EES_Optimal_Ascent

April 30, 2019

In [10]:
```
# -*- coding: utf-8 -*-
# Adapted from 2017 Interstellar Technologies Inc. All Rights Reserved.

from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
from OpenGoddard.optimize import Problem, Guess, Condition, Dynamics
from math import *

class Orbiter:
    def __init__(self):

        #note, if d_ is used, it refers to dimensional quantity in SI units.

        AU = 1.5*10**11;
        mu_sun = (6.67*10**-11)*(2*10**30) #SI units
        year = 3.154*10**7 #seconds

        #set up non-d params

        self.lref = 1*AU; #1 AU
        self.vref = sqrt(mu_sun/self.lref) #v of earths orbit
        self.tref = self.lref/self.vref
        self.aref = (self.vref)**2/self.lref
        self.wref = 1/self.tref;

        self.d_mdry = 2800; #kg

        #assign operating limits
        self.d_a_max = 0.5/self.d_mdry; #assuming 0.5N for vehicle mass of 5000 kg

        self.d_tf_max = 10*year;


        #assign initial conditions
        self.d_r0 = 1*AU;
        self.d_vr0 = 0;
```

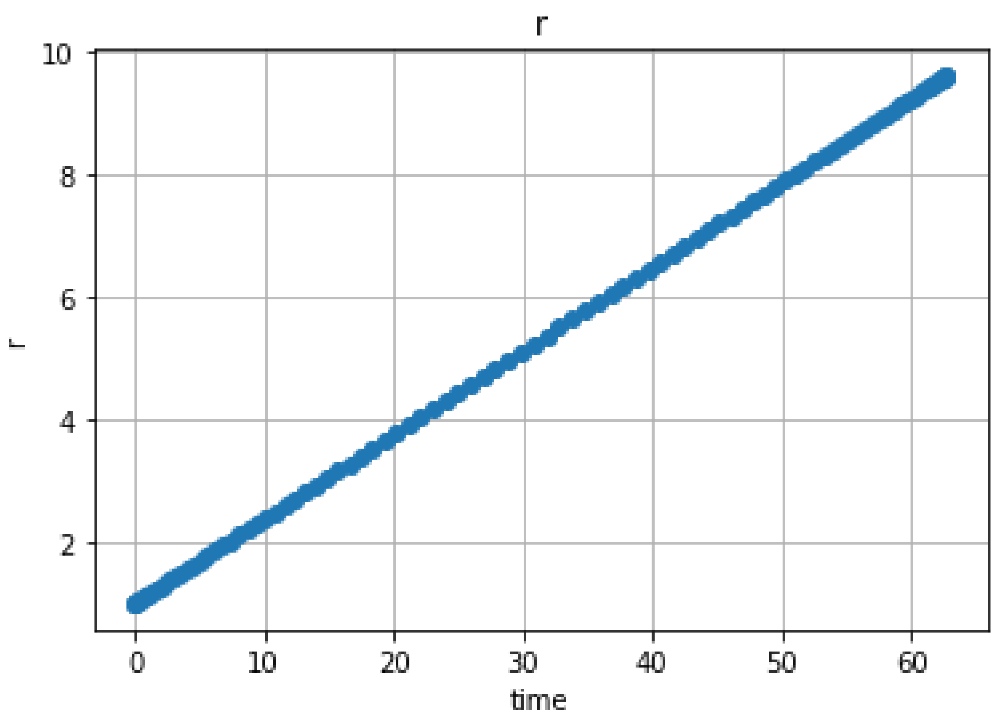
```python
        self.d_vt0 = sqrt(mu_sun/(1*AU));
        self.d_w0 =self.d_vt0/self.d_r0; #earths orbital speed



        self.d_rf = 9.6*AU;
        self.d_vrf = 0;
        self.d_final_apogee = 9.6*AU;
        self.d_final_semimajor = (self.d_rf+self.d_final_apogee)/2;
        self.d_vtf = sqrt(mu_sun*(2/self.d_rf - 1/self.d_final_semimajor)); #vis viva e
        self.d_wf = self.d_vtf/self.d_rf;


        #non-dimensionalise
        self.u_max  = self.d_a_max/self.aref;
        self.r0     = self.d_r0 /self.lref;
        self.vr0    = self.d_vr0/self.vref;
        self.vt0    = self.d_vt0/self.vref;
        self.rf     = self.d_rf /self.lref;
        self.vrf    = self.d_vrf/self.vref;
        self.vtf    = self.d_vtf/self.vref;
        self.tf_max = self.d_tf_max/self.tref;


def dynamics(prob, obj, section):
    r   = prob.states(0, section)
    vr  = prob.states(1, section)
    vt  = prob.states(2, section)
    ur1 = prob.controls(0, section)
    ur2 = prob.controls(1, section)
    ut1 = prob.controls(2, section)
    ut2 = prob.controls(3, section)

    dx = Dynamics(prob, section)
    dx[0] = vr
    dx[1] = vt**2 / r - 1 / r**2 + (ur1 - ur2)
    dx[2] = - vr * vt / r + (ut1 - ut2)
    return dx()


def equality(prob, obj):
    r   = prob.states_all_section(0)
    vr  = prob.states_all_section(1)
    vt  = prob.states_all_section(2)
    ur1 = prob.controls_all_section(0)
    ur2 = prob.controls_all_section(1)
    ut1 = prob.controls_all_section(2)
    ut2 = prob.controls_all_section(3)
```

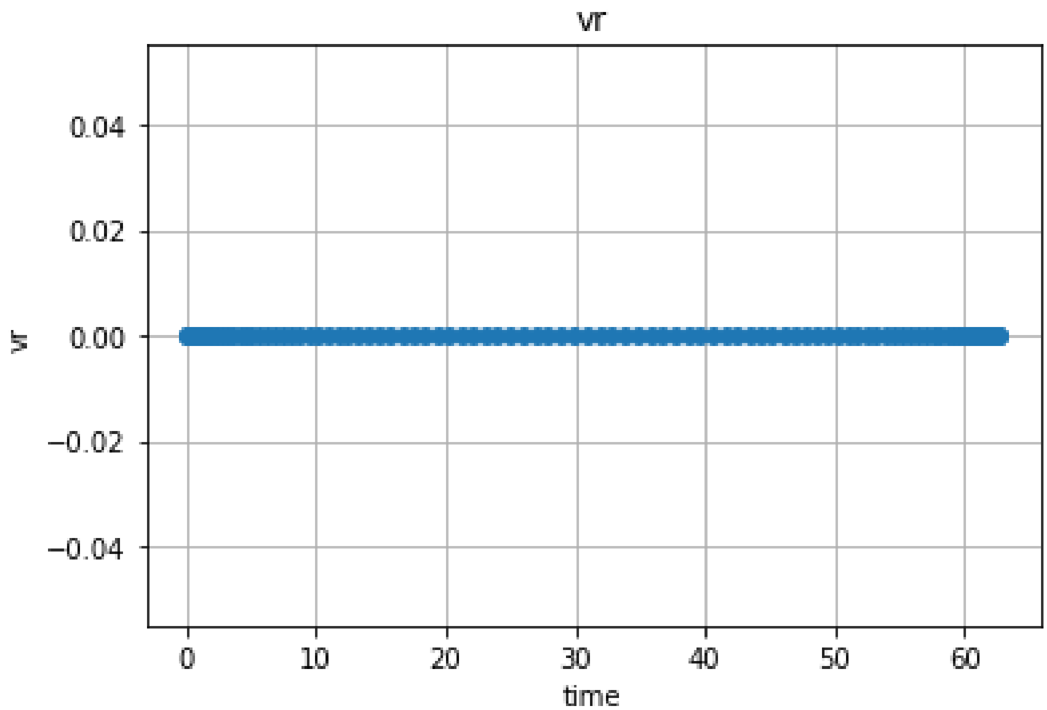
```
    tf  = prob.time_final(-1)

    result = Condition()

    # event condition
    result.equal(r[0], obj.r0)
    result.equal(vr[0], obj.vr0)
    result.equal(vt[0], obj.vt0)
    result.equal(r[-1], obj.rf)
    result.equal(vr[-1], obj.vrf)


    return result()



def inequality(prob, obj):
    r   = prob.states_all_section(0)
    vr  = prob.states_all_section(1)
    vt  = prob.states_all_section(2)
    ur1 = prob.controls_all_section(0)
    ur2 = prob.controls_all_section(1)
    ut1 = prob.controls_all_section(2)
    ut2 = prob.controls_all_section(3)
    tf  = prob.time_final(-1)

    result = Condition()

    # lower bounds
    result.lower_bound(r, 0.5*min(obj.r0,obj.rf))
    result.lower_bound(ur1, 0.0)
    result.lower_bound(ut1, 0.0)
    result.lower_bound(ur2, 0.0)
    result.lower_bound(ut2, 0.0)
    result.lower_bound(tf, 0.0)
    result.lower_bound(vt[-1],obj.vtf) #requries minimum final tangential speed, no upp

    # upper bounds
    result.upper_bound(r, 1*max(obj.r0,obj.rf))
    result.upper_bound(np.sqrt(np.square(ur1-ur2) + np.square(ut1-ut2)), obj.u_max)
    #result.upper_bound(ut1, obj.u_max)
    #result.upper_bound(ur2, obj.u_max)
    #result.upper_bound(ut2, obj.u_max)
    result.upper_bound(tf, obj.tf_max)




    return result()
```

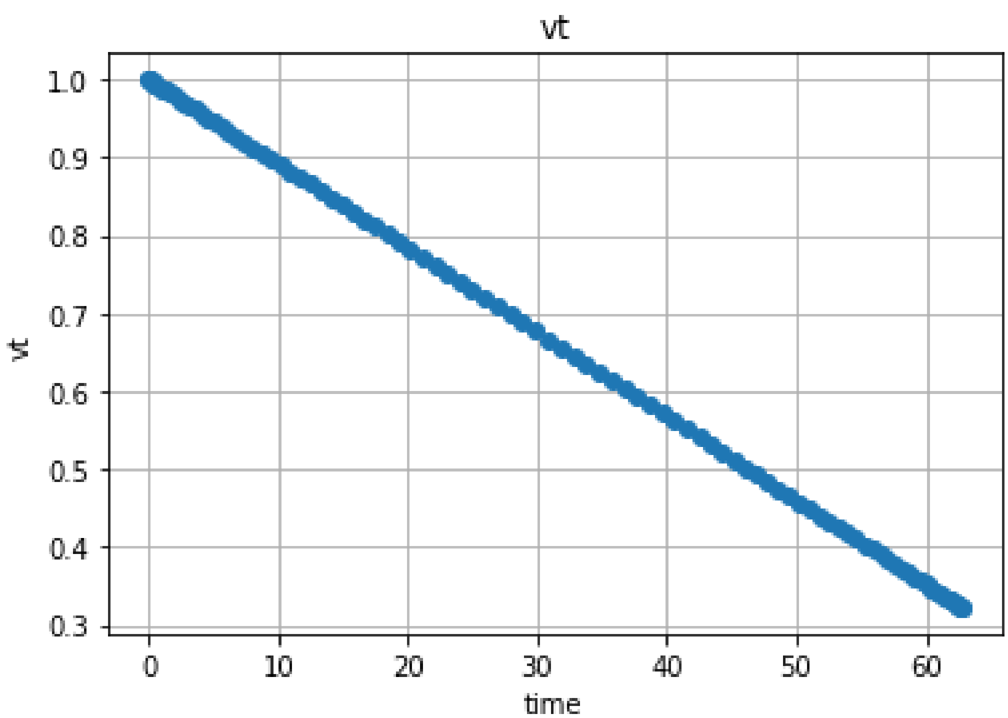
```
def cost(prob, obj):
    return 0.0


def running_cost(prob, obj):
    ur1 = prob.controls_all_section(0)
    ur2 = prob.controls_all_section(1)
    ut1 = prob.controls_all_section(2)
    ut2 = prob.controls_all_section(3)

    netaccel = np.sqrt(np.square(ur1 + ur2) + np.square(ut1 + ut2));

    if 0:
        print(type(ur1))
        print(ur1.size)
        print((ur1+ur2+ut1+ut2).size)
        print(netaccel.size)



    return netaccel
```

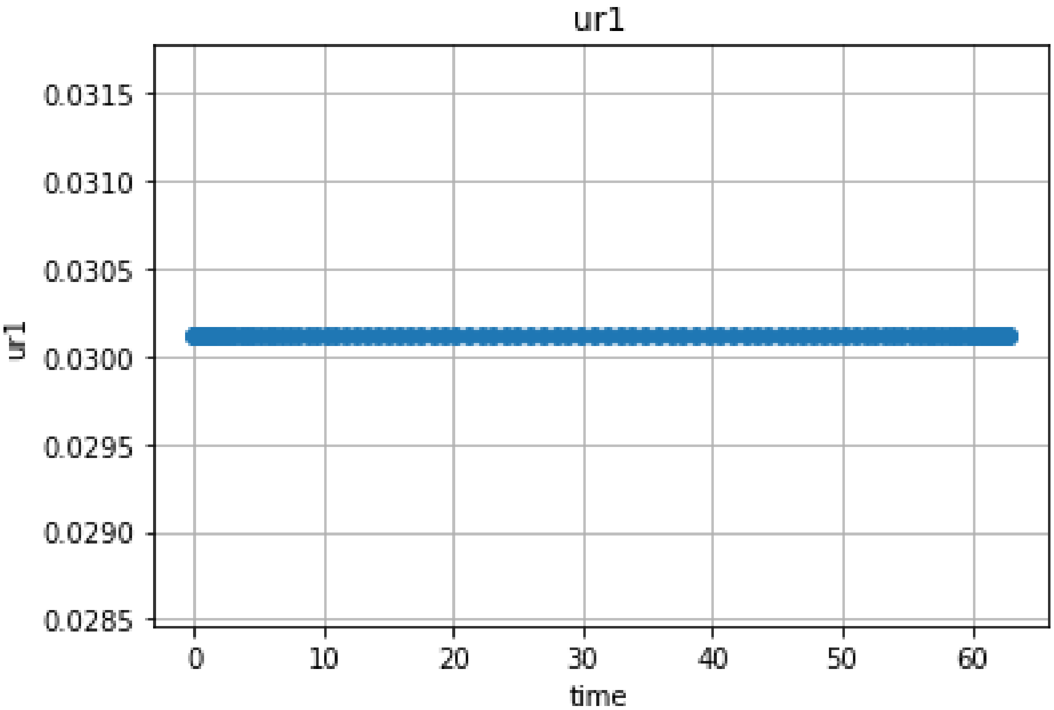
In [11]:
```
# ========================
plt.close("all")
plt.ion()
# Program Starting Point

obj = Orbiter()


time_init = [0.0, obj.tf_max]
n = [100]
num_states = [3]
num_controls = [4]
max_iteration = 10

flag_savefig = False

savefig_dir = "10_Low_Thrust_Orbit_Transfer/"

# ------------------------
# set OpenGoddard class for algorithm determination
prob = Problem(time_init, n, num_states, num_controls, max_iteration)


# ========================
```

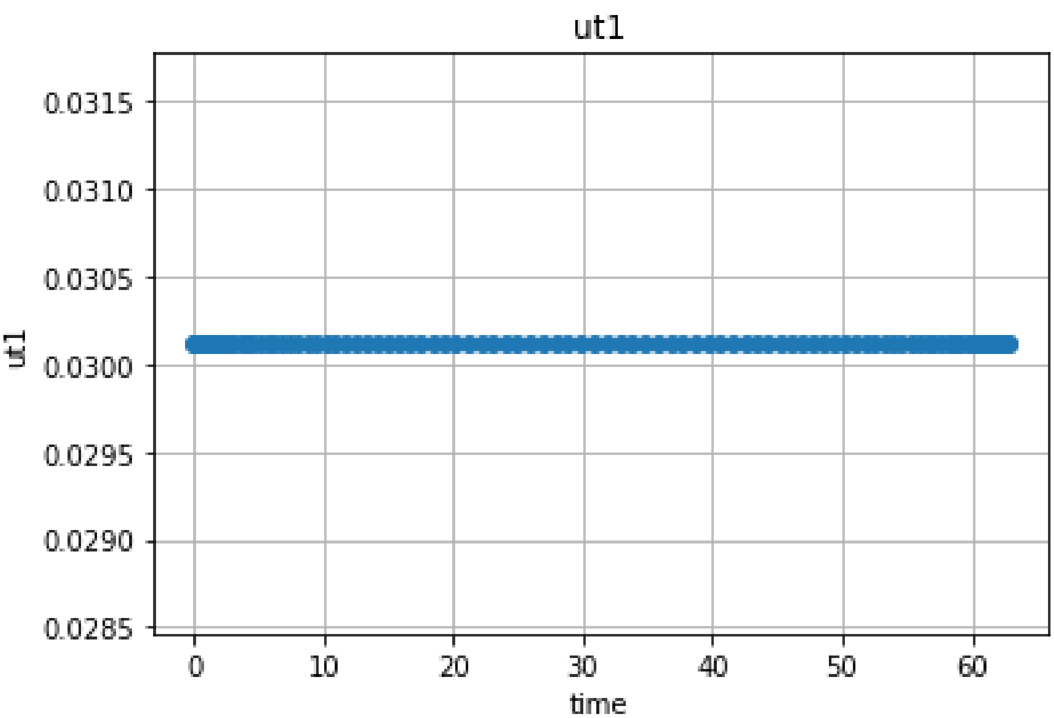
```
# Initial parameter guess
r_init = Guess.linear(prob.time_all_section, obj.r0, obj.rf)
Guess.plot(prob.time_all_section, r_init, "r", "time", "r")
if(flag_savefig):plt.savefig(savefig_dir + "guess_r" + savefig_add + ".png")

vr_init = Guess.linear(prob.time_all_section, obj.vr0, obj.vrf)
Guess.plot(prob.time_all_section, vr_init, "vr", "time", "vr")
if(flag_savefig):plt.savefig(savefig_dir + "guess_vr" + savefig_add + ".png")

vt_init = Guess.linear(prob.time_all_section, obj.vt0, obj.vtf)
Guess.plot(prob.time_all_section, vt_init, "vt", "time", "vt")
if(flag_savefig):plt.savefig(savefig_dir + "guess_vt" + savefig_add + ".png")

ur1_init = Guess.linear(prob.time_all_section, obj.u_max, obj.u_max)
Guess.plot(prob.time_all_section, ur1_init, "ur1", "time", "ur1")
if(flag_savefig):plt.savefig(savefig_dir + "guess_ur1" + savefig_add + ".png")

ut1_init = Guess.linear(prob.time_all_section, obj.u_max, obj.u_max)
Guess.plot(prob.time_all_section, ut1_init, "ut1", "time", "ut1")
if(flag_savefig):plt.savefig(savefig_dir + "guess_ut1" + savefig_add + ".png")
```




vr
0.04
0.02
0.00
−0.02
−0.04
vr
0
10
20
30
40
50
60
time


vt
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
vt
0
10
20
30
40
50
60
time


ur1
0.0315
0.0310
0.0305
0.0300
0.0295
0.0290
0.0285
ur1
0
10
20
30
40
50
60
time


ut1
0.0315
0.0310
0.0305
0.0300
0.0295
0.0290
0.0285
ut1
0
10
20
30
40
50
60
time

In [12]:
```
prob.set_states_all_section(0, r_init)
prob.set_states_all_section(1, vr_init)
prob.set_states_all_section(2, vt_init)
prob.set_controls_all_section(0, ur1_init)
prob.set_controls_all_section(2, ut1_init)

# ========================
# Main Process
# Assign problem to SQP solver
prob.dynamics = [dynamics]
prob.knot_states_smooth = []
prob.cost = cost
prob.running_cost = running_cost
prob.equality = equality
prob.inequality = inequality


def display_func():
    tf = prob.time_final(-1)
    print("tf: {0:.5f}".format(tf))


prob.solve(obj, display_func, ftol=1e-6)
```

```
---- iteration : 1 ----
Iteration limit exceeded    (Exit mode 9)
            Current function value: 0.0303987176329822
            Iterations: 26
            Function evaluations: 18295
            Gradient evaluations: 26
Iteration limit exceeded
tf: 62.70508

---- iteration : 2 ----
Iteration limit exceeded    (Exit mode 9)
            Current function value: 0.02589161264142747
            Iterations: 26
            Function evaluations: 18289
            Gradient evaluations: 26
Iteration limit exceeded
tf: 62.70508

---- iteration : 3 ----
Iteration limit exceeded    (Exit mode 9)
            Current function value: 0.024664341148715924
            Iterations: 26
            Function evaluations: 18292
            Gradient evaluations: 26
```

```
Iteration limit exceeded
tf: 62.70508

---- iteration : 4 ----
Iteration limit exceeded    (Exit mode 9)
            Current function value: 0.0240943740579277
            Iterations: 26
            Function evaluations: 18289
            Gradient evaluations: 26
Iteration limit exceeded
tf: 62.70508

---- iteration : 5 ----
Iteration limit exceeded    (Exit mode 9)
            Current function value: 0.023689852797767946
            Iterations: 26
            Function evaluations: 18292
            Gradient evaluations: 26
Iteration limit exceeded
tf: 62.70508

---- iteration : 6 ----
Iteration limit exceeded    (Exit mode 9)
            Current function value: 0.023458069945614307
            Iterations: 26
            Function evaluations: 18297
            Gradient evaluations: 26
Iteration limit exceeded
tf: 62.70508

---- iteration : 7 ----
Iteration limit exceeded    (Exit mode 9)
            Current function value: 0.023213624797851346
            Iterations: 26
            Function evaluations: 18291
            Gradient evaluations: 26
Iteration limit exceeded
tf: 62.70508

---- iteration : 8 ----
Iteration limit exceeded    (Exit mode 9)
            Current function value: 0.023097141544382783
            Iterations: 26
            Function evaluations: 18301
            Gradient evaluations: 26
Iteration limit exceeded
tf: 62.70508
```

```
---- iteration : 9 ----
Iteration limit exceeded    (Exit mode 9)
            Current function value: 0.023017800003718867
            Iterations: 26
            Function evaluations: 18305
            Gradient evaluations: 26
Iteration limit exceeded
tf: 62.70508

---- iteration : 10 ----
Iteration limit exceeded    (Exit mode 9)
            Current function value: 0.022925736042640258
            Iterations: 26
            Function evaluations: 18304
            Gradient evaluations: 26
Iteration limit exceeded
tf: 62.70508
```

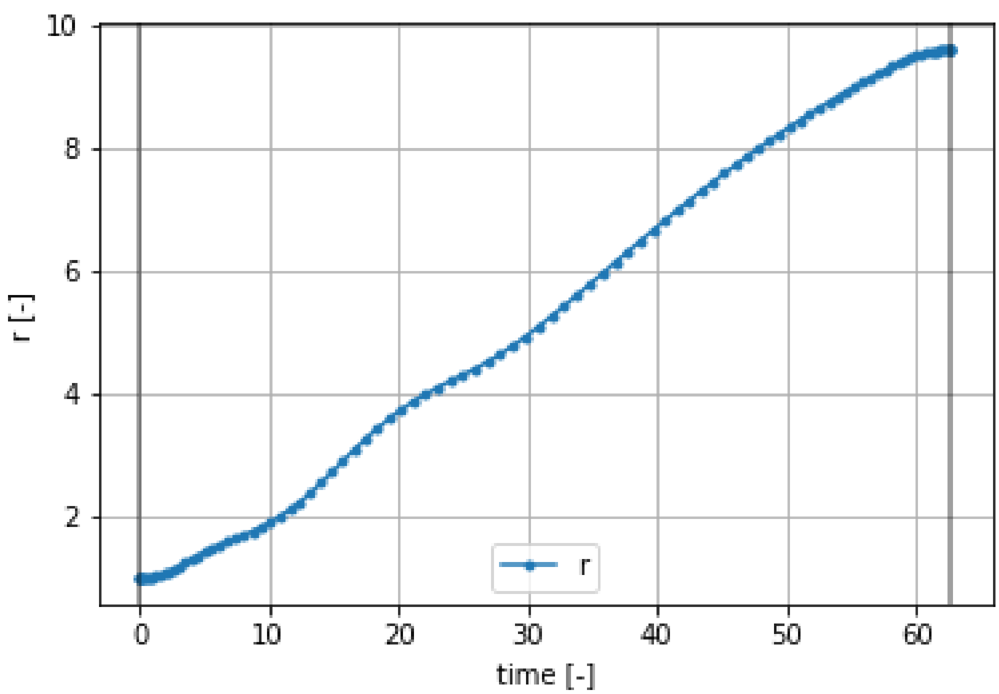
In [13]:
```
# ========================
# Post Process
# ------------------------
# Convert parameter vector to variable
r = prob.states_all_section(0)
vr = prob.states_all_section(1)
vt = prob.states_all_section(2)
ur1 = prob.controls_all_section(0)
ur2 = prob.controls_all_section(1)
ut1 = prob.controls_all_section(2)
ut2 = prob.controls_all_section(3)
time = prob.time_update()

year = 3.154*10**7 #seconds

accel_mag=np.sqrt(np.square(ur1-ur2)+np.square(ut1-ut2))



# ------------------------
# Visualizetion
plt.figure()
plt.plot(time, r, marker=".", label="r")
for line in prob.time_knots():
    plt.axvline(line, color="k", alpha=0.5)
plt.grid()
plt.xlabel("time [-]")
plt.ylabel("r [-]")
```

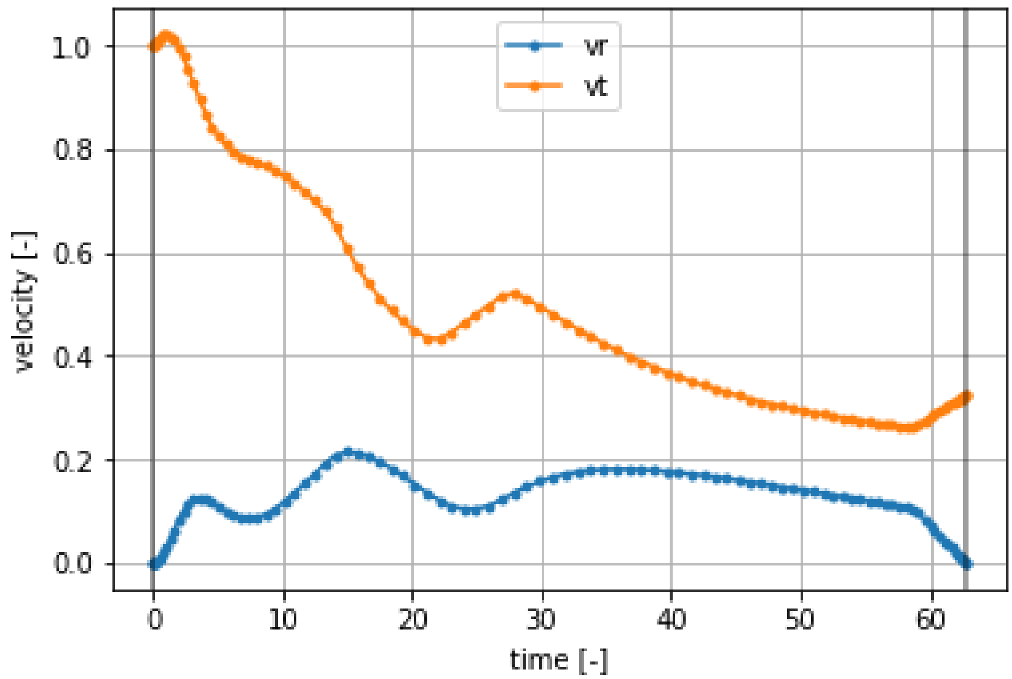
```
plt.legend(loc="best")
if(flag_savefig): plt.savefig(savefig_dir + "r" + ".png")

plt.figure()
plt.plot(time, vr, marker=".", label="vr")
plt.plot(time, vt, marker=".", label="vt")
for line in prob.time_knots():
    plt.axvline(line, color="k", alpha=0.5)
plt.grid()
plt.xlabel("time [-]")
plt.ylabel("velocity [-]")
plt.legend(loc="best")
if(flag_savefig): plt.savefig(savefig_dir + "velocity" + ".png")

plt.figure()
plt.plot(time*obj.tref/year, (ur1 - ur2)*obj.aref, marker=".", label="ar")
plt.plot(time*obj.tref/year, (ut1 - ut2)*obj.aref, marker=".", label="at")
plt.grid()
plt.xlabel("time [years]")
plt.ylabel("accel [m/s2]")
# plt.ylim([-0.02, 0.6])
plt.legend(loc="best")

plt.figure()
plt.plot(time, accel_mag, marker=".", label="a_mag")
plt.plot([0,time[-1]],[obj.u_max,obj.u_max],'k:')
plt.grid()
plt.xlabel("time [-]")
plt.ylabel("accel_mag [-]")
plt.legend(loc="best")

plt.figure()
plt.plot(time, ur1, marker=".", label="ar_out")
plt.plot(time, ur2, marker=".", label="ar_in")
plt.grid()
plt.xlabel("time [-]")
plt.ylabel("accel_radial [-]")
# plt.ylim([-0.02, 0.6])
plt.legend(loc="best")

plt.figure()
plt.plot(time, ut1, marker=".", label="at_pro")
plt.plot(time, ut2, marker=".", label="at_retro")
plt.grid()
plt.xlabel("time [-]")
plt.ylabel("accel_tangential [-]")
# plt.ylim([-0.02, 0.6])
plt.legend(loc="best")
```

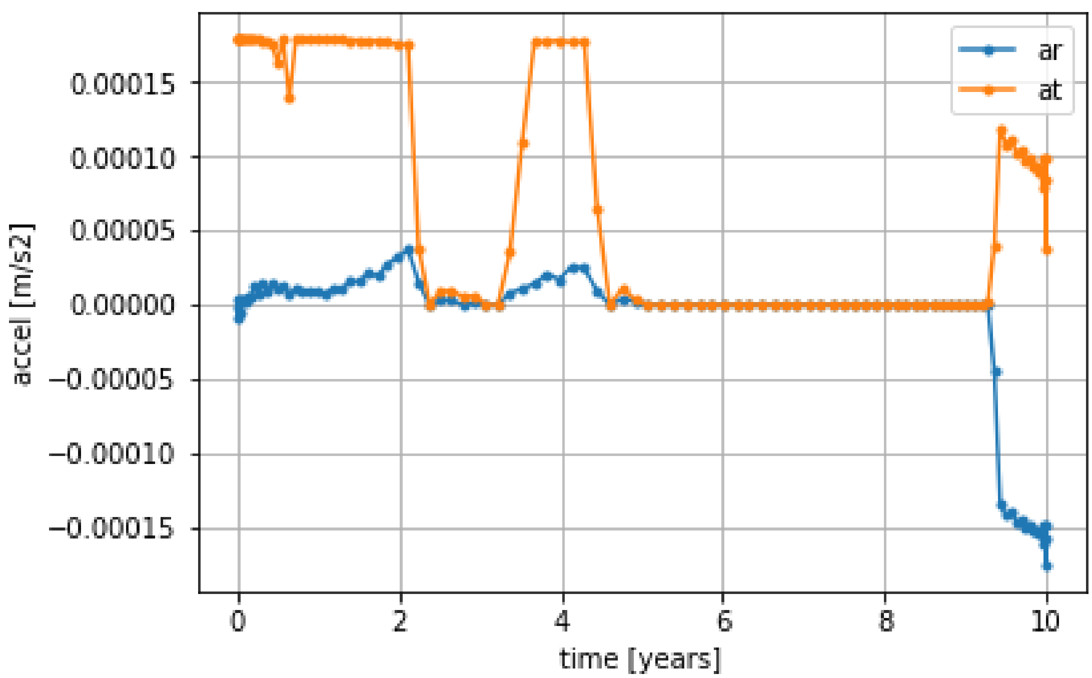
```python
if(flag_savefig): plt.savefig(savefig_dir + "thrust" + ".png")


from scipy import integrate
from scipy import interpolate

theta = integrate.cumtrapz(vt / r, time, initial=0)
theta_f = interpolate.interp1d(time, theta)
r_f = interpolate.interp1d(time, r)
time_fine = np.linspace(time[0], time[-1], 1000)
r_fine = r_f(time_fine)
theta_fine = theta_f(time_fine)

fig = plt.figure()
# plt.plot(r*np.cos(theta), r*np.sin(theta))
plt.plot(r_fine*np.cos(theta_fine), r_fine*np.sin(theta_fine))
ax = fig.add_subplot(111)
circle0 = plt.Circle((0.0, 0.0), obj.r0, ls="--", fill=False, fc='none')
circlef = plt.Circle((0.0, 0.0), obj.rf, ls="--", fill=False, fc='none')
ax.add_patch(circle0)
ax.add_patch(circlef)
plt.grid()
plt.axis('equal')
plt.ylim((-1.5*max(obj.r0,obj.rf),1.5*max(obj.r0,obj.rf)))
plt.xlim((-1.5*max(obj.r0,obj.rf),1.5*max(obj.r0,obj.rf)))
plt.xlabel('x (AU)')
plt.ylabel('y (AU)')
if(flag_savefig): plt.savefig(savefig_dir + "trajectry" + ".png")


plt.show()


#plot the burn direction vector

helioxaccel = (ur1 - ur2)*np.cos(theta) - (ut1-ut2)*np.sin(theta)
helioyaccel = (ur1 - ur2)*np.sin(theta) + (ut1-ut2)*np.cos(theta)

fig = plt.figure()
plt.plot(time,helioxaccel,label='helio x')
plt.plot(time,helioyaccel, label='helio y')

plt.show()

fig = plt.figure()
# plt.plot(r*np.cos(theta), r*np.sin(theta))
plt.quiver(r*np.cos(theta), r*np.sin(theta), helioxaccel,helioyaccel)
```

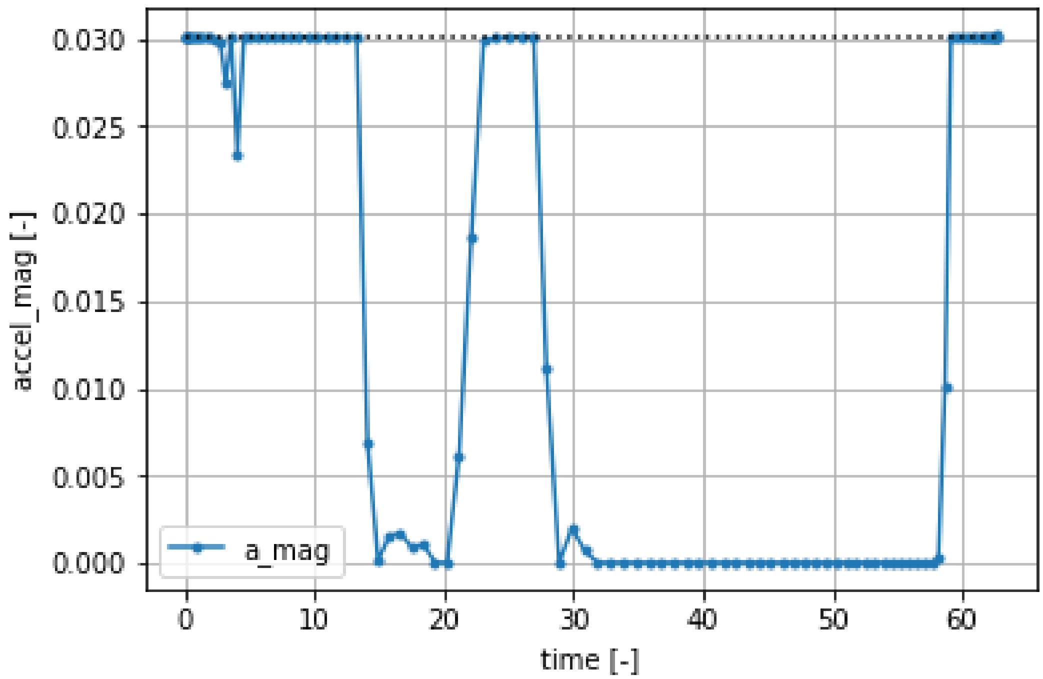
```
ax = fig.add_subplot(111)
circle0 = plt.Circle((0.0, 0.0), obj.r0, ls="--", fill=False, fc='none')
circlef = plt.Circle((0.0, 0.0), obj.rf, ls="--", fill=False, fc='none')
ax.add_patch(circle0)
ax.add_patch(circlef)
plt.grid()
plt.axis('equal')
plt.ylim((-1.5*max(obj.r0,obj.rf),1.5*max(obj.r0,obj.rf)))
plt.xlim((-1.5*max(obj.r0,obj.rf),1.5*max(obj.r0,obj.rf)))
plt.xlabel('x (AU)')
plt.ylabel('y (AU)')
if(flag_savefig): plt.savefig(savefig_dir + "trajectry" + ".png")


plt.show()
```

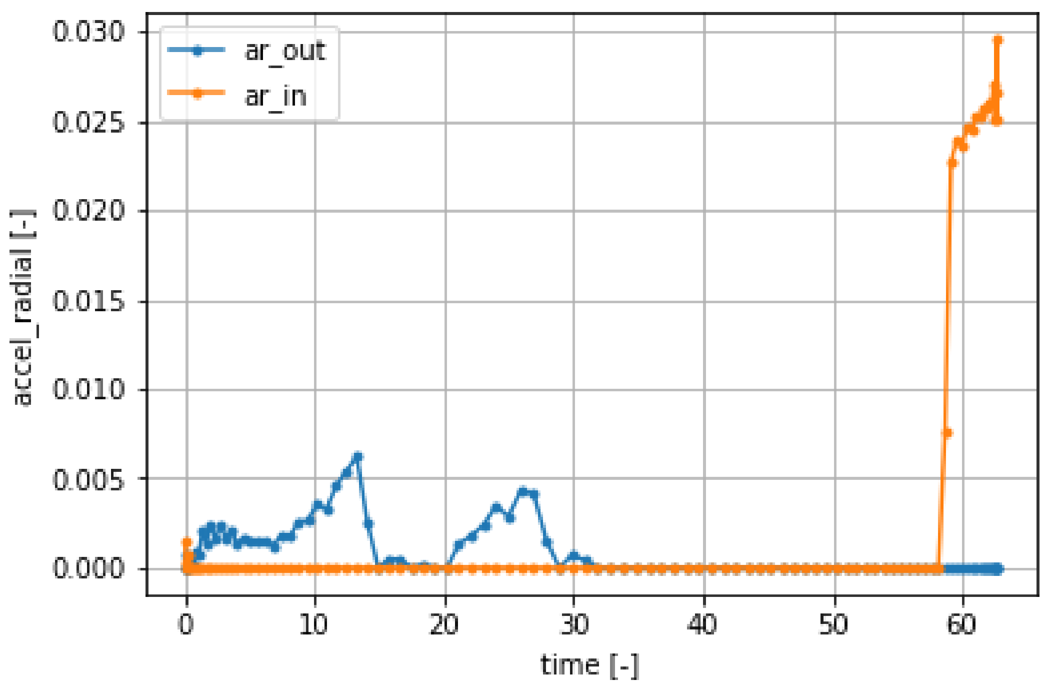
```
/Users/Devansh/anaconda3/lib/python3.6/site-packages/matplotlib/cbook/deprecation.py:107: Matplo
  warnings.warn(message, mplDeprecation, stacklevel=1)
```

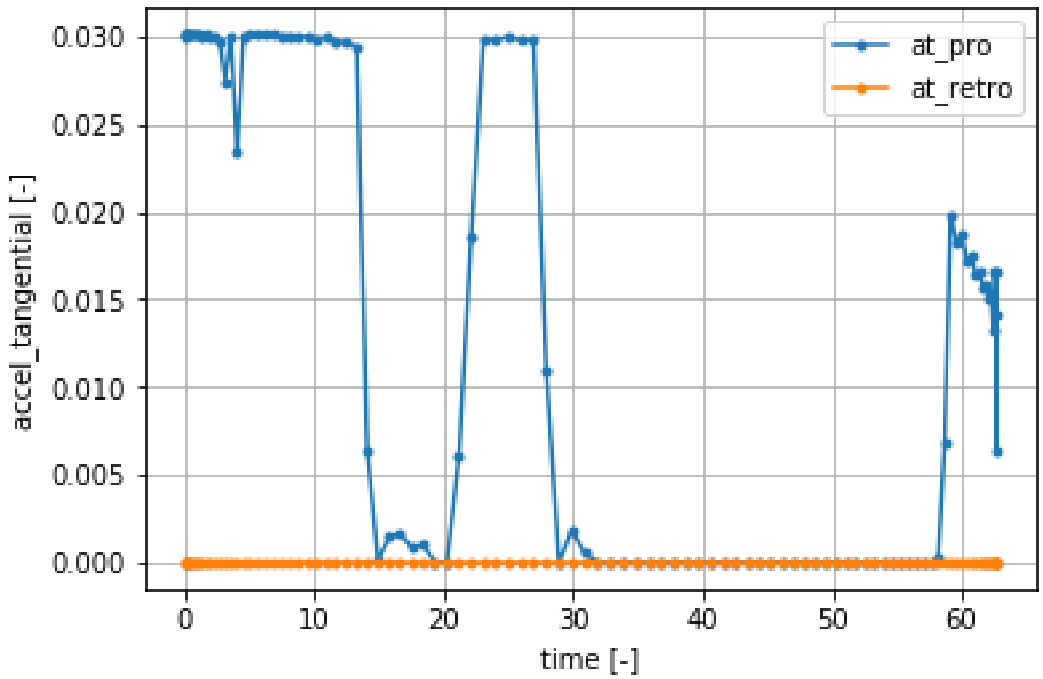



vr
vt
velocity [-]
time [-]
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
60

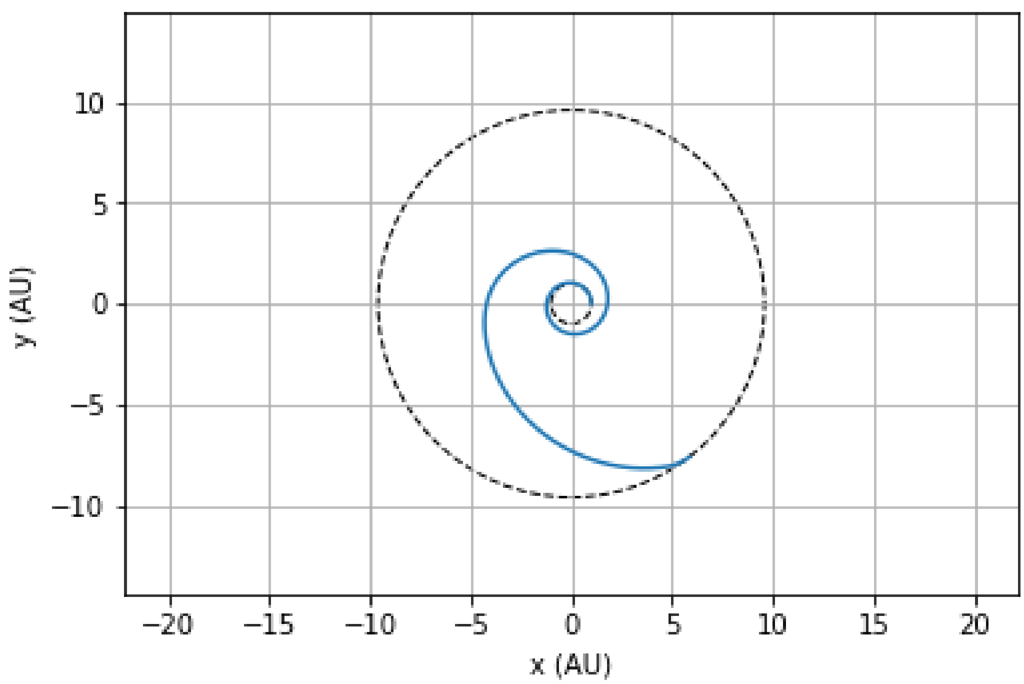

ar
at
accel [m/s2]
time [years]
0.00015
0.00010
0.00005
0.00000
−0.00005
−0.00010
−0.00015
0
2
4
6
8
10


0.030
0.025
0.020
0.015
0.010
0.005
0.000
accel_mag [-]
a_mag
0
10
20
30
40
50
60
time [-]

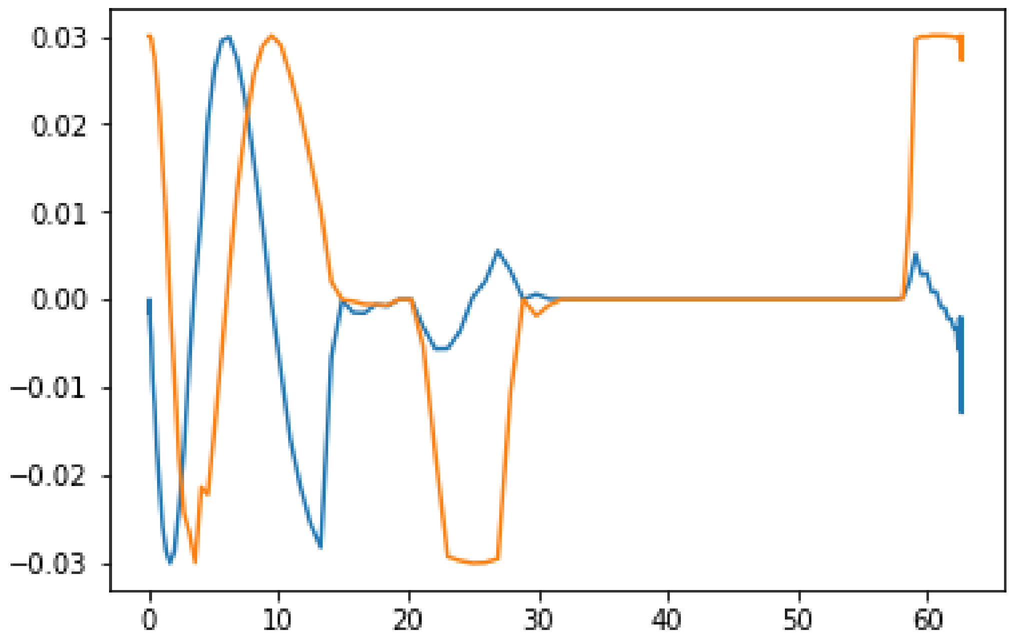

0.030
0.025
0.020
0.015
0.010
0.005
0.000
accel_radial [-]
ar_out
ar_in
0
10
20
30
40
50
60
time [-]

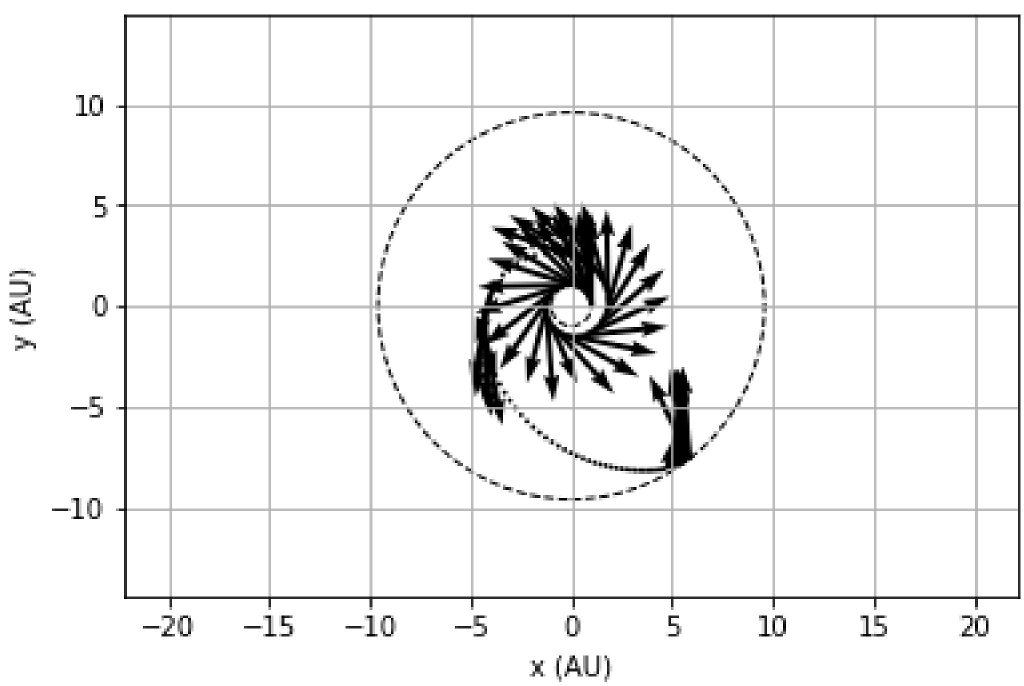


0.030
0.025
0.020
0.015
0.010
0.005
0.000
accel_tangential [-]
0
10
20
30
40
50
60
time [-]
at_pro
at_retro



10
5
0
−5
−10
y (AU)
−20
−15
−10
−5
0
5
10
15
20
x (AU)





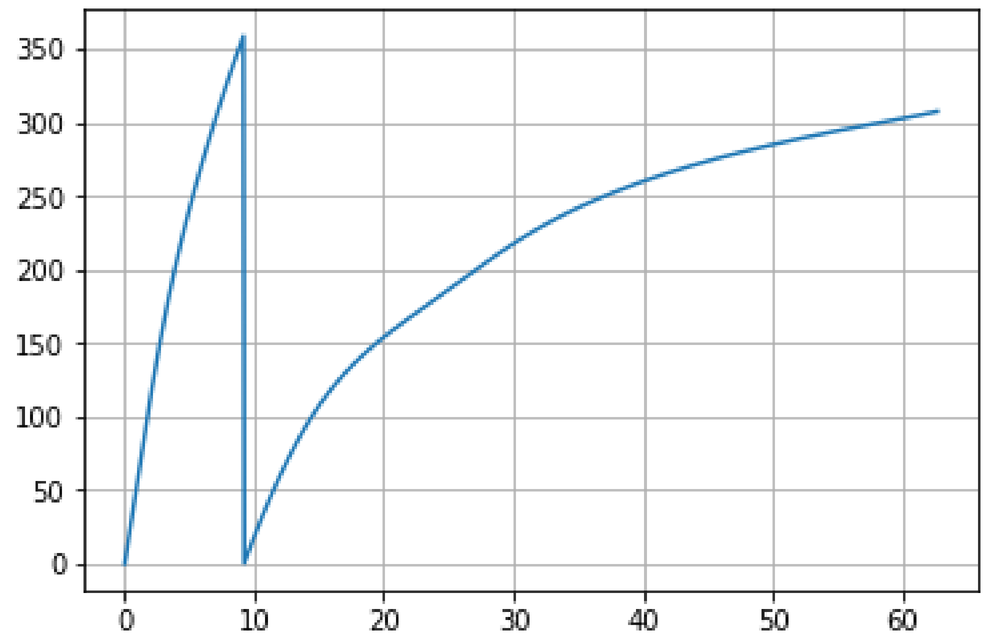

```
In [18]: len(ur2)
```

```
Out[18]: 100
```

In [19]:
```
len(theta)
```

Out[19]:
```
100
```

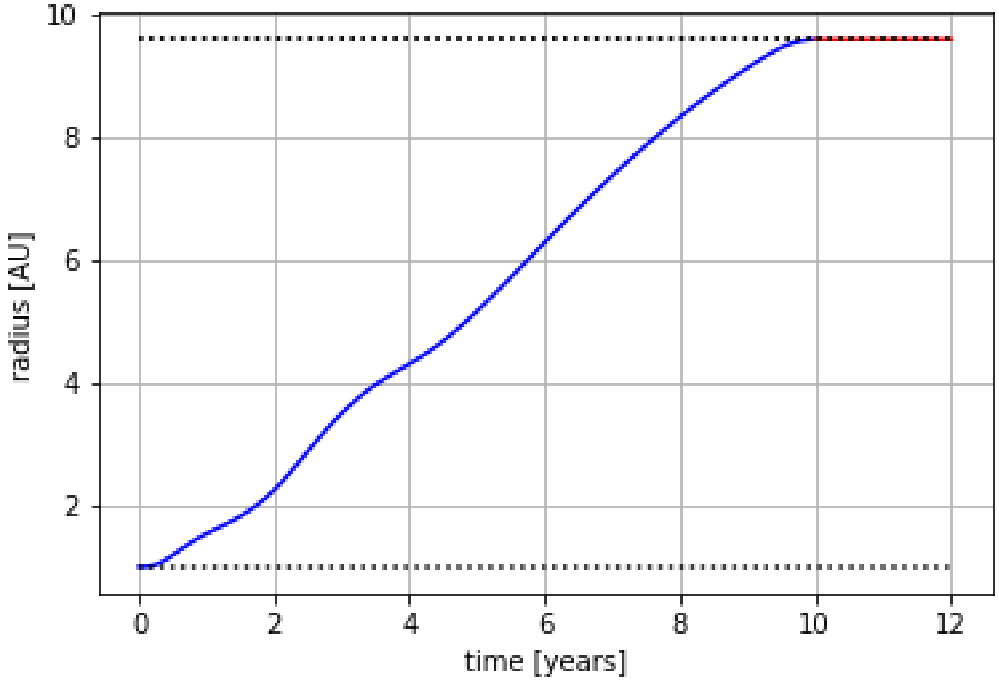
In [20]:
```
plt.figure()
plt.plot(time_fine,360*((theta_fine/(2*pi))%1))
plt.grid()
```



In [22]:
```
#determine how long the transfer will take
final_time = time[-1]
d_final_time = final_time*obj.tref
year = 3.154*10**7 #seconds

print("Transfer time: ",d_final_time/year, "years")

#determine delta v and mass fraction
delv=np.trapz(y=accel_mag,x=time)
d_delv = delv*obj.vref
print("Required Delta-v: ", d_delv, "m/s")
mpm0=1-exp(-d_delv/(2020*9.81))
print("Propellant mass fraction: ", mpm0)

#final phase angle:
print("Arriving phase: ", theta_fine[-1], "rad")
print("Arriving phase: ", 360*((theta_fine[-1]/(2*pi))%1))
```

```
Transfer time:  10.0000000024 years
Required Delta-v:  21441.030839 m/s
Propellant mass fraction:  0.6610813180747914
Arriving phase:  11.6525431488 rad
Arriving phase:  307.641543023
```

In [23]:
```
#integrate the dynamics from the end point

#define the passive dynamics
from scipy.integrate import odeint
def passiveDynamics(x,t):
    r = x[0]
    vr = x[1]
    vt = x[2]

    dx = [0,0,0]
    dx[0] = vr
    dx[1] = vt**2 / r - 1 / r**2
    dx[2] = - vr * vt / r

    return dx

#integrate using odeint
t_passive = np.linspace(final_time,final_time+2*year/obj.tref,10000)
x_passive = odeint(passiveDynamics,[r[-1],vr[-1],vt[-1]],t_passive)

#extract the radial position
r_passive = x_passive[:,0]


#plot the final solution
plt.figure()
plt.plot(time*obj.tref/year, r,'b')
plt.plot(t_passive*obj.tref/year, r_passive,'r')
plt.plot([0,final_time*obj.tref/year + 2],[obj.r0,obj.r0],'k:')
plt.plot([0,final_time*obj.tref/year + 2],[obj.rf,obj.rf],'k:')
plt.plot([0,final_time*obj.tref/year + 2],[obj.d_final_apogee/obj.lref,obj.d_final_apog
plt.grid()
plt.xlabel("time [years]")
plt.ylabel("radius [AU]")

plt.show()
```



10
8
6
4
2
radius [AU]
0
2
4
6
8
10
12
time [years]