PHY505: Computational Physics 1
Final Exam

Hsuan-Hao Fan

Person No. 5009-7813

December 12, 2014

Contents

1 Problem 1

Consider a rocket of total mass m_0, with a mass of m_1 when it is empty of propellant. Assume the rocket expels propellent at a constant exhaust velocity v_e, and the mass linearly decreases with time with rate B until it is spent:

$$m(t) = \max(m_1, m_0 - Bt) \tag{1}$$

Consider the "Saturn V" rocket with $m_0 = 2,970,000$ kg and $m_1 = 130,000$ kg after the first stage, which can apply 34020 kN of force to move ("thrust"). The burn time is 165 s. This burn time corresponds to B=17212 kg/s.

`http://en.wikipedia.org/wiki/Saturn_V`

1.1 Part a

In the absence of air resistance, calculate the escape velocity off the surface of the earth if a rocket is fired straight up.

Solution:

Escape velocity is the speed to let the rocket go to infinite distance. That means the total energy of the rocket is zero at $r \to \infty$, which r is the distance from the center of Earth. Assume there is no air resistance. The mass of Earth (M) is 5.97219×10^{24} kg and the mean radius of Earth (R) is 6371×10^3 m.[1] $G = 6.67384 \times 10^{-11}(\mathrm{m}^3\mathrm{kg}^{-1}\mathrm{s}^{-2})$ is the gravitational constant.[2] Let h be the distance from the surface of Earth. From the conservation of energy, we can calculate the escape velocity (v_e):

$$\begin{aligned} &\frac{1}{2}mv_e^2 - \frac{GMm}{R} = 0 \\ \Rightarrow \quad & v_e = \sqrt{\frac{2GM}{R}} = \sqrt{\frac{2 \times 6.67384 \times 10^{-11} \times 5.97219 \times 10^{24}}{6371 \times 10^3}} = 11185 m/s \approx 11.2 km/s \end{aligned} \tag{2}$$

This is consistent with the results on Wikipedia, `http://en.wikipedia.org/wiki/Escape_velocity#cite_note-Wimmer-6`.

1.2 Part b

The "balle" program from Lecture 20 handles projectile motion in a gravitational field, close to the surface of the earth. Modify the "balle" program from Lecture 20 to account for the changing mass of a rocket, and the full gravitational potential from the earth for arbitrary radius. Plot the distance from the surface of the earth $r(t)$, as well as the velocity $v(t)$, as a function of time for the first stage of the Saturn V rocket. Does the rocket achieve escape velocity like this?

Solution:

Assume the mass of Earth M, the radius of Earth R, and h the distance from the surface of Earth. The gravitational acceleration g becomes

$$g = \frac{GM}{(R+h)^2}. \tag{3}$$

[1] http://en.wikipedia.org/wiki/Earth

[2] `http://en.wikipedia.org/wiki/Gravitational_constant`

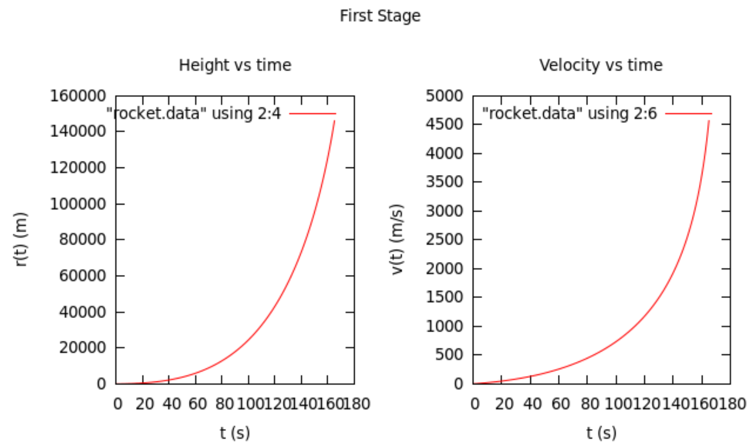
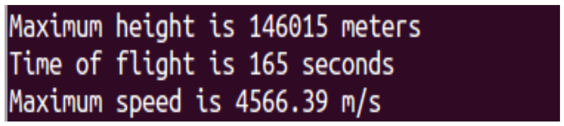


Figure 1: (left) $r(t)$ vs t (right) $v(t)$ vs t



```
Maximum height is 146015 meters
Time of flight is 165 seconds
Maximum speed is 4566.39 m/s
```

Figure 2: The final position and velocity

As $h = 0$, we can get the gravitation acceleration on the surface of Earth, $g = 9.8196$, as we expected. During the first stage, the mass of the rocket decreases with increasing time,

$$m(t) = m_0 - Bt. \tag{4}$$

Here $m_0 = 2,970,000$ kg, the burn time is 165 s, and $B = 17212$ kg/s. Therefore, after the first stage, the mass of the rocket becomes 130,020 kg. It means there are 20 kg of propellent left. The C++ code, "*Problem1_Rocket.cpp*", is shown in Appendix A.1. We plot $r(t)$ versus t and $v(t)$ versus t as shown in Figure 1. From the results, it does make sense because as time increases, the mass of rocket decreases. That means the constant applied force $F_{\text{app}} = 34020kN$ leads to larger acceleration (upward). Moreover, the gravitational force (downward) becomes smaller as explained by Eq. 3. That is, the total acceleration increases as time (upward).

From Figure 2, the final speed in the end is 4566.39 m/s. It does not achieve escape velocity calculated in part a, $v_e = 11185$ (m/s). It does make sense because if it is larger than escape velocity, then the rocket will goes to infinity. :)

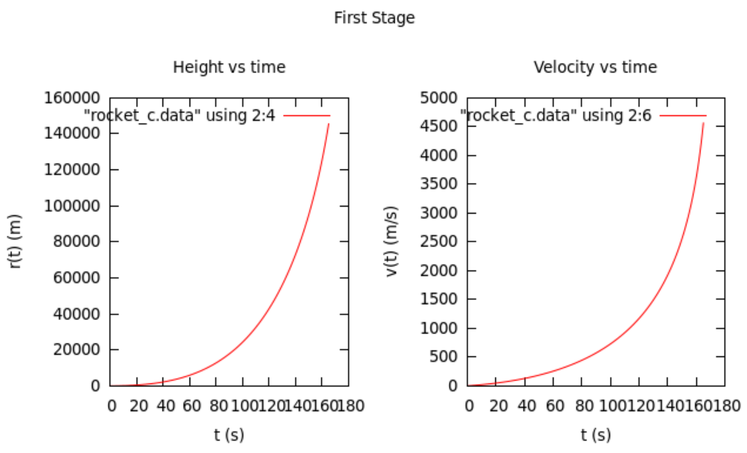
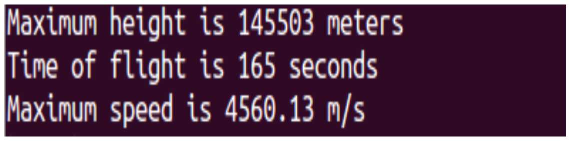


Figure 3: (left) $r(t)$ vs t (right) $v(t)$ vs t with air resistance



```
Maximum height is 145503 meters
Time of flight is 165 seconds
Maximum speed is 4560.13 m/s
```

Figure 4: The final position and velocity as we consider air resistance

1.3 Part c

Now consider air resistance in the same problem. Imagine that the cross-sectional area of the rocket is $25m^2$, and that the density of air in unit of kg/m^3 is equal to

$$\rho(h) = 1.2e^{-h/h_0}, \tag{5}$$

where h is the height off the earth's surface in meters, and $h_0 = 10,000$ m. With the same initial parameters as in (b), compare the results from (b) to the case with air resistance.

Solution:

We use the same C++ code as that in part b. But now we consider the air resistance. The results are shown in Figure 3. From these results, they are similar to those in part b. From Figure 4, we can see the difference clearly. The maximum speed and height are smaller than what we got in part b.

In order to compare the results obtained in part b, we plot them together as shown in Figure 5. They are very close!

Actually, we cannot compare them in that way because the scale we see is too large to be difficult to notice the difference. Therefore, we use the C++ code, "*Read_data.cpp*", to calculate the difference between the results we got in part b and c. Then re-plot $r(t)$ versus t and $v(t)$ versus t as shown in Figure 6.

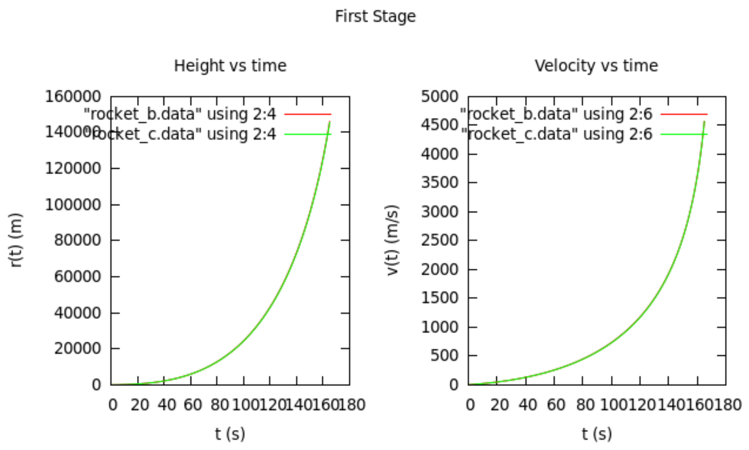


Figure 5: (left) $r(t)$ vs t (right) $v(t)$ vs t with air resistance. The results got in part b (c) are denoted by red (green) line.

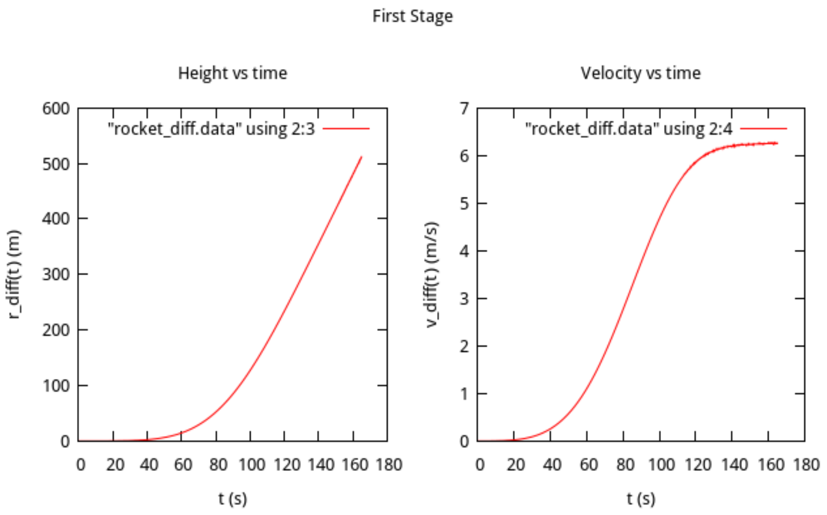


Figure 6: (left) difference in $r(t)$ vs t (right) difference in $v(t)$ vs t.

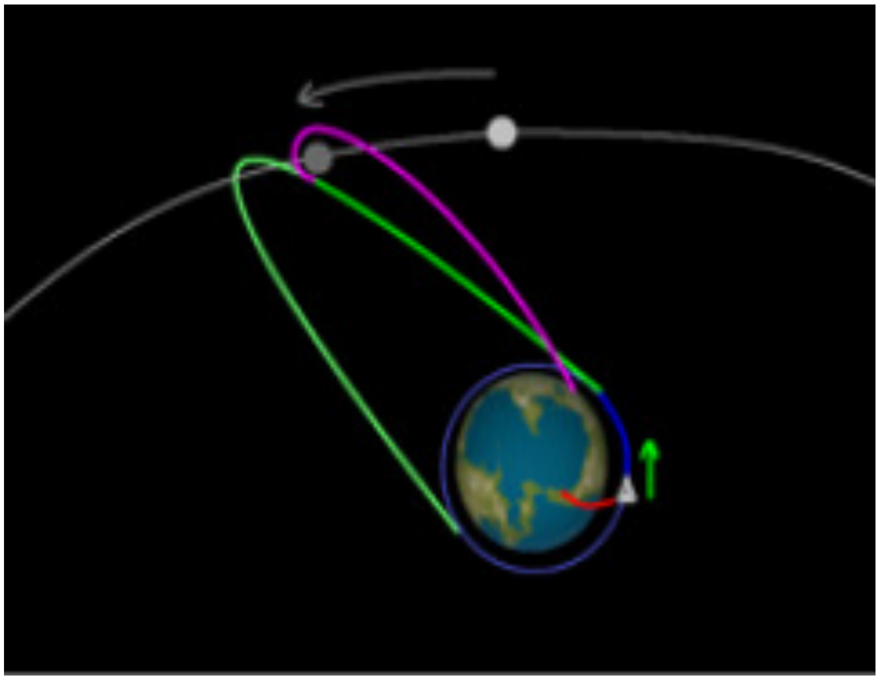


Figure 7: From exam sheet

2 Problem 2

In Kerbal Space Program (KSP), orbital mechanics are simplified using a "patched" 2-body approach, rather than solving the full N-body problem. That is, the potential is a two-body gravitational potential, and only the LARGEST gravitational force on the object is considered. Assume you have a Saturn V (as in Problem 1) initially in a circular orbit with an altitude of 100 km (this would correspond to the third stage of the Saturn V rocket, with $m = 13,500$ kg). Assume you can apply a maneuver in negligible time, such that the velocity is imparted as an impulse, with a final imparted velocity of 8000 m/s. (This is in addition to the orbital velocity of a rocket at an altitude of 100 km.)

2.1 Part a

Write an expression for $V(r)/m$, where $V(r)$ is the potential acting on the rocket, and m is the mass of the rocket, for both the "true" and "patched" earth-moon-rocket systems. Compare the two graphically by plotting $V(r)/m$ for both cases in the line connecting the earth and moon.
Solution:

Assume the distance between moon and earth is $\mathbf{R}$; between rocket and earth is $\mathbf{r}$; between rocket and moon is $\mathbf{d}$. Set moon, rocket, and earth are in line at the beginning as shown in Figure 8. From Kepler's third law, we can know that the period of rocket is less than that of moon:

$$\frac{r^3}{T_R^2} = \frac{R^3}{T_M^2}$$
$$\Rightarrow \because r < R \therefore T_R < T_M.$$

Here T_R is the orbital period of rocket and T_M is the orbital period of moon. We can consider earth is fixed here because the mass of earth is very larger than the mass of moon and rocket. We

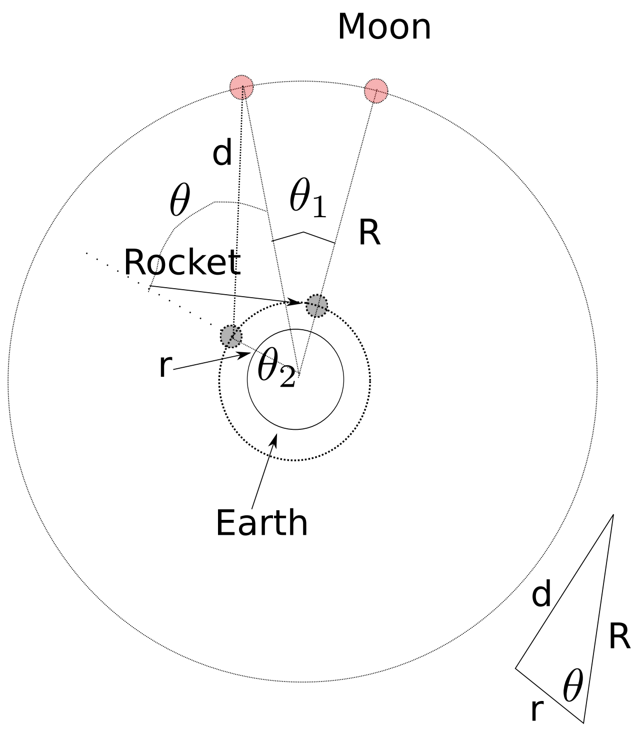


Figure 8: Earth-moon-rocket system

will see this in part c.

Next, let us write down the expression for $V(r)$, which is the potential acting on the rocket. Assume the mass of the rocket m, the mass of the moon M_m, and the mass of the earth M_E. If we consider the "**true**" earth-moon-rocket systems, then the potential acting on the rocket is written as

$$\begin{aligned} V_{\text{true}}(r) &= -\frac{GM_Em}{r} - \frac{GM_mm}{d} \quad (\because d^2 = r^2 + R^2 - 2rR\cos\theta) \\ &= -\frac{GM_Em}{r} - \frac{GM_mm}{\sqrt{r^2 + R^2 - 2rR\cos\theta}} \end{aligned} \tag{6}$$

Here G is the gravitational constant, constant R is the distance between earth and moon, and *theta* is the angle between the line connecting the earth and rocket, and the line connecting the earth and moon. In contrast, if we consider the "**patched**" earth-moon-rocket systems, then we only consider the largest gravitational force on the rocket, that is, the gravitational force of earth. Thus the expression for $V_{\text{patched}}(r)$ is given by

$$V_{\text{patched}}(r) = -\frac{GM_Em}{r}. \tag{7}$$

Finally, let us plot $V(r)/m$ in these two cases as rocket is in the line between earth and moon,

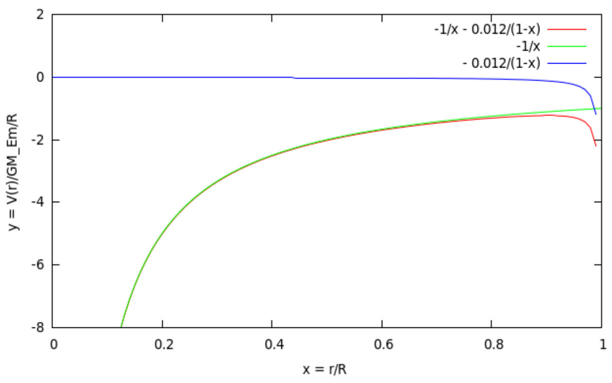


Figure 9: Earth-moon-rocket system : The green line represents the case for "patched" potential as shown in Eq. 9; the red line represents the case for "true" potential as shown in Eq. 8; the blue line represents the potential attributed by moon.

$\theta = 0$ and $0 < r < R$. In order to do so, we modify Eqs. 6 and 7 as follows:

$$\frac{V_{\text{true}}(r)}{GM_E m/R} = -\frac{1}{r/R} - \frac{M_m/M_E}{1 - r/R} \tag{8}$$

$$\frac{V_{\text{patched}(r)}}{GM_E m/R} = -\frac{1}{r/R}. \tag{9}$$

Note that G, M_E, and R are constants so we can get $V(r)/m$ by multiplying $RV(r)/GM_E m$ by GM_E/R. Thus it will not change the behavior of the results we obtain as shown in Figure 9. Here $M_m/M_E = 0.012$ is the ratio of mass of moon and mass of earth and $0 < r/R < 1$. We let $RV(r)/GM_E m$ as y and x as r/R. That means $V(r)/m$ is in unit of GM_E/R and r is in unit of R.

From Figure 9, we can see that there are distinguished differences as the rocket close to the moon, say $r/R \to 1$. However, because the second term in Eq. 8 is approximately equal to $M_m/M_E = 0.012$ as the rocket close to earth, say $r/R \approx 0$, there are no distinguished differences when $r/R \to 0$. Finally, the blue line in Figure 9 represents the potential attributed by moon. It is clear that as $0 < 1 - r/R < 0.2$, we need to take this into account in "patched" potential. This is very important when we write the code in part C.

2.2 Part b

Calculate the orbital velocity of the Saturn V rocket at an altitude of 100 km.

Solution: Now let us calculate the orbital velocity of the Saturn V rocket at an altitude of 100 km, $r = 100km$. We do not need to consider the effect of moon here because $r = 100km$ is very close to earth. From below relation, we can calculate the orbital velocity of the Saturn V easily,

$$\begin{aligned} \frac{GM_E m}{r^2} &= m \times \frac{v^2}{r} \\ \Rightarrow \quad v &= \sqrt{\frac{GM_E}{r}} = \sqrt{\frac{6.67384 \times 10^{-11}(\text{m}^3\text{kg}^{-1}\text{s}^{-2}) \times 5.972 \times 10^{24}(\text{kg})}{100 \times 10^3(\text{m})}} \\ &\approx 63131(m/s) = 63.13(km/s) \end{aligned}$$

Note that v is proportional to $\frac{1}{\sqrt{r}}$, so the orbital speed of moon (v_M) is less than the orbital speed of rocket (v_R).

2.3 Part c

Modify the "planar3body" code in Lecture 22 to handle BOTH the "true" earth-moon-rocket potential and the KSP "patched" potential. Code a "Moon encounter" for Saturn V, such that your rocket is deflected by the Moon's gravity (in real life, the astronauts would then "burn retrograd" to slow down and be captured by the moon). Do this for both the "true" and "patched" potentials. Plot the trajectory of your "Moon shot".

Consider the earth to be at rest for this purpose (also be sure to change the initial x3,y3,vx3,vy3 to be 0,0,0,0). HINT: It is easiest to work in coordinates with the earth at the origin, and use units such that the radii and orbital velocity of the moon are 1.0 and 2π, respectively.

Solution:

First of all, let us consider the "true" earth-moon-rocket potential. The mass of Earth is $M_E = 5.972 \times 10^{24}$ kg, the mass of moon is $M_m = 7.3477 \times 10^{22}$ kg, and the mass of rocket is $m = 13,500$ kg. We set the parameters for m_1 (rocket), m_2 (moon), and m_3 (earth) in unit of mass of earth. That is,

$$m_1 = \frac{m}{M_E} \approx 2.260 \times 10^{-21}\,; m_2 = \frac{M_m}{M_E} = 0.012\,; m_3 = \frac{M_E}{M_E} = 1.0. \tag{10}$$

We regard earth as fixed. That means $x_3, y_3, v_{x,3}, v_{y,3} = 0.0$. The initial position of rocket and moon are set as $x_1 = 0.0$, $y_1 = r/R = 100/384399 = 0.00026$ and $x_2 = 0.0$, $y_2 = R/R = 1.0$. Here we use $r = 100$ km and $R = 384399$ km.[3] The initial velocity of moon is $v_{x,2} = -2\pi = -6.283$ and $v_{y,2} = 0.0$. $v_{x,2}$ is given by $v_{x,2} = \frac{2\pi R}{T_m}$ and $T_m = 1, R = 1$, where T_m is the orbital period of moon and R is the orbital radius of moon. Thus the time and the length we use in the code is in the unit of the orbital period of moon and the orbital radius of moon, respectively. Next, from the Kepler's third law, we can know the relation between the orbital period of rocket (T_r) and the orbital period of moon (T_m):

$$\frac{r^3}{T_r^2} = \frac{R^3}{T_m^2} \Rightarrow \frac{T_r}{T_m} = \left(\frac{r}{R}\right)^{3/2} = (0.00026)^{3/2} = 4.195 \times 10^{-6}. \tag{11}$$

Then the orbital speed of rocket v is

$$v = \frac{2\pi r}{T_r} = \frac{2\pi R}{T_m} \times \frac{r/R}{T_r/T_m} = v_m \times \frac{r/R}{T_r/T_m} = 2\pi \times \frac{0.00026}{4.195 \times 10^{-6}} = 389.326. \tag{12}$$

Note that the speed is in unit of R/T_m. That is, we can transform the unit to km/s to check whether the results we got in part b is correct or not:

$$v = 389.326 \times \frac{384399}{27.321 \times 24 \times 60 \times 60} = 63.39(km/s). \tag{13}$$

It is consistent with the results we got in part b! Here we use the orbital period of moon $T_m = 27.321$ (days).[4] In the end, we transform the unit of final imparted velocity of 8000 m/s as follows:

$$\frac{8.0(km/s)}{63.39(km/s)} \times 389.326 = 49.134. \tag{14}$$

Therefore, we set $v_{y,1} = -389.326 + 49.134 = -340.19$ and $v_{x,1} = 0.0$. Note that the imparted velocity must slow down the original velocity of rocket because from part b, we know that the orbital velocity is proportional to $\sqrt{\frac{1}{r}}$, if we want to increase the radius of orbital, then we must

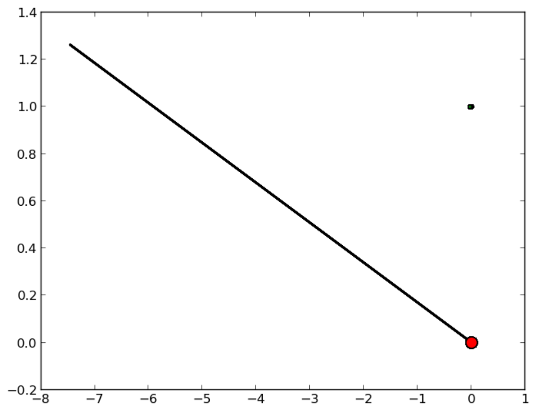


Figure 10: Earth-moon-rocket system in true potential case.

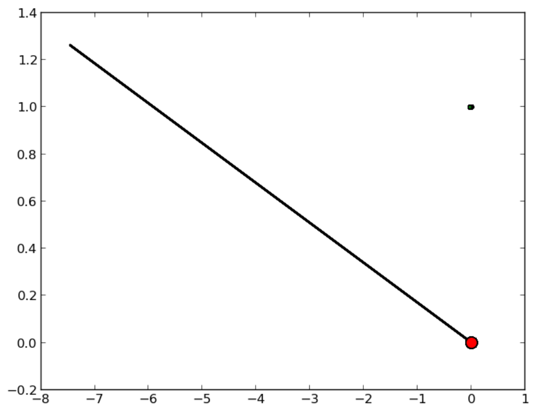


Figure 11: Earth-moon-rocket system in patched potential case.

decrease the orbital velocity. All parameters and C++ code,' '*Problem2_Truepotentail.cpp*", we use for this part is shown in Appendix A.2. The results are shown in Figure 10.

For the patched case, C++ code, "*planar3body_Patchedpotential.cpp*", is shown in Appendix A.2. We only need to change the part for acceleration. Now for earth and moon, they only feel each other. As for the rocket, it always feel the gravitational force exerted by earth. But as it is very close to moon ($r_{12} < 0.2$ in the code), then we need to add the force exerted by moon. We set the condition $r_{12} < 0.2$ because of the results of part b. The results are shown in Figure 11.

[3]See semi-major axix at http://en.wikipedia.org/wiki/Moon.
[4]See orbital period at http://en.wikipedia.org/wiki/Moon.

3 Problem 3

Consider a Gaussian wavepacket moving with initial wavenumber k_0 in 1 dimension:

$$\psi(x,0) = \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^{1/4} e^{ik_0x - \frac{(x-x_0)^2}{4\sigma^2}} \tag{15}$$

Imagine that it evolves under the time-dependent Schrödinger equation:

$$i\hbar\frac{\partial\psi(x,t)}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\psi(x,t)}{\partial x^2} + V(x)\psi(x,t) \equiv (\mathcal{T}+\mathcal{V})\psi(x,t) \tag{16}$$

Investigate the case where $V(x)$ is a quantum harmonic oscillator:

$$V(x) = \frac{1}{2}m\omega^2x^2 \tag{17}$$

For this problem, use units such that $\hbar = m = 1$, and use the "wavepacket" code from Lecture 28 to examine the time evolution of the wavepacket.

3.1 Part a

Modify the "wavepacket" code in Lecture 28 to solve the QHO potential. Plasce the minimum of the QHO potential at $L/2$, and set $\omega = \sqrt{2/L}$. Show the time evolution for $E = 1$ (a simple set a snapshots is best).

Solution:

Because we use the unit $m = 1$ and $\omega = \sqrt{2/L}$, we set V_0 in python code as

$$V_0 = \frac{1}{2}m\omega^2 = \frac{1}{2}\times\frac{2}{L} = \frac{1}{L}. \tag{18}$$

Thus, the potential is given by

$$V(x) = V_0(x-\frac{L}{2})^2. \tag{19}$$

Moreover, we set the initial position of wavepacket at the center of potential, say $L/2$. Here we set $L = 100.0$, so center of potential is $L/2 = 50.0$ and we can also calculate the width of potential at given energy E as follows:

$$E = V(x) \Rightarrow E = \frac{1}{L}(x-\frac{L}{2})^2 \Rightarrow x - \frac{L}{2} = \pm\sqrt{LE} \tag{20}$$

That means the width of potential is $2\sqrt{LE}$. The python code "*QHO.py*" is shown in Appendix A.3.

The results are as shown in Figure 12. Note that the red lines are denoted as boundaries, $E = V(x)$. From the results, we can see the wavepacket is moving back and forth between two boundaries. And there is a small probability to tunnel out of the boundary as we have learned in Quantum Mechanics course. That is out side of boundary, the wavepacket behaves like a exponential decay.

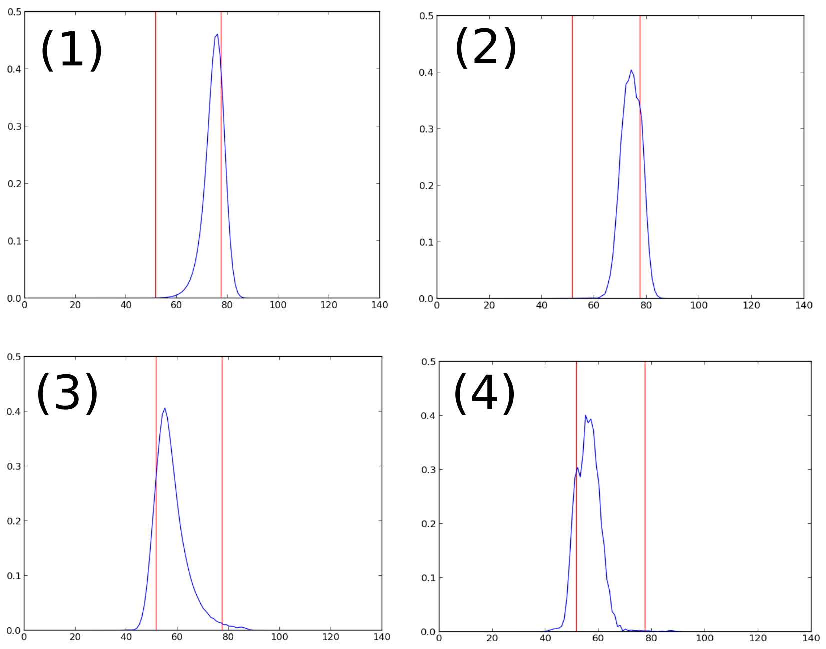


Figure 12: $E = 1.0, V(x) = V_0(x - L/2)^2$

3.2 Part b

Repeat (a), except now use the potential

$$V(x) = \frac{1}{2}m\omega^2(x^2 + x^4). \tag{21}$$

Solution:

We still use the same code as part (a). The only thing we need to do is change "self.gaussian = True" in the code from False to True. The results are shown in Figure 13.

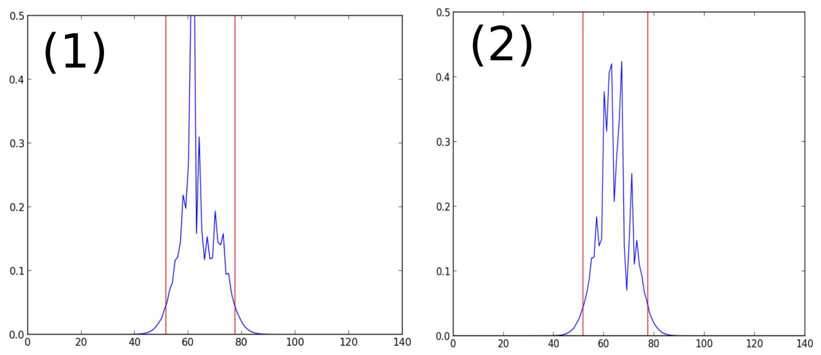


Figure 13: $E = 1.0, V(x) = V_0[(x - L/2)^2 + (x - L/2)^4]$

3.3 Part c

Do you expect the initial state that you've constructed in (a) to remain coherent throughout the evolution? Why of why not? Did you observe what you expect?

Solution:

No. I expect the initial state I've constructed in (a) cannot be coherent throughout as shown in Figure 13 because the potential $V(x) = \frac{1}{2}m\omega^2(x^2 + x^4)$ becomes very narrow at the energy level $E = 1$. We can see this from Figure 14. From the figure, we can see clearly that the boundary is at ± 3 as $V(x) = V_0(x^2 + x^4)$ and $E = 1.0$. However, we construct the initial width of the wavepacket in (a) as $L/20.0 = 5.0$ with $L = 100$. It is larger than the width of the potential. That is why we can see the wavepacket is coherent at the beginning, but after few seconds, it becomes non-coherent. Therefore, we change the initial width of the wavepacket as $L/100.0 = 1.0$. It becomes better as shown in Figure 15.

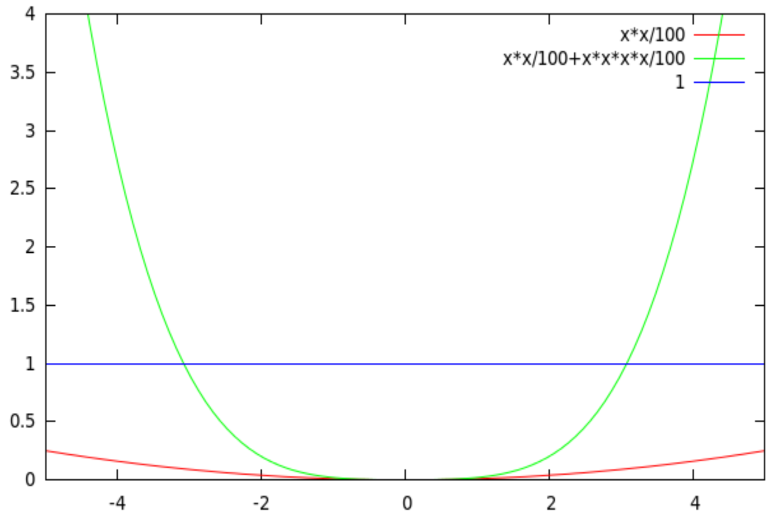


Figure 14: Plot $E = 1$, $V(x) = V_0 x^2$, and $V(x) = V_0(x^2 + x^4)$ with $V_0 = 1/L = 1/100$

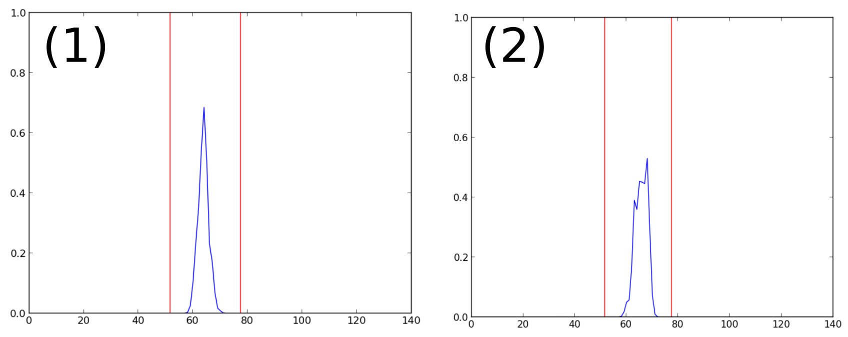
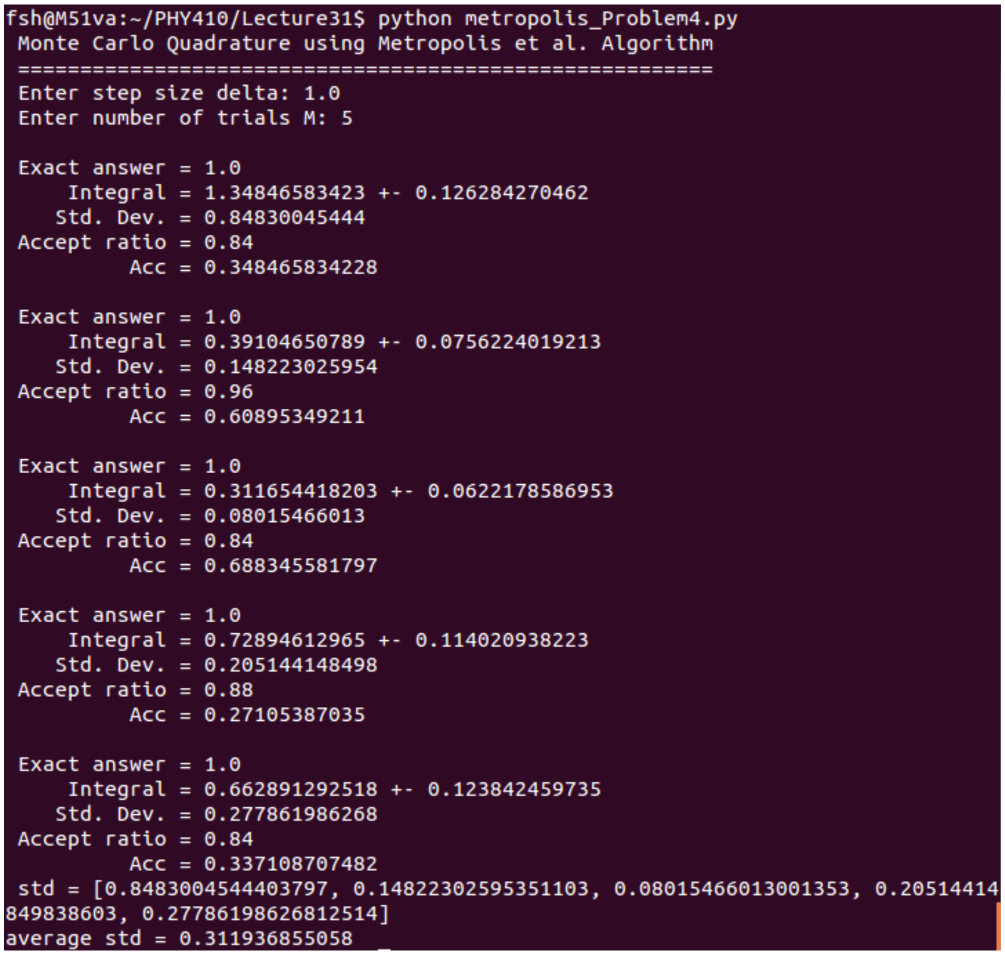


Figure 15: Set the initial width of wavepacket as $L/100 = 1.0$.



```
fsh@M51va:~/PHY410/Lecture31$ python metropolis_Problem4.py
 Monte Carlo Quadrature using Metropolis et al. Algorithm
 ========================================================
 Enter step size delta: 1.0
 Enter number of trials M: 5

 Exact answer = 1.0
     Integral = 1.34846583423 +- 0.126284270462
    Std. Dev. = 0.84830045444
 Accept ratio = 0.84
          Acc = 0.348465834228

 Exact answer = 1.0
     Integral = 0.39104650789 +- 0.0756224019213
    Std. Dev. = 0.148223025954
 Accept ratio = 0.96
          Acc = 0.60895349211

 Exact answer = 1.0
     Integral = 0.311654418203 +- 0.0622178586953
    Std. Dev. = 0.08015466013
 Accept ratio = 0.84
          Acc = 0.688345581797

 Exact answer = 1.0
     Integral = 0.72894612965 +- 0.114020938223
    Std. Dev. = 0.205144148498
 Accept ratio = 0.88
          Acc = 0.27105387035

 Exact answer = 1.0
     Integral = 0.662891292518 +- 0.123842459735
    Std. Dev. = 0.277861986268
 Accept ratio = 0.84
          Acc = 0.337108707482
 std = [0.8483004544403797, 0.14822302595351103, 0.08015466013001353, 0.20514414
849838603, 0.27786198626812514]
average std = 0.311936855058
```

Figure 16: A=5

4 Problem 4

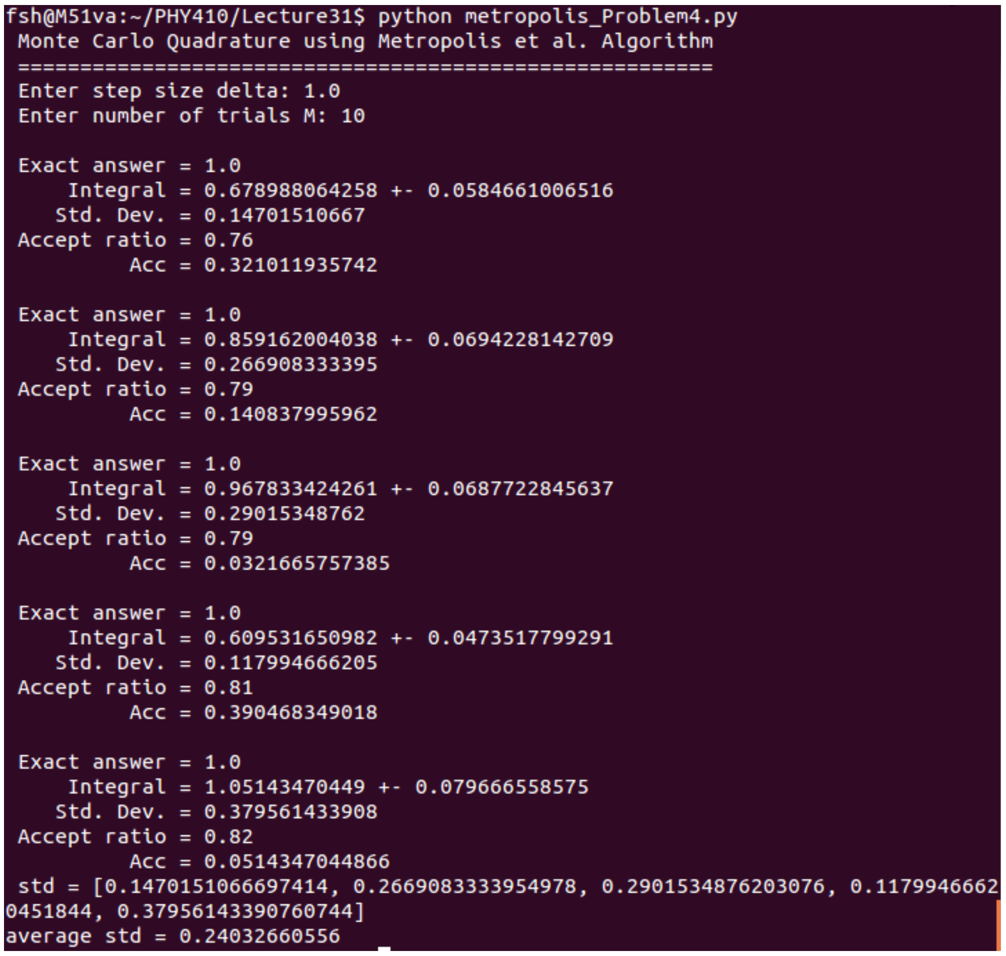
Consider the MC integration of a Gaussian distribution with the Metropolis algorithm ("metropolis" code in Lecture 31).

4.1 Part a

Modify the "metropolis" program to perform Metropolis integrations with the same number (A) for both the number of trials (M) and the number of Metropolis steps (N). Check the values of A=5, 10, 50, 100, 500, 1000, and report the accuracy of the answer (variable "ans", compared to the true value). Perform this 5 times and report the average deviation for each choice of A.

Solution:

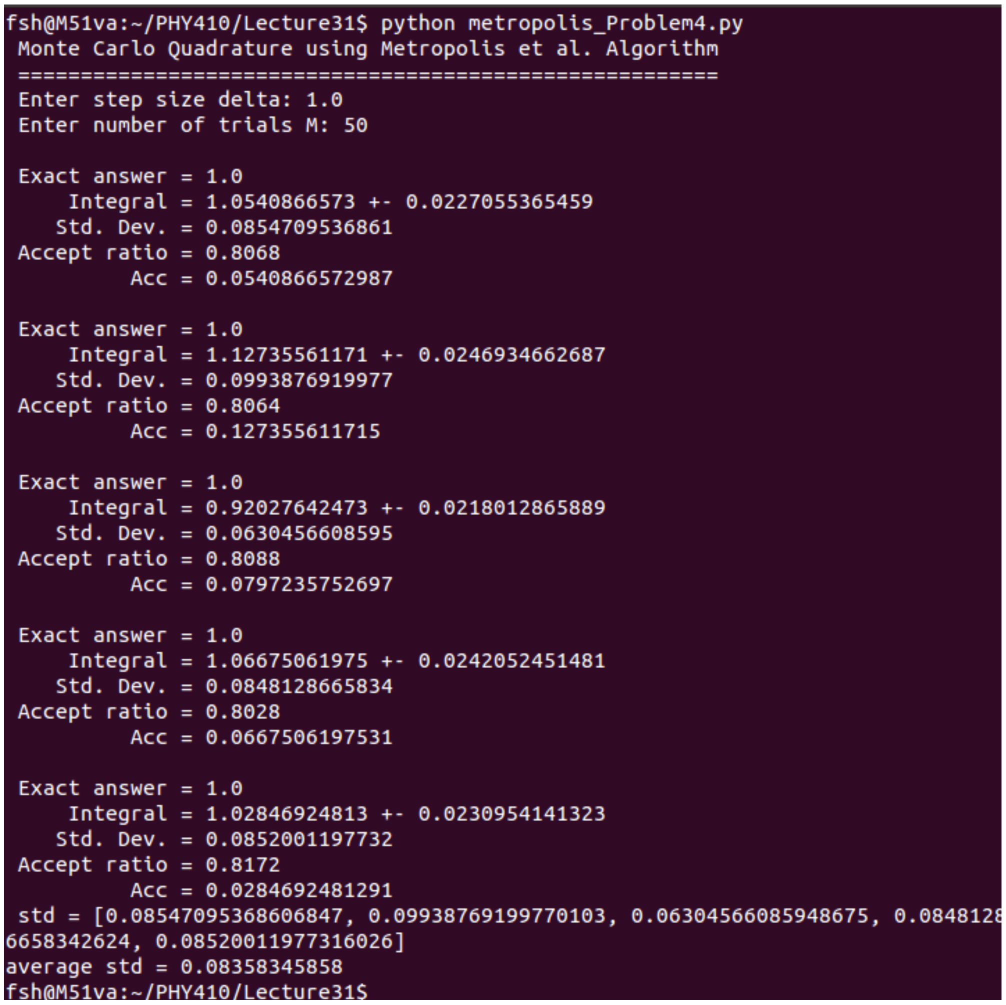
The python code, "*metropolis_Problem4.py*", is shown in Appendix A.4. In the problem, we all use the step size, 1.0. First of all, let us consider $A = 5$ case. The results are shown in Figure 16. The average deviation is 0.311936855058.



```
fsh@M51va:~/PHY410/Lecture31$ python metropolis_Problem4.py
 Monte Carlo Quadrature using Metropolis et al. Algorithm
 ========================================================
 Enter step size delta: 1.0
 Enter number of trials M: 10

 Exact answer = 1.0
     Integral = 0.678988064258 +- 0.0584661006516
    Std. Dev. = 0.14701510667
 Accept ratio = 0.76
          Acc = 0.321011935742

 Exact answer = 1.0
     Integral = 0.859162004038 +- 0.0694228142709
    Std. Dev. = 0.266908333395
 Accept ratio = 0.79
          Acc = 0.140837995962

 Exact answer = 1.0
     Integral = 0.967833424261 +- 0.0687722845637
    Std. Dev. = 0.29015348762
 Accept ratio = 0.79
          Acc = 0.0321665757385

 Exact answer = 1.0
     Integral = 0.609531650982 +- 0.0473517799291
    Std. Dev. = 0.117994666205
 Accept ratio = 0.81
          Acc = 0.390468349018

 Exact answer = 1.0
     Integral = 1.05143470449 +- 0.079666558575
    Std. Dev. = 0.379561433908
 Accept ratio = 0.82
          Acc = 0.0514347044866
 std = [0.1470151066697414, 0.2669083333954978, 0.2901534876203076, 0.1179946662
0451844, 0.37956143390760744]
average std = 0.24032660556
```

Figure 17: A=10

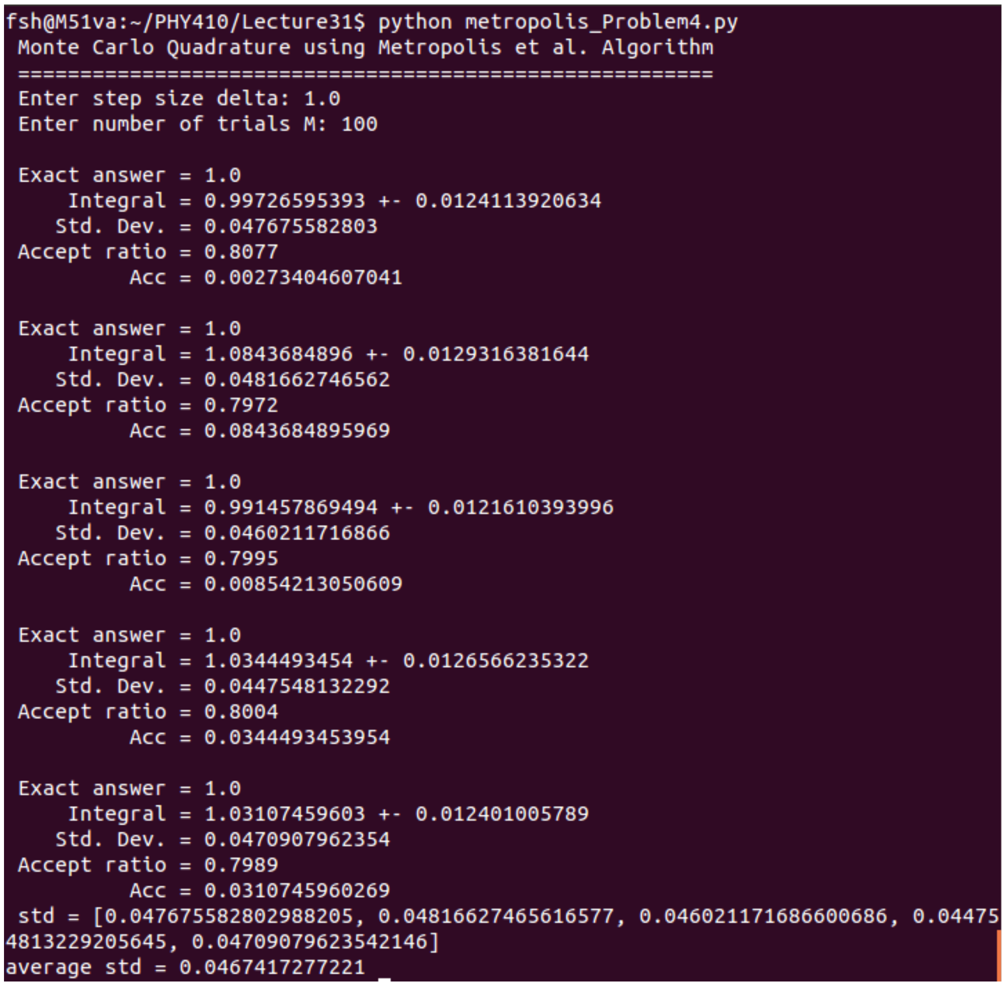
Next we consider $A = 10$. The results are shown in Figure 17. The average deviation is 0.24032660556.



```
fsh@M51va:~/PHY410/Lecture31$ python metropolis_Problem4.py
 Monte Carlo Quadrature using Metropolis et al. Algorithm
 ========================================================
 Enter step size delta: 1.0
 Enter number of trials M: 50

 Exact answer = 1.0
     Integral = 1.0540866573 +- 0.0227055365459
    Std. Dev. = 0.0854709536861
 Accept ratio = 0.8068
          Acc = 0.0540866572987

 Exact answer = 1.0
     Integral = 1.12735561171 +- 0.0246934662687
    Std. Dev. = 0.0993876919977
 Accept ratio = 0.8064
          Acc = 0.127355611715

 Exact answer = 1.0
     Integral = 0.92027642473 +- 0.0218012865889
    Std. Dev. = 0.0630456608595
 Accept ratio = 0.8088
          Acc = 0.0797235752697

 Exact answer = 1.0
     Integral = 1.06675061975 +- 0.0242052451481
    Std. Dev. = 0.0848128665834
 Accept ratio = 0.8028
          Acc = 0.0667506197531

 Exact answer = 1.0
     Integral = 1.02846924813 +- 0.0230954141323
    Std. Dev. = 0.0852001197732
 Accept ratio = 0.8172
          Acc = 0.0284692481291
 std = [0.08547095368606847, 0.09938769199770103, 0.06304566085948675, 0.0848128
6658342624, 0.08520011977316026]
average std = 0.08358345858
fsh@M51va:~/PHY410/Lecture31$
```

Figure 18: A=50

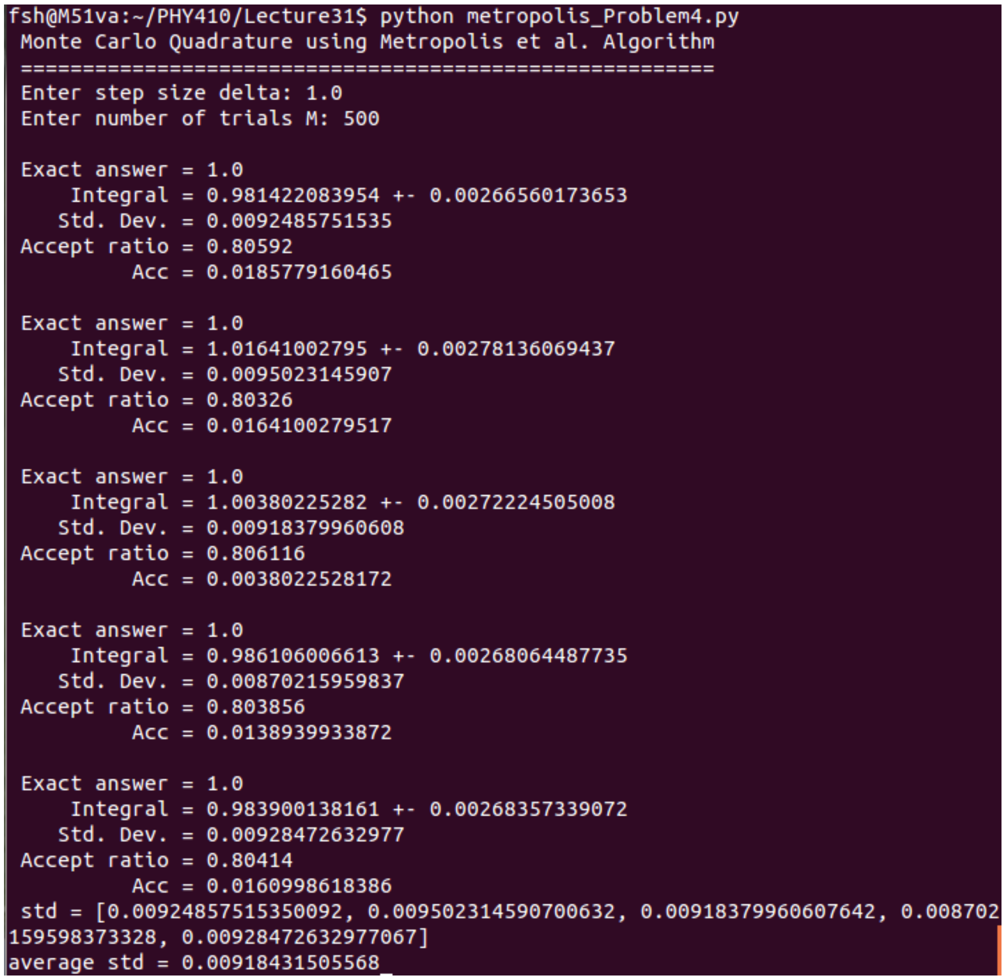
Next we consider $A = 50$. The results are shown in Figure 18. The average deviation is 0.08358345858.



```
fsh@M51va:~/PHY410/Lecture31$ python metropolis_Problem4.py
 Monte Carlo Quadrature using Metropolis et al. Algorithm
 ========================================================
 Enter step size delta: 1.0
 Enter number of trials M: 100

 Exact answer = 1.0
     Integral = 0.99726595393 +- 0.0124113920634
    Std. Dev. = 0.047675582803
 Accept ratio = 0.8077
          Acc = 0.00273404607041

 Exact answer = 1.0
     Integral = 1.0843684896 +- 0.0129316381644
    Std. Dev. = 0.0481662746562
 Accept ratio = 0.7972
          Acc = 0.0843684895969

 Exact answer = 1.0
     Integral = 0.991457869494 +- 0.0121610393996
    Std. Dev. = 0.0460211716866
 Accept ratio = 0.7995
          Acc = 0.00854213050609

 Exact answer = 1.0
     Integral = 1.0344493454 +- 0.0126566235322
    Std. Dev. = 0.0447548132292
 Accept ratio = 0.8004
          Acc = 0.0344493453954

 Exact answer = 1.0
     Integral = 1.03107459603 +- 0.012401005789
    Std. Dev. = 0.0470907962354
 Accept ratio = 0.7989
          Acc = 0.0310745960269
 std = [0.04767558280298205, 0.04816627465616577, 0.04602117168660068 6, 0.04475
4813229205645, 0.04709079623542146]
average std = 0.0467417277221
```

Figure 19: A=100

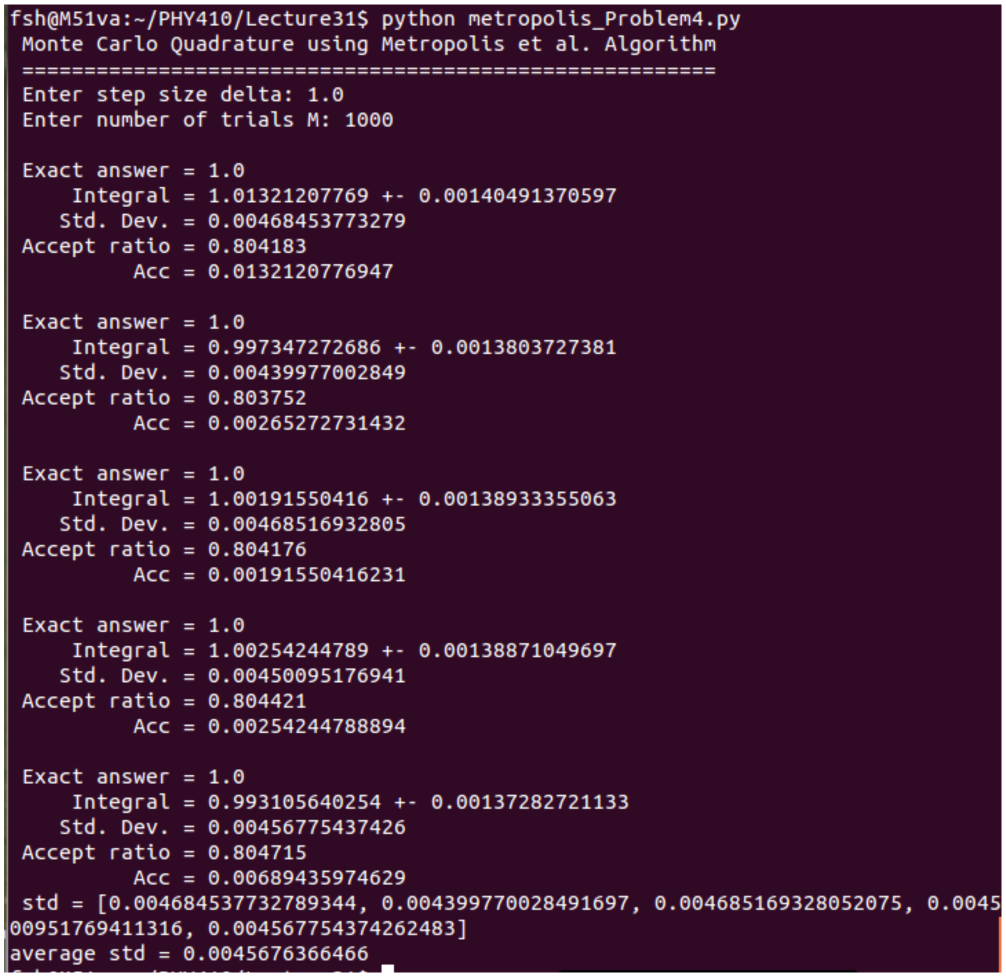
Next we consider $A = 100$. The results are shown in Figure 19. The average deviation is 0.0467417277221.



```
fsh@M51va:~/PHY410/Lecture31$ python metropolis_Problem4.py
 Monte Carlo Quadrature using Metropolis et al. Algorithm
 ========================================================
 Enter step size delta: 1.0
 Enter number of trials M: 500

 Exact answer = 1.0
     Integral = 0.981422083954 +- 0.00266560173653
    Std. Dev. = 0.0092485751535
 Accept ratio = 0.80592
          Acc = 0.0185779160465

 Exact answer = 1.0
     Integral = 1.01641002795 +- 0.00278136069437
    Std. Dev. = 0.0095023145907
 Accept ratio = 0.80326
          Acc = 0.0164100279517

 Exact answer = 1.0
     Integral = 1.00380225282 +- 0.00272224505008
    Std. Dev. = 0.00918379960608
 Accept ratio = 0.806116
          Acc = 0.0038022528172

 Exact answer = 1.0
     Integral = 0.986106006613 +- 0.00268064487735
    Std. Dev. = 0.00870215959837
 Accept ratio = 0.803856
          Acc = 0.0138939933872

 Exact answer = 1.0
     Integral = 0.983900138161 +- 0.00268357339072
    Std. Dev. = 0.00928472632977
 Accept ratio = 0.80414
          Acc = 0.0160998618386
 std = [0.00924857515350092, 0.00950231459070632, 0.00918379960607642, 0.008702
159598373328, 0.00928472632977067]
average std = 0.00918431505568
```

Figure 20: A=500

Next we consider $A = 500$. The results are shown in Figure 20. The average deviation is 0.00918431505568.



```
fsh@M51va:~/PHY410/Lecture31$ python metropolis_Problem4.py
 Monte Carlo Quadrature using Metropolis et al. Algorithm
 ========================================================
 Enter step size delta: 1.0
 Enter number of trials M: 1000

 Exact answer = 1.0
     Integral = 1.01321207769 +- 0.00140491370597
    Std. Dev. = 0.00468453773279
 Accept ratio = 0.804183
          Acc = 0.0132120776947

 Exact answer = 1.0
     Integral = 0.997347272686 +- 0.0013803727381
    Std. Dev. = 0.00439977002849
 Accept ratio = 0.803752
          Acc = 0.00265272731432

 Exact answer = 1.0
     Integral = 1.00191550416 +- 0.00138933355063
    Std. Dev. = 0.00468516932805
 Accept ratio = 0.804176
          Acc = 0.00191550416231

 Exact answer = 1.0
     Integral = 1.00254244789 +- 0.00138871049697
    Std. Dev. = 0.00450095176941
 Accept ratio = 0.804421
          Acc = 0.00254244788894

 Exact answer = 1.0
     Integral = 0.993105640254 +- 0.00137282721133
    Std. Dev. = 0.00456775437426
 Accept ratio = 0.804715
          Acc = 0.00689435974629
 std = [0.004684537732789344, 0.004399770028491697, 0.004685169328052075, 0.0045
00951769411316, 0.004567754374262483]
average std = 0.0045676366466
```

Figure 21: A=1000

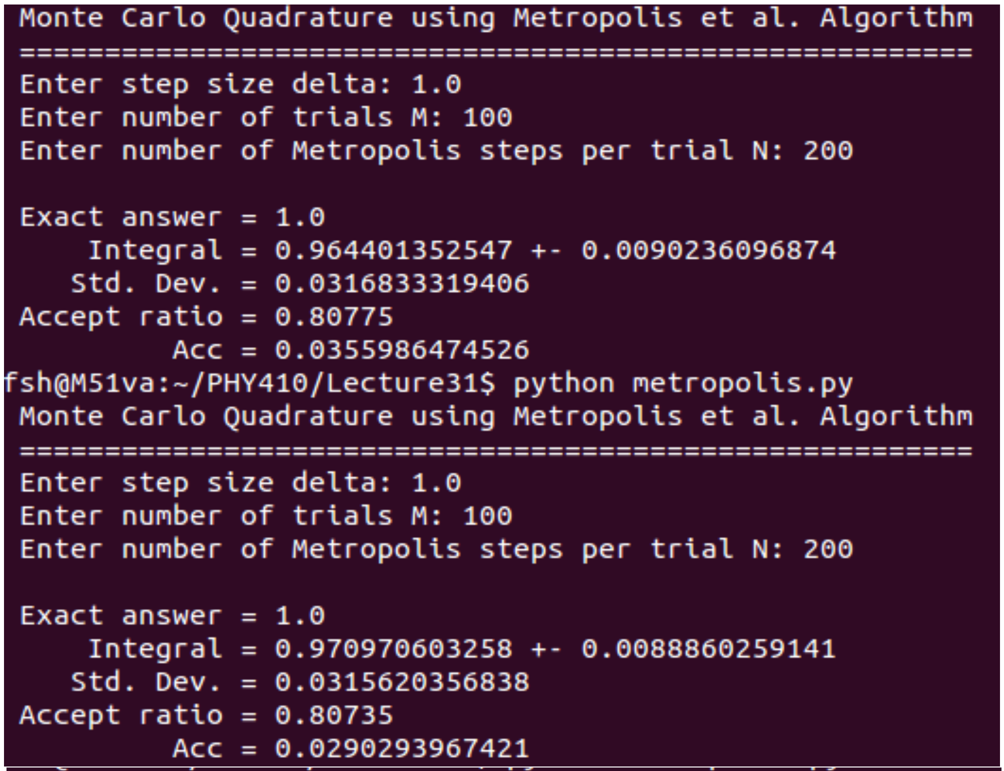
Finally, we consider $A = 1000$. The results are shown in Figure 21. The average deviation is 0.0045676366466.

4.2 Part b

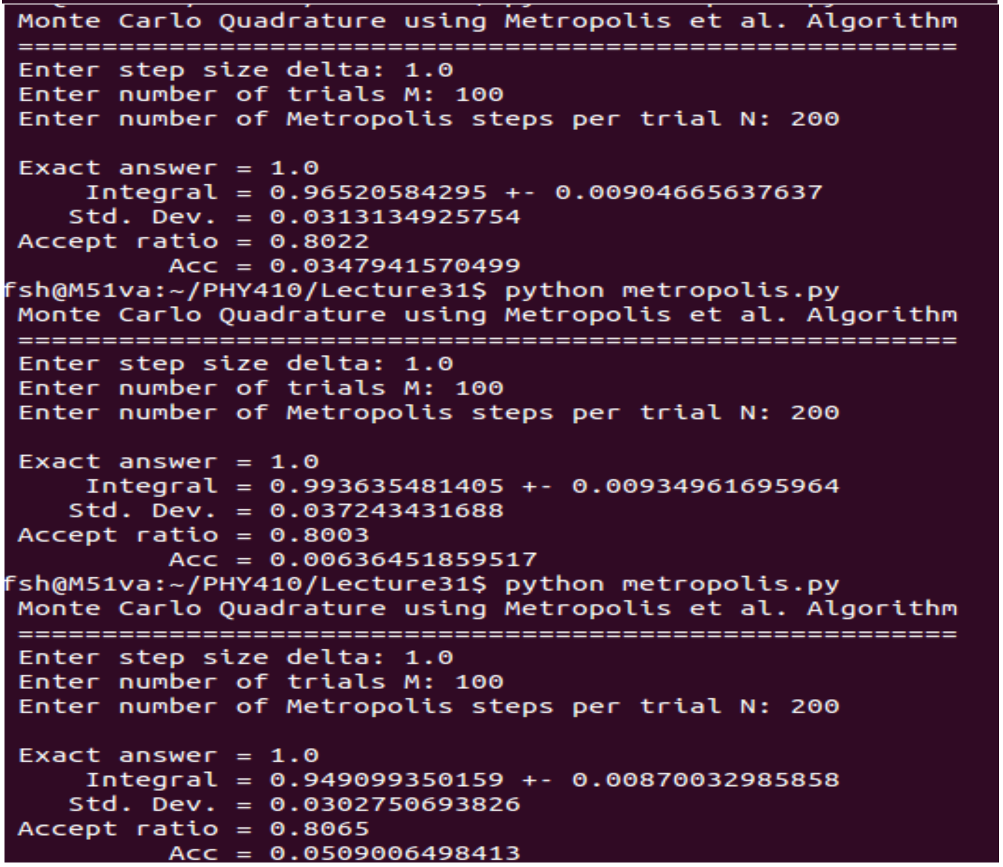
At what point do you achieve accuracy within 10% of the true answer? If you had to change either M or N (but not both) by a factor of two to achieve better accuracy, which would you do? Why? Support your case with examples from the code.

Solution:

From part a, at A=100, all trials achieve accuracy within 10% of the true answer. I will change number of trials M because more trials can lead to more accurate results. It is like tossing a coin. If we do few trials, then we cannot get the correct probability of geting head or tail. In order to prove my conclusion, I show the results with $M = 100, N = 200$ at first as shown in Figure 22. Then I show the case $M = 200, N = 100$ in Figure 23. Choosing $M = 200, N = 100$ is better than choosing $M = 100, N = 200$. Moreover, it improves the accuracy compared with the results by using $M = N = 100$.

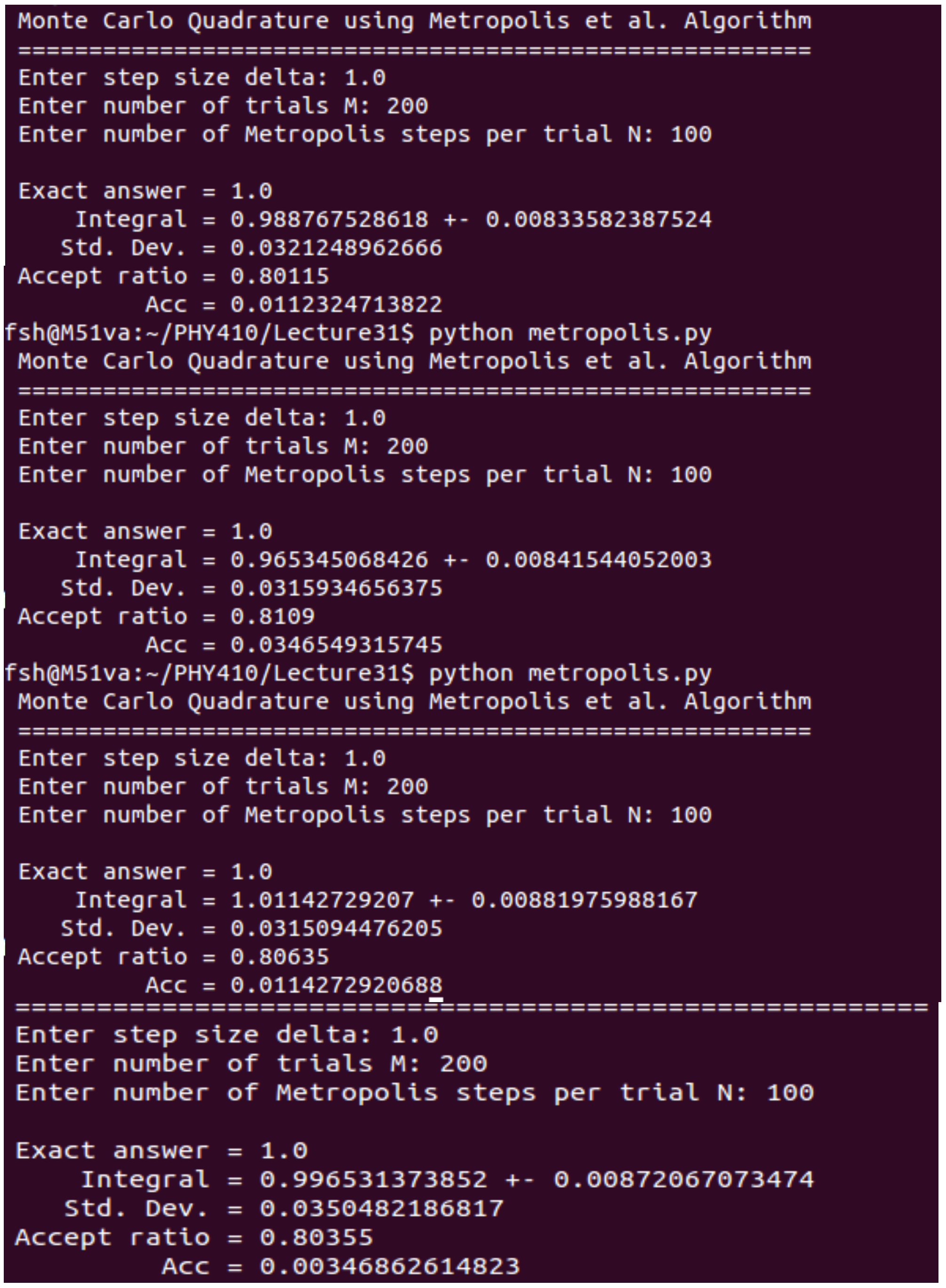


```
 Monte Carlo Quadrature using Metropolis et al. Algorithm
 ========================================================
 Enter step size delta: 1.0
 Enter number of trials M: 100
 Enter number of Metropolis steps per trial N: 200

 Exact answer = 1.0
     Integral = 0.964401352547 +- 0.0090236096874
    Std. Dev. = 0.0316833319406
 Accept ratio = 0.80775
          Acc = 0.0355986474526
fsh@M51va:~/PHY410/Lecture31$ python metropolis.py
 Monte Carlo Quadrature using Metropolis et al. Algorithm
 ========================================================
 Enter step size delta: 1.0
 Enter number of trials M: 100
 Enter number of Metropolis steps per trial N: 200

 Exact answer = 1.0
     Integral = 0.970970603258 +- 0.0088860259141
    Std. Dev. = 0.0315620356838
 Accept ratio = 0.80735
          Acc = 0.0290293967421
```



```
 Monte Carlo Quadrature using Metropolis et al. Algorithm
 ========================================================
 Enter step size delta: 1.0
 Enter number of trials M: 100
 Enter number of Metropolis steps per trial N: 200

 Exact answer = 1.0
     Integral = 0.96520584295 +- 0.00904665637637
    Std. Dev. = 0.0313134925754
 Accept ratio = 0.8022
          Acc = 0.0347941570499
fsh@M51va:~/PHY410/Lecture31$ python metropolis.py
 Monte Carlo Quadrature using Metropolis et al. Algorithm
 ========================================================
 Enter step size delta: 1.0
 Enter number of trials M: 100
 Enter number of Metropolis steps per trial N: 200

 Exact answer = 1.0
     Integral = 0.993635481405 +- 0.00934961695964
    Std. Dev. = 0.037243431688
 Accept ratio = 0.8003
          Acc = 0.00636451859517
fsh@M51va:~/PHY410/Lecture31$ python metropolis.py
 Monte Carlo Quadrature using Metropolis et al. Algorithm
 ========================================================
 Enter step size delta: 1.0
 Enter number of trials M: 100
 Enter number of Metropolis steps per trial N: 200

 Exact answer = 1.0
     Integral = 0.949099350159 +- 0.00870032985858
    Std. Dev. = 0.0302750693826
 Accept ratio = 0.8065
          Acc = 0.0509006498413
```

Figure 22: M=100, N =200



```
 Monte Carlo Quadrature using Metropolis et al. Algorithm
 ========================================================
 Enter step size delta: 1.0
 Enter number of trials M: 200
 Enter number of Metropolis steps per trial N: 100

 Exact answer = 1.0
     Integral = 0.988767528618 +- 0.00833582387524
    Std. Dev. = 0.0321248962666
 Accept ratio = 0.80115
          Acc = 0.0112324713822
fsh@M51va:~/PHY410/Lecture31$ python metropolis.py
 Monte Carlo Quadrature using Metropolis et al. Algorithm
 ========================================================
 Enter step size delta: 1.0
 Enter number of trials M: 200
 Enter number of Metropolis steps per trial N: 100

 Exact answer = 1.0
     Integral = 0.965345068426 +- 0.00841544052003
    Std. Dev. = 0.0315934656375
 Accept ratio = 0.8109
          Acc = 0.0346549315745
fsh@M51va:~/PHY410/Lecture31$ python metropolis.py
 Monte Carlo Quadrature using Metropolis et al. Algorithm
 ========================================================
 Enter step size delta: 1.0
 Enter number of trials M: 200
 Enter number of Metropolis steps per trial N: 100

 Exact answer = 1.0
     Integral = 1.01142729207 +- 0.00881975988167
    Std. Dev. = 0.0315094476205
 Accept ratio = 0.80635
          Acc = 0.0114272920688
 ========================================================
 Enter step size delta: 1.0
 Enter number of trials M: 200
 Enter number of Metropolis steps per trial N: 100

 Exact answer = 1.0
     Integral = 0.996531373852 +- 0.00872067073474
    Std. Dev. = 0.0350482186817
 Accept ratio = 0.80355
          Acc = 0.00346862614823
```

Figure 23: M=200, N =100

A Appendix

A.1 C++ code of problem 1

The following C++ code was used to obtain the results of problem 1 in this report:

```
//Problem1_Rocket.cpp

// balle - Program to compute the trajectory of a baseball
//         using the Euler method.
// Adapted from Garcia, Numerical Methods for Physics, 2nd Edition
#include <iostream>
#include <fstream>
#include <math.h>
#include <vector>
using namespace std;

int main() {
  /*In what follows, we set all vectors upward be positive. */

  //* Set initial position and velocity of the rocket
  double r1[2], v1[2], r[2], v[2], accel[2];

  //Enter initial height (meters) from the surface of earth:
  double y1 = 0.0;
  r1[0] = 0;  r1[1] = y1;      // Initial vector position
  //Enter initial speed in y-direction (m/s):
  double speed =0.0;
  const double pi = 3.141592654;
  v1[0] = 0.0;                    // Initial velocity (x)
  v1[1] = speed;                  // Initial velocity (y)
  r[0] = r1[0];  r[1] = r1[1];    // Set initial position and velocity
  v[0] = v1[0];  v[1] = v1[1];

  double airFlag, rho;
  cout << "Air resistance? (Yes:1, No:0): ";
  cin >> airFlag; //Consider air resistance or not.

  /*Choose Algorithm*/
  int euler = 0;
  cout << "Use Euler (0), Euler-Cromer (1), or Midpoint (2) ? ";
  cin >> euler;

  //* Loop until the maxStep
  // Enter time step, tau (sec):
  double tau = 0.1;
  int iStep, maxStep = 1651;    //1651 Maximum number of steps
  //Record position and velocity with considering Air resistance
  std::vector<double> time  (maxStep);
```

```
std::vector<double> xplot  (maxStep);
std::vector<double> yplot  (maxStep);
std::vector<double> vyplot(maxStep);
std::vector<double> vxplot(maxStep);


//* Set physical parameters (mass, Cd, etc.)
double Cd = 0.35;         // Drag coefficient (dimensionless)
double area = 25.0;  // Cross-sectional area of projectile (m^2)
double grav = 9.81;      // Gravitational acceleration (m/s^2)
double mass = 2970000.0;    // Initial Mass of rocket (kg)
double G = 6.67384e-11; // Gravitational constant (m^3 kg^-1 s^-2)
double M = 5.97219e24;   // Mass of Earth (kg)
double R = 6371e3;        // Mean radius of Earth (m)
double B = 17212.0;       // Burning rate (kg/s)

for( iStep=0; iStep<maxStep; iStep++ ) {

  //* Record position (computed and theoretical) for plotting
  xplot[iStep] = r[0];    // Record trajectory for plot
  yplot[iStep] = r[1];
  vxplot[iStep] = v[0];
  vyplot[iStep] = v[1];
  double t = (iStep)*tau;       // Current time
  time[iStep] = t;
  // Gravitation change with the distance
  grav = G*M/((r[1]+R)*(r[1]+R));
  // Mass change with time
  double m1 = mass - B*t; // final mass of rocket
  // Applied force by burning (N)
  double F_app = 34020e3;
  // Acceleration from applied force (m/s)
  double Accel_app = F_app/m1;

  if( airFlag == 0 )
  rho = 0;  // No air resistance
  else
  rho = 1.2*exp(-r[1]/10000.0); // Density of air (kg/m^3)
  double air_const = -0.5*Cd*rho*area/mass;  // Air resistance constant

  //Check code whether correct or not
  cout << "t= " << t << ";m= "<< m1 << ";h= " << r[1]
       << ";x= "<< r[0] << ";time= " << time[iStep]
       << ";rho= " << rho << "; g= " << grav
       << ";Acce_app = " << Accel_app << endl;

  //* Calculate the acceleration of the ball
  double normV = sqrt( v[0]*v[0] + v[1]*v[1] );
```

```
    accel[0] = air_const*normV*v[0];    // Air resistance
    accel[1] = air_const*normV*v[1];    // Air resistance
    accel[1] += (-1.0)*grav + Accel_app;    // Gravity & applied force

    //* Calculate the new position and velocity using Euler method
    if ( euler == 0 ) {         // Euler step
      r[0] += tau*v[0];
      r[1] += tau*v[1];
      v[0] += tau*accel[0];
      v[1] += tau*accel[1];
    } else if ( euler == 1 ) {// Euler-Cromer step
      v[0] += tau*accel[0];
      v[1] += tau*accel[1];
      r[0] += tau*v[0];
      r[1] += tau*v[1];
    } else {                    // Midpoint step
      double vx_last = v[0];
      double vy_last = v[1];
      v[0] += tau*accel[0];
      v[1] += tau*accel[1];
      r[0] += tau*0.5*(v[0] + vx_last);
      r[1] += tau*0.5*(v[1] + vy_last);
    }
    /*
    //* If ball reaches ground (y<0), break out of the loop
    if( r[1] < 0 )  {
      xplot[iStep] = r[0];  // Record last values computed
          yplot[iStep] = r[1];
      break;                    // Break out of the for loop
    }
    */

  }

  //* Print maximum range and time of flight
    cout << "Maximum height is " << yplot[iStep-1] << " meters" << endl;
    cout << "Time of flight is " << (iStep-1)*tau << " seconds" << endl;
    cout << "Maximum speed is " << vyplot[iStep-1] << " m/s" << endl;
    //Note that iStep becomes 1651, not 1650.


  //* Print out the plotting variables:
  //    xplot, yplot, xNoAir, yNoAir
  ofstream dataFile("rocket.data");
     dataFile << "#" << '\t' << "time" << '\t'
          << "xplot" << '\t' << "yplot" << '\t'
          << "vxplot" << '\t' << "vyplot" << '\t'
          << endl;
    for (int s = 0; s < maxStep; s++) {
      dataFile << s << '\t' << time[s] << '\t'
               << xplot[s] << '\t' << yplot[s] << '\t'
               << vxplot[s] << '\t' << vyplot[s] << '\t'
               << '\n';
    }
    dataFile.close();
/*
  ofstream plotOut("plot.txt"),
    noAirOut("noAirPlot.txt");
  int i;
  for( i=0; i<iStep; i++ ) {
    plotOut << xplot[i] << " ";
    plotOut << yplot[i] << endl;
  }
  for( i=0; i<iStep; i++ ) {
    noAirOut << xNoAir[i] << " ";
    noAirOut << yNoAir[i] << endl;
  }
*/
  return 0;
}
```

```
// read_data.cpp
#include <algorithm>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <math.h>
#include <vector>
using namespace std;
//#include "least_squares.hpp"

int main()
{


    int n= 1651; //number of data points

    //Record position and velocity with considering Air resistance
    std::vector<double> time  (n);
    std::vector<double> yplot (n);
    std::vector<double> yplot_air (n);
    std::vector<double> vyplot(n);
    std::vector<double> vyplot_air(n);
    std::vector<double> ydiff (n);
    std::vector<double> vydiff(n);

    // read the rocket_b.data
    string file_name("rocket_b.data");

    ifstream data_file(file_name.c_str());
    if (data_file.fail()) {
        cerr << "cannot open " << file_name << endl;
        return 0;
    } else
      cout << " reading data file: " << file_name << endl;

    string line;
    int j = 0;
    while (getline(data_file, line) && j < n) {
        if (line.c_str()[0] != '#') {
          stringstream sline(line);
          int number=0;
          double t=0., x=0., y=0., vx=0., vy=0.;
          sline >> number >> t >> x >> y >> vx >> vy;


          time[j] = t;
```

```
            yplot[j] = y;
            vyplot[j] = vy;
            ++j;
        }
    }
    data_file.close();

    // read the rocket_c.data
    string Rocket_air("rocket_c.data");

    ifstream data_file2(Rocket_air.c_str());
    if (data_file2.fail()) {
        cerr << "cannot open " << Rocket_air << endl;
        return 0;
    } else
      cout << " reading data file: " << Rocket_air << endl;

    string line2;
    int k = 0;
    while (getline(data_file2, line2) && k < n) {
        if (line2.c_str()[0] != '#') {
          stringstream sline(line2);
          int number=0;
          double t=0., x=0., y=0., vx=0., vy=0.;
          sline >> number >> t >> x >> y >> vx >> vy;

          yplot_air[k] = y;
          vyplot_air[k] = vy;
          ++k;
        }
    }
    data_file2.close();

    //Calculate the difference
    for(int i=0; i<n; i++){
      ydiff[i] = yplot[i] - yplot_air[i];
      vydiff[i] = vyplot[i] - vyplot_air[i];
    }

    //* Print out the plotting variables:
    // time, ydiff, vydiff
  ofstream dataFile("rocket_diff.data");
     dataFile << "#" << '\t' << "time" << '\t'
          << "y_diff" << '\t' << "vy_diff"<< endl;
     for (int s = 0; s < n; s++) {
       dataFile << s << '\t' << time[s] << '\t'
                << ydiff[s] << '\t' << vydiff[s] << '\n';
     }
    dataFile.close();


  return 0;
}
```

A.2 C++ code of problem 2

The following C++ code was used to obtain the results of problem 2 for true potential in this report:

```
//Problem2_Truepotentail.cpp
#include <cmath>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

#include "diffeq.hpp"
using namespace cpt;


class Planar3Body {

public :
  Planar3Body( double im1, double im2, double im3 ) :
    m1( im1), m2(im2), m3(im3), G( 4 * pi * pi / (m1 + m2 + m3) )
  {
  }


  // represent point in extended phase space by 13-component vector
  // [ t, r1, v1, r2, v2, r3, v3 ] = [ t, x1, y1, vx1, vy1, ... ]

  Matrix<double,1> operator()(Matrix<double,1>& trv) const {

    double t = trv[0];
    double x1 = trv[1], y1 = trv[2],  vx1 = trv[3],  vy1 = trv[4],
      x2 = trv[5], y2 = trv[6],  vx2 = trv[7],  vy2 = trv[8],
      x3 = trv[9], y3 = trv[10], vx3 = trv[11], vy3 = trv[12];

    double r12 = sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2)),
      r13 = sqrt((x1 - x3)*(x1 - x3) + (y1 - y3)*(y1 - y3)),
      r23 = sqrt((x2 - x3)*(x2 - x3) + (y2 - y3)*(y2 - y3));

    double ax1 = - G*m2*(x1 - x2)/(r12*r12*r12) - G*m3*(x1 - x3)/(r13*r13*r13),
      ay1 = - G*m2*(y1 - y2)/(r12*r12*r12) - G*m3*(y1 - y3)/(r13*r13*r13);

    double ax2 = - G*m1*(x2 - x1)/(r12*r12*r12) - G*m3*(x2 - x3)/(r23*r23*r23),
      ay2 = - G*m1*(y2 - y1)/(r12*r12*r12) - G*m3*(y2 - y3)/(r23*r23*r23);

    double ax3 = - G*m1*(x3 - x1)/(r13*r13*r13) - G*m2*(x3 - x2)/(r23*r23*r23),
      ay3 = - G*m1*(y3 - y1)/(r13*r13*r13) - G*m2*(y3 - y2)/(r23*r23*r23);
    Matrix<double,1> f(trv.size());
    f[0] = 1.0;
    f[1] = vx1,  f[2] = vy1,  f[3] = ax1,  f[4] = ay1;
    f[5] = vx2,  f[6] = vy2,  f[7] = ax2,  f[8] = ay2;
    f[9] = vx3, f[10] = vy3, f[11] = ax3, f[12] = ay3;

    return f;
  }



  static const double pi;


protected :

  // set physical constants

  double m1;          // lightest body (Rocket)
  double m2;          // next heavier body (Moon)
  double m3;          // heaviest body (Earth)
  // use units such that G*(m1+m2+m3) = 4*pi**2
  double G;

};

const double Planar3Body::pi = 4 * atan(1.0);

int main() {

    double m1, m2, m3;
    cout << " Gravitational planar 3-body problem" << endl;
    cout << " Enter standard (0) or adaptive (1) RK4 scheme : " << endl;
    bool stdrk4 = true;
    cin >> stdrk4;
    cout << " Enter m1 m2 m3: " << endl;
    cin >> m1 >> m2 >> m3;
    double x1, y1, vx1, vy1;
    cout << " Enter x1 y1 vx1 vy1: " << endl;
    cin >> x1 >> y1 >> vx1 >> vy1;
    double x2, y2, vx2, vy2;
    cout << " Enter x2 y2 vx2 vy2: " << endl;
    cin >> x2 >> y2 >> vx2 >> vy2;

    // Set position and velocity of m3 assuming center of mass at origin
    double x3 = 0.0, y3 = 0.0,
           vx3 = 0.0,
           vy3 = 0.0;
```

```
cout << " x3 = " << x3 << " y3 = " << y3
     << " vx3 = " << vx3 << " vy3 = " << vy3 << endl;

// compute net angular velocity
double
I = m1*(x1*x1 + y1*y1) + m2*(x2*x2 + y2*y2) + m3*(x3*x3 + y3*y3),
L = m1*(x1*vy1 - y1*vx1) +  m2*(x2*vy2 - y2*vx2) +  m3*(x3*vy3 - y3*vx3),
omega = L / I;
cout << " Net angular velocity = " << omega << endl;

// We can transform to the co-rotating frame v -> v - omega x r
// this non-inertial transformation will change the trajectories!
cout << " Enter 1 to transform to co-rotating frame, 0 otherwise: "<< endl;
bool yes;
cin >> yes;
if (yes) {
    vx1 = vx1 + omega * y1 ; vy1 = vy1 - omega * x1;
    vx2 = vx2 + omega * y2 ; vy2 = vy2 - omega * x2;
    vx3 = vx3 + omega * y3 ; vy3 = vy3 - omega * x3;
    cout << " vx1 = " << vx1 << " vy1 = " << vy1 << endl;
    cout << " vx2 = " << vx2 << " vy2 = " << vy2 << endl;
    cout << " vx3 = " << vx3 << " vy3 = " << vy3 << endl;
}

double t_max, dt;
cout << " Enter time step dt: ";
cin >> dt;
cout << " Enter total time to integrate: ";
cin >> t_max;
string file_name;
cout << " Enter data file name: ";
cin >> file_name;
ofstream file(file_name.c_str());


Planar3Body planar3Body( m1, m2, m3 );

double t = 0;
Matrix<double,1> trv(13);
trv[0] = t;
trv[1] = x1, trv[2]  = y1, trv[3]  = vx1, trv[4]  = vy1;
trv[5] = x2, trv[6]  = y2, trv[7]  = vx2, trv[8]  = vy2;
trv[9] = x3, trv[10] = y3, trv[11] = vx3, trv[12] = vy3;
while (t <= t_max) {
    for (int i = 0; i < trv.size(); i++)
        file << " " << trv[i];
    file << "\n";
    if ( stdrk4 )
```

```
            RK4_step(trv, dt, planar3Body );
        else
            RK4_adaptive_step(trv, dt, planar3Body, 1e-6);
        t = trv[0];
    }
    file.close();
}
```

```
//planar3body_Patchedpotential.cpp
#include <cmath>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

#include "diffeq.hpp"
using namespace cpt;


class Planar3Body {

public :
  Planar3Body( double im1, double im2, double im3 ) :
    m1( im1), m2(im2), m3(im3), G( 4 * pi * pi / (m1 + m2 + m3) )
  {
  }


  // represent point in extended phase space by 13-component vector
  // [ t, r1, v1, r2, v2, r3, v3 ] = [ t, x1, y1, vx1, vy1, ... ]

  Matrix<double,1> operator()(Matrix<double,1>& trv) const {

    double t = trv[0];
    double x1 = trv[1], y1 = trv[2],  vx1 = trv[3],  vy1 = trv[4],
      x2 = trv[5], y2 = trv[6],  vx2 = trv[7],  vy2 = trv[8],
      x3 = trv[9], y3 = trv[10], vx3 = trv[11], vy3 = trv[12];


    double r12 = sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2)),
      r13 = sqrt((x1 - x3)*(x1 - x3) + (y1 - y3)*(y1 - y3)),
      r23 = sqrt((x2 - x3)*(x2 - x3) + (y2 - y3)*(y2 - y3));

    double ax1,ay1;
    if(r12 < 0.2){
      ax1 = - G*m2*(x1 - x2)/(r12*r12*r12) - G*m3*(x1 - x3)/(r13*r13*r13);
      ay1 = - G*m2*(y1 - y2)/(r12*r12*r12) - G*m3*(y1 - y3)/(r13*r13*r13);
     }else{
      ax1 = - G*m3*(x1 - x3)/(r13*r13*r13);
      ay1 = - G*m3*(y1 - y3)/(r13*r13*r13);
     }

    double ax2 = - G*m3*(x2 - x3)/(r23*r23*r23),
      ay2 = - G*m3*(y2 - y3)/(r23*r23*r23);
    double ax3 = - G*m2*(x3 - x2)/(r23*r23*r23),
      ay3 = - G*m2*(y3 - y2)/(r23*r23*r23);

    Matrix<double,1> f(trv.size());
    f[0] = 1.0;
    f[1] = vx1,  f[2] = vy1,  f[3] = ax1,  f[4] = ay1;
    f[5] = vx2,  f[6] = vy2,  f[7] = ax2,  f[8] = ay2;
    f[9] = vx3, f[10] = vy3, f[11] = ax3, f[12] = ay3;

    return f;
  }



  static const double pi;


protected:

  // set physical constants

  double m1;        // lightest body (Rocket)
  double m2;        // next heavier body (Moon)
  double m3;        // heaviest body (Earth)
  // use units such that G*(m1+m2+m3) = 4*pi**2
  double G;

};

const double Planar3Body::pi = 4 * atan(1.0);

int main() {

    double m1, m2, m3;
    cout << " Gravitational planar 3-body problem" << endl;
    cout << " Enter standard (0) or adaptive (1) RK4 scheme : " << endl;
    bool stdrk4 = true;
    cin >> stdrk4;
    cout << " Enter m1 m2 m3: " << endl;
    cin >> m1 >> m2 >> m3;
    double x1, y1, vx1, vy1;
    cout << " Enter x1 y1 vx1 vy1: " << endl;
    cin >> x1 >> y1 >> vx1 >> vy1;
    double x2, y2, vx2, vy2;
    cout << " Enter x2 y2 vx2 vy2: " << endl;
    cin >> x2 >> y2 >> vx2 >> vy2;

    // Set position and velocity of m3 assuming center of mass at origin
```

```
double x3 = 0.0, y3 = 0.0,
       vx3 = 0.0,
       vy3 = 0.0;
cout << " x3 = " << x3 << " y3 = " << y3
     << " vx3 = " << vx3 << " vy3 = " << vy3 << endl;

// compute net angular velocity
double
I = m1*(x1*x1 + y1*y1) + m2*(x2*x2 + y2*y2) + m3*(x3*x3 + y3*y3),
L = m1*(x1*vy1 - y1*vx1) +  m2*(x2*vy2 - y2*vx2) +  m3*(x3*vy3 - y3*vx3),
omega = L / I;
cout << " Net angular velocity = " << omega << endl;

// We can transform to the co-rotating frame v -> v - omega x r
// this non-inertial transformation will change the trajectories!
cout << " Enter 1 to transform to co-rotating frame, 0 otherwise: "<< endl;
bool yes;
cin >> yes;
if (yes) {
    vx1 = vx1 + omega * y1 ; vy1 = vy1 - omega * x1;
    vx2 = vx2 + omega * y2 ; vy2 = vy2 - omega * x2;
    vx3 = vx3 + omega * y3 ; vy3 = vy3 - omega * x3;
    cout << " vx1 = " << vx1 << " vy1 = " << vy1 << endl;
    cout << " vx2 = " << vx2 << " vy2 = " << vy2 << endl;
    cout << " vx3 = " << vx3 << " vy3 = " << vy3 << endl;
}

double t_max, dt;
cout << " Enter time step dt: ";
cin >> dt;
cout << " Enter total time to integrate: ";
cin >> t_max;
string file_name;
cout << " Enter data file name: ";
cin >> file_name;
ofstream file(file_name.c_str());


Planar3Body planar3Body( m1, m2, m3 );

double t = 0;
Matrix<double,1> trv(13);
trv[0] = t;
trv[1] = x1, trv[2]  = y1, trv[3]  = vx1, trv[4]  = vy1;
trv[5] = x2, trv[6]  = y2, trv[7]  = vx2, trv[8]  = vy2;
trv[9] = x3, trv[10] = y3, trv[11] = vx3, trv[12] = vy3;
while (t <= t_max) {
    for (int i = 0; i < trv.size(); i++)
```

```
            file << " " << trv[i];
        file << "\n";
        if ( stdrk4 )
          RK4_step(trv, dt, planar3Body );
        else
          RK4_adaptive_step(trv, dt, planar3Body, 1e-6);
        t = trv[0];
    }
    file.close();
}
```

==

The following are the parameters we used in problem 2:

```
1
2.26e-21 0.012 1.0
0.0 0.00026 -340.19 0.0
0.0 1.0 -6.283 0.0
0
0.000001
0.005
earthmoonrocket_cpp.txt
```

A.3 Python code of problem 3

The following python code was used to obtain the results of problem 3 in this report:

```
#QHO.py
import math
import cmath
import time
import matplotlib
matplotlib.use('TkAgg')
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation


class Wavepacket :

    def __init__(self, N=600, L=100., dt=0.1, periodic=True):

        self.h_bar = 1.0              # Planck's constant / 2pi in natural units
        self.mass = 1.0               # particle mass in natural units

        # The spatial grid
        self.N = N                    # number of interior grid points
        self.L = L                    # system extends from x = 0 to x = L
        self.dx = L / float(N + 1)    # grid spacing
        self.dt = dt                  # time step
        self.x = []                   # vector of grid points
        self.periodic = periodic      # True = periodic, False = Dirichlet boundary


        # Initial wave packet
        self.x_0 = L / 2.0            # location of center
        self.E = 1.0                  # average energy
        self.sigma_0 = L / 20.0       # initial width of wave packet
        self.psi_norm = 1.0           # norm of psi
        self.k_0 = 0.0                # average wavenumber
        self.velocity = 0.0           # average velocity

        self.t = 0.0                  # time
        self.psi = []                 # complex wavefunction
        self.chi = []                 # wavefunction for simplified Crank-Nicholson
        self.a = []
        self.b = []                   # to represent tridiagonal elements of matrix
        self.c = []
        self.alpha = 0.0
        self.beta = 0.0               # corner elements of matrix Q

        # The potential V(x)
        self.V_0 = 1.0 / L              # V0 of potential (the coefficient)
        self.V_width = 2.0*math.sqrt(self.E * L)  # width of potential at energy le
        self.V_center = 0.50 * L        # center of harmonic potential
        self.gaussian = False           # True = add x^4 term into harmonic potential
                                        # False = harmonic potential

        # reset global vectors
        self.psi = [0 + 1j*0 for j in range(N)]
        self.chi = [0 + 1j*0 for j in range(N)]

        # reset time and the lattice
        self.t = 0.0
        self.dx = L / float(self.N + 1)
        self.x = [ float(j * self.dx) for j in range(self.N) ]

        # initialize the packet
        self.k_0 = math.sqrt(2*self.mass*self.E - self.h_bar**2 / 2 / self.sigma_0*
        self.velocity = self.k_0 / self.mass
        self.psi_norm = 1 / math.sqrt(self.sigma_0 * math.sqrt(math.pi))
        for j in range(self.N):
            exp_factor = math.exp( - (self.x[j] - self.x_0)**2 / (2 * self.sigma_0*
            self.psi[j] = (math.cos(self.k_0 * self.x[j]) + 1j * math.sin(self.k_0
            self.psi[j] *= exp_factor * self.psi_norm

        # elements of tridiagonal matrix Q = (1/2)(1 + i dt H / (2 hbar))
        for j in range(self.N):
            self.a.append( - 1j * self.dt * self.h_bar / (8 * self.mass * self.dx**
            self.b.append( 0.5 + 1j * self.dt / (4 * self.h_bar) *
                       (self.V(self.x[j]) + self.h_bar**2 / (self.mass * self.dx**2)
            self.c.append( - 1j * self.dt * self.h_bar / (8 * self.mass * self.dx**
        self.alpha = self.c[N-1]
        self.beta = self.a[0]




    def V(self, x):
        half_width = abs(0.5 * self.V_width)
        if self.gaussian:
            return self.V_0 * ( (x-self.V_center)*(x-self.V_center) + math.pow(x-se
        else:
            return self.V_0 * (x -self.V_center) * (x-self.V_center)



    def solve_tridiagonal(self, a, b, c, r, u):
        n = len(r)
        gamma = [ 0 + 1j*0 for j in range(n) ]
        beta = b[0]
        u[0] = r[0] / beta
        for j in range(1, n):
            gamma[j] = c[j-1] / beta
            beta = b[j] - a[j] * gamma[j]
            u[j] = (r[j] - a[j] * u[j-1]) / beta
        for j in range(n-2, -1, -1):
            u[j] -= gamma[j+1] * u[j+1]

    def solve_tridiagonal_cyclic(self, a, b, c, alpha, beta, r, x):
        n = len(r)
        bb = [0 + 1j*0 for j in range(self.N)]
        u = [0 + 1j*0 for j in range(self.N)]
        z = [0 + 1j*0 for j in range(self.N)]
        gamma = -b[0]
        bb[0] = b[0] - gamma
        bb[n-1] = b[n-1] - alpha * beta / gamma
        for i in range(1, n-1):
            bb[i] = b[i]
        self.solve_tridiagonal(a, bb, c, r, x)
        u[0] = gamma
        u[n-1] = alpha
        for i in range(1, n-1):
            u[i] = 0
        self.solve_tridiagonal(a, bb, c, u, z)
        fact = x[0] + beta * x[n-1] / gamma
        fact /= 1.0 + z[0] + beta * z[n-1] / gamma
        for i in range(n):
            x[i] -= fact * z[i]

#    T = 5.0                    # time to travel length L


class Animator :


    def __init__(self, periodic=True,wavepacket=None):
        self.avg_times = []
        self.periodic = periodic
        self.wavepacket = wavepacket
        self.t = 0.
        self.fig, self.ax = plt.subplots()

        self.myline = plt.axvline( x=(self.wavepacket.V_center - 0.5 * self.wavepa
                                   color='r'
            )
        self.myline = plt.axvline( x=(self.wavepacket.V_center + 0.5 * self.wavepa
                                   color='r'
            )
        self.ax.set_ylim(0,0.5)
        initvals = [ abs(ix) for ix in self.wavepacket.psi]
        self.line, = self.ax.plot(initvals)


    def update(self, data) :
        self.line.set_ydata(data)
        return self.line,

    def time_step(self):
        while True :
            start_time = time.clock()
            if self.periodic:
                self.wavepacket.solve_tridiagonal_cyclic(self.wavepacket.a, self.w
                                                         self.wavepacket.c, self.w
                                                         self.wavepacket.psi, self.
            else:
                self.wavepacket.solve_tridiagonal(self.wavepacket.a, self.wavepack
                                                  self.wavepacket.c, self.wavepack
            for j in range(self.wavepacket.N):
                self.wavepacket.psi[j] = self.wavepacket.chi[j] - self.wavepacket.
            self.t += self.wavepacket.dt;
            end_time = time.clock()
            print 'Tridiagnonal step in ' + str(end_time - start_time)
            yield [abs(ix) for ix in self.wavepacket.psi]

    def create_widgets(self):
        self.QUIT = Button(self, text="QUIT", command=self.quit)
        self.QUIT.pack(side=BOTTOM)

        self.draw = Canvas(self, width="600", height="400")
        self.draw.pack(side=TOP)

    def animate(self) :
        self.ani = animation.FuncAnimation( self.fig,        # Animate our figure
                                            self.update,     # Update function draw
                                            self.time_step,  # "frames" function do
                                            interval=50,     # 50 ms between iterat
                                            blit=False       # don't blit anything
                                            )


wavepacket = Wavepacket(N=128)
animator = Animator(periodic=True,wavepacket=wavepacket)
animator.animate()
plt.show()
```

A.4 Python code of problem 4

The following python code was used to obtain the results of problem 4 in this report:

```
#metropolis_Problem4.py
import math
import random

class Metropolis :
    def __init__(self, delta=1.0) :
        self.x = 0.0              # initial position of walker
        self.delta = delta        # step size
        self.accepts = 0          # number of steps accepted

    def step(self):
        x_trial = self.x + self.delta * random.uniform(-1, 1)
        ratio = self.P(x_trial) / self.P(self.x)
        if (ratio > random.random()):
            self.x = x_trial
            self.accepts += 1

    def P(self, x):
        # normalized Gaussian function
        return math.exp(-x**2 / 2.0) / math.sqrt(2 * math.pi)

    def f_over_w(self):
        # integrand divided by weight function
        return float(self.x**2)


print " Monte Carlo Quadrature using Metropolis et al. Algorithm"
print " =========================================================="
delta = float(input(" Enter step size delta: "))
M = int(input(" Enter number of trials M: "))
#number of Metropolis steps per trial N
# = number of trials M

Nsteps = [M,M,M,M,M]
std = []

for k in range(5):
    f_sum = 0.0           # accumulator for f(x) values
    f2_sum = 0.0          # [f(x)]**2 values
    err_sum = 0.0         # error estimates
    Acc = 0.0
    metropolis = Metropolis( delta=delta )
    for i in range(Nsteps[k]):
        avg = 0.0
        var = 0.0
        for j in range(Nsteps[k]):
            metropolis.step()
            fx = metropolis.f_over_w()
            avg += fx
            var += fx**2
        avg /= float(Nsteps[k])
        var /= float(Nsteps[k])
        var = var - avg**2
        err = math.sqrt(var / Nsteps[k])
        f_sum += avg
        f2_sum += avg**2
        err_sum += err
    ans = f_sum / float(Nsteps[k])
    acc = abs(ans - 1.0)/ 1.0
    std_dev = math.sqrt(f2_sum / Nsteps[k] - ans * ans)
    std_dev /= math.sqrt(Nsteps[k]-1.0)
    err = err_sum / float(Nsteps[k])
    err /= math.sqrt(Nsteps[k])
    std.append(std_dev)
    print ""
    print " Exact answer =", 1.0
    print "     Integral =", ans, "+-", err
    print "    Std. Dev. =", std_dev
    print " Accept ratio =", metropolis.accepts / float(Nsteps[k]*Nsteps[k])
    print "          Acc =", acc

print " std =", std #Check answer

avg_std = 0.0
for l in range(5):
  avg_std += std[l]
avg_std /= float(5)
print "average std =", avg_std
```