Assignment 4: Operational Maneuvers

Abstract

In this assignment the student will learn how to model a sensor cone and perform nadir tracking by a satellite to perform sensor acquisition.

1 Objective

The objective with this assignment is to gain insight into nadir pointing, how to model a sensor cone, and how to visualize it using Pyvista.

2 Nadir Pointing

Nadir is the direction directly below the satellite. The orbit frame is defined to have the $\mathbf{x}^o$ axis coincide with the radius vector, meaning that if you align the body frame with the orbit frame, the satellite is pointing away from the Earth, which typically is not very useful if the scientific instrument is mounted on the positive $\mathbf{x}^b$ axis. The desired quaternion relative to the orbit frame can be defined as $\mathbf{q}_{o,d} = \begin{bmatrix} \cos(\frac{\vartheta_{o,d}}{2}) & \mathbf{k}_{o,d}^\top \sin(\frac{\vartheta_{o,d}}{2}) \end{bmatrix}^\top$ where $\vartheta_{o,d}$ is the desired angle of rotation, around the unit vector $\mathbf{k}_{o,d}$.

Detumbling

1. Create the desired quaternion relative to the orbit frame $\mathbf{q}_{o,d}$ that points towards the Earth (along the negative $\mathbf{x}^o$ axis.)
2. What is the desired angular velocity $\boldsymbol{\omega}_{o,d}^d$ needed to remain aligned with the negative $\mathbf{x}^o$ axis?
3. Show that by using the proportional_derivative_attitude_controller() and this desired quaternion and angular velocity that you are able to point towards the Earth doing nadir tracking. Create an animation and take a few screen-shots for the report.

3 Sensor Cone

Any scientific instrument has a predefined cone angle that can be found as function of the field-of-view.

3.1 Finding Line-Sphere Intersections

As we will be moving the cone over the Earth, we would like parts of the cone that are inside the Earth to be removed. This can be achieved by using line-sphere intersection consider Figure 1.

A line can be parameterized as

$$\mathbf{p} = \mathbf{p}_1 + u(\mathbf{p}_2 - \mathbf{p}_1)$$

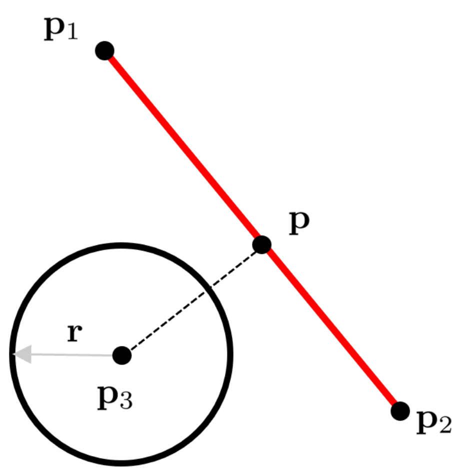

Figure 1: Line-sphere intersection diagram.

and on component form

$$\begin{aligned} x &= x_1 + u(x_2 - x_1) \\ y &= y_1 + u(y_2 - y_1) \\ z &= z_1 + u(z_2 - z_1). \end{aligned}$$

A sphere located at point $\mathbf{p}_3(x_3, y_3, z_3)$ can be defined as

$$(x - x_3)^2 + (y - y_3)^2 + (z - z_3)^2 = r^2$$

By inserting the line equation into sphere equation and after some manipulation it can be shown that (Bourke, 1992) it is possible to write this as a second order equation

$$au^2 + bu + c = 0,$$

where

$$\begin{aligned} a &= (x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2 \\ b &= 2[(x_2 - x_1)(x_1 - x_3) + (y_2 - y_1)(y_1 - y_3) + (z_2 - z_1)(z_1 - z_3)] \\ c &= x_3^2 + y_3^2 + z_3^2 + x_1^2 + y_1^2 + z_1^2 - 2[x_3x_1 + y_3y_1 + z_3z_1] - r^2 \end{aligned}$$

which can be solved in terms of intersection points using

$$u_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Where the line intersection can be determined by $b^2 - 4ac$:

- If $b^2 - 4ac < 0$ it does not intersect
- If $b^2 - 4ac = 0$ it intersects at one point
- If $b^2 - 4ac > 0$ it intersects at two points.

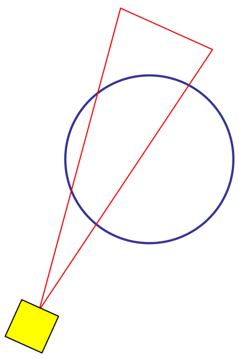

Figure 2: Raycasting using cone sphere interactions

Note that there are existing code implementations available at Paul Bourke's webpage, which can serve as inspirations for an implementation at: `https://paulbourke.net/geometry/circlesphere/index.html#linesphere`

Line-Sphere Intersection

1. Create a new file called `sensors.py`. Inside it, create a new function called def calculate_line_sphere_intersection_point(p1, p2, p3, radius=6378e3): where point p3 is the center of the Earth set at $\mathbf{p}_3 = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^\top$. The function must take basis in the above equations and return:
   - The closest intersection point (if there are two)
   - A boolean if there is an intersection or not
2. Create a line comprised by two points, p1 and p2, and a sphere that is in between. Show that you are able to detect the intersection.

3.2 Raycasting

One way to create the sensor cone is through raycasting, where a number of lines are projected from the satellite in the body frame, and by comparing the lines with the sphere to detect intersections, the closest points can be selected, such that the resulting line points will lie on the surface of the Earth, which then allows for building the sensor cone. Figure 2 shows how the lines will intersect the Earth, where the objective is to identify intersections, while Figure 3 shows the relationships between the field of view angle, the raycasting length and the radius of the cone.

The raycasted points can be constructed as a circle a distance raycasting_length along the $\mathbf{x}^b$ axis. This can be achieved through the following function which returns the raycasted points in the inertial frame.

```
def calculate_raycasting_points(R_i_b, r_i, raycasting_length
    =10000e3, field_of_view_half=30/2*np.pi/180,
    number_of_raycasting_points=10):
```

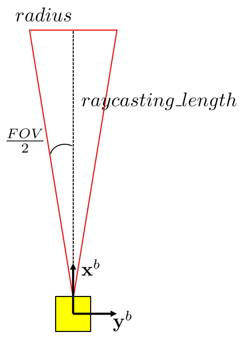

Figure 3: Showing relationship between field of view angle, raycasting length and radius of the cone.

```
    # Initializes the raycasting points
    raycasting_points_i = []
    radius = raycasting_length*np.tan(field_of_view_half)
    # Generating a number of points in a sphere at a point
    along the x^b axis
    for theta in np.linspace(0, 2*np.pi,
    number_of_raycasting_points, endpoint=False):
        points_i = R_i_b @ np.array([raycasting_length,
        radius*np.cos(theta),
        radius*np.sin(theta)])
        raycasting_points_i.append(r_i + points_i)

    raycasting_points_i = np.array(raycasting_points_i)

    return raycasting_points_i
```

Based on the raycasting points and the intersections, it is possible to find the intersection points in to either be at the intersection, or equal to the raycasting points (in case of no intersection). This can be achieved through the following function:

```
def calculate_intersection_points_in_inertial_frame(r_i, R_i_b,
     raycasting_length, field_of_view_half_deg,
    number_of_raycasting_points):

    fov_half_rad = field_of_view_half_deg*np.pi/180
    raycasting_points_i = calculate_raycasting_points(R_i_b,
    r_i, raycasting_length=raycasting_length, field_of_view_half
    =fov_half_rad, number_of_raycasting_points=
    number_of_raycasting_points)

    # Satellite and Earth are fixed
    p1 = r_i
    p3 = np.array([0, 0, 0]) # Earth is at the origin

    intersection_points_i = np.zeros_like(raycasting_points_i)

    # Looping through the different raycasting points
    for i in range(0, len(raycasting_points_i[:,0])):
        p2 = raycasting_points_i[i]
        closest_intersection_point, is_line_intersecting =
    calculate_line_sphere_intersection_point(p1, p2, p3, radius
    =6378e3)
        if is_line_intersecting == True:
            intersection_points_i[i] =
    closest_intersection_point
        else:
            intersection_points_i[i] = raycasting_points_i[i]

    return intersection_points_i
```

Raycasting

1. Implement the function calculate_raycasting_points()
2. Implement the function calculate_intersection_points_in_inertial_frame()
3. Use Matplotlib to plot the resulting points and satellite

You should now have all functions needed to apply the results that can be done through the visualization script.

3.3 Visualization

To visualize the sensor cone, We can create the following function to create the sensor cone using triangles using two intersection points and the position of the satellite. By looping through all intersection points, the complete cone can be constructed.

```
def create_sensor_cone(plotter, r_i, R_i_b, raycasting_length,
    field_of_view_half_deg, number_of_raycasting_points):

    # Finding the intersection points
    intersection_points_i =
    calculate_intersection_points_in_inertial_frame(r_i, R_i_b,
    raycasting_length, field_of_view_half_deg,
    number_of_raycasting_points)

    # Create triangles for the cone
    triangles = []
    for i in range(len(intersection_points_i[:,0]) - 1):
        triangles.append([r_i, intersection_points_i[i+1],
    intersection_points_i[i]])

    # Close the cone by connecting the last point to the first
    triangles.append([r_i, intersection_points_i[0],
    intersection_points_i[-1]])

    # Create the mesh
    cone_mesh = pv.PolyData()
    for tri in triangles:
        cone_mesh += pv.Triangle(tri)
    plotter.add_mesh(cone_mesh, opacity=0.25, color="green")

    return cone_mesh
```

If we create the triangles at each time-step the simulation gets really slow. To that end, the objective is instead to update the sensor cone points dynamically over time, something that can be achieved through the following function.

```
def update_sensor_cone_points(plotter, line_actor,
    sensor_cone_mesh, r_i, R_i_b, raycasting_length,
    field_of_view_half_deg, number_of_raycasting_points):

    #########################
    # Updating Cone Triangles
    #########################

    # Recalculates the intersection points
    intersection_points_i =
    calculate_intersection_points_in_inertial_frame(r_i, R_i_b,
    raycasting_length, field_of_view_half_deg,
    number_of_raycasting_points)

    # Combine origin with intersection points to form new
    vertex positions
    new_points = np.vstack([r_i, intersection_points_i])

    # Update the mesh points
    sensor_cone_mesh.points = new_points

    #########################
    # Update the circle lines
    #########################
    circle_lines = []
    for i in range(len(intersection_points_i) - 1):
        circle_lines.append([intersection_points_i[i],
    intersection_points_i[i + 1]])
        circle_lines.append([intersection_points_i[-1],
    intersection_points_i[0]])  # Close the circle

    # Convert lines to a NumPy array
    line_points = np.array(circle_lines).reshape(-1, 3)

    # Remove the old lines if line_actor exists
    if line_actor is not None:
        plotter.remove_actor(line_actor)

    line_actor = plotter.add_lines(line_points, color="green",
    width=2)

    return line_actor
```

A first step before running a complete animation, is to simply add the code inside the visualize_scene() function:

```
# Sensor Cone
raycasting_length = 10000e3
field_of_view_half_deg = 15/2
number_of_raycasting_points = 100
R_i_b = pyvista_rotation_matrix_from_euler_angles(Theta_ib)
sensor_cone_mesh = create_sensor_cone(plotter, r_i, R_i_b,
    raycasting_length=raycasting_length, field_of_view_half_deg=
    field_of_view_half_deg, number_of_raycasting_points=
    number_of_raycasting_points)

line_actor = plotter.add_lines(np.empty((0, 3)), color="green",
     width=2)  # Initialize with an empty actor
line_actor = update_sensor_cone_points(plotter, line_actor,
    sensor_cone_mesh, r_i, R_i_b, raycasting_length,
    field_of_view_half_deg, number_of_raycasting_points)
```

While to create an animation, the animate_satellite() function can be updated as

```

#...old code


# Sensor Cone
raycasting_length = 10000e3
field_of_view_half_deg = 15/2
number_of_raycasting_points = 100
R_i_b = pyvista_rotation_matrix_from_euler_angles(Theta_ib)
sensor_cone_mesh = create_sensor_cone(plotter, r_i, R_i_b,
    raycasting_length=raycasting_length, field_of_view_half_deg=
    field_of_view_half_deg, number_of_raycasting_points=
    number_of_raycasting_points)

line_actor = plotter.add_lines(np.empty((0, 3)), color="green",
     width=2)  # Initialize with an empty actor

for i in range(len(t)):
    time = t[i]
    r_i = r_i_array[i]

    # other code

    line_actor = update_sensor_cone_points(plotter, line_actor,
     sensor_cone_mesh, r_i, R_i_b, raycasting_length,
    field_of_view_half_deg, number_of_raycasting_points)

    plotter.write_frame()
```

Visualization

1. Implement the following functions:
    - create_sensor_cone()
    - update_sensor_cone_points()
2. Show that you can create the sensor cone inside the visualize_scene() function
3. Show that you are able to animate the sensor cone during operations.
4. Document your results with pictures and explainations of what you have done and achieved.