Session 3: Objectives

30 min Hands on

- set up and run earth orbit acc(single trajectory run)

30 min Theory

- understand the high level architecture of the codebase
- what goes into a single trajectory run

Bonus:

- run GPU accelerated 2D paramter sweep(multiple parallel trajectory runs)

Hands on: running 1 trajectory around earth

- segments
  - clone code base - learn how to use `git`
  - `Pkg` init in julia `repl` NOT your shell!
  - installing packages and running `earth_orbit_acc.jl`
- workflow for each segment:
  - live demonstration & show plots
  - then go around help others set up

Theory section: explaining Earth Orbit Acc

- single trajectory run for 1 set of initial conditions

Please open the file `earth_orbit_acc.jl` on your device or on the github site. It is located in the folder `examples/`

Define Run-time and Initial Conditions

Comment sections: `#1-3` : Before we run simulations we ask:

Run-time

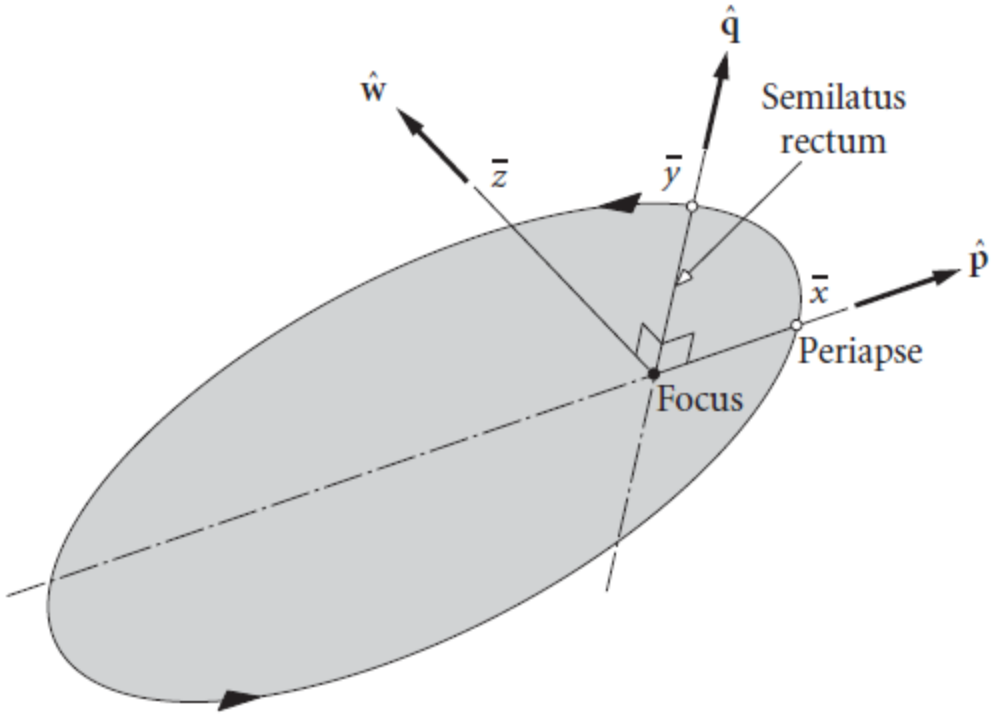
- How long to propagate? Integration time step? Include torques?
- Do we use accurately updated sun positions or simply assume a fixed sun position?

Initial Conditions

- What kind of orbit and where to orbit?
- What is the spacecraft's initial position,velocity,orientation?

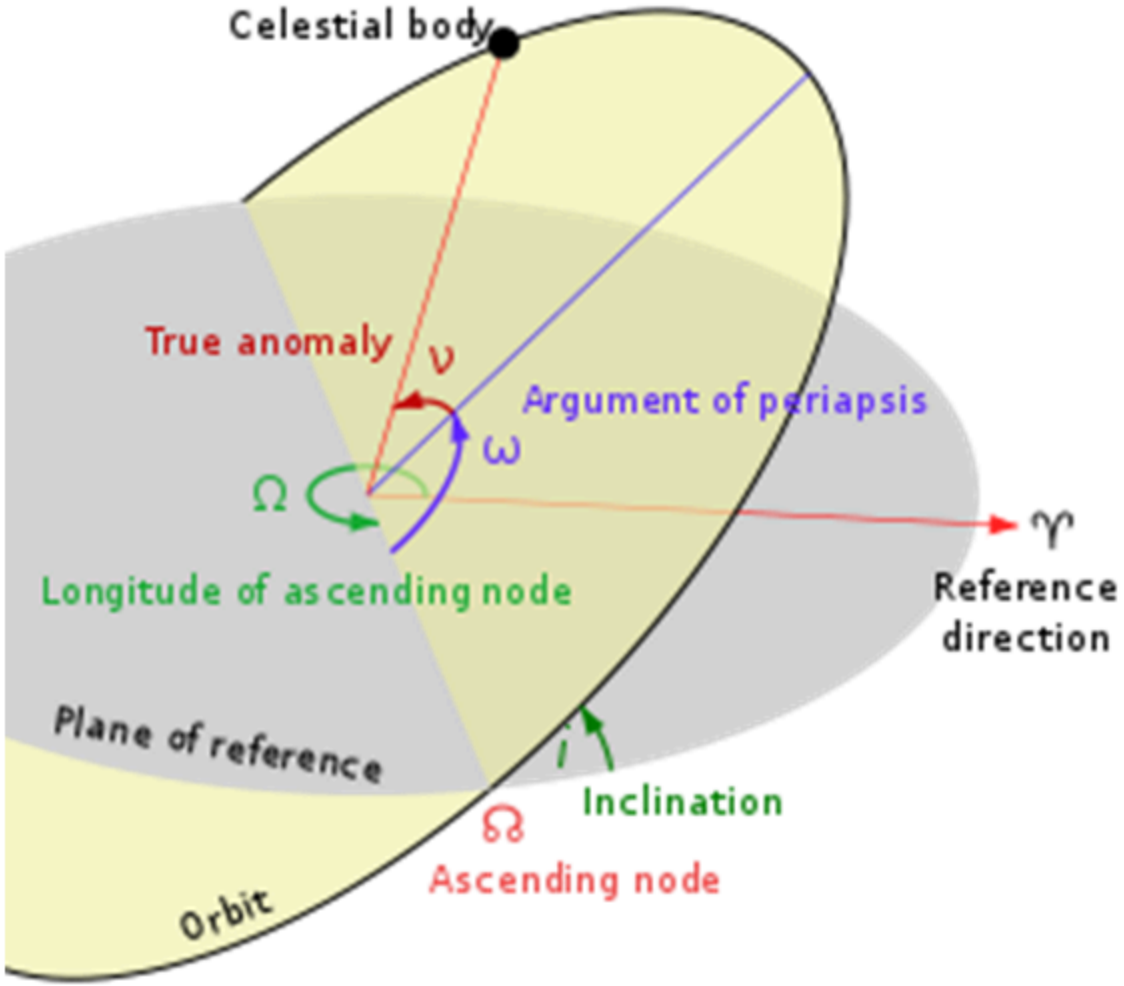
Perifocal frame

POV: you are now the spacecraft zooming around in orbit. The orbital plane is what you see



Inertial frame(ECI)

Zoom out! Now let us consider the orientation and position of this orbital plane relative to that of the earth



Putting together how do we go from `perifocal` to `inertial` frame?

- Perifocal position & velocity

$$\mathbf{r}_{pf} = \frac{p}{1 + e\cos\nu}\begin{pmatrix}\cos\nu\\ \sin\nu\\ 0\end{pmatrix}, \qquad \mathbf{v}_{pf} = \sqrt{\frac{\mu}{p}}\begin{pmatrix}-\sin\nu\\ e+\cos\nu\\ 0\end{pmatrix}, \quad \text{where } p = a(1-e^2)$$

- Rotation matrices

$$R_1(\theta) = \begin{pmatrix}1 & 0 & 0\\ 0 & \cos\theta & -\sin\theta\\ 0 & \sin\theta & \cos\theta\end{pmatrix}, \quad R_3(\theta) = \begin{pmatrix}\cos\theta & \sin\theta & 0\\ -\sin\theta & \cos\theta & 0\\ 0 & 0 & 1\end{pmatrix}$$

3 times: rotate this amount this direction. You will then find yourself on the other plane

$$\mathbf{r} = Q\mathbf{r}_{pf}, \qquad \mathbf{v} = Q\mathbf{v}_{pf}, \qquad Q = R_3(-\Omega)\,R_1(-i)\,R_3(-\omega)$$

Cartesian Elements

Basically the code we called upon `coe2rv` located in our custom module `OrbitalState.jl` to convert elements $(a, e, i, \Omega, \omega, v)$ defining the orbital plane to the familiar `x,y,z` positional and velocity vector.

$$\boxed{\text{Orbital Elements}} \rightarrow \boxed{\text{Cartesian Elements}}$$

- `r::SVector{3,Float64}` : Position vector [m]
- `v::SVector{3,Float64}` : Velocity vector [m/s]

Additionally we have these other Cartesian elements to describe rotational orientation and speed of the sail

- `q::SVector{4,Float64}` : Quaternion (attitude)
- `ω::SVector{3,Float64}` : Angular velocity [rad/s]

Why this specific set of run-time and initial conditions? Well we have a whole paper on that...for simplicity I'll cover 2 key concepts governing this

```
# 0.  Run-time options
const YEARS        = T(1) / T(4)     # 0.25 years
const DT           = T(100)
const WARMUP_DT    = T(1)
const PROGRESS_BAR = true
const RATE         = T(1.0)
const PITCH        = T(50)
const MODEL_TORQUES = false
```

```
# Orbital elements
a_km = T(6378.137) + T(1000.0)  # semi-major axis in km
e     = T(0.001)
i     = T(99.79)                # degrees
Ω     = T(58.6)
ω     = T(0.0)
ν     = T(0.0)
r, v = coe2rv(a_km * T(1e3), e, i, Ω, ω, ν, earth.mu)
```

Sun Synchronous Orbit

RAAN change earth = RAAN change sun. In simple terms we want the orbital plane to have a fixed relative orientation with respect to that of the "sun-rays"

In simpler terms when you are in orbit and you pass a certain fixed point in that orbital path, you guaranteed to see the same illumination conditions. "The sun rises at the same orbital time each cycle".

Such consistent illumination conditions allows for

- easier to spot trends, interpret data
- find/validate crucial scaling laws(analytical equations that govern desirable spacecraft dynamics)

So how to maintain SSO? How is this related to initial conditions?

Earth is not a perfect sphere. Due to oblateness effects, the orbital plane perturbs by a certain amount known as J2 which is a function of semi-major axis(a) and inclination(i). Essentially to maintain SSO at a certain altitude certain inclination values must be set.

Next we ensure the thrust vector which is actually the sail normal i.e the normal to our reflective surface propelled by solar radiation lies within the orbital plane. If not the inclination of the plane will change and SSO can't be maintained.

Maximum semi-major axis(SMA) increase

First we want SSO for good data basically. Next we want to maximize SMA. It turns out that the semi-major axis gain is reflective of the amount of energy gained by the sail. That is the whole point of the sail, we want to **increase orbital energy** of our sail through **solar radiation pressure**

Once we can verify this with our earth LEO test, we take on the "full power of the sun" by performing special kinds of heliocentric orbits that allow us to gain enough energy to leave the solar system as desired!

How to maximize SMA then?

Beyond structural parameters and recall we can't change certain initial conditions like inclination and have face certain limitations like keep our thrust vector within the orbital plane.

We instead try to find the best initial pitch angle(determines the **orientation/attitude**) of the sail as well as the **RAAN**(the orbital position in which the sail rises above the reference plane).

There are many ways to do so including the use of optimization frameworks like **genetic algorithms**. But for the sake of simplicity we present the most straightforward of them all...a good old GPU toasting brute force session. Before that I briefly explain the last few parts of the code

```
q0 = fixed_in_plane_attitude(r, v; angle_deg = PITCH)
ω0 = SVector{3,T}(T(0), T(0), RATE)
```

- `fixed_in_plane_attitude`

> - **Orbital Plane**: The plane defined by the spacecraft's position vector (r) and velocity vector (v). The angular momentum vector (h = r × v) is perpendicular to this plane.
> - **angle_deg**: The attitude is rotated by a specified angle (default: 45 degrees) from the radial direction (r̂) toward the tangential direction (perpendicular to r̂ within the plane). This angle is in-plane, meaning it doesn't tilt out of the orbital plane.

- `angular_velocity_along_body_z`

Defines initial spin about the z axis which is also our sail normal. Recall we have a "spin stabilized sail". This then gives rise to the moments of inertia formulas as in comment section `#4` which model the sail as pair of cross braced rods aligned along the principle axis.

Finally we have defined our drag and reflective coefficients. Then as the functions `add_force` and `add_torque` suggests, it allows us to plug in our coefficients and our design parameters to the SRP, Albedo, Gravitational and drag models as discussed by Shiki last week, which then contribute to the net torques and forces that influence the evolution of the sails state vector(position,orientation,velocity etc) at each time step

in `#6` the propagator function argument `law` allows us to introduce active control laws that can actively manipulate the attitude of the sail. In our case we are too broke and hence do not have the means to do such a thing. Hence we are "passively stabilized".

the rest of the code is just functionality to plot our results

Earth Ensemble CPU/GPU

As promised we now go over the GPU accelerated brute force demonstration.

- multiple trajectories over a range of initial conditions
- 2D parameter sweep

Plan:

- Live demonstration of `earth_ensemble_gpu` . Show plots
- Then assist those still strugging with set-up or would like to challenge themselves to run the gpu code.