HW 3

Tory Smith

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint, solve_ivp
```

1. Assume an orbit plane coordinate system with a gravitational parameter of 1, i.e., μ = 1.

a. Generate a “true” solution by numerically integrating the equations of motion for the initial condition

Save the values of the state vector $X(t_i)$ for t_i = i ·10 time units (TU); i = 0,...,10. Provide X(ti) for t1 and t10 in the writeup. In your write-up, please indicate which integrator you used, what the tolerance was set to, and any other details necessary. Note, if you use a fixed time-step integrator, set the time-step to be smaller than 10 TU, but only save the data at 10 TU intervals.

In [264…
```
t = np.arange(0, 110, 10)
mu = 1
#initial Conditions
X0 = np.array([1, 0, 0, 1])

#equations of motion
def two_body(t, R, mu):
    r = np.linalg.norm(R[0:2])
    r_ddot = -R[0:2]/r**3
    return np.concatenate([R[2:4], r_ddot])


#numeric integration
sol_two_body = solve_ivp(two_body, [0, 100], X0, args=(mu,), t_eval=t, rtol=3E-14, atol=1E
```

In [267…
```
X = sol_two_body.y.T
print("X(t_0):", X[0])
print("X(t_1):",X[1])
print("X(t_10):",X[10])
```

```
X(t_0): [1. 0. 0. 1.]
X(t_1): [-0.83907153 -0.54402111  0.54402111 -0.83907153]
X(t_10): [ 0.86231887 -0.50636564  0.50636564  0.86231887]
```

b. Perturb the previous set of initial conditions by an amount X∗(t0) = X(t0) −δX(t0). Numerically integrate this “nominal” trajectory along with the associated state transition matrix to find X∗(ti) and Φ(ti,t0) at ti = i ·10 TU; i = 0,...,10.

In [185…
```
def two_body_stm(t, R, mu):
    r = np.linalg.norm(R[0:2])

    phi = R[4:].reshape((4, 4))
    A = np.zeros([4, 4])
    A[0, 2] = 1
    A[1, 3] = 1
    A[2, 0] = 3 * mu * R[0]**2 / r**5 - mu/r**3
    A[3, 0] = 3 * mu * R[0] * R[1] / r**5
    A[2, 1] = 3 * mu * R[0] * R[1] / r**5
    A[3, 1] = 3 * mu * R[1]**2 / r**5 - mu / r**3

    phi_dot = np.matmul(A, phi)
```

```
    r_ddot = -R[0:2]/r**3

    return np.concatenate([R[2:4], r_ddot, phi_dot.ravel()])
```

In [144…

```
t = np.arange(0, 110, 10)
dX = np.array([1E-6, -1E-6, 1E-6, 1E-6])

#initial conditions
Xd0 = X0 - dX
phi = np.eye(4)
R0 = np.concatenate([Xd0, phi.ravel()])

#numeric integration
sol_two_body_stm = solve_ivp(two_body_stm, [0, 110], R0, args=(mu,), t_eval=t, rtol=3E-14,
```

Provide X∗(ti) and Φ(ti,t0) at t1 and t10 in the write-up.

In [270…

```
print("X*(t_0):",sol_two_body_stm.y.T[0, 0:4])
print("phi(t_0, t_0):\n ", sol_two_body_stm.y.T[0, 4:].reshape((4, 4)))
print("X(t_1):",sol_two_body_stm.y.T[1, 0:4])
print("phi(t_1, t_0):\n ",sol_two_body_stm.y.T[1, 4:].reshape((4, 4)))
print("X(t_10):",sol_two_body_stm.y.T[10, 0:4])
print("phi(t_10, t_0):\n ",sol_two_body_stm.y.T[10, 4:].reshape((4, 4)))
```

```
X*(t_0): [ 9.99999e-01  1.00000e-06 -1.00000e-06  9.99999e-01]
phi(t_0, t_0):
  [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
X(t_1): [-0.8390311  -0.54407149  0.54407612 -0.83904124]
phi(t_1, t_0):
  [[-19.29631747  -1.00059195  -1.54462409 -20.59227468]
 [ 24.5395369    2.54304004   3.38202244  24.99596383]
 [-26.62844858  -1.24704108  -2.08602899 -27.54137483]
 [-15.07542265  -1.45709728  -2.00114421 -14.66741225]]
X(t_10): [ 0.86262336 -0.50584396  0.50584569  0.8626233 ]
phi(t_10, t_0):
  [[-1.51284032e+02 -6.96433460e-02 -5.75183991e-01 -1.52539455e+02]
 [-2.60234514e+02  8.81235607e-01  1.91322895e-02 -2.60670088e+02]
 [ 2.59154448e+02  3.74643453e-01  1.23674844e+00  2.60026380e+02]
 [-1.52127911e+02  3.66712857e-01 -1.38829570e-01 -1.51639213e+02]]
```

c. For this problem, Φ(ti,t0) is symplectic. Demonstrate this for Φ(t10,t0) by multiplying it by Φ−1(t10,t0), given by Eq. 4.2.22 in the text. Provide Φ−1(t10,t0) and show that the product with Φ(t10,t0) is the identity matrix.

In [271…

```
phi10 = sol_two_body_stm.y.T[10, 4:].reshape((4, 4))
print("phi(t_10, t_0)^-1:\n", np.linalg.inv(phi10))
print("phi(t_10, t0)*phi(t_10, t_0)^-1: \n", np.matmul(phi10, np.linalg.inv(phi10)).round(
```

```
phi(t_10, t_0)^-1:
 [[ 1.23674844e+00 -1.38829570e-01  5.75183991e-01 -1.91322894e-02]
 [ 2.60026380e+02 -1.51639213e+02  1.52539455e+02  2.60670088e+02]
 [-2.59154448e+02  1.52127911e+02 -1.51284032e+02 -2.60234514e+02]
 [-3.74643453e-01 -3.66712857e-01 -6.96433461e-02  8.81235607e-01]]
phi(t_10, t0)*phi(t_10, t_0)^-1:
 [[ 1.  0. -0. -0.]
 [-0.  1.  0.  0.]
 [ 0.  0.  1. -0.]
 [ 0.  0. -0.  1.]]
```

d. Calculate the perturbation vector, δX(ti), by the following methods:

(1) δX(ti) = X(ti) −X∗(ti)

(2) δX(ti) = Φ(ti,t0)δX(t0)

and compare the results of (1) and (2). Provide the numeric results of (1) and (2) at t1 and t10 in the write-up, along with δX(ti) −Φ(ti,t0)δX(t0). How closely do they compare?

In [273…
```python
Xstar = sol_two_body_stm.y.T[:, 0:4]
phi1 = sol_two_body_stm.y.T[1, 4:].reshape((4, 4))
dX0 = np.array([1E-6, -1E-6, 1E-6, 1E-6])
dX1_0 = X[1] - Xstar[1]
print("dX(t_1) = X(t_1)-X*(t_1): ", dX1_0)
dX1_1 = np.dot(phi1, dX0)
print("phi(t_1, t_0)*dX(t_0): ", dX1_1)
print("dX(t_1) - phi(t_1, t_0)*dX(t_0): ", dX1_0 - dX1_1)
```

```
dX(t_1) = X(t_1)-X*(t_1):  [-4.04310379e-05  5.03755902e-05 -5.50095267e-05 -3.02848903e-0
5]
phi(t_1, t_0)*dX(t_0):  [-4.04326243e-05  5.03744831e-05 -5.50088113e-05 -3.02868818e-05]
dX(t1) - phi(t_1, t_0)*dX(t_0):  [ 1.58643818e-09  1.10708336e-09 -7.15365896e-10  1.99149
098e-09]
```

In [274…
```python
phi10 = sol_two_body_stm.y.T[10, 4:].reshape((4, 4))
dX0 = np.array([1E-6, -1E-6, 1E-6, 1E-6])
dX10_0 = X[10]- Xstar[10]
print("X(t_10)-X*(t_10): ", dX10_0)
dX10_1 = np.dot(phi10, dX0)
print("phi(t_10, t_0)*dX(t_0): ", dX10_1)
print("dX(t_10) - phi(t_10, t_0)*dX(t_0): ", dX10_0 - dX10_1)
```

```
X(t_10)-X*(t_10):  [-0.00030449 -0.00052168  0.00051995 -0.00030443]
phi(t_10, t_0)*dX(t_0):  [-0.00030433 -0.00052177  0.00052004 -0.00030427]
dX(t_10) - phi(t_10, t_0)*dX(t_0):  [-1.58362570e-07  8.87747287e-08 -9.10120600e-08 -1.58
109594e-07]
```

2) Given the observation state relation $y = Hx + \hat{\epsilon}$

a. Using the batch processing algorithm, what is $\hat{x}$? In the write-up, outline the method employed in the code.

I used the batch processing algorithm descibed on lecture slide 11 from module 7. It starts by initializing the values according to the initial conditions provided as well as Λ and N. Since this is a scalar example $\phi = 1$ and $A = 0$, resulting in $\dot{\phi} = 0$. This makes ϕ always 1. Thus the Propogation to Next Observation step can be skipped. Next I go through the Accumulate and Map to Epoch step to find the new values of Λ and N. Those are then used to solve the Normal Equations to find $\hat{x}$.

In [283…
```python
y_bar = np.array([1, 2, 1])
W = np.array([[2, 0, 0], [0, 1, 0], [0, 0, 1]])
H_dbar = np.array([1, 1, 1]).T

x_bar = 2#1x1
W_bar = 2
lam = W_bar
N = W_bar*x_bar
x_hat = 0

#batch processing algorithm
phi_t = np.array([1])
lam = W_bar
N = W_bar*x_bar
yi = y_bar

#Accumulate and map to epoch
Hi_tilda = H_dbar
Hi = Hi_tilda*phi_t
lam = lam + np.dot(np.dot(Hi.T, W), Hi)
N = N + np.dot(np.dot(Hi.T, W), yi)

#Solve for normal equation
```

```
x_hat = N/lam

print("x_hat: ", x_hat)
```

```
x_hat:  1.5
```

b. What is the best estimate of the observation error, $\hat{\epsilon}$?

In [278…

```
e_hat = yi-H_dbar*x_hat_new
print("e_hat: ", e_hat)
```

```
e_hat:  [-0.5  0.5 -0.5]
```