Homework 2

Luke Verlangieri

October 12, 2025

1. CubeSat

A)

```
#returns the specific angular momentum as a vector (h = r X v)
def spec_ang_momentum_from_v_and_r(v, r) -> list:
  return vector_functions.cross_product(r,v)
```

B)

Specific angular momentum h is equivalent to $\frac{H}{mass}$. Where H is the angluar momentum of an object. Multiplying the vector given by the specific angular momentum function with the mass of the object gives the angular momentum vector.

```
r = [7000, 0, 0] #km
v = [0, 8, 0] #km/s
mass = 6 #kg

h = orbital_equations_of_motion.spec_ang_momentum_from_v_and_r(v,r)
print(f' Specific Angular Momentum Vector {h} km^2/s \n'
      f' Angular Momentum Vector {[direction * mass for direction in h]} kg * km^2 /s\n'
      f' Angular Momentum Magnitude {vector_functions.magnitude(h) * mass} kg * km^2 /s\

OUTPUT:
    Specific Angular Momentum Vector [0, 0, 56000] km^2/s
    Angular Momentum Vector [0, 0, 336000] kg * km^2 /s
    Angular Momentum Magnitude 336000.0 kg * km^2 /s
```

For context, magnitude is another helper function that I wrote. I figured that function would be useful later. Here is the commented code for that function.

```
#returns a scalar, takes in a vector (list) of any size
def magnitude(v) -> float:
 inside_sqrt = 0
 for var in v : inside_sqrt += math.pow(var, 2) #square each value
 return math.sqrt(inside_sqrt) #take sqrt & return
```

Regardless, as can be seen from the above code. The angular momentum for the given position and velocity is $0i + 0j + 336000k$ and the magnitude of this vector is 336000 $\frac{kg*km^2}{s}$.

C & D & E)

This is the code that I used for the remaining problems and the output that resulted from it, the 1st, 2nd, and 3rd index of each list are the corresponding vectors for parts C, D, and E respectively. This is the vector I used for part E.

$$r = 3500i - 3500j + 0k \quad \& \quad v = 8i + 8j + 0k$$

```
        #Part C              Part D         Part E
r_vals = [[3500, 6062, 0], [0, 7000, 0], [3500, -3500, 0]]
v_vals = [[-6.928, 4.000, 0], [8, 0, 0], [8, 8, 0]]

for i in range(len(r_vals)):

  r = r_vals[i]
  v = v_vals[i]
  h = orbital_equations_of_motion.spec_ang_momentum_from_v_and_r(v,r)
  print(f' Specific Angular Momentum Vector {h} km^2/s \n'
        f' Angular Momentum Vector {[direction * mass for direction in h]} kg * km^2 /s\
        f' Angular Momentum Magnitude {vector_functions.magnitude(h) * mass} kg * km^2 /

OUTPUT:
 Specific Angular Momentum Vector [0.0, -0.0, 55997.536] km^2/s
 Angular Momentum Vector [0.0, -0.0, 335985.216] kg * km^2 /s
 Angular Momentum Magnitude 335985.216 kg * km^2 /s

 Specific Angular Momentum Vector [0, 0, -56000] km^2/s
 Angular Momentum Vector [0, 0, -336000] kg * km^2 /s
 Angular Momentum Magnitude 336000.0 kg * km^2 /s

 Specific Angular Momentum Vector [0, 0, 56000] km^2/s
 Angular Momentum Vector [0, 0, 336000] kg * km^2 /s
 Angular Momentum Magnitude 336000.0 kg * km^2 /s
```

Part C: Angular momentum in this case is not entirely conserved, however its possible to see small shifts like this, as they are farily miniscule and can happen from perturbations such as objects colliding with the satellite or body in orbit. This could be a computational error in the computer as well, floats can have rounding issues and what not. If I were to be critical, this would not be the same orbit, but the shift is small enough to confidently say that this is likely the same orbit from before.

Part D: Contrary to Part C, while the magnitude of angular momentum is the same, the direction of the orbit is entirely reversed from that of the original orbit. That wouldn't be

possible without a crazy torque being applied to this body, therefore this is not the same orbit.

Part E: This also works, angular momentum is conserved in this case and the direction is the same from beginnging to end. I came up with these numbers by picking a pair of vectors that would have a resluting cross product of $0i + 0j + 56,000k$. As taking the magnitude of this vector results in the same angluar momentum as the original orbit. Because this has the same direction and magnitude, this is as valid position and velocity.

2. Wobble

A)

Taking (0,0) to be the center of mass of the larger object, the distance to the center of mass (the wobble) from the larger mass is given by

$$\frac{\Sigma m_n r_n}{\Sigma m_n} = \frac{m_{small} r}{m_{large} + m_{small}}$$

```
#returns the wobble between two bodies in the given radius units
#If percent difference is true, will return a precent diff instead
#assumes m1 to be (0,0)
def wobble(m1, m2, r, percent_diff = False):
  wobble = (m2 * r)/(m2+m1) #compute center of mass
  #returns wobble or percent distance from m1 relative to r
  return wobble if not percent_diff else wobble/r
```

B)

Bodies	Larger Mass (kg)	Smaller Mass (kg)	Distance (km)	Wobble (km)	Percent of Distance (%)
Sun & Earth	1.989×10^{30}	5.974×10^{24}	1.496×10^{8}	449.33	3.00×10^{-4}
Sun & Jupiter	1.989×10^{30}	1.899×10^{27}	7.788×10^{8}	742850.9	0.0954
Earth & Moon	5.974×10^{24}	7.348×10^{22}	3.844×10^{5}	4670.66	1.215
Earth & CubeSat	5.974×10^{24}	3	6821	3.425×10^{-21}	5.022×10^{-23}
Earth & ISS	5.974×10^{24}	510000	6771	5.780×10^{-16}	8.54×10^{-18}

Here is the code that resulted in values within this table

```
RADIUS_EARTH = 6371 # km

# in km, relative to the given orbit
#values pulled from Curtis Appendix A
wobbles = {
  'Sun to Earth'    : ('Sun', 'Earth', 149.6*math.pow(10,6)),
  'Sun to Jupiter'  : ('Sun', 'Jupiter', 778.8*math.pow(10,6)),
  'Earth to Moon'   : ('Earth', 'Moon', 384.4*math.pow(10,3)),
```

```
  'Earth to CubeSat': ('Earth', 'CubeSat',450 + RADIUS_EARTH),
  'Earth to ISS'    : ('Earth', 'ISS', 400 + RADIUS_EARTH)
}

#in kg
masses = {
  'Sun'     : 1.989*math.pow(10,30),
  'Earth'   : 5.974*math.pow(10,24),
  'Moon'    : 73.48*math.pow(10,21),
  'CubeSat' : 3,
  'ISS'     : 510000,
  'Jupiter' : 1.899*math.pow(10,27)
}

for wobble in wobbles:

  (body1, body2, radius) = wobbles[wobble]
  m1 = masses[body1]
  m2 = masses[body2]

print(f'Wobble relative to larger mass: {wobble}. \n'
f'Wobble abs dist (km) {wobble(m1, m2, radius)}\n'
f'Wobble percent diff (\%) {wobble(m1, m2, radius, percent_diff=True)})
```

These results make sense, I want to take a careful look when comparing the results between the Sun & Jupiter and the Moon & Earth. Even though the Sun and Jupiter are massive bodies, the percent wobble between the Earth and the Moon is much greater than the Sun and Jupiter. To be fair, the Moon and Earth are closer in mass than than the Sun and Jupiter, and closer in distance. But I still find it interesting.

With that being said 1.215% is a considerable amount, I figure that in industry that wobble is taken into account when calculating orbits of the Moon around the Earth. I would also like to comment and say that the percent differnce is a more useful metric than the wobble in my opinion. Having the total distance is useful, but the percent of distance puts it more to scale with the other systems.

Its funny to think that a there is a wobble between a 3kg CubeSat and the Earth, even if it is utterly negligible.

3. Kepler's 2nd Law

A)

We can appraoch this problem by breaking the velocity vector into its unique compononents

$$v = v_{\perp}\hat{u_{\perp}} + v_r\hat{u}_r$$

From Curtis 2.4, equations for these components are defined as

$$r = \frac{h^2}{\mu}\frac{1}{1 + e\cos(\theta)}, \quad v_{\perp} = \frac{h}{r} = \frac{\mu(1 + e\cos(\theta))}{h}, \quad v_r = \frac{\mu}{h}e\sin(\theta)$$

Plugging $v_{\perp}$ and v_r into the vector derivation of velocity, we result in

$$v^2 = \frac{\mu^2}{h^2}(1 + e\cos(\theta))^2 + \frac{\mu^2}{h^2}e^2\sin^2(\theta)$$

Which can be resolved to be

$$v = \frac{\mu}{h}\sqrt{1 + e(e + 2\cos(\theta))}$$

B)

Putting this equation into python

```
#returns a scalar state of distance and velocity.
#returns None,None in the event that 1 + ecos(theta) = 0 (div by 0 err)
#values are returned in m, m/s
def keplers_scalar_state(h, mu, e, theta) -> tuple:

  if (1+e*math.cos(theta)) == 0 : return None, None

  #Keplers equation r = h^2/mu * (1/1+ecos(theta))
  radius = (math.pow(h,2)/mu)*(1/(1+e*math.cos(theta)))

  #Derived velocity equation
  velocity = (mu/h)*math.sqrt((1+e*(e+2*math.cos(theta))))

  return radius, velocity #return the value as a tuple
```

Using this code, I wrote this loop to extract values from this function

```
theta_vals = [0, math.pi/2, -math.pi/2, math.pi]
e_vals = [0, 0.5, 3, 1]
h = 1
mu = 1
```

```
for e in e_vals:
  print(f'For e of {e}: ')
  for theta in theta_vals:
    radius, velocity = keplers_scalar_state(h, mu, e, theta)
    print(f'Theta: {round(theta,4)} radius: {radius} m; velocity: {velocity}')
```

This resulted in the values (Where h = and $\mu = 0$, r and v have units m and m/s respectively)

e	0	$\frac{\pi}{2}$	$\frac{-\pi}{2}$	π
0	r:1.0 v:1.0	r:1.0 v:1.0	r:1.0 v:1.0	r:1.0 v:1.0
0.5	r:0.666 v:1.5	r:1.0 v:1.118	r:1.0 v:1.118	r:2.0 v:0.5
3	r:0.25 v:4	r:0.99 v:3.162	r:0.99 v:3.162	r:-0.5 v:2.0
1	r:0.5 v:2.0	r:1.0 v:1.414	r:1.0 v:1.414	div/0 error

The general trend that I notice with these values is for values of $0 < e < 1$ we see a controlled elliptical orbit. This is revealed in the velocity reducing at high theta and high r, and the velocity increasing at low r and theta (periapsis). This orbit can be visualized as an elliptical orbit.

On the other hand for e of 0, radius and velocity are just a function of h & μ, because these constants = 1 in this case, all values at e of 0 are 1. e of 0 is a perfect circular orbit, which explains the constant behavior.

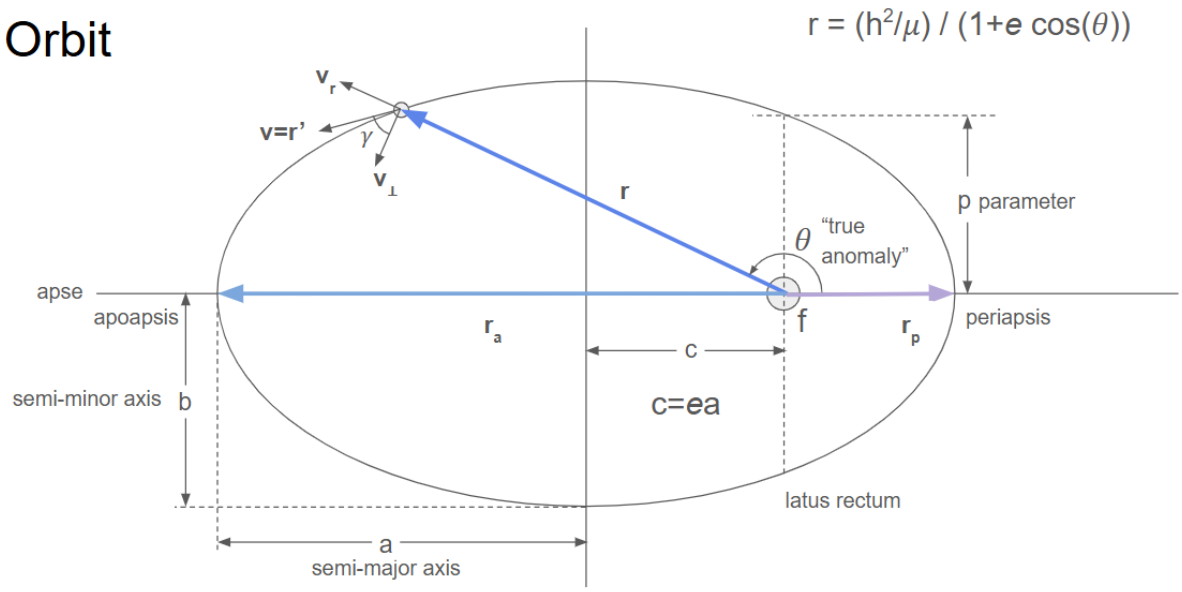
Looking at e = 3, the function resulted in a strange value for radius, this is because 1+e*cos(pi) is a negative number when e = 3, this means that the radius is not physically defined, and that we are using a value of theta outside of the domain of the true anomaly.

Lastly when e = 1, we are on the cusp of controlled orbit and because of that we see that the object is at greatest velocity at the closest true anomaly (0) and as the true anomaly grows, the velocity begins to decrease and eventually the orbit becomes undefined at a true anomaly of pi (this is an asymptote).

Key takeaways: At periapsis, the object is always at its highest velocity, and at apoapsis the object is always at its slowest velocity. e of 0 represents a circular orbit, $0 < e < 1$ is a elliptical orbit, e of 1 is a parabolic orbit, and $e > 1$ is a hyperbolic orbit.

4. Orbiting the Moon

Consider the diagram below



A)

Using the period equation defined in class, and a feature of orbit

$$T = 2\pi\sqrt{\frac{a^3}{\mu}},\ r_p = a(1-e)$$

These equations can be rewritten to take the forms

$$a = \sqrt[3]{\frac{T^2\mu}{4\pi^2}},\ e = 1 - \frac{r_p}{a}$$

From the problem statement we are given that we are orbiting the moon, periapsis is at an distance of 3000km, and the period of orbit is 7 days. From Kepler appendix B, we find that the moon has a mass of $73.48 \cdot 10^{21}$ kg. Approximating $M_{moon} + M_{Sat} \approxeq M_{moon}$, we find that $\mu \approxeq GM_{moon}$. Adding the given altitude to the radius of the moon, we find that $r_p = 3000$ Km and $\mu = GM_{\text{moon}} = (6.6743 \times 10^{-11})\,\frac{\text{m}^3}{\text{kg}\cdot\text{s}^2} \times (73.48 \times 10^{21})\,\text{kg} = 4.90 \times 10^{12}\,\frac{\text{m}^3}{\text{s}^2}$

Using these values, we find a to be

$$a = \sqrt[3]{\frac{(7\text{ day} \times 86400\,\frac{s}{day})^2(4.90 \times 10^{12}\,\frac{m^3}{s^2})}{4 \times \pi^2}} = 35674\text{ Km}$$

Then solving for e

$$e = 1 - \frac{3000\text{ Km}}{35674\text{ Km}} = 0.91612$$

B)

Here is my code for solving an orbital state

```
#Returns a dictionary with all useful values of a defined orbit
#expects si units
def orbital_state(a,e,mu) -> dict:
  r_p = a * (1-e) #rp = a(1-e)
  r_a = a * (1+e) #ra = a(1+e)
  T = (2*math.pi/math.sqrt(mu))*(a**(3/2)) # T = 2(pi) * sqrt(a^3/mu)
  spec_e = -mu/(2*a) #-mu/(2*a)
  h = math.sqrt(a*mu*(1-math.pow(e,2))) # h = sqrt(mu*a*(1-e^2))
  b = a * math.sqrt(1-(math.pow(e,2)))
  v_p = h/r_p # vp = h/rp
  v_a = h/r_a # va = h/va

  state = {
    'r_p (m)'       : r_p,
    'r_a (m)'       : r_a,
    'v_p (m/s)'     : v_p,
    'v_a (m/s)'     : v_a,
    'b (m)'         : b,
    'h (m^2/s)'     : h,
    'T (s)'         : T,
    'spec_e (J/kg)' : spec_e
   }

   return state
```

C)

Using the following code along with the above function

```
e  = 0.91512
a  = 35674000 #m
mu = 4.90 * math.pow(10,12)

orbital_state = orbital_equations_of_motion.orbital_state(a,e,mu)

for state in orbital_state:
  print(f'{state} : {orbital_state[state)}')

OUTPUT:
    r_p (m) : 2992335.1199999982
    r_a (m) : 68355664.88
    v_p (m/s) : 1771.3495735741005
    v_a (m/s) : 77.54253503506851
```

```
    b (m) : 14301855.008053133
    h (m^2/s) : 5300471538.802802
    T (s) : 604796.8775405113
    spec_e (J/kg) : -68677.46818411168
```

D)

Here is the code that I used to solve this problem. I started by solving a system of equations to find a and e

```
RADIUS_MOON = 1737000 #m
r_p = 50000 + RADIUS_MOON
r_a = 3000000

e = (r_a-r_p)/(r_a+r_p)
a = r_p/(1-e)

#Use the same mu defined above for the moon
orbital_state = orbital_equations_of_motion.orbital_state(a,e,mu)
#Uses the same loop as above
orbital_equations_of_motion.print_state(orbital_state)

OUTPUT:
    r_p (m) : 1787000.0
    r_a (m) : 3000000.0
    v_p (m/s) : 1853.8717015857478
    v_a (m/s) : 1104.2895769112438
    b (m) : 2315383.3375922875
    h (m^2/s) : 3312868730.7337313
    T (s) : 10510.708121838788
    spec_e (J/kg) : -1023605.5984959265
```

As can be seen from the output, the velocity required to be in this orbit at apoapsis is 1104.29 (m/s) or 1.104 (km/s).