    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,
                        h,
                        f,
                        de.rungeKutta4)
    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'],
         label="%.2f deg" % maxPhi)
plt.ylabel('y [m]')
plt.xlabel('x [m]')
plt.grid()
plt.legend()
```



Σχήμα 3: Παρεμβολή με πολυώνυμα Newton 3ου και 4ου βαθμού



Σχήμα 4: Παρεμβολή με πολυώνυμα Newton 3ου και 4ου βαθμού

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, h, f):
    return yi + h * f(ti, yi)


def rungeKutta4(ti, yi, h, f):
    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)

```

```python
def solve(Y0, h, f, method, N=None, tmin=None, tmax=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, h, f)  # 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}
```

Εκτέλεση Προγραμμάτων

```
Python 3.6.3 |Anaconda custom (64-bit)| (default, Nov  8 2017, 15:10:56) [MSC v.1900 64
bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 6.2.1 -- An enhanced Interactive Python.

Restarting kernel...
```

```
In [1]: runfile('C:/Users/tzave/ComputationalPhysics/Assignment4/ex1.py', wdir='C:/Users/
tzave/ComputationalPhysics/Assignment4')
Trapezoid
Integral: 130.50117588
Deviation: 0.00117587954807

Simpson 1/3
Integral: 130.500231743
Deviation: 0.000231742858887

In [2]: runfile('C:/Users/tzave/ComputationalPhysics/Assignment4/ex2.py', wdir='C:/Users/
tzave/ComputationalPhysics/Assignment4')
Reloaded modules: integration

In [3]: runfile('C:/Users/tzave/ComputationalPhysics/Assignment4/ex3.py', wdir='C:/Users/
tzave/ComputationalPhysics/Assignment4')
Reloaded modules: diffEquation
Max Range @27.12 deg: 3.0662
Max Range @27.63 deg: 3.0641

In [4]:
```