SIMULATION OF THE TRAJECTORY OF THE CASSINI MISSION TO SATURN AND BEYOND

Max Murmann

DEN318 Third Year Project (Aero) April 2019
Supervisor: Dr R Vepa

SCHOOL OF ENGINEERING AND MATERIALS SCIENCE

ENGINEERING / MATERIALS

THIRD YEAR PROJECT

DEN318/ MAT500

APRIL 2019

DECLARATION

This report entitled:

Simulation of the Trajectory of the Cassini Mission to Saturn and Beyond

Was composed by me and is based on my own work. Where the work of the others has been used, it is fully acknowledged in the text and in captions to table illustrations. This report has not been submitted for any other qualification.

Name	Max Murmann
Signed	____________
Date	____________

Abstract

In 1997, the Cassini probe launched on a trajectory towards Saturn. It did not, however, fly directly there. It performed a series of gravity assists around planets to reach the necessary orbital energy to reach Saturn. This trajectory involved fly-bys of Venus twice, Earth and Jupiter. The aim of this report is to build a system which can reliably recreate the Cassini trajectory. This was done by creating a program which simulates the Solar System, and then simulates the forces on the Spacecraft at each point along its trajectory. The system then uses a trial and error method to find the optimal trajectory for reaching Saturn. The trajectory found was accurate to within a few days for most of the trajectory, with a bigger discrepancy at the end, with the final approach to Saturn being about a year ahead of Cassini.

List of Symbols

Symbol	**Meaning**
v	Velocity
v_{inf}	Velocity at the point of infinity of an orbit
r	Orbital radius
r_p	Orbital radius at periapsis
E	Energy constant of an orbit
F	Force
G	Gravitational Constant
M/m	Mass of an object, capital M indicates larger body
d	Distance
a	Acceleration
B	Asymptotic Miss Distance
μ	Standard Gravitational Parameter
α	Turning angle in a hyperbolic orbit
AU	Astronomical unit, equal to the distance from the earth to the sun ($1.49*10^8$km)
SOI	Sphere of Influence
DSM	Deep Space Maneuver
dV/Delta V	Change in velocity required for a maneuver

List of Tables

Contents

Intro

Gravity assist success

Gravity assists have been used by many interplanetary missions. The first official use of a gravity assist was used by the Luna 3 probe, launched in 1959 by the Soviet Union. This trajectory utilised a gravity assist around the Moon to gain enough energy to exit the Earth's sphere of influence (SOI). The first interplanetary gravity assist was used on the Mariner 10 mission, which used a gravity assist around Venus to lose enough energy to be able to do a fly-by of Mercury, a mission which would normally be very costly in terms of fuel usage. [1]

Since these initial missions took place, gravity assist trajectories have been getting more and more complicated in terms of their design. Almost every mission to the outer Solar System has used a gravity assist of some kind, most famously with the Voyager 1 and 2 probes. The trajectory that these two probes used took them on a "grand tour" of the Solar System, passing by Jupiter, Saturn, Neptune and Uranus, which caused a gain in energy large enough to put them on an escape trajectory from the Solar System. [1]

The main purpose of a gravity assist is to be able to gain orbital energy without needing to have as long of a burn time. The saving of this burn time can either allow for a smaller launch vehicle being required and therefore cause the mission to be more cost effective, or it can allow for more payload to be carried by the Spacecraft. [2]

Cassini

In 1997, the Cassini mission to Saturn launched from Cape Canaveral on top of a Titan IV launch stage. The Spacecraft performed an escape maneuver headed towards Venus, and proceeded to do gravity assists around Venus twice, Earth, and finally Jupiter before it had gained enough energy to be able to arrive at Saturn, after nearly 7 years of traveling in space. In between the two passes of Venus, a large burn was made to put the Spacecraft back on a trajectory to pass Venus again. [3]

This mission was considered a massive success due to the large amount of scientific research that was completed. Its success was in part due to the Spacecraft design that included a wide range of scientific instruments. In its 13 years spent in the Saturn system, it took over 450,000 photos and collected 635GB of scientific data. The Spacecraft was also carrying a lander, the Huygens probe, which is the first man made object to land on a body in the outer Solar System. [3]

This data lead to almost 4,000 scientific papers being published and 6 new moons of Saturn being discovered. Considering the initial mission was only to have a duration of 3 years, Cassini exceeded expectations, with two extra phases to the mission being added. Its final descent into the atmosphere of Saturn was well documented, with the final transmission being sent on the 15th September 2017. [3]

The trajectory that the Spacecraft took is what is going to be analysed and simulated in this report. The probe was launched on a trajectory towards Venus, and completed passes of Venus, Earth, and Jupiter on its way to Saturn. This trajectory allowed for a much larger payload to be carried, which attributed to the success of the mission. [3]

Aims

The aim of this project is to simulate the trajectory taken by the Cassini mission, from its launch to the arrival at Saturn. Importantly, it's going to focus on creating a computer program to assist finding these trajectories, as opposed to solely simulating the trajectory. The computer system that is designed must be able to find a trajectory from Earth to Saturn, via gravity assists. Whether this identically matches the trajectory taken by Cassini is less important than showing the actual program is functional and can work as intended.

Project Report Contents

This report will contain a short background and explanation of the mathematical calculations behind gravity assists. It will then go into detail on how the program created functions and provide an example calculation. The results of the found trajectory will then shown and compared to the Cassini probe trajectory. Limitations of the project and suggestions for further work are then provided.

Background

Orbits and Gravity Assists

In an orbit, the amount of energy you have remains constant throughout a fixed orbit. As the Spacecraft approaches its periapsis (the lowest point in the orbit), the velocity increase as it loses potential energy. The opposite occurs when the Spacecraft is heading towards the apoapsis (highest point in the orbit). This relationship between orbital velocity and altitude is shown by the Visa-Viva equation: [4]

$$\frac{1}{2}v^2 - \frac{\mu}{r} = E \equiv constant$$

Equation 1

Therefore, if the orbit needs to be changed, energy needs to be added or subtracted. From the equation, if the velocity is increased, the energy is increased by the velocity squared. This will cause the orbit to have a greater energy throughout the orbit. If this energy is added at the periapsis, the apoapsis will increase.

The simplest way to add energy in a Spacecraft is through burning the thrusters to increase the velocity. Using this method to plan an intercept with another body is known as a *Hohmann Transfer*. These involved two burns, one at the beginning of the transfer to extend the apoapsis of the orbit to match that of the target body, and a second burn to match the velocity of the second planet. The calculation of one of these trajectories is shown in Ap. 1

The main problem with these trajectories is for outer planets is they have very large fuel requirements. For example, to get to Saturn, around 7.5kms^{-1} of velocity gain is required when leaving earth to gain the required energy. As stated before, lowering the energy requirements allows for a cheaper mission, or a larger payload. To gain the necessary energy to reach these outer planets, a gravity assist maneuver is used. [4], [1]

A planetary gravity assist involves the Spacecraft flying very close to a planet and gaining energy as it flies by. The force between the Spacecraft and the planet is governed by the equation: [4]

$$F = \frac{G(M - m)}{d^2}$$

Equation 2

Due to the fact that the mass of the Spacecraft is so small compared to the mass of the planet, this term can effectively be ignored, and we are left with: [4]

$$F = \frac{GM}{d^2}$$

Equation 3

This can then be combined with the equation relating force and acceleration, to find the acceleration of the objects: [4]

$$F = ma$$

Equation 4

$$ma = \frac{GM}{d^2}$$

Equation 5

Two things can be drawn from these equations. Firstly, the closer the Spacecraft gets to the planet, the larger the force gets, with a ratio of d^2. Secondly, the acceleration of the planet will be very small compared to the acceleration of the Spacecraft, to the point where it will essentially be 0. [4] Therefore, as the Spacecraft gets dragged into the planet's gravity well, it will continue accelerating until the periapsis, and then begin to decelerate. In a two-body problem, this would mean that the Spacecraft gained as much energy as it would then lose. However, the Spacecraft will also gain some of the orbital energy of the planet in the solar reference frame, due to the planet moving away from the Spacecraft as the Spacecraft falls towards it, extending the acceleration time of the Spacecraft. [2] From this, it's apparent that the larger the planet, the larger effect of a gravity assist it has. Also, if the planet is closer to the sun, it will be traveling faster, therefore more energy can be gained.

The theoretical maximum energy gain from each planet is shown in the table below:

Planet	**Gravitational Parameter (μ)**	**Semi Major Axis (au)**	**ΔEmax (m^2s^{-2})**
Jupiter	1.27E+17	5.2028	5.83E+08
Saturn	3.79E+16	9.5388	2.62E+08
Venus	3.25E+14	0.7233	2.55E+08
Earth	3.99E+14	1	2.40E+08
Mercury	2.20E+13	0.38	1.74E+08
Uranus	5.79E+15	19.182	1.08E+08
Mars	4.28E+13	1.5237	9.57E+07
Neptune	6.84E+15	30.0577	9.20E+07

Table 1 - Maximum Energy Gain from the Planets [5]

Gravity assists will also change the direction of the Spacecraft. The magnitude of this turning angle is governed by how close the Spacecraft gets to the planet, and is given by the following equation: [5]

$$\alpha = 2tan^{-1}\left(\frac{\mu}{BV_{inf}^2}\right)$$

Equation 6

Where B is the asymptotic approach distance, given as: [5]

$$B = \left\{r_p^2(1 + (\frac{2\mu}{r_p V_{inf}^2}\right\}^{1/2}$$

Equation 7

Then, by performing vector calculations, we can find the total velocity increase: [6]

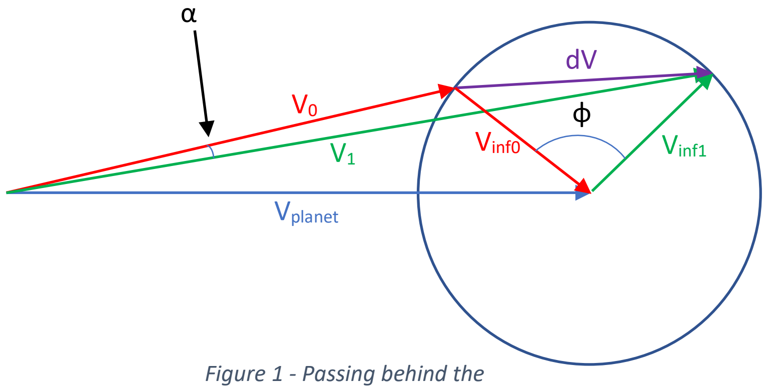

Figure 1 - Passing behind the planet causing a dV increase, with V0 being the initial velocity, and V1 being the post assist velocity in the in the heliocentric frame [6]

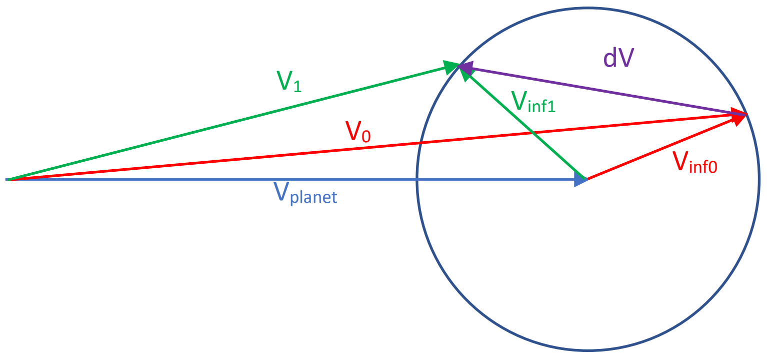

Figure 2 - The Spacecraft passing in front of the planet, causing a dV decrease [6]

As is shown by the vectors, passing either in front or behind the planet will either increase or decrease the orbital velocity, therefore increasing the orbital energy. As only the closer planet needs to be reached to perform the gravity assist maneuver, large amounts of fuel can be saved attempting to reach planets in the outer Solar System. These assists have been used by many missions, due to the fuel savings, and therefore cost savings. [6]

Method

General method for finding trajectory

The method that was chosen for this trajectory finder is a simple trial and error method. Firstly, a simulation of the planets in the Solar System was made, which simplifies their trajectories to circular orbits, and has them orbit around the sun. The planets were put on fixed trajectories to simplify the amount of computation that had to be performed every time step. The initial positions of the planets are determined by the date that the program is started.

The Spacecraft is then simulated by using the equation for the force due to gravity (Eq 3) to determine the forces from all the planets that are being simulated. This force is then applied to the Spacecraft and the velocity change is calculated, and then by extension the change of position. This simulation is then stepped through and the positions of the planets and Spacecraft are updated for the whole time period specified by the user.

In order to check as many trajectories as possible, the program has a loop nested within another loop. The outer loop scans through the different initial dates, and the inner loop then scans through the different initial launch velocities. For each set of initial conditions, the simulation is run and the closest approaches to the target body are recorded. These approaches are then printed out, allowing for them to be analysed by the user and the best trajectory chosen to be analysed in further detail. This method of having the user decide the best trajectory was chosen because there are many variables that need to be considered to find optimal trajectories, such as travel time, launch velocity and approach velocity. The final trajectory is then drawn out by a separate program onto a graph.

Code application

The program that was written was split into 3 different sections: the simulator itself, the trajectory scanner, and the orbit drawer. This was done to allow for modular use of the simulator by both the scanner and the orbit drawer, and theoretically any other program that could need it.

Simulator

2 different classes are used for this simulation: “Body” and “Spacecraft”. Body contains the necessary information about a Solar System body, such as the mass, position, radius, etc.

```
class Body():
    def __init__(self, name, pos, ang, orbrad, rad, mass, SOI, fixed, color):
        self.name = name
        self.pos = [0,0]
        self.orbrad = orbrad
        self.fixed = fixed
        self.rad = rad
        self.color = color
        self.mass = mass
        self.ang = ang
        self.SOI = SOI*self.rad
        if orbrad != 0:
            self.pos[0] = self.orbrad*math.cos(math.radians(self.ang))
            self.pos[1] = self.orbrad*math.sin(math.radians(self.ang))
            self.orbvel = math.sqrt(GMSun/orbrad)
            self.orbper = math.sqrt((4*pi**2*orbrad**3)/(GMSun))
            self.stepAng = 360/self.orbper
```

Spacecraft contains the information about the Spacecraft, such as the position, velocity, closest approach information, etc.

```
class Spacecraft(object):
    def __init__(self, name, vel, pos, color, closest, startBody, final, closeTime, closeVel):
        self.name = name
        self.vel = vel
        self. pos = pos
        self.color = color
        self.closest = closest
        self.startBody = startBody
        self.final = final
        self.closeTime = closeTime
        self.closeVel = closeVel
```

The simulator subroutine takes 6 inputs:

- “bodies” – The list of bodies in the Solar System
- “Spc” – the Spacecraft that will be used for the trajectory
- “totalTime” – the total time for the simulation to run
- “currentDV” – the initial velocity of the Spacecraft
- “timeStep” – the amount of time between each calculation in the simulation
- “currentDate” – the start date for the simulation

The simulation starts by setting all the bodies and the Spacecraft to their initial positions, which are calculated using the “skyfield” library, which return the planet’s locations at any given date. The Spacecraft is then placed on the edge of the Solar System’s SOI, and given the initial velocity specified.

The program then enters a While loop which runs until the total time is reached, iterating the Spacecraft and the planets each time it passes through. The planets are simulated on a fixed path, and their next position is simply found through the equation for a position on a circle

```python
def stepPlanet(bodies, timeStep):
    for body in bodies:
        if body.fixed is False:
            angle = body.ang + body.stepAng*timeStep
            x = body.orbrad*math.cos(math.radians(angle))
            y = body.orbrad*math.sin(math.radians(angle))
            body.pos = [x,y]
            body.ang = angle
```

To calculate the next position of the Spacecraft, an n-body approach was taken to find the forces on the Spacecraft. This is done by calculating the effect of gravity on the Spacecraft of every body in the Solar System in turn. Using the equation of gravitational forces, (Eq 3) the force is calculated as a vector with an X and Y component. The force of each body is added together to find the total force on the Spacecraft. This is then used to calculate the change in velocity of the Spacecraft, which then this is used to calculate this distance traveled by the Spacecraft per each timeStep. The simulation also checks if the current location of the planet is closer than the saved closest approach and adjusts this value accordingly.

```python
def stepSpacecraft(bodies,spc, timeStep, currentDate,time):
    totForceX = totForceY = 0
    count = 0
    for body in bodies:
        dx = body.pos[0]-spc.pos[0]
        dy = body.pos[1]-spc.pos[1]
        dtotan=abs(dy)/abs(dx)
        gravAngle = math.atan(dtotan)
        d = math.sqrt((dx)**2+(dy)**2)
        f = (G*body.mass)/(d**2)
        fx = math.cos(gravAngle) * f
        fy = math.sin(gravAngle) * f
        if dx <= 0 and dy <= 0:
            fx = -fx
            fy = -fy
        elif dx > 0 and dy <= 0:
            fy = -fy
        elif dx<0 and dy>=0:
            fx= -fx
        if spc.closest[count] > d:
            spc.closest[count] = d
            spc.closeTime[count] = currentDate+timedelta(seconds=time)
            spc.closeVel[count] = math.sqrt((spc.vel[0])**2+(spc.vel[1])**2)
        count += 1
        totForceX += fx
        totForceY += fy
    spc.vel[0] += totForceX * timeStep
    spc.vel[1] += totForceY * timeStep
    spc.pos[0] += spc.vel[0] * timeStep
    spc.pos[1] += spc.vel[1] * timeStep
```

The full code for the simulator is available in the appendix under Ap 3

Trajectory Scanner

The trajectory scanner is the main part of this program, which runs the simulation from a different starting position and velocity. It starts at the indicated start date, and runs until the end date, flagging trajectories which are considered close enough to the target body.

The program begins by taking inputs from the user with regard to the program that is going to run. These inputs dictate the initial conditions for the orbit, with the start body for the Spacecraft, the start date, initial velocity, and the target body. It also takes the inputs for the step values for the time, initial velocity, and start times. This dictates how many calculations will be run for the program.

After these initial conditions are specified, a short test calculation is run and timed by the program. By calculating the total number of calculations, an estimate for the total time taken for the program is found and outputted to the user. These approximate timings were found to be fairly accurate, to within a few seconds of the actual time taken. After the user accepts the approximate time, another timer is started, and the main body of the program begins.

The main body of this program is centered around a nested while loop. The outer while loop dictates the start date of the simulation, and the inner loop sets the initial velocity to be tested. After setting both of these values, the program runs the simulator. Once the simulation is completed, an initial check of the closest approach of the planets is done, to check if the closest approach is inside a planet, indicating a collision. If this is the case, the trajectory is flagged and not included (if the Spacecraft trajectory ends up inside the target planet, this is not flagged). The closest approaches to the target body are then checked, and if any are within the target range, the trajectory is added to a separate list of trajectories that have been found to approach the target body. The initial conditions are then stepped forward by one step and the simulation is run again. This keeps repeating until the current conditions match the end conditions. The timer is then stopped, and the output recorded.

```
while currentDate <= endDate:
    currentDV = dvStart
    while currentDV <= dvEnd:
        currentCalc += 1
        #clear()
        print(str(round((currentCalc*100/totalCalc),2))+"%")
        simu.simulation(bodies,spc, totalTime, currentDV, timeStep, currentDate)
        farEnough = True
        bodyCountClose = 0
        for closer in spc.closest:
            if closer < bodies[bodyCountClose].rad and bodyCountClose != targetBodyNum:
                farEnough = False
            bodyCountClose += 1
        if spc.closest[targetBodyNum] < bodies[targetBodyNum].SOI*targetDistance and farEnough == True:
            finalTrajectories.append(simu.trajectoryInfo(currentDate,currentDV,0,(spc.closest[targetBodyNum]/bodies[targetBodyNum].SOI),spc.closeTime[targetBodyNum],spc.closeVel[targetBodyNum]))
            foundCount +=1
        currentDV += stepDV
    currentDate += stepStartDate
```

Once the while loops have finished, the timer is stopped, and the total time taken is calculated. All of the saved trajectories that have found a close approach are then written to an output file, along with the closest approach number and the total time taken.

The user can then inspect the found trajectories and choose which ones to pursue further to find a more accurate trajectory. This is not done through the program as the trajectory that is chosen is not just based on the least delta V, or the closest approach, or the slowest approach velocity, hence why it is difficult to find an algorithm that finds the “ideal” trajectory. Having the user choose the trajectory is a much more effective way of deciding, unless some kind of algorithm was designed to weigh up the options, or perhaps machine learning.

Full code for this part of the program is available under “Trajectory Scanner” in Ap 3

Orbit Drawer

The last part of this code is the orbit drawer. Using the “matplotlib” library in python, the simulator is run and at every time step of the simulation, the position of the Spacecraft is printed. This simply runs by taking the initial conditions of the orbit to be drawn, and then passing it into the simulator, with the “final” Boolean set to true so that the simulator is triggered to draw out the ouput. The positions at each step of the simulator are printed onto the matplotlib graph. After the simulation is carried out, a secondary routine is run to plot the final position of the objects on the graph. The closest approaches to each body in the simulation is then printed out to a text document.

Full code is available under Ap. 3

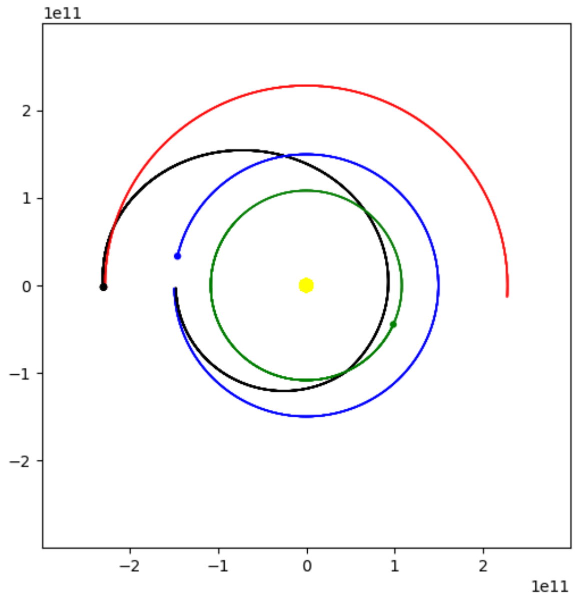

Figure 3 – Example output trajectory from Earth to Mars via Venus Gravity assist

Example Calculation (Earth Venus Mars)

Firstly, a wide search is run to find trajectories that get close to Mars. The target radius was set to 100 (which equates to 100 times the radius of the SOI of Mars), initial velocity step set to 10ms^{-1}, and time step of 1 day. The scan was run from 26 Kms^{-1} to 28Kms^{-1}. These initial values for velocity were found by doing a calculation for a Hohmann Transfer (Ap. 2) from Earth to Venus. The dates span for the year of 1996. This took about 3 hours to complete, and is by far the longest scan time of the whole trajectory finding

The initial scan found two trajectories that get close to Mars:

Traj	Launch Date	Launch DV(m/s)	Closest App (Body SOI)	App Time	Total Time
1	1996-03-18	27210	87.50955	1997-01-30	319 days
2	1996-03-22	26640	74.62968	1997-01-29	318 days

Table 2 - Wide scan of trajectories to Mars via Venus

Trajectory 1 was chosen to be investigated further as it has the least DV requirements. As shown in Figure 4, there is a small increase in velocity from the assist around Venus.

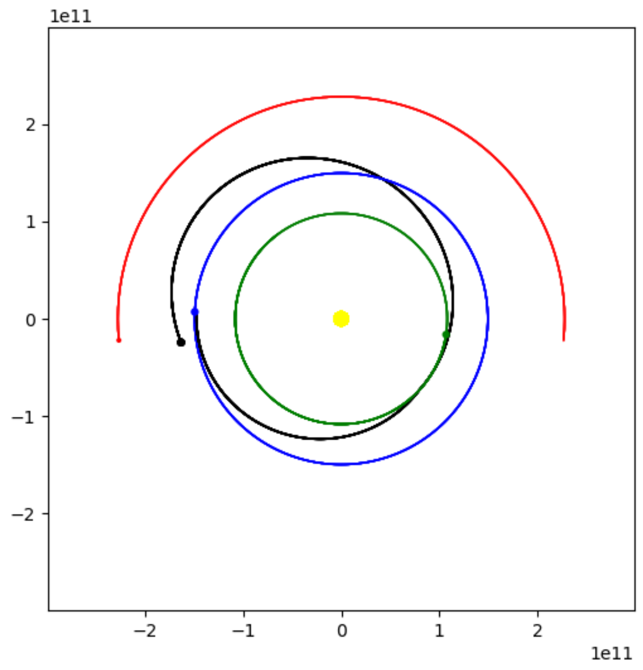

Figure 4 - Initial found trajectory

The search was then repeated, with tighter steps for the DV and start time. A step velocity of 0.01 ms^{-1} was used, at 1-hour intervals, centered around 3/18/1996. A trajectory was found that came within 3.6 SOI radii, at 00:00 on the 18th. This search took 4min and 26 seconds.

A final run was done with step DV of 0.0001 ms^{-1} and centered around midnight on the 18th with 1-minute intervals. This took 3 minutes and found a trajectory that came within 2.4 times the SOI radius, which is a small enough margin that trajectory correction maneuvers can be used to fine tune the approach.

The final values found for the trajectory are:

Launch Date	Launch DV(m/s)	Closest App (Body SOI)	App Time	Total Time
1996-03-18	27206.49	2.411468	1997-03-17	364 days

Table 3 - Final trajectory found to Mars via Venus

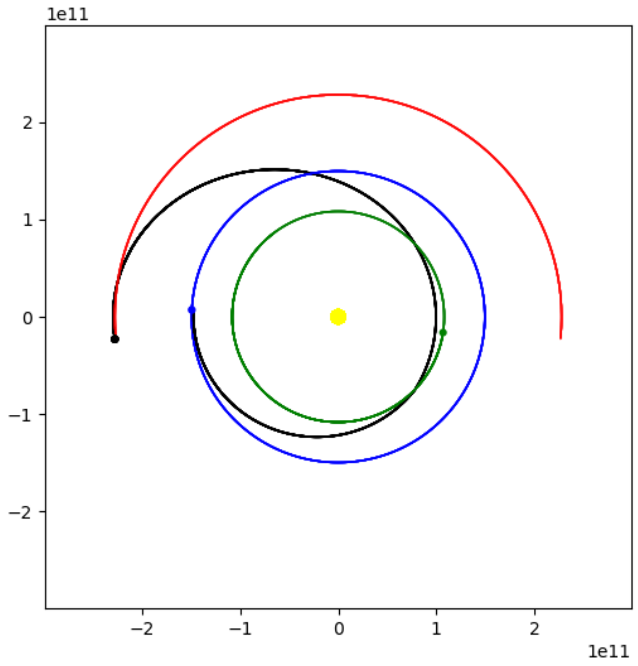

Figure 5- Final trajectory to Mars

Results

The final trajectory was found in two sections, the first section to place the Spacecraft into the position for the Deep Space Maneuver (DSM), and the second from the DSM to Saturn. This was done because the system only allows for the velocity to be set at the beginning of the trajectory.

The coloured lines on the graph represent the different bodies in the Solar System, with the black line showing the trajectory of the Spacecraft. The scale of the graphs is shown on the axis, to the scale of 10^{11}m.

Colour	Object	Colour	Object
Yellow	The Sun	Orange	Jupiter
Green	Venus	Yellow	Saturn
Blue	Earth	Black	Spacecraft
Red	Mars		

Table 4 – Legend for the output graphs

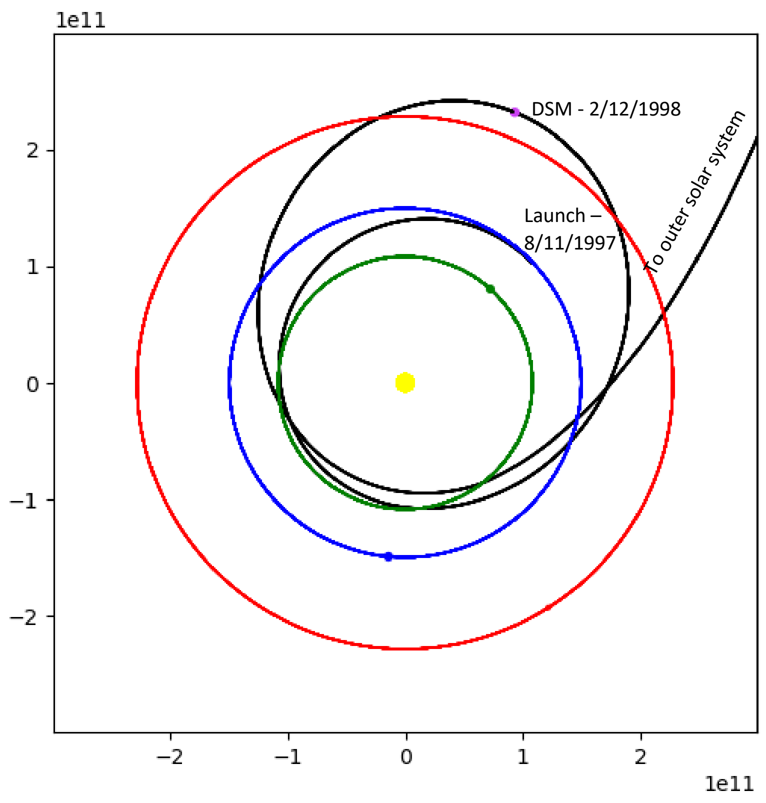

Figure 6 – Inner Planet View of the Trajectory, with the DSM marked in purple

The first section is the initial phase up to the DSM, starting with a retrograde burn to put the Spacecraft onto a Venus intercept, which propels the Spacecraft into the position required. This initial burn required an increase in velocity of 3663.59ms^{-1} when leaving Earth, launching on the 8th November 1997.

For the first section, it was clear that there was only one trajectory that would result in the Spacecraft being in the correct position when it reached the date for the DSM. However, past the DSM there were a couple of options, that mainly centered around two types of trajectory.

The first trajectory involved a much faster trajectory towards Saturn, making more use of the Earth and Jovian gravity assists to allow the Spacecraft to reach Saturn by August 2001. However, this resulted in a very high approach speed that would require a large burn upon the arrival at Saturn. This trajectory required a burn of 809.42ms^{-1}

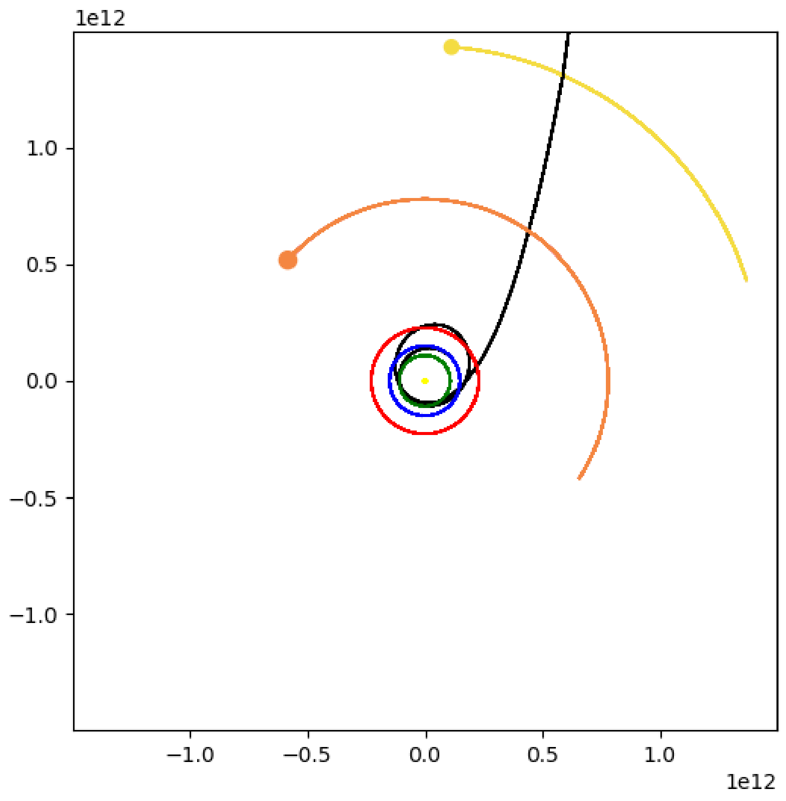

Figure 7 - Fast flyby of Saturn

The second trajectory does not get as close to Jupiter, and therefore does not gain the same amount of velocity. This allows for a much slower approach to Saturn and would require much less fuel to enter the system. This came at the cost of increasing the arrival time to the 12th October 2004. This trajectory took a burn of 809.40ms^{-1}

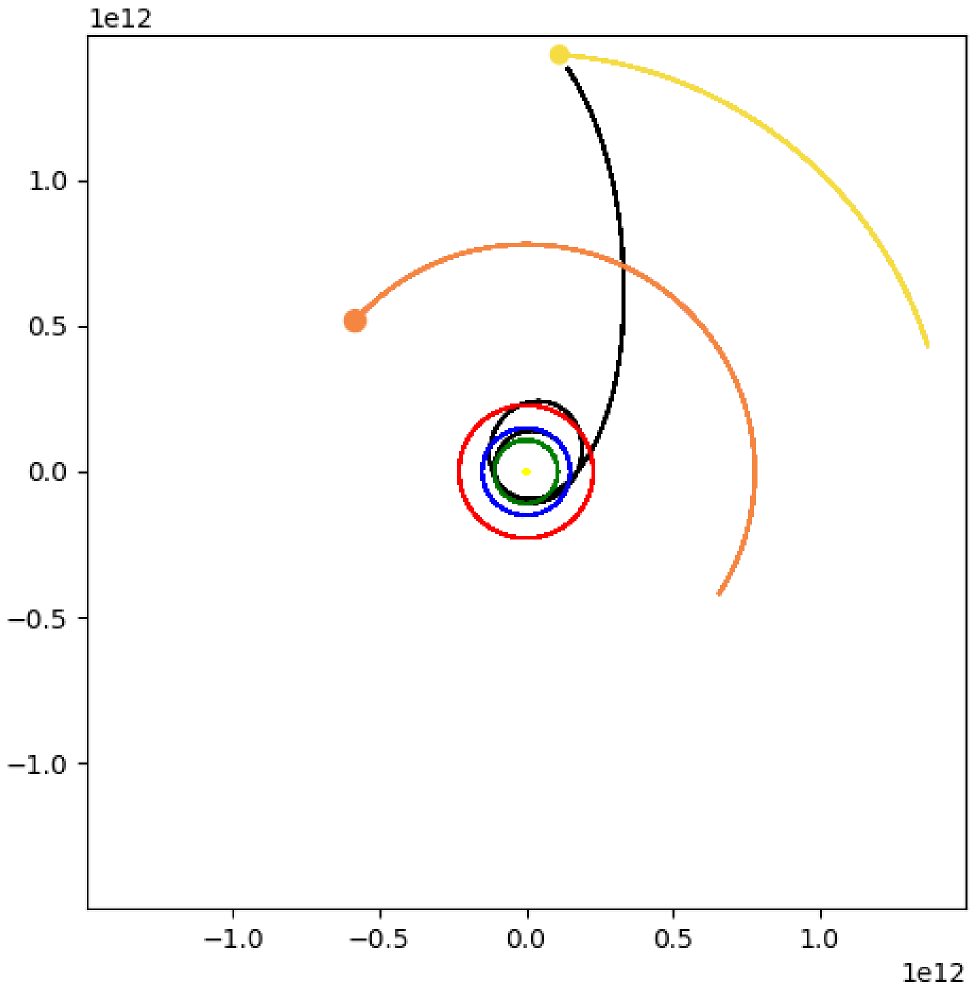

Figure 8 - Slower approach to Saturn

The fly-by dates of each planet are shown in the following table, as well as the total time for the mission:

	Planet						Total Time
	Launch	Venus 1	Venus 2	Earth	Jupiter	Saturn	
Cassini [7]	15/10/1997	26/04/1998	24/06/1999	18/08/1999	30/12/2000	18/5/2004	2451 Days
Found Trajectory	8/11/1997	25/04/1998	8/8/1999	29/08/1999	12/20/2000	19/6/2003	2049 Days
Difference	16 Days	1 Day	45 Days	11 Days	18 Days	333 Days	448 Days

Table 5 - Flyby dates of the planets by the Spacecraft

The DeltaV requirements for this mission are shown in the following table. An arbitrary circular parking orbit of 400km altitude was chosen. The calculation for the DV required at launch is shown in Ap 2

	Launch (ms^{-1})	DSM (ms^{-1})	Saturn Insertion(ms^{-1})	Total(ms^{-1})
Cassini [7]	3687.29[1]	450	626	4137.29
DSM	3663.59	809.40	1746.8	6219.79
Difference	53.7	359.6	1120.8	2082.5

Table 6 - DeltaV requirements for the trajectories

[1] This number is an approximation based on an image provided by NASA, as the real value does not seem to have been released.

Conclusion

Of the two found trajectories, the second one was chosen as it would reduce the burn required at Saturn to slow the Spacecraft down. This would increase the total flight time by a large margin, however the insertion burn for this trajectory would be in the tens of thousands of delta V, which would not be feasible for the Cassini probe.

The intercept dates of the two trajectories match up within a few days for the early encounters. There is a difference of two weeks for the launch date due to the slight difference in the initial launch trajectory which is caused by a limitation of the system. The system only checks for prograde and retrograde burns from Earth, whereas the Cassini mission had a radial component to the velocity on launch. This also causes a small discrepancy of 53.7ms^{-1} in the delta V. The first fly-past of Venus is within a day of the Cassini trajectory, and provides the required gravity assist to put the Spacecraft into the orbit that is required for the DSM. The position of the DSM maneuver also closely matches the position of the Cassini mission, with it being performed just outside the radius of Mars' orbit. This section of the trajectory is the least complicated, so it is understandable that the trajectories are very closely matched.

The next section of the trajectory is governed by the DSM, which occurs on the 2nd of December. This is a day before the Cassini probe performed a similar burn. The purpose of this burn is to fine tune the second approach to Venus, to get the required gravity assist in order for the rest of the trajectory to be possible. There was quite a large difference in the velocity of this DSM, with the Cassini mission requiring 359.6ms^{-1} less delta V. This caused a slightly different second Venus intercept than Cassini, however the Spacecraft ended up on a similar trajectory once on its way out of the inner Solar System.

The second part of the trajectory matches the Cassini trajectory to a certain degree as well. The Earth and Jupiter intercept dates are within weeks of the Cassini mission, which shows that the same trajectory is being followed, with the only main difference being the timing of the approach to Saturn. This is about a year off the final mission, and results in a higher approach speed to Saturn.

The Cassini probe only had to perform a 626ms^{-1} burn to stay within the SOI of Saturn [5], and this mission would require approximately 3.5kms^{-1}, which is significantly higher, and would ultimately not have been possible by the Cassini probe. This is the main drawback of the trajectory that was found.

The system performed well, providing a trajectory that is similar to the Cassini trajectory, and performed all tasks such as drawing out the trajectories, marking closest approaches to planets, etc. The main limitation of the system running is the time taken for each calculation. Some of the more complicated parts of the trajectory, such as the second section after the DSM, took a few hours to run. The other main limitation of the program, and most likely the reason that the trajectory differed in the end from the Cassini mission, was that it only allowed for a velocity change that was either directly prograde or retrograde from the starting planet's velocity. Many of the burns done by Cassini were not directly prograde or retrograde, and this allowed finer control over the Spacecraft's trajectory.

Another limitation is the planet's trajectories all being modeled on fixed paths around the sun with a constant radius. This removes a certain level of accuracy from the system, but it was done to reduce the time taken for the calculations.

Further work into this system would likely involve adding a functionality that could take a burn that had various different components, which would be closer to the way that the maneuvers were performed on the Cassini mission, along with many other similar Spacecraft missions. This functionality was not added for this program as the extra time needed to check exponentially more trajectories would not have been feasible.

Another change that would greatly increase the accuracy of this system is to place the planets on their exact trajectories, as opposed to the way they are simulated by this system. The "Skyfield" python library that the system uses to place the planets in their initial positions is capable of providing this functionality. The only reason it was not used for this system is that it would greatly increase the time taken for the system to run each step, therefore increasing the overall runtime.

Both of these limitations are based around reducing the time taken for the calculations to be performed down to reasonable time limits. Currently this system is running in the RAM of the computer being used. The more accuracy required by the program, the greater the computing power must be. Ways of increasing this computing power include converting the program to be able to run using GPU computing, cloud computing, or just simply using a more powerful computer. For example, NASA use their Pleiades Supercomputer to perform their trajectory calculations, which is multiple magnitudes bigger than the Personal Computer that was being used to run this program.

Another method that could be followed is optimising the code so that it can perform the calculations more effectively. For example, having a program that uses machine learning to learn the best way to begin trajectories could reduce the total number of calculations required. There is a large amount of new studies being currently conducted into trajectory finding programs, such as the work done by Shirobokov, showing that the optimization of trajectory design is very relevant to modern study.

Bibliography

[1] M. Vasile, "On the Preliminary Design of Multiple Gravity-Assist Trajectories," *Journal of Spacecraft and Rockets,* pp. 794-805, 2006.

[2] J. V. Allen, "Gravitational Assist in Celestial Mechanics," *Association of Phsyics,* pp. 448-451, 2002.

[3] T. Greicius, "Cassini," 06 04 2019. [Online]. Available: https://www.nasa.gov/mission_pages/cassini/main/index.html.

[4] R. S. Dunning, The Orbital Mechanics of Flight Mechanics, NASA, 1973.

[5] T. Goodson, "Cassini Maneuver Experiece: Launch and Early Cruise," vol. 4224, no. 98, 1998.

[6] K. Qadir, "Multi-gravity assist design tool for interplanetary trajectory optimisation," Cranfield University, 2009.

[7] P. F, "Cassini Interplanetary Trajectory Design," vol. 0818, no. 95, 1995.

Acknowledgements

I’d firstly like to say thank you to Dr Vepa for his help on this project, his input helped shape this project from the beginning and I would not have been as successful without him.

I’d also like to thank the Queen Mary Climbing society for providing me with the support, friendship, and coffee to help me get through this report. I’d also like to thank Lara for helping me proofread and correcting my awful grammar.

Appendix

Ap. 1 – Hohmann Transfer Calculation

Firstly, we take the Eq.1 and set it equivalent to the gravitational constant divided by twice the semi major axis:

$$\frac{1}{2}v^2 - \frac{\mu}{r} = -\frac{\mu}{2a}$$

Next, we know that the semi major axis is the average between the apoapsis and the periapsis. For our Hohmann Transfer, we know this needs to be the radius of our initial orbit, and the radius of our target orbit r_a:

$$a = \frac{r_a + r_p}{2}$$

$$\frac{1}{2}v^2 - \frac{\mu}{r} = -\frac{\mu}{r_a + r_p}$$

Rearranging this for v and setting r to either the periapsis or apoapsis location depending on which way the transfer is going, we can find an equation for v: [4]

$$v_p = \sqrt{2\left(\frac{\mu}{r_p} - \frac{\mu}{r_a + r_p}\right)}$$

$$v_a = \sqrt{2\left(\frac{\mu}{r_a} - \frac{\mu}{r_a + r_p}\right)}$$

Ap. 2 - V cir to V inf

If we take Eq. 1 and take it out to r=∞, we can see that this term will drop out and leave us with the equation for v_∞:

$$\frac{1}{2}v_\infty^2 = E$$

If we set this equal to the energy equation, due to the fact that energy remains constant in the orbit, we can find the velocity required at the periapsis:

$$\frac{1}{2}v_\infty^2 = \frac{1}{2}v_p^2 - \frac{\mu}{r_p}$$

$$v_p = \sqrt{2\left(\frac{\mu}{r_p} + \frac{1}{2}v_\infty^2\right)}$$

Then, using a variant of this equation to work out the orbital velocity of our parking orbit, we can subtract these two values to find out the required dV: [4]

$$v_c = \sqrt{\frac{\mu}{r}}$$

$$v_p - v_c = dV$$

Ap. 3 – Full Code

Simulator

```
import matplotlib.pyplot as graph
import math
from astropy import constants as const
from skyfield.api import Loader, utc
from datetime import datetime, timedelta
import os
clear = lambda: os.system('cls')

#Needed for skyfield data to be loadable
load = Loader('~/skyfield-data')
ts = load.timescale()
bodiesData = load("de421.bsp")

#Constants
G = const.G.value
AU = const.au.value
GMSun = const.GM_sun.value
pi = math.pi
close = AU*50

class location():
    def __init__(self, pos, vel, velAng,bodyVelAng, date):
        self.pos = []
        self.vel = vel
        self.velAng = velAng
        self.bodyVelAng = bodyVelAng
        self.date = date

#Class for saved trajectories
class trajectoryInfo(object):
    def __init__(self,startDate,launchDV, timeforJourney, closestApp, closestAppTime, appVel, apo):
        self.startDate = startDate
        self.launchDV = launchDV
        self.timeforJourney = timeforJourney
        self.closestApp = closestApp
        self.closestAppTime = closestAppTime
        self.appVel = appVel
        self.apo = apo
#Class for bodies, eg sun and planets
class Body():
    def __init__(self, name, pos, ang, orbrad, rad, mass, SOI, fixed, color):
        self.name = name
        self.pos = [0,0]
        self.orbrad = orbrad
        self.fixed = fixed
        self.rad = rad
        self.color = color
        self.mass = mass
        self.ang = ang
        self.SOI = SOI*self.rad
        if orbrad != 0:
            self.pos[0] = self.orbrad*math.cos(math.radians(self.ang))
            self.pos[1] = self.orbrad*math.sin(math.radians(self.ang))
            self.orbvel = math.sqrt(GMSun/orbrad)
            self.orbper = math.sqrt((4*pi**2*orbrad**3)/(GMSun))
            self.stepAng = 360/self.orbper
```

```

#Class for the Spacecraft
class Spacecraft(object):
    def __init__(self, name, vel, pos, color, closest, startBody, final, closeTime, closeVel):
        self.name = name
        self.vel = vel
        self. pos = pos
        self.color = color
        self.closest = closest
        self.startBody = startBody
        self.final = final
        self.closeTime = closeTime
        self.closeVel = closeVel

#Initialise Solar System bodies and create lists of bodies
sun = Body(name="Sun",pos=[0,0],orbrad=0,ang = 0, rad=const.R_sun.value,mass=const.M_sun.value,SOI = 1, fixed=True, color = "Yellow")
venus = Body(name="Venus",pos=[0,0],orbrad=AU*0.7233, ang = 0, rad=const.R_earth.value*0.9488,mass=const.M_earth.value*0.815, SOI = 102, fixed=False, color = "green")
earth = Body(name="Earth",pos=[0,0],orbrad=AU, ang = 0, rad=const.R_earth.value,mass=const.M_earth.value, SOI = 145, fixed=False, color = "blue")
mars = Body(name="Mars",pos=[0,0],orbrad=AU*1.5237, ang = 0,  rad=const.R_earth.value*0.53,mass=const.M_earth.value*0.107, SOI = 170, fixed=False, color = "red")
jupiter = Body(name="Jupiter",pos=[0,0],orbrad=AU*5.2044, ang = 0,  rad=const.R_earth.value*11.209,mass=const.M_earth.value*317.8, SOI = 687, fixed=False, color = "#f48641")
saturn = Body(name="Saturn",pos=[0,0],orbrad=AU*9.5826, ang = 0,  rad=const.R_earth.value*9.449,mass=const.M_earth.value*95.159, SOI = 1025, fixed=False, color = "#f4dc42")

sunSky = bodiesData["sun"]
venusSky = bodiesData["VENUS BARYCENTER"]
earthSky = bodiesData["EARTH BARYCENTER"]
marsSky = bodiesData["MARS BARYCENTER"]
jupiterSky = bodiesData["JUPITER BARYCENTER"]
saturnSky = bodiesData["SATURN BARYCENTER"]

bodies = [sun,venus,earth,mars, jupiter, saturn]
bodiesSky = [sunSky,venusSky,earthSky,marsSky, jupiterSky, saturnSky]
apo = location([0,0],0,0,0,datetime(1997,11,8,0,0,0).replace(tzinfo=utc))

#Draw position of planets on graph
def drawPos(bodies,spc):
    for body in bodies:
        place = graph.Circle(body.pos, radius=body.rad*10, color=body.color)
        graph.gca().add_patch(place)
    x = spc.pos[0]
    y = spc.pos[1]
    place1 = graph.Circle((x,y), radius=200000000, color=spc.color)
    graph.gca().add_patch(place1)

#Draw final position of planets on graph
def drawPosDone(bodies, spc):
    for body in bodies:
        if body.name != "Sun":
            place = graph.Circle(body.pos, radius=body.rad*500, color=body.color)
            graph.gca().add_patch(place)
        x = spc.pos[0]
        y = spc.pos[1]
        place1 = graph.Circle((x,y), radius=bodies[2].rad*500, color=spc.color)
        graph.gca().add_patch(place1)

#Find initial positions of bodies at a given date, from skyfield API
```

```
def findBodyPos(body,currentDate, bodyNum):
    timeNow = ts.utc(currentDate)
    bodyPos = bodiesSky[bodyNum].at(timeNow).position.km*1000
    xSky = bodyPos[0]
    ySky = bodyPos[1]
    dSky = math.sqrt(xSky**2+ySky**2)
    body.ang = math.degrees(math.acos(xSky/dSky))
    if ySky < 0:
        body.ang = -body.ang
    initX = body.orbrad*math.cos(math.radians(body.ang))
    initY = body.orbrad*math.sin(math.radians(body.ang))
    return [initX,initY]

#Set up simulator first positions, setting planet and Spacecraft initial locations
def initialSim(bodies, currentDate,spc,currentDV):
    countStepBody=0
    spc.closest.clear()
    spc.closeTime.clear()
    spc.closeVel.clear()
    apo = location([0,0],0,0,0,datetime(1997,11,8,0,0,0).replace(tzinfo=utc))
    for body in bodies:
        spc.closest.append(close)
        spc.closeTime.append(datetime(1997,10,15,0,0,0).replace(tzinfo=utc))
        spc.closeVel.append(0)
        if not body.fixed:
            body.pos = findBodyPos(body, currentDate, countStepBody)
            if spc.startBody == body.name:
                spc.pos[0] = body.pos[0] - body.SOI*math.sin(math.radians(90-body.ang))
                spc.pos[1] = body.pos[1] - body.SOI*math.cos(math.radians(90-body.ang))
                spcvelAng = body.ang + 90
                if spc.pos[1] < 0:
                    spcvelAng = -spcvelAng
                velX = currentDV*math.cos(math.radians(spcvelAng))
                velY = currentDV*math.sin(math.radians(spcvelAng))
                spc.vel[0] = velX
                spc.vel[1] = velY
            if spc.startBody == "DSM":
                spc.pos[0] = 91894034111.96698
                spc.pos[1] = 232145539946.29926
                spcD = (math.sqrt(spc.pos[0]**2+spc.pos[1]**2))
                spcvelAng = math.degrees(math.acos(spc.pos[0]/spcD))
                velX = currentDV*math.cos(math.radians(spcvelAng+90))
                velY = currentDV*math.sin(math.radians(spcvelAng+90))
                spc.vel[0] = velX
                spc.vel[1] = velY

        countStepBody+=1

#Step Spacecraft through one timestep
def stepSpacecraft(bodies,spc, timeStep, currentDate,time):
    totForceX = totForceY = 0
    count = 0
    for body in bodies:
        dx = body.pos[0]-spc.pos[0]
        dy = body.pos[1]-spc.pos[1]
        dtotan=abs(dy)/abs(dx)
        gravAngle = math.atan(dtotan)
        d = math.sqrt((dx)**2+(dy)**2)
        f = (G*body.mass)/(d**2)
        fx = math.cos(gravAngle) * f
        fy = math.sin(gravAngle) * f
        if dx <= 0 and dy <= 0:
            fx = -fx
            fy = -fy
        elif dx > 0 and dy <= 0:
            fy = -fy
        elif dx<0 and dy>=0:
            fx= -fx
        if spc.closest[count] > d:
            spc.closest[count] = d
            spc.closeTime[count] = currentDate+timedelta(seconds=time)
            spc.closeVel[count] = math.sqrt((spc.vel[0])**2+(spc.vel[1])**2)

        count += 1
        totForceX += fx
        totForceY += fy
    spc.vel[0] += totForceX * timeStep
    spc.vel[1] += totForceY * timeStep
    spc.pos[0] += spc.vel[0] * timeStep
    spc.pos[1] += spc.vel[1] * timeStep

#Step planets through one time step
def stepPlanet(bodies, timeStep):
    for body in bodies:
        if body.fixed is False:
            angle = body.ang + body.stepAng*timeStep
            x = body.orbrad*math.cos(math.radians(angle))
            y = body.orbrad*math.sin(math.radians(angle))
            body.pos = [x,y]
            body.ang = angle

#Simulation looper, runs through from given initial date to end date.
def simulation(bodies,spc,totalTime,currentDV, timeStep, currentDate):
    time = 0
    farRad = 0
    initialSim(bodies,currentDate,spc,currentDV)
    while time < totalTime:
        spc.foundClose = False
        stepPlanet(bodies,timeStep)
        stepSpacecraft(bodies,spc,timeStep,currentDate,time)
        if spc.final == True:
            drawPos(bodies,spc)
            spcD = (math.sqrt(spc.pos[0]**2+spc.pos[1]**2))
            if spcD>farRad:
                apo.date = currentDate+timedelta(seconds=time)
                apo.pos = spc.pos
                apo.vel = spc.vel
                #apo.vel = math.sqrt(spc.vel[0]**2+spc.vel[1]**2)
                apo.velAng = math.degrees(math.tan(spc.vel[0]/spc.vel[1]))
                apo.bodyVelAng = bodies[5].ang
                farRad = spcD
        time += timeStep

#Returns target number in list of planets
def findTarget(bodies, target):
    countTarget = 0
    for body in bodies:
        if body.name == target:
            return countTarget
        countTarget += 1

#Returns list of bodies, used to get list for use when simulator.py is called in other programs
```

```python
def returnBodyList():
    return bodies

def returnLocation():
    return apo


```

Scanner

```python
import math
from astropy import constants as const
import Simulator as simu
import datetime
from skyfield.api import utc
import xlsxwriter
import os
clear = lambda: os.system('cls')

G = const.G.value
AU = const.au.value
GMSun = const.GM_sun.value
pi = math.pi

resetDate = datetime.datetime(1990,1,1,0,0,0).replace(tzinfo=utc)
spc = simu.Spacecraft(name="Cassini", vel = [0,0], pos = [AU-1*10**9,0], color="black", closest = [], startBody= "Earth", final = False, closeTime = [], closeVel = [])
spcFinal = simu.Spacecraft(name="CassiniOpt", vel = [0,0], pos = [AU-1*10**9,0], color="black", closest = [], startBody= "Earth",  final = True, closeTime = [], closeVel = [])
spcReset = simu.Spacecraft(name="Cassini", vel = [0,0], pos = [AU-1*10**9,0], color="black", closest = [], startBody= "Earth", final = False, closeTime = [], closeVel = [])
bodies = simu.returnBodyList()

finalTrajectories = []

targetDistance = 100

timeStep = 3600
totalTime = 3600*24*365*6

stepStartDate = datetime.timedelta(days=1)
startDate = datetime.datetime(1997,11,1,0,0,0).replace(tzinfo=utc)
endDate = datetime.datetime(1997,11,15,2,0,0).replace(tzinfo=utc)
currentDate = startDate

stepDV = 1

dvStart = 26788.326
dvEnd = 26788.326

currentDV = dvStart

targetBody = "Venus"

for body in bodies:
    spcFinal.closest.append(AU*50)
```

```python

currentCalc = 0
totalCalc = ((endDate+stepStartDate-startDate)/stepStartDate) + ((dvEnd-dvStart)/stepDV)*((endDate+stepStartDate-startDate)/stepStartDate)
targetBodyNum = simu.findTarget(bodies, targetBody)

startapproxtime= datetime.datetime.now()
simu.simulation(bodies,spc, totalTime, currentDV, timeStep, currentDate)
endapproxtime = datetime.datetime.now()

approxTimePerCalc = endapproxtime-startapproxtime
approxTime = totalCalc*approxTimePerCalc

print("Total Calculations: " + str(totalCalc))
print("Approx time: " + str(approxTime))
input("Ensure Excel is closed, and press enter to continue")

startTime = datetime.datetime.now()
farEnough = True
bodyCountClose = 0
foundCount = 0

while currentDate <= endDate:
    currentDV = dvStart
    while currentDV <= dvEnd:
        currentCalc += 1
        #clear()
        print(str(round((currentCalc*100/totalCalc),2))+"%")
        simu.simulation(bodies,spc, totalTime, currentDV, timeStep, currentDate)
        farEnough = True
        bodyCountClose = 0
        for closer in spc.closest:
            if closer < bodies[bodyCountClose].rad*1.2 and bodyCountClose != targetBodyNum:
                farEnough = False
            bodyCountClose += 1
        if spc.closest[targetBodyNum] < bodies[targetBodyNum].SOI*targetDistance and farEnough == True:
            finalTrajectories.append(simu.trajectoryInfo(currentDate,currentDV,0,(spc.closest[targetBodyNum]/bodies[targetBodyNum].SOI),spc.closeTime[targetBodyNum],(spc.closeVel[targetBodyNum]-bodies[targetBodyNum].orbvel),simu.returnLocation()))
            foundCount +=1
        currentDV += stepDV
    currentDate += stepStartDate

finishTime = datetime.datetime.now()
finishTime = finishTime-startTime
try:
    workbook = xlsxwriter.Workbook("Outputs/trajectories.xlsx")
except:
    input("Close Excel")
    workbook = xlsxwriter.Workbook("Outputs/trajectories.xlsx")

worksheet = workbook.add_worksheet()
worksheet.write(0,0,"Traj")
worksheet.write(0,1,"Launch Date")
worksheet.write(0,2,"Launch DV")
worksheet.write(0,3,"Closest App")
worksheet.write(0,4,"App Time")
worksheet.write(0,5,"App Vel")
worksheet.write(0,6,"Total Time")
worksheet.write(0,7,"Apo Time")
worksheet.write(0,9,"Closest Trajectory")
worksheet.write(0,10,"Total Calculations")
worksheet.write(0,11,"Estimated Time")
worksheet.write(0,12,"Total Time Taken")

try:
    closestTraj = finalTrajectories[0]
except IOError:
    input("No trj Found")
countTraj = 0
row = 1
count = 1

for final in finalTrajectories:
    if closestTraj.closestApp > final.closestApp:
        closestTraj = final
        countTraj = count
    worksheet.write(row,0,count)
    worksheet.write(row,1,str(final.startDate))
    worksheet.write(row,2,final.launchDV)
    worksheet.write(row,3,final.closestApp)
    worksheet.write(row,4,str(final.closestAppTime))
    worksheet.write(row,5,final.appVel)
    totalTimeDate = final.closestAppTime - startDate
    worksheet.write(row,6,str(totalTimeDate))
    worksheet.write(row,7,str(final.apo.date))

    count +=1
    row+=1
string = 'internal:Sheet1!A'+str(countTraj+1)
worksheet.write_url(1,9,string)
worksheet.write(1,9,countTraj)
worksheet.write(1,10,currentCalc)
worksheet.write(1,11,str(approxTime))
worksheet.write(1,12,str(finishTime))
chart = workbook.add_chart({'type': 'line'})
chart.add_series({'values': ["Sheet1",1,3,row,3]})
worksheet.insert_chart(3,9,chart)
workbook.close()
print("Done")
input()

```

Orbit Drawer

```python
import matplotlib.pyplot as graph
from astropy import constants as const
from skyfield.api import utc
import math
import datetime
import Simulator as simu

#Constants
G = const.G.value
AU = const.au.value
GMSun = const.GM_sun.value
pi = math.pi

```

```python
#Initialise graph
graphSize = AU*10
graphDim = [-graphSize, graphSize, -graphSize, graphSize]
graphScale = 1000000
graph.figure(figsize=(6,6))
graph.axis(graphDim)

#Initialise Bodies and Spacecraft
spc = simu.Spacecraft(name="Cassini", vel = [0,0], pos = [AU-1*10**9,0], color="black", closest = [], startBody= "Venus", final = True, closeTime = [], closeVel = [])
bodies = simu.returnBodyList()

#Set initial conditions for orbit
timeStep = 3600
totalTime = 3600*24*1660
startDate = datetime.datetime(1998,12,2,2,0,0).replace(tzinfo=utc)
dvInf =40277.11

targetBody = "Saturn"
targetBodyNum = simu.findTarget(bodies,targetBody)

#Simulation run
print("Running...")
simu.simulation(bodies,spc,totalTime,dvInf,timeStep,startDate)
simu.drawPosDone(bodies,spc)
print("Done, loading graph")

#Write outputs to file and save graph
outputFile = open("Outputs/BestFound.txt", "w")
apo = simu.returnLocation()
outputFile.write("Launch DV:" + str(dvInf))
outputFile.write("\nStart Date:" + str(startDate))
totalTimeDate = spc.closeTime[targetBodyNum] - startDate
outputFile.write("\nTotal Flight Time:" + str(totalTimeDate))
outputFile.write("\nLocation Pos:" + str(apo.pos))
outputFile.write("\nLocation Rad:" + str(((math.sqrt(apo.pos[0]**2+apo.pos[1]**2))/AU)))
outputFile.write("\nLocation Time:" + str(apo.date))
outputFile.write("\nLocation Vel:" + str(apo.vel))
outputFile.write("\nLocation Ang:" + str(apo.velAng))
outputFile.write("\nLocation Body Ang:" + str(apo.bodyVelAng))
outputFile.write("\n")
count = 0
outputFile.write("\nSpacecraft Closest Approaches")
outputFile.write("\n")
for space in spc.closest:
    outputFile.write("\nBody: " + bodies[count].name + "   Closest: " + str(round((spc.closest[count]/bodies[count].SOI),2)) + " Body SOI")
    outputFile.write("\nTime of closest approach: " +str(spc.closeTime[count]))
    outputFile.write("\n")
    count += 1

outputFile.close()
graph.savefig("Outputs\graph.png")
print("Graph saved")
input()
```