Luke Verlangieri
AA310 HW6

1.

In HW5 P2 the arrival to Mars was presented (below) with arrival from "right to left", which put the periapsis on the left side of the planet - meaning that the "origin" of the perifocal frame points left. Matlab/Python default to the "origin" pointing right

(a) Create a 2D rotation matrix that would allow you to plot the standard output of your Matlab/Python functions but align it with the "arrival" picture, rather than in the default right direction.

Considering just this arrival case, the periapsis directional vector plotted in python is 180° out of phase with peripasis pointing vector of the orbit. Therefore we need to create a rotation matrix that rotates a given vector by 180°. Hence,

$$Q = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \quad \theta = 180^\circ \longrightarrow \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$

This is the rotation matrix we need to use in order to have our periapsis direction vector, and standard output, face the direction as the periapsis vector of mars perifocal frame.

(b) Make that same matrix into a 3D matrix, which correctly handles the 3rd dimension during the rotation.

This follow up step is not much different from that of the previous step, because the periapsis direction vector plotted by python is in the same plane as the vector on the diagram, and we are rotating about the out of plane vector (ω in the perifocal frame), we don't want our rotation matrix to change the orientation of the ω vector, therefore the rotation matrix is

$$Q = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(c) What are the 2 2D rotation matrices to rotate from the original transfer orbit axis? (provide numerical answers)

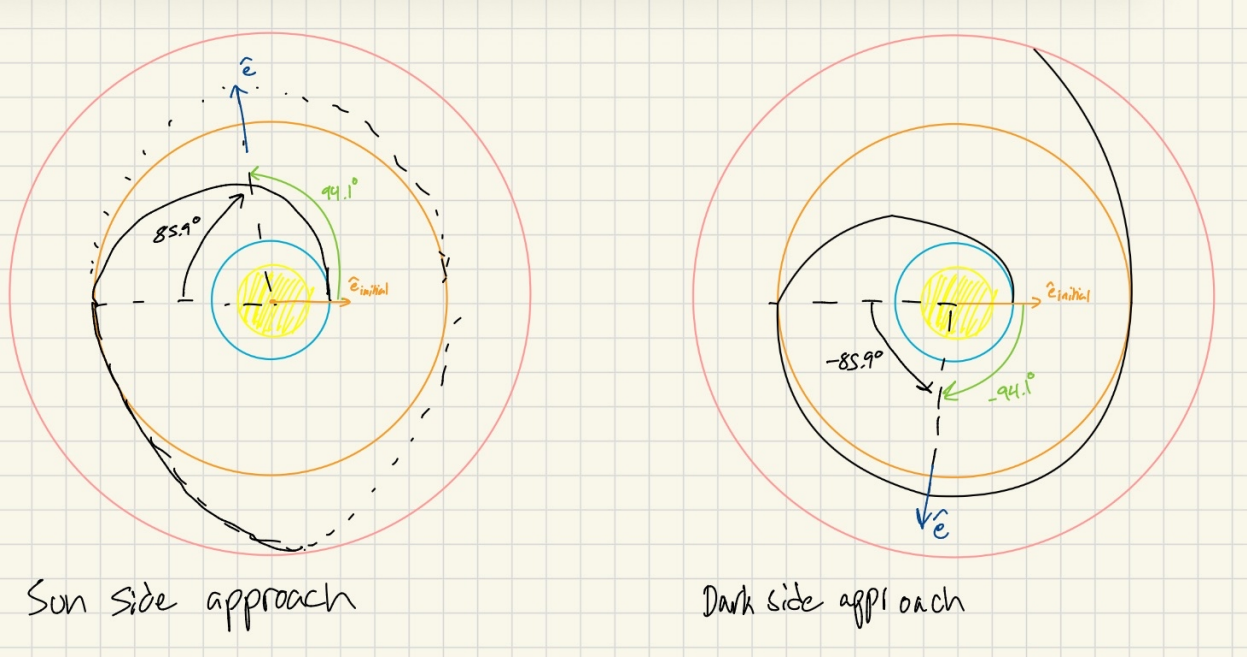

Figure 1: Comparing $\hat{e}$, sunside and darkside

In both cases, the $\hat{e}$ points toward the right of the figure in the original transfer orbit. in the sun side approach, the angle between the original transfer $\hat{e}$ and the final transfer $\hat{e}$ is $180° - 85.9° = 94.1°$. Because we are rotating in 2D, the rotation matrix takes a similar form to part a.

$$Q_{sun\ side} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \quad \theta = 94.1° \longrightarrow \begin{bmatrix} -0.0715 & 0.9974 \\ -0.9974 & -0.0715 \end{bmatrix}$$

In the dark side case, the angle of rotation is instead $180° + 85.9° = -94.1°$ which results in a dark side rotation matrix of

$$Q_{dark\ side} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \quad \theta = -94.1° \longrightarrow \begin{bmatrix} -0.0715 & -0.9974 \\ 0.9974 & -0.0715 \end{bmatrix}$$

2.

The elliptical orbit of an Earth satellite has the following orbital elements:

(a) $\Omega = 90°$ right ascension of the ascending node

(b) $i = 90°$ inclination

(c) $\omega = -45°$ argument of perigee

(d) $r_p = 12000$ km radius of perigee

(e) $e = 0.5000$ eccentricity

(f) $\theta = 45°$ true anomaly

(a) Carefully sketch the orbit (in 3D) in the geocentric reference frame XYZ. Clearly identify in your sketch the orbital parameters Ω, i, ω, θ and the position vector $\mathbf{r}$.

For this problem, I wrote a script to plot create a 3d plot of the system. Here is the script I used and my annotated plot.

```
def geocentric_3d_plot(omega, i, w, theta, a, e, deg=True):

    if deg:
        omega = np.deg2rad(omega)
        i = np.deg2rad(i)
        w = np.deg2rad(w)
        theta = np.deg2rad(theta)

    # Rotation matrices
    #Curtis 4.34
    R3_W = np.array([[ np.cos(omega),  np.sin(omega), 0],
                     [-np.sin(omega),  np.cos(omega), 0],
                     [0, 0, 1]])
    #Curtis 4.32
    R1_i = np.array([[1, 0, 0],
                     [0, np.cos(i), np.sin(i)],
                     [0, -np.sin(i), np.cos(i)]])
    #Curtis 4.34
    R3_w = np.array([[ np.cos(w),  np.sin(w), 0],
                     [-np.sin(w),  np.cos(w), 0],
                     [0, 0, 1]])

    #Rotation matrix - Curtis 4.49
    Q_peri_geo = (R3_w @ R1_i @ R3_W).T
    # the @ is syntactical sugar for matrix multiply! pretty awesome I just found
    # out about that while working on this

    # Current radius vector
```

```
r_mag = a*(1-e**2)/(1 + e*np.cos(theta))
rc = r_mag * np.array([np.cos(theta), np.sin(theta), 0])
rc_geo = Q_peri_geo @ rc

theta_deg = np.arange(0, 360)
p = a*(1-e**2)/(1 + e*np.cos(np.deg2rad(theta_deg))) * np.cos(np.deg2rad(theta_deg))
q = a*(1-e**2)/(1 + e*np.cos(np.deg2rad(theta_deg))) * np.sin(np.deg2rad(theta_deg))
w_vec = np.zeros_like(theta_deg)
r_peri = np.vstack((p, q, w_vec))
r_geo = Q_peri_geo @ r_peri

h = Q_peri_geo @ np.array([0,0,1])
h = (a/2) * h # h_vec
N = np.cross([0,0,1], h) # Node line
rp = a*(1-e) * (Q_peri_geo @ np.array([1,0,0])) #rp

#equitorial plane
x = RADIUS_EARTH * np.cos(np.deg2rad(theta_deg))
y = RADIUS_EARTH * np.sin(np.deg2rad(theta_deg))
z = np.zeros_like(theta_deg)
eq_vec = np.vstack((x,y,z))

# Plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot(
    r_geo[0], r_geo[1], r_geo[2], linewidth=1, label='Orbit')
ax.plot(
    eq_vec[0], eq_vec[1], eq_vec[2],linewidth=1, label='Equitorial Plane', color = 'gr
ax.quiver(
    0,0,0, rc_geo[0], rc_geo[1], rc_geo[2], color='red', linewidth=1, label='Current r
ax.quiver(
    0,0,0, N[0], N[1], N[2], color='green', linestyle='-', label='Node line')
ax.quiver(
    0,0,0, rp[0], rp[1], rp[2], color='orange', linestyle='-', label='Periapsis')
ax.scatter(
    [0],[0],[0], color='black', s=100, label='Earth')

ax.set_xlabel('X [km]')
ax.set_ylabel('Y [km]')
ax.set_zlabel('Z [km]')
ax.set_box_aspect([1,1,1])
max_val = a*(1+e)
ax.set_xlim([-max_val, max_val])
ax.set_ylim([-max_val, max_val])
ax.set_zlim([-max_val, max_val])
ax.legend()
plt.show()
```

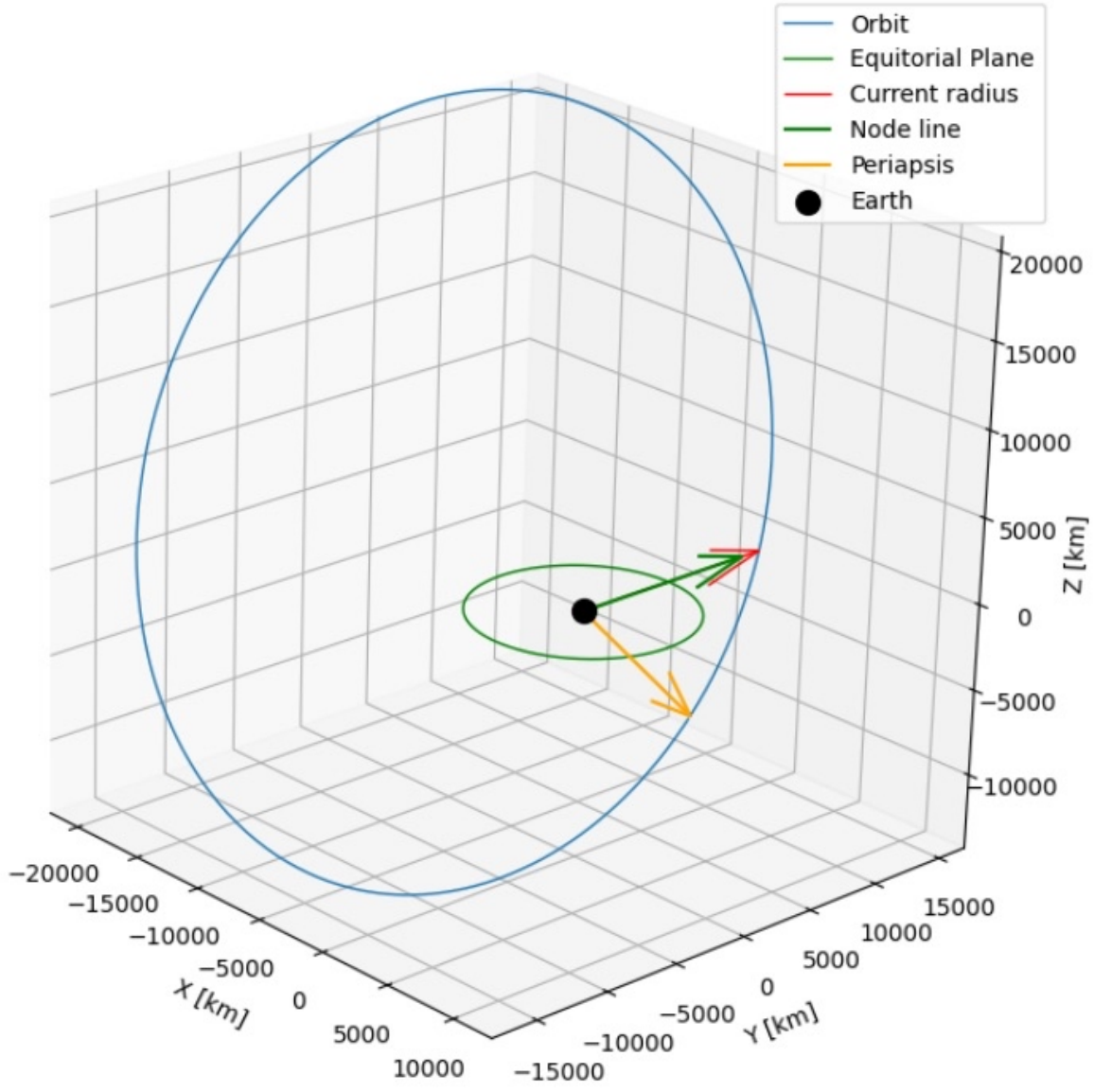

Figure 2: Raw function plot output

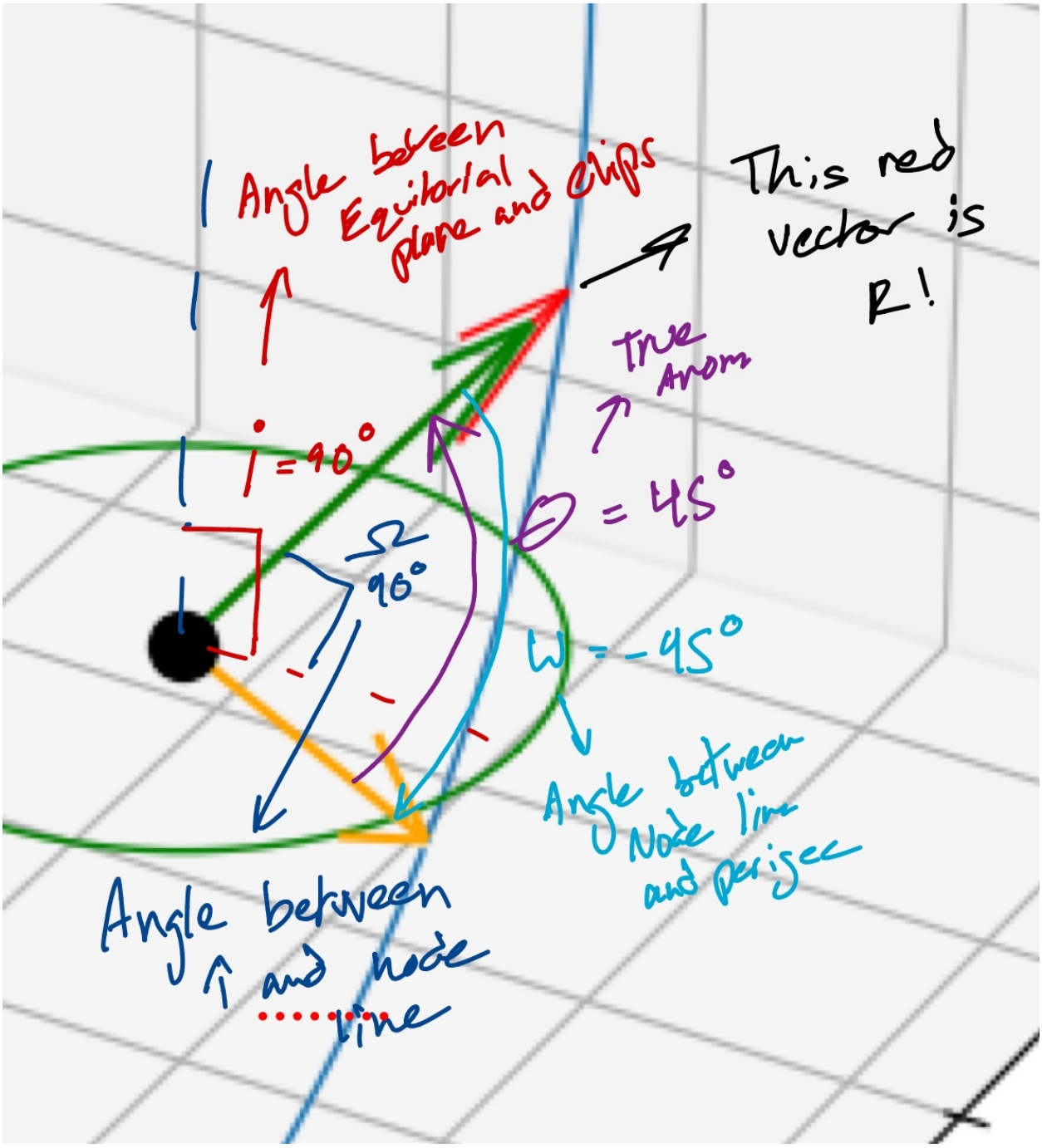

Figure 3: Annotated plot

(b) What is the position vector r in the geocentric reference frame? [Partial answer: |r| = 13,298km]

Going through my script, printing out `rc_geo` will give the position vector in the geocentric reference frame, running the script and printing that out, we see

```
[8.14287879e-13 1.32983303e+04 0.00000000e+00]
```

Because 10^{-13} might as well be 0, this can be rewritten with correct sig figs to be

$$0\hat{i} + 13298\ [km]\hat{j} + 0\hat{k}$$

3.

If we know r and v in the geocentric equatorial frame we can know everything about an orbit. Given:

$$\mathbf{r} = 1000\hat{\mathbf{I}} + 2000\hat{\mathbf{J}} + 20000\hat{\mathbf{K}} \text{ (km)}$$

$$\mathbf{v} = 3\hat{\mathbf{I}} + 3\hat{\mathbf{J}} - 0.3\hat{\mathbf{K}} \text{ (km/s)}$$

calculate all the orbital parameters $[\Omega, i, \omega, e, h, \theta]$ by coding Curtis Algorithm 4.2 into Matlab/Python and showing all your code and intermediate answers. [Partial answer: $i = 92.010°$; $e = 0.093685$]

Here is the function I used to find these parameters using Curtis alg 4.2

```
#expects r and v vectors in the for i j k
#uses curtis alg 4.2
def get_geocentric_orbital_params(r, v):

  #get magnitudes
  r_mag = vector_functions.magnitude(r)
  v_mag = vector_functions.magnitude(v)

  #Dot v with the unit vector of r to get velocity in radial dir
  v_r = (1/r_mag) * vector_functions.dot_product(r,v)

  #specific angular momentum vector
  h = vector_functions.cross_product(r,v)
  h_mag = vector_functions.magnitude(h)

  # get inclination
  i = math.acos(h[2]/h_mag) # 4.7

  #get node line vector k x h
  N = vector_functions.cross_product([0,0,1], h)
  N_mag = vector_functions.magnitude(N)

  #calculate right ascension
  if (N[1] >= 0):
    omega = math.acos(N[0]/N_mag)
  else:
    omega = math.pi * 2 - math.acos(N[0]/N_mag)

  mu = planetary_data.MU_EARTH_KM

  #get e vector
  vec1 = [((v_mag**2)- (mu/r_mag)) * component for component in r]
  vec2 = [(r_mag * v_r) * component for component in v]
  vec3 = vector_functions.subtraction(vec1, vec2)
  e = [(1/mu) * component for component in vec3]
```

```
  #get eccentricity
  e_mag = vector_functions.magnitude(e)

  #arument of perigee
  if(e[2] >= 0):
    w = math.acos(vector_functions.dot_product(N,e)/(N_mag * e_mag))
  else:
    w = math.pi * 2 - math.acos(vector_functions.dot_product(N,e)/(N_mag * e_mag))

  #get the true anomaly
  e_unit = vector_functions.unit_vector(e)
  r_unit = vector_functions.unit_vector(r)

  if(v_r >= 0):
    theta = math.acos(vector_functions.dot_product(e_unit, r_unit))
  else:
    theta = math.pi * 2 - math.acos(vector_functions.dot_product(e_unit, r_unit))

  state = {
    'h (km^2/s)'  : h_mag,
    'omega (deg)' : np.rad2deg(omega),
    'i (deg)'     : np.rad2deg(i),
    'w (deg)'     : np.rad2deg(w),
    'e'           : e_mag,
    'theta (deg)' : np.rad2deg(theta)
  }

  print(
    f'Node Vector {fmt_list(N)}, \n'
    f'H vector {fmt_list(h)} \n'
    f'e vector {fmt_list(e)}'
  )

  return state
```

Running this function with the given parameters we get

```
r = [1000, 2000, 20000]
v = [3, 3, -0.3]

geo_state = orbital_equations_of_motion.get_geocentric_orbital_params(
  r, v
)

orbital_equations_of_motion.print_state(geo_state, label='Geocentric')

OUTPUT:
```

```
Node Vector [-60300, -60600, 0] Km^2/s
H vector [-60600, 60300, -3000] km^2/s
e vector [-0.026893, -0.031213, -0.084153]

============ State of Geocentric ============
h (km^2/s) : 85542
omega (deg) : 225.14
i (deg) : 92.01
w (deg) : 296.01
e : 0.093698
theta (deg) : 160.04
```

4.

A measurement taken from the UW Jacobson Observatory (Latitude: 47.660503°, Longitude: −122.309424°, Altitude: 220.00 feet) when its local sidereal time is 36.00° makes the following observations of a space object (Based on Curtis Problems 5.12 + 5.13):

- Azimuth: 8.0000°
- Azimuth rate: 0.050000°/s.
- Elevation: 24.000°
- Elevation rate: 0.02000°/s
- Range: 8250.0 km
- Range rate: −0.25000 km/s

(a) What are the r & v vectors (the state vector) in geocentric coordinates? (Answer r = [230.51 1464.0 12199] km v = [−1.0662 7.1295 0.31939] km/s)

We can use Curtis 5.4 to solve this, here is my python implemenation of that algorithm

```
def alg_5_4(A, Ar, a, ar, p, pr, lat, sidr, H, deg=True):

  if deg:
    A  = vector_functions.deg2rad(A)
    Ar = vector_functions.deg2rad(Ar)
    a  = vector_functions.deg2rad(a)
    ar = vector_functions.deg2rad(ar)
    lat = vector_functions.deg2rad(lat)
    sidr = vector_functions.deg2rad(sidr)

  f = planetary_data.EARTH_FLATTENING_FACTOR
  Re = planetary_data.RADIUS_EARTH_KM

  big_term = (Re/math.sqrt(1-(2*f - (f**2))*(math.sin(lat)**2)))

  # Curtis 5.56
  R = [
    (big_term + H)*math.cos(lat)*math.cos(sidr),
    (big_term + H)*math.cos(lat)*math.sin(sidr),
    ((big_term*(1-f)**2) + H)*math.sin(lat)
  ]

  #5.83a
  delta = math.asin(math.cos(lat)*math.cos(A)*math.cos(a) +
  math.sin(lat)*math.sin(a))

  #5.83b
```

```
temp = math.acos((math.cos(lat)*math.sin(a) -
math.sin(lat)*math.cos(A)*math.cos(a))/math.cos(delta))
h = math.pi * 2 - temp if A < math.pi else temp

#5.83c
alpha = sidr - h

#5.68 & 5.71
p_unit = [
  math.cos(delta)*math.cos(alpha),
  math.cos(delta)*math.sin(alpha),
  math.sin(delta)
]

p_vec = [ele * p for ele in p_unit]

#5.63
r = vector_functions.addition(R, p_vec)

#2.67
omega = [0,0,planetary_data.ANG_VEL_EARTH]
R_dot = vector_functions.cross_product(omega, R)

#5.84
delta_dot = (1/math.cos(delta))*(
  (-Ar*math.cos(lat)*math.sin(A)*math.cos(a)) +
  ar * (math.sin(lat)*math.cos(a) - math.cos(lat)*math.cos(A)*math.sin(a)))

#5.85
alpha_dot = planetary_data.ANG_VEL_EARTH + (
  (Ar*math.cos(A)*math.cos(a) - ar*math.sin(A)*math.sin(a) +
  delta_dot*math.sin(A)*math.cos(a)*math.tan(delta))/
  (math.cos(lat)*math.sin(a) - math.sin(lat)* math.cos(A)* math.cos(a))
)

# 5.69
pr_unit= [
  (-alpha_dot*math.sin(alpha)*math.cos(delta) -
  delta_dot * math.cos(alpha) * math.sin(delta)),
  (alpha_dot*math.cos(alpha)*math.cos(delta) -
  delta_dot * math.sin(alpha) * math.sin(delta)),
  (delta_dot * math.cos(delta))
]

p1 = [pr * ele for ele in p_unit]
p2 = [p * ele for ele in pr_unit]

v = vector_functions.addition(R_dot, vector_functions.addition(p1, p2))
```

```
  return (r, v)
```

Here is the code I used with this function

```
#given
A = 8
Ar = 0.05
a = 24
ar = 0.02
p = 8250
pr = -0.25
H = 220/3.28
sidr = 36
lat = 47.660503

r,v = orbital_equations_of_motion.alg_5_4(
  A, Ar, a, ar, p, pr, lat, sidr, H
)

vec_units = ['I','J','K']
vector_functions.print_vector_formatted(
    r, vec_units, units='km', label='r')
vector_functions.print_vector_formatted(
    v, vec_units, units='km/s',label='v')

OUTPUT:

    r:  267.02 I 1490.5 J 12248 K km
    v:  -1.0681 I 7.1322 J 0.31939 K km/s
```

(b) Calculate the orbital parameters $[\Omega, i, \omega, e, h, \theta]$ of the satellite. (For your thoughts: what type of object could this be?) (Partial Answer $e = 0.61784$, $i = 87.902$)

We can get the rest of the parameters of the geocentric state using the function from question 3, plugging r and v in to that function, we get

```
============ State of Geocentric ============
h (km^2/s) : 87942
omega (deg) : 278.62
i (deg) : 87.721
w (deg) : 72.647
e : 0.62586
theta (deg) : 24.024
```