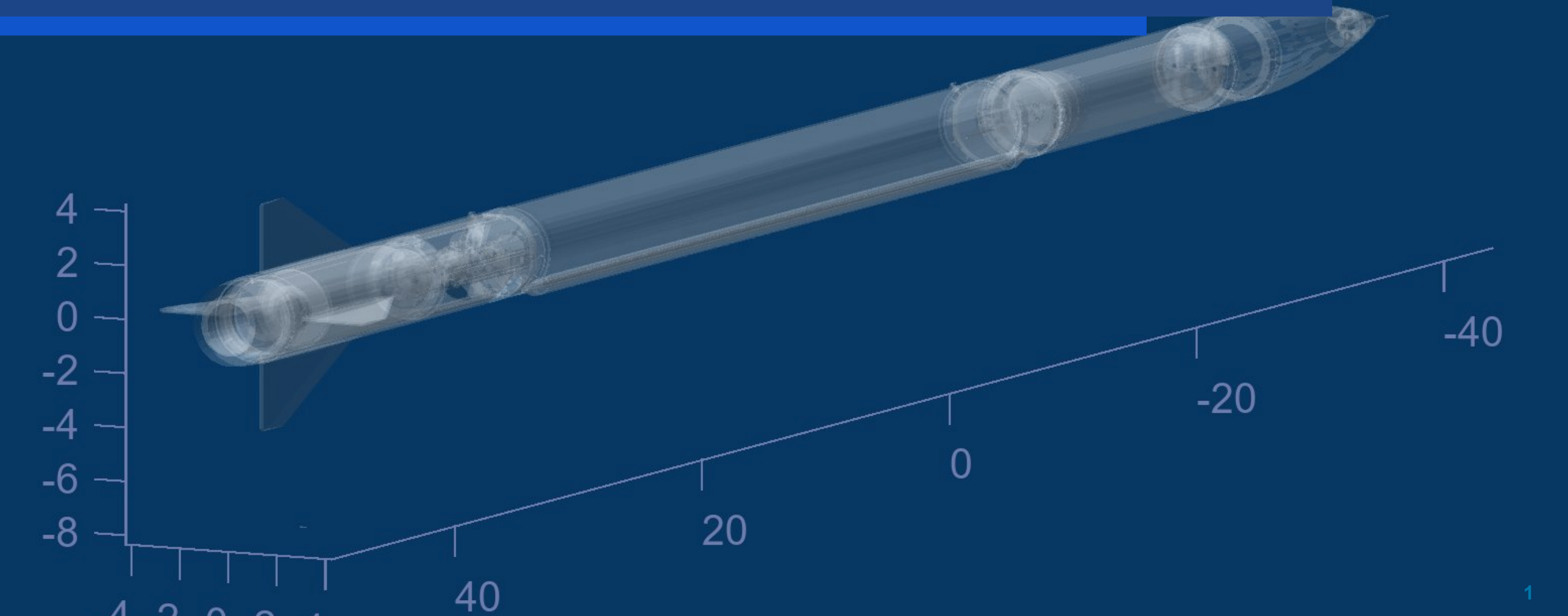
Rocketry simulation in MATLAB

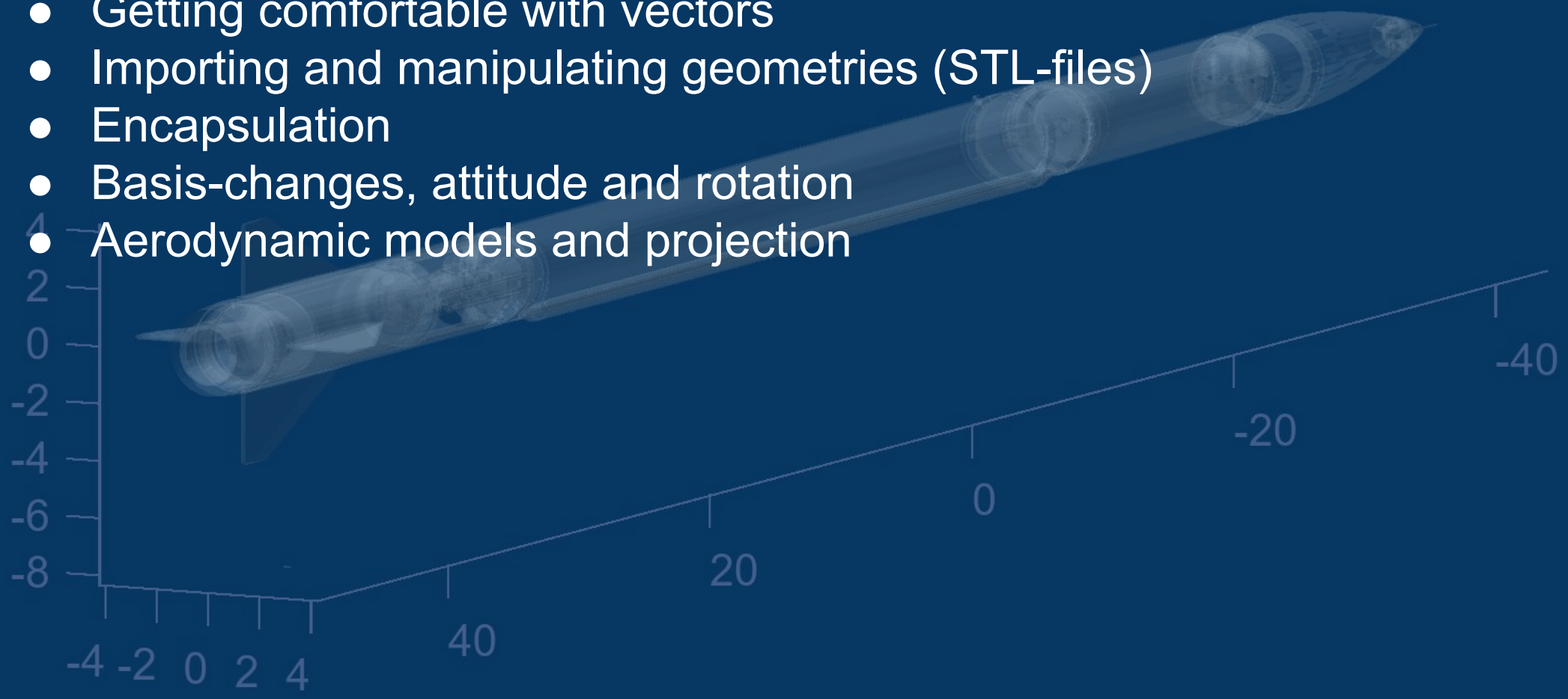
How to simulate 3D-bodies using linear algebra



What goes into making a simulation from scratch?

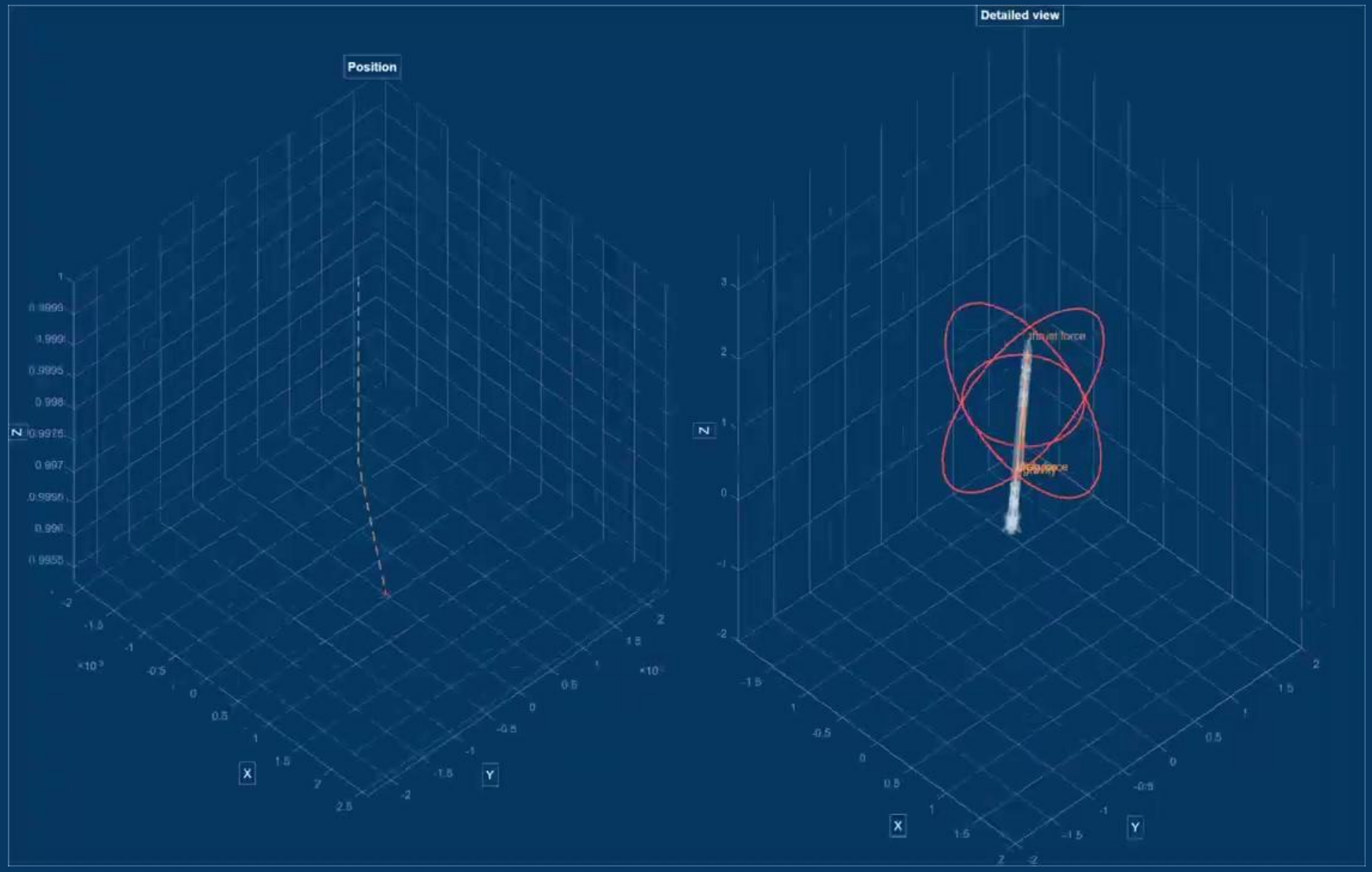
How do you cook a rocket-stew?

- Getting comfortable with vectors
- Importing and manipulating geometries (STL-files)
- Encapsulation
- Basis-changes, attitude and rotation
- Aerodynamic models and projection



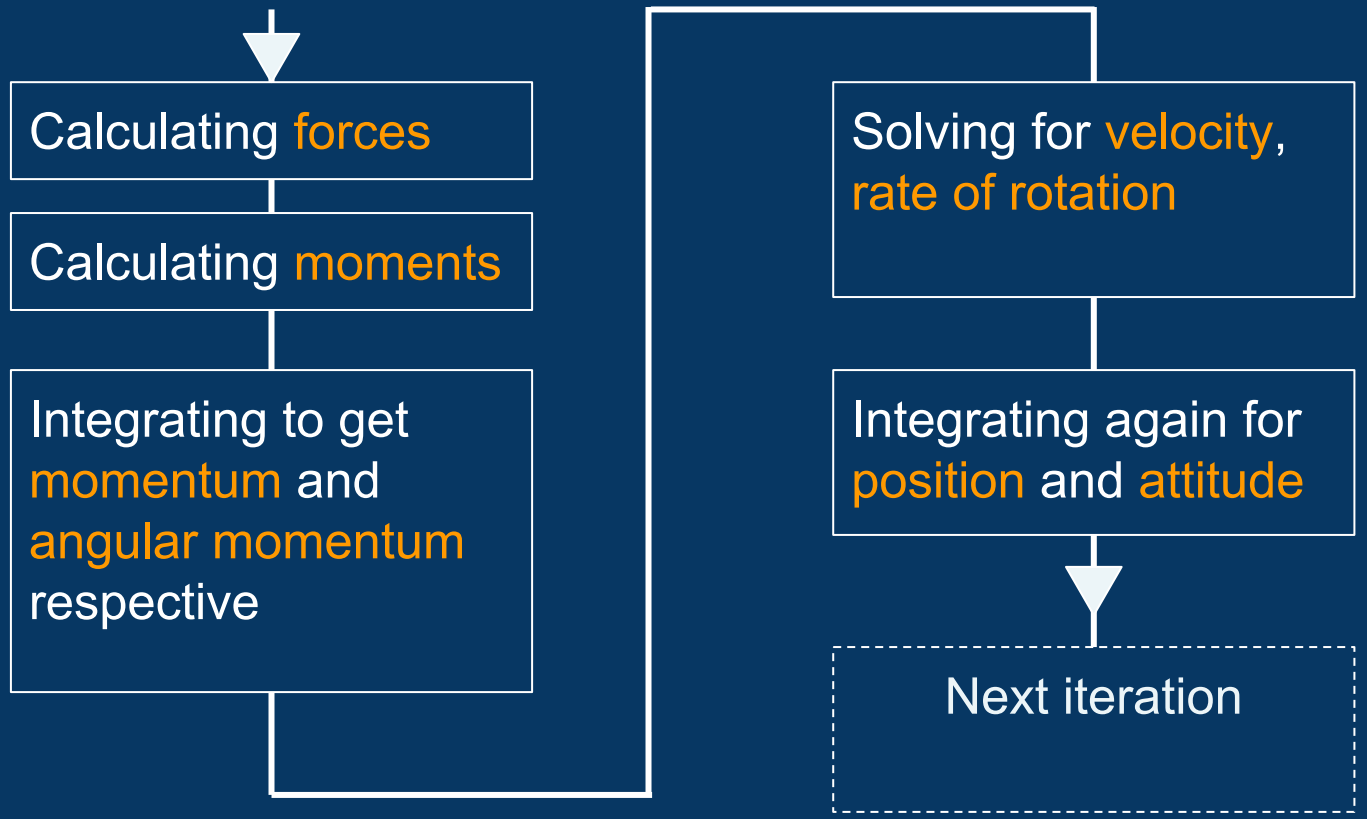
What goes into making a simulation from scratch?

How do you cook a rocket-stew?



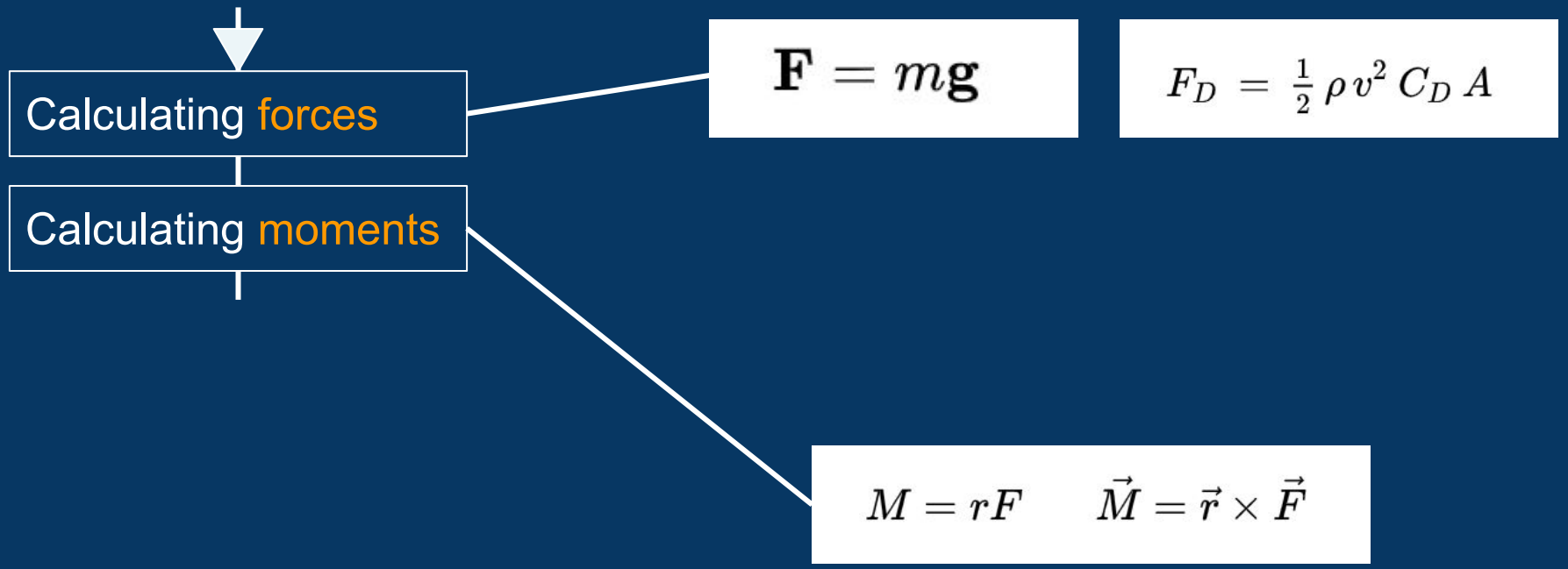
What we're building up to.

The physics underlying what we'll do today:



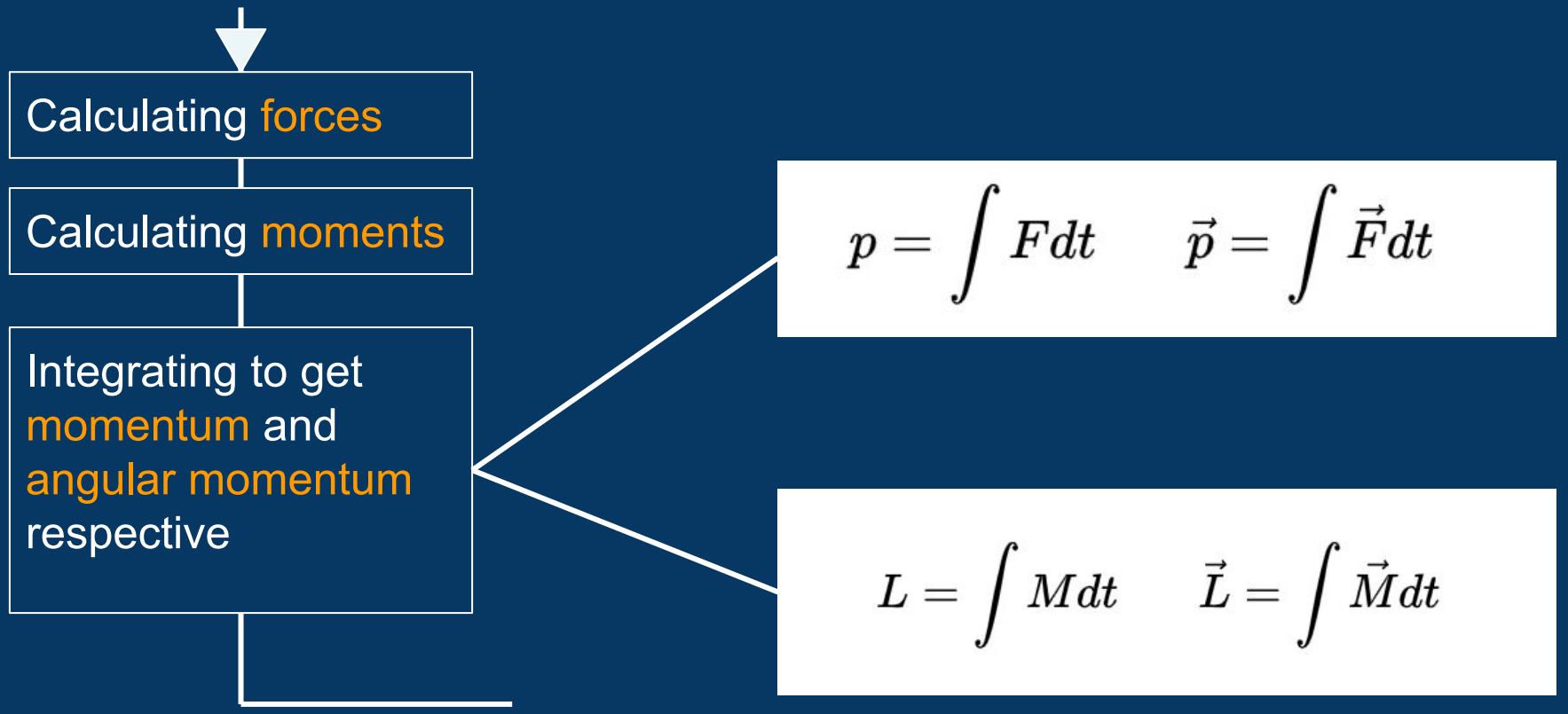
When in doubt, Wikipedia! - Galileo probably

The physics underlying what we'll do today:



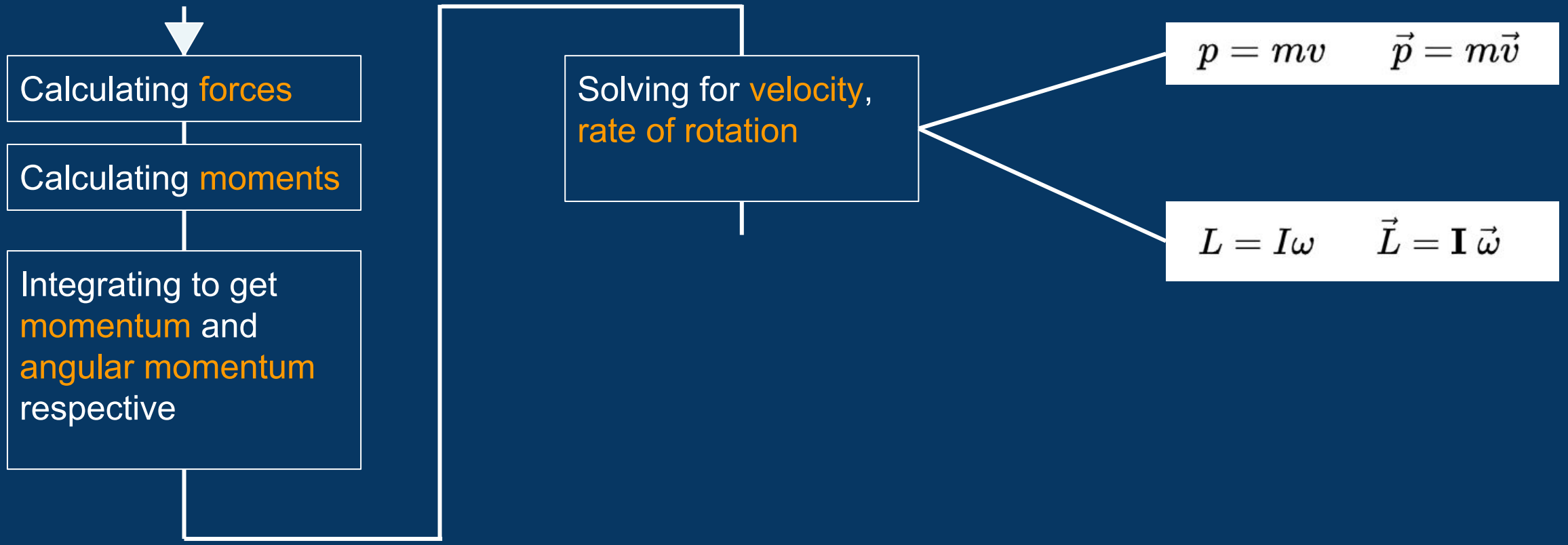
When in doubt, Wikipedia! - Galileo probably

The physics underlying what we'll do today:



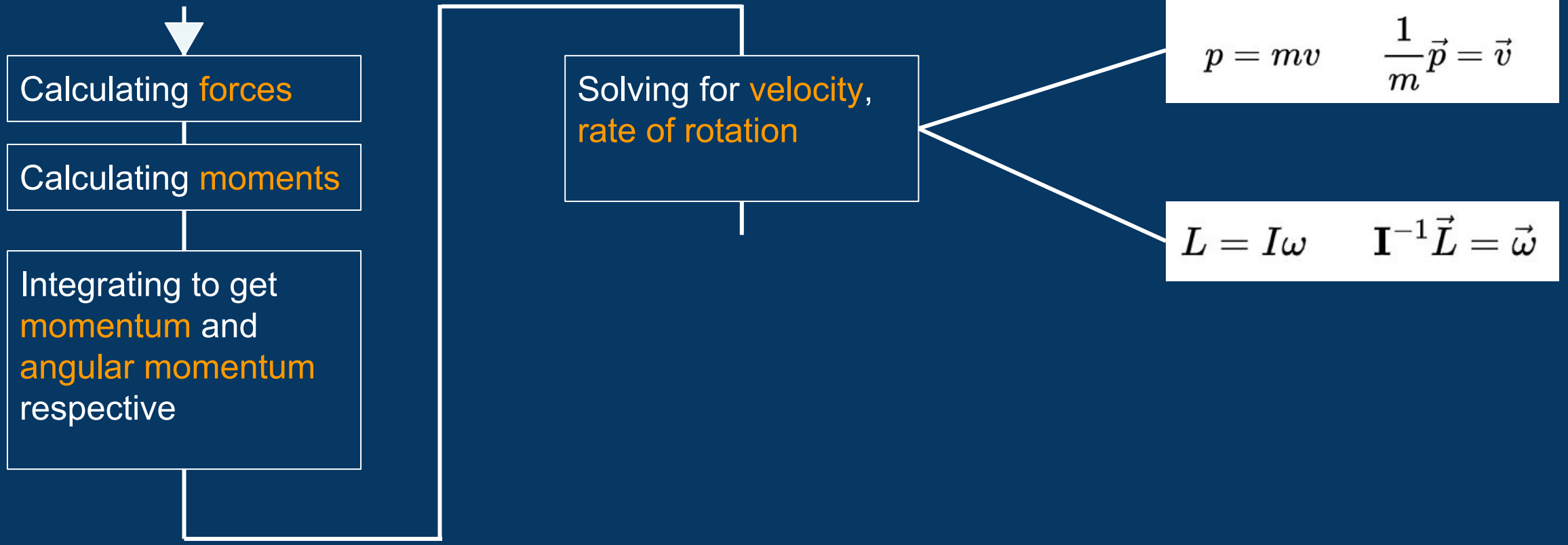
When in doubt, Wikipedia! - Galileo probably

The physics underlying what we'll do today:



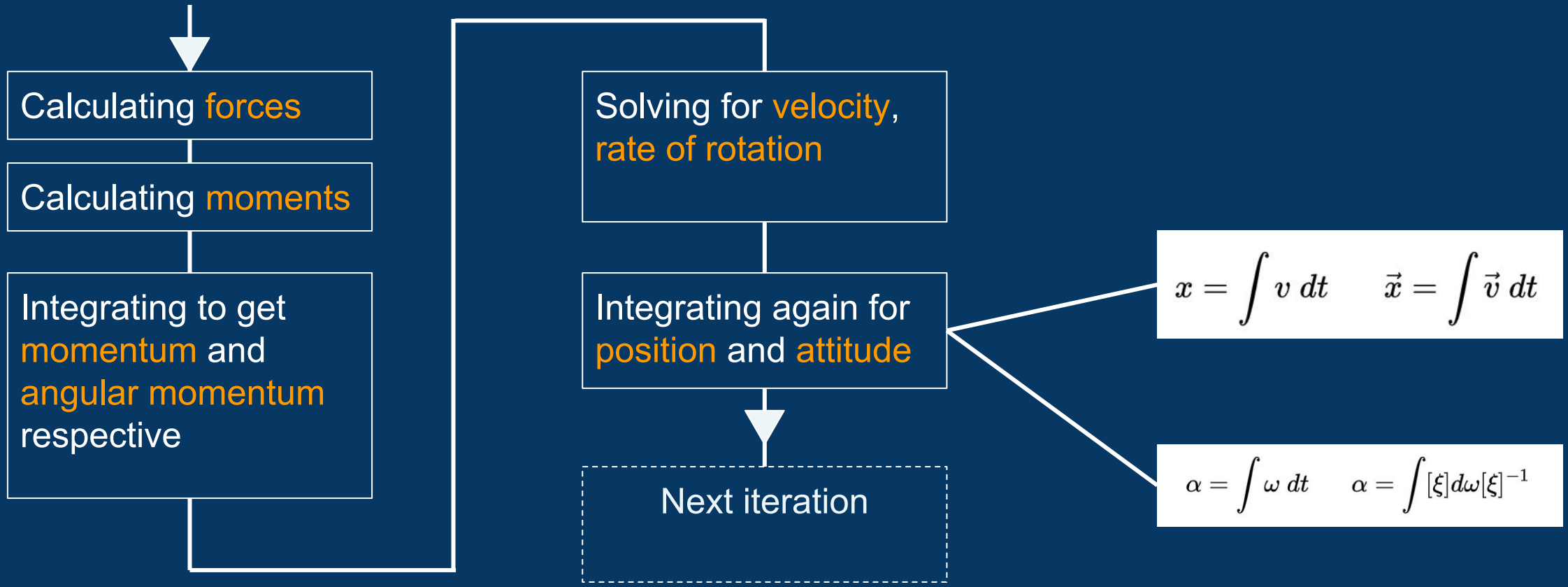
When in doubt, Wikipedia! - Galileo probably

The physics underlying what we'll do today:



When in doubt, Wikipedia! - Galileo probably

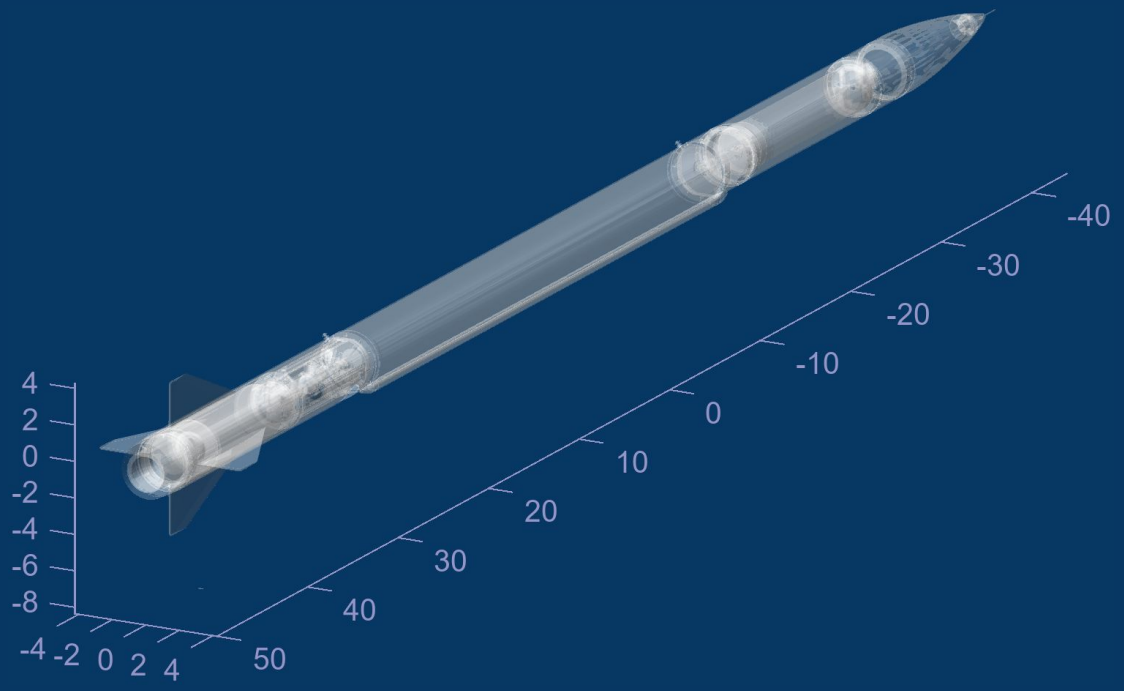
The physics underlying what we'll do today:



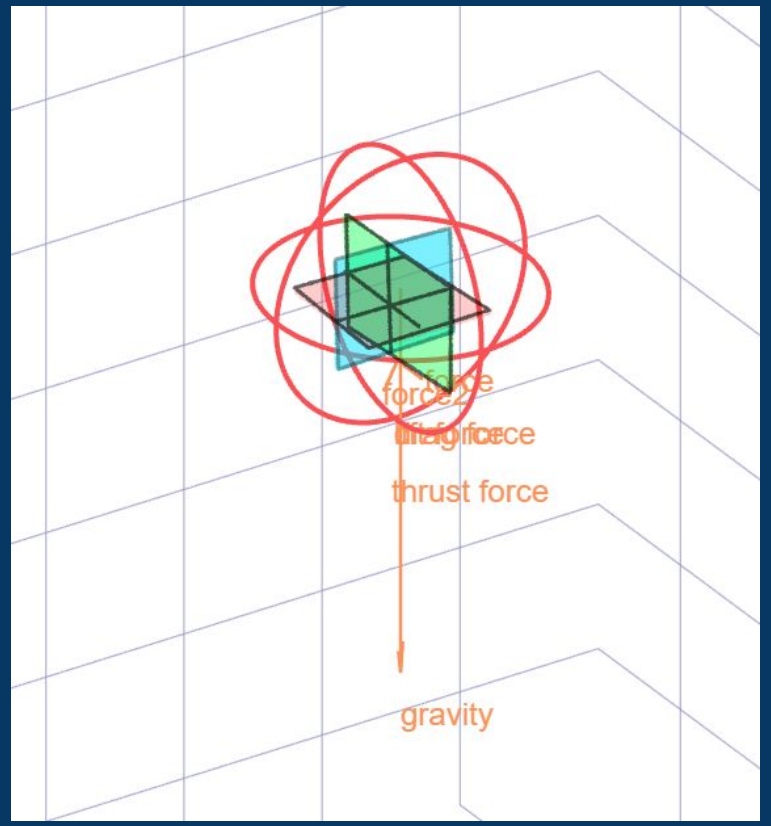
When in doubt, Wikipedia! - Galileo probably

That’s the plan!

Before we make this for real though, we’re gonna need some tools.



How a rocket looks in your head



How a rocket looks to the simulation

Recipe for a rocket:

Rockets vector-valued properties:

- Center of mass
- Center of pressure
- Area
- …

Force and moment-vectors acting on the rocket:

- Lift-vector
- Drag-vector
- Gravity
- Thrust

```
mass                    = 100;
center_of_mass          = [0;0;0.5];
center_of_pressure      = [0;0;-0.5];

area                    = [0.1;0.1;0.1^2];

friction_coefficient    = [1;1;1]*0.2;

forces("gravity")       = force(g*mass*[0;0;-1], center_of_mass);
moments("moment 1")     = moment([10;0;-1], center_of_mass);

position                = [0;0;1];
velocity                = [1;1;1 ];

attitude                = eye(3);
dimensions              = [1;1;1];
angular_momentum        = zeros(3,1);
rotation_rate           = zeros(3,1);
pressure_coefficient    = eye(3)*0.2;
friction_coefficient    = ones(3,1)*0.2;
```

Recipe for a rocket:

Rockets vector-valued properties:

- Center of mass
- Center of pressure
- Area
- …

Force and moment-vectors acting on the rocket:

- Lift-vector
- Drag-vector
- Gravity
- Thrust

```
mass                    = 100;
center_of_mass          = [0;0;0.5];
center_of_pressure      = [0;0;-0.5];

area                    = [0.1;0.1;0.1^2];

friction_coefficient    = [1;1;1]*0.2;

forces("gravity")       = force(g*mass*[0;0;-1], center_of_mass);
moments("moment 1")     = moment([10;0;-1], center_of_mass);

position                = [0;0;1];
velocity                = [1;1;1 ];

attitude                = eye(3);
dimensions              = [1;1;1];
angular_momentum        = zeros(3,1);
rotation_rate           = zeros(3,1);
pressure_coefficient    = eye(3)*0.2;
friction_coefficient    = ones(3,1)*0.2;
```

```
apply_aerodynamics(mass, center_of_mass, center_of_pressure, area, ...
                   friction_coefficient, forces, moments, position, ...
                   velocity, dimensions, angular_momentum, ...
                   rotation_rate, pressure_coefficient, ...
                   friction_coefficient)
```

Recipe for a rocket:

How to package your variables to avoid your rocket turning into a soup

```
classdef rocket
properties
mass                        = 100;
center_of_mass              = [0;0;0.5];
center_of_pressure          = [0;0;-0.5];

area                        = [0.1;0.1;0.1^2];

friction_coefficient        = [1;1;1]*0.2;

position                    = [0;0;1];
velocity                    = [1;1;1 ];

 ....
end



methods

function self = apply_aerodynamics(self)
    .... some stuff
    ... some other stuff....
end

end

end
```

classes

- Object-orientation standard for programming
- Allows you to give your data “responsibility” over the things you can do to it

```
my_rocket = rocket();


my_rocket.area = ...;

my_rocket.apply_aerodynamics();
```

Recipe for a rocket:

How to package your variables to avoid your rocket turning into a soup

```
rocket = struct();

rocket.mass                     = 100;
rocket.center_of_mass           = [0;0;0.5];
rocket.center_of_pressure       = [0;0;-0.5];

rocket.area                     = [0.1;0.1;0.1^2];

rocket.friction_coefficient     = [1;1;1]*0.2;

rocket.position                 = [0;0;1];
rocket.velocity                 = [1;1;1 ];

 ....
```

```
apply_aerodynamics(rocket)
```

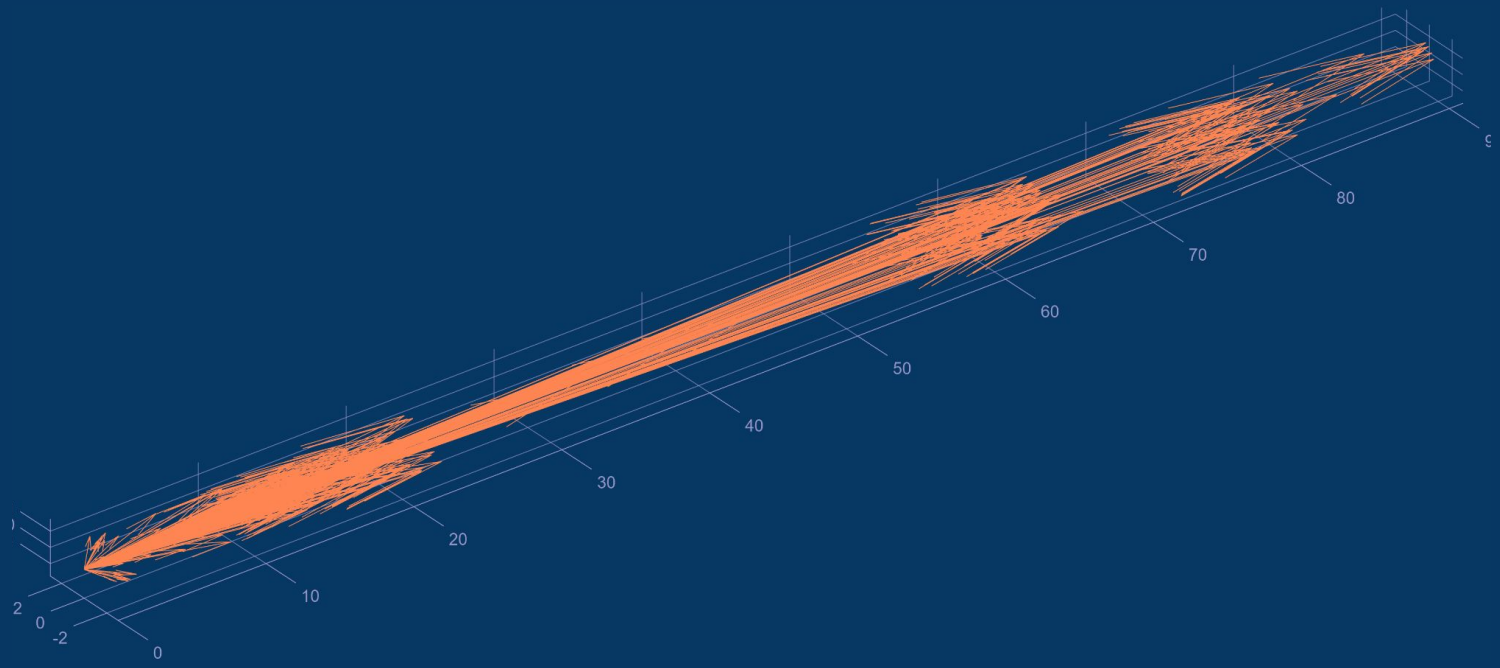
struct

- A dumpster-fire of random variables.
- Like a class, except you can add and remove variables as you feel like it.
- Less syntax! Less cumbersome to work with.

Working with vectors

Concepts that you're probably familiar with, but that we will now apply

```
ax = axes();
my_mesh = stlread("AM_00 Mjollnir Full CAD v79 low_poly.stl");


origo = 0*(1:200:height(new_mesh.Points))';
quiver3(ax, origo,                          origo,                          origo, ...
            new_mesh.Points(1:200:end,3),   new_mesh.Points(1:200:end,2), new_mesh.Points(1:200:end,1))

axis("equal")
```

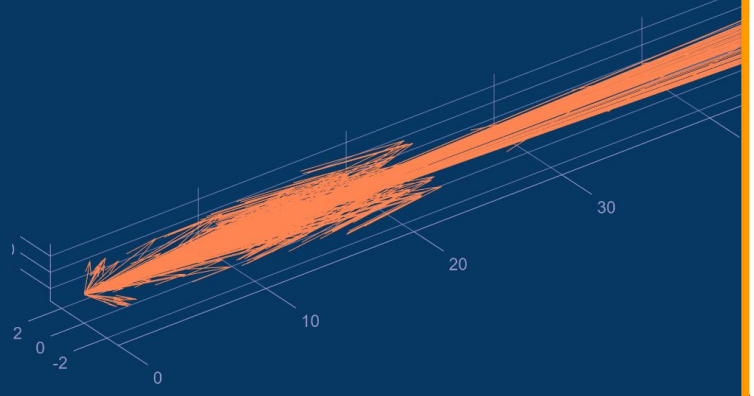


- Vector-differences
- Projection
- Cross-products
- Matrix multiplication
- Base-changes

Working with vectors

Concepts that you're probably familiar with, but that we will now apply

```
ax = axes();
my_mesh = stlread("AM_00 Mjollnir Full CAD v79 low_poly.stl");


origo = 0*(1:200:height(new_mesh.Points))';
quiver3(ax, origo,                          origo,
            new_mesh.Points(1:200:end,3),   new_mesh.Poi

axis("equal")
```



Add-On Explorer

Contribute | Manage Add-Ons

Search for add-ons

Installed

STL File Reader

Version 1.2.0.0 (1.6 MB) by Eric Johnson

STLREAD imports geometry from a binary stereolithography (STL) file into MATLAB.

Function

(75)

37.2K Downloads

Updated 20 Jul 2011

View License

Open Folder | Manage

Overview | Functions | Examples | Version History | Reviews (75) | Discussions (21)

In addition to the STLREAD import function, this submission also includes a small demo that loads an STL model of a human femur bone.

FV = STLREAD(FILENAME) imports triangular faces from the binary STL file indicated by FILENAME, and returns the patch struct FV, with fields 'faces' and 'vertices'.

[F,V] = STLREAD(FILENAME) returns the faces F and vertices V separately.

[F,V,N] = STLREAD(FILENAME) also returns the face normal vectors.

The faces and vertices are arranged in the format used by the PATCH plot object.

Cite As

Eric Johnson (2023). STL File Reader (https://www.mathworks.com/matlabcentral/fileexchange/22409-stl-file-reader), MATLAB Central File Exchange. Retrieved October 14, 2023.

MATLAB Release Compatibility

Created with R2008b
Compatible with any release

Platform Compatibility

☑ Windows ☑ macOS ☑ Linux

Categories

MATLAB > Data Import and Analysis

Physical Modeling > Simscape Multibody > Model Import

MATLAB > Data Import and Analysis > Data Import and Export > Standard File Formats > STL (STereoLithography)

Tags Add Tags

cad | data export | data import | faces

erences

ucts

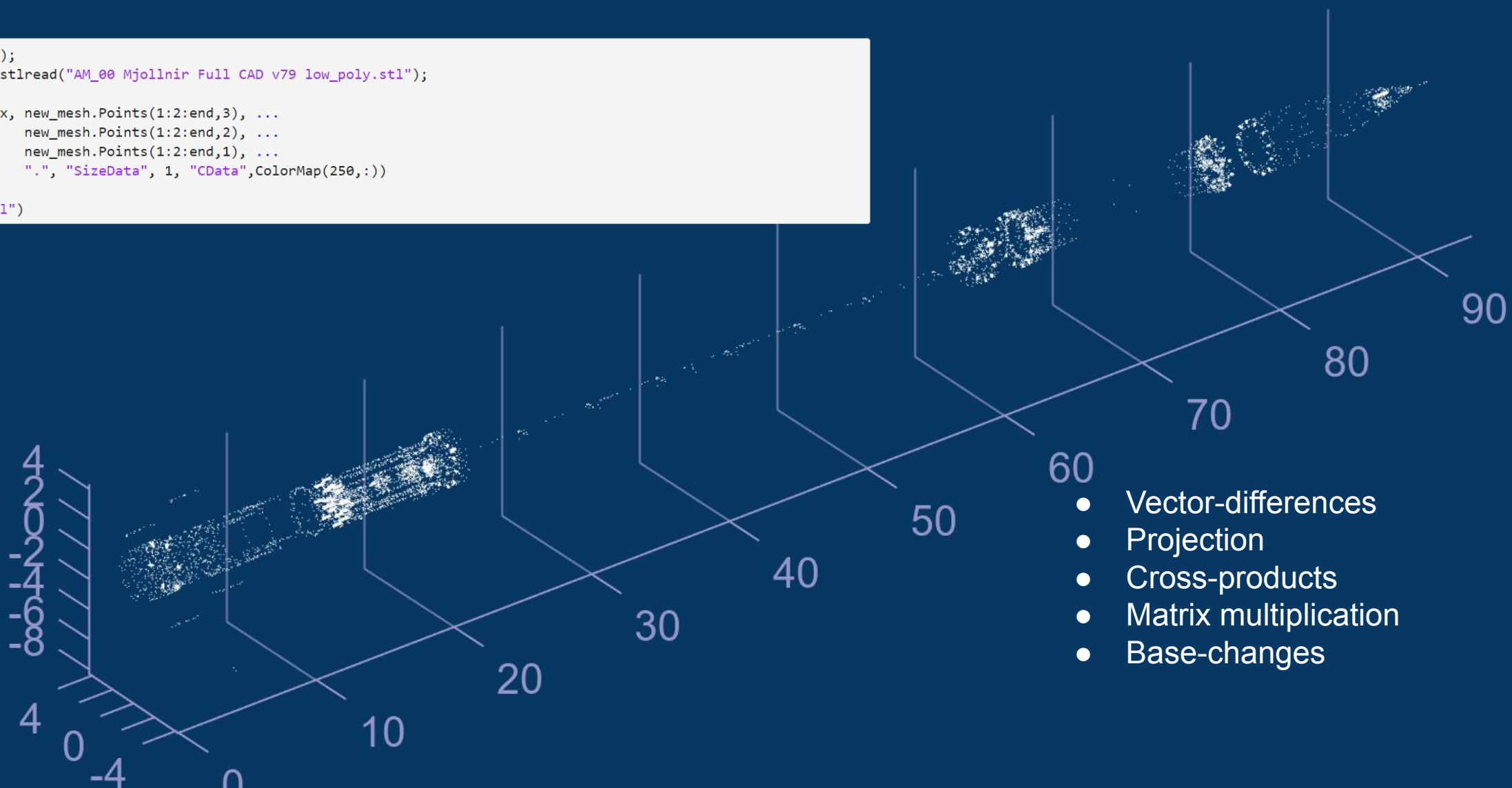
iplication

ges

Working with vectors

Concepts that you're probably familiar with, but that we will now apply

```
ax = axes();
my_mesh = stlread("AM_00 Mjollnir Full CAD v79 low_poly.stl");

scatter3(ax, new_mesh.Points(1:2:end,3), ...
             new_mesh.Points(1:2:end,2), ...
             new_mesh.Points(1:2:end,1), ...
             ".", "SizeData", 1, "CData",ColorMap(250,:))

axis("equal")
```

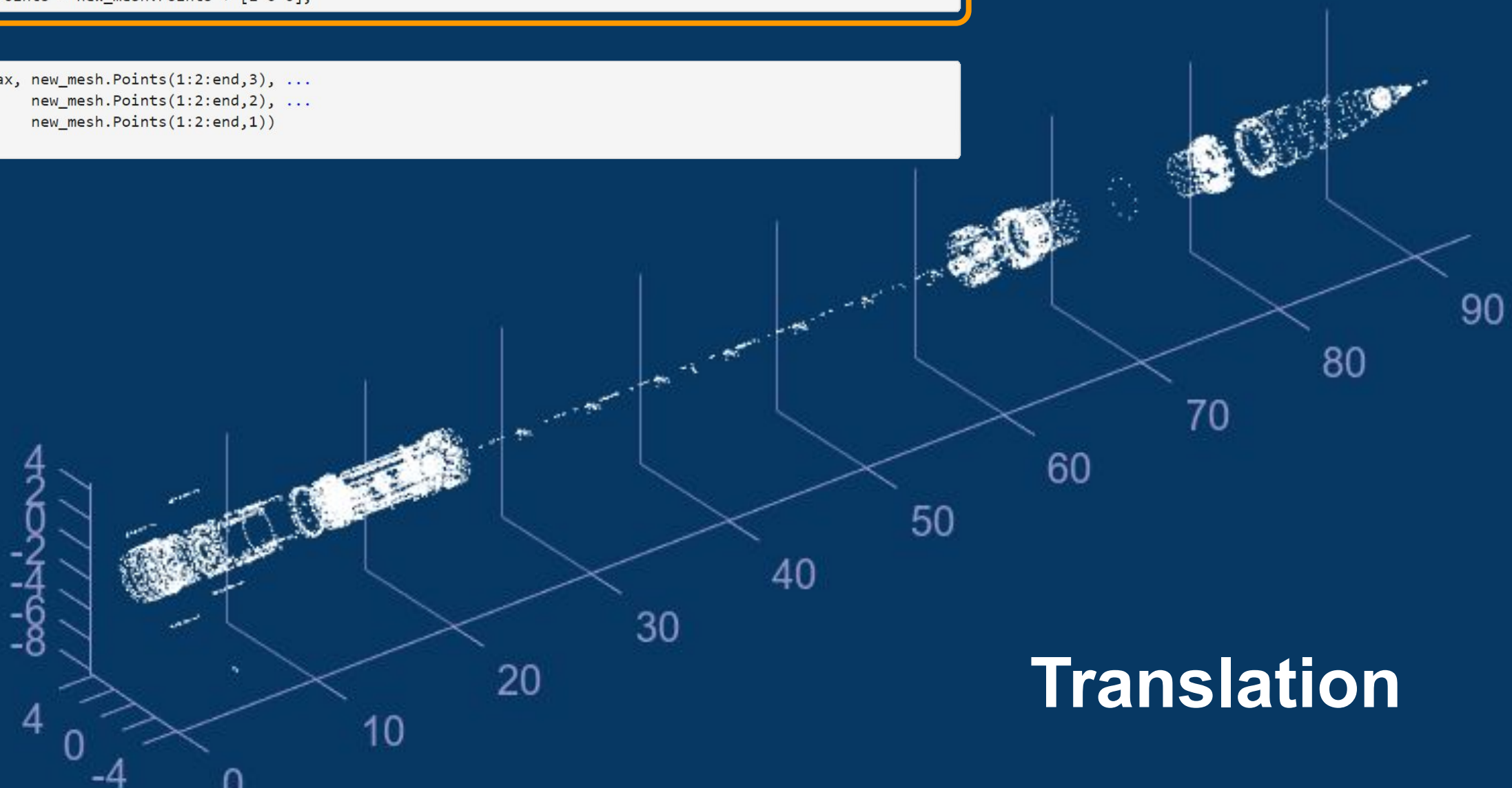


- Vector-differences
- Projection
- Cross-products
- Matrix multiplication
- Base-changes

```
ax = axes();
my_mesh = stlread("AM_00 Mjollnir Full CAD v79 low_poly.stl");
```

```
new_mesh = my_mesh;
new_mesh.Points = new_mesh.Points + [1 0 0];
```

```
scatter3(ax, new_mesh.Points(1:2:end,3), ...
             new_mesh.Points(1:2:end,2), ...
             new_mesh.Points(1:2:end,1))
```

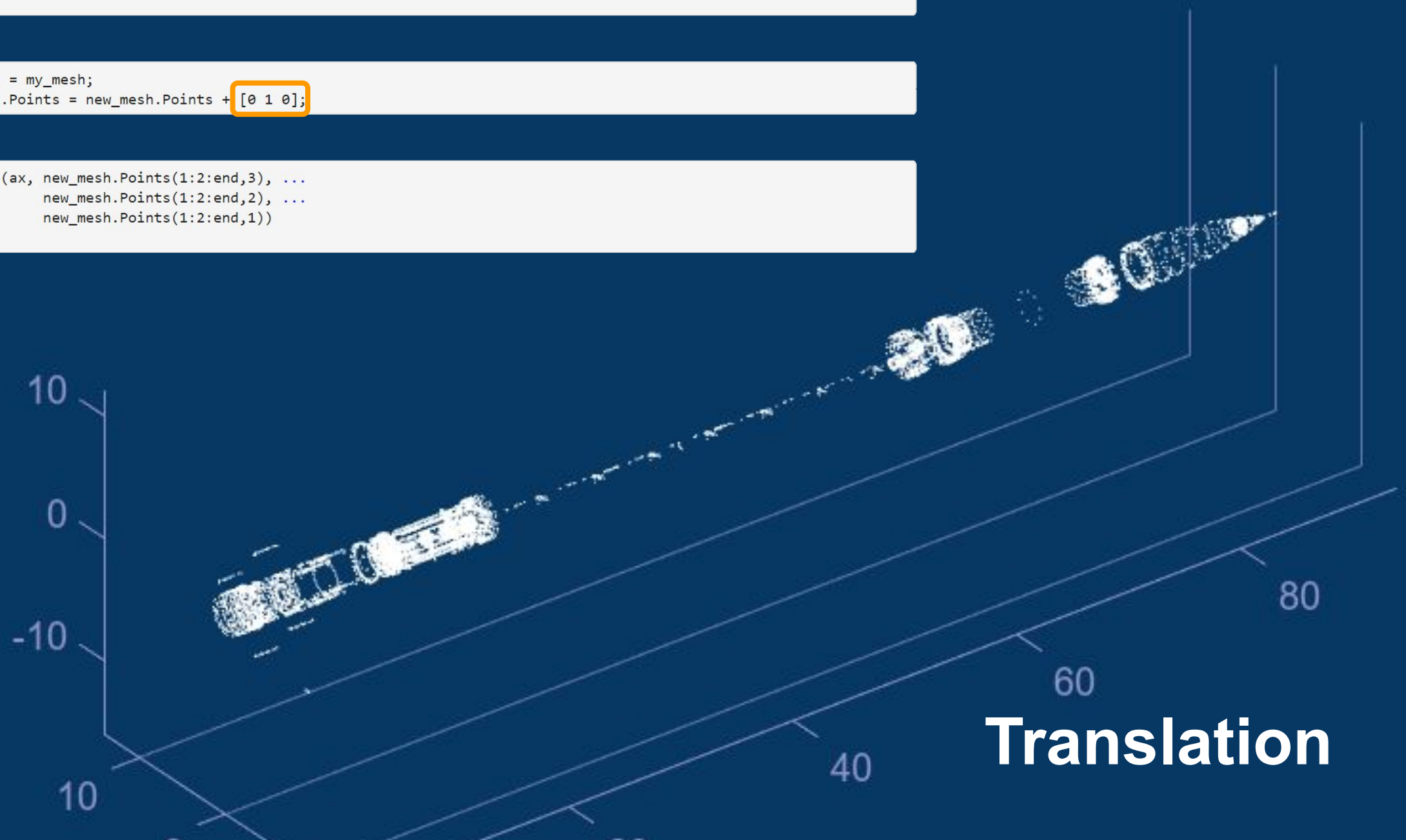


Translation

```
ax = axes();
my_mesh = stlread("AM_00 Mjollnir Full CAD v79 low_poly.stl");
```

```
new_mesh = my_mesh;
new_mesh.Points = new_mesh.Points + [0 1 0];
```

```
scatter3(ax, new_mesh.Points(1:2:end,3), ...
             new_mesh.Points(1:2:end,2), ...
             new_mesh.Points(1:2:end,1))
```

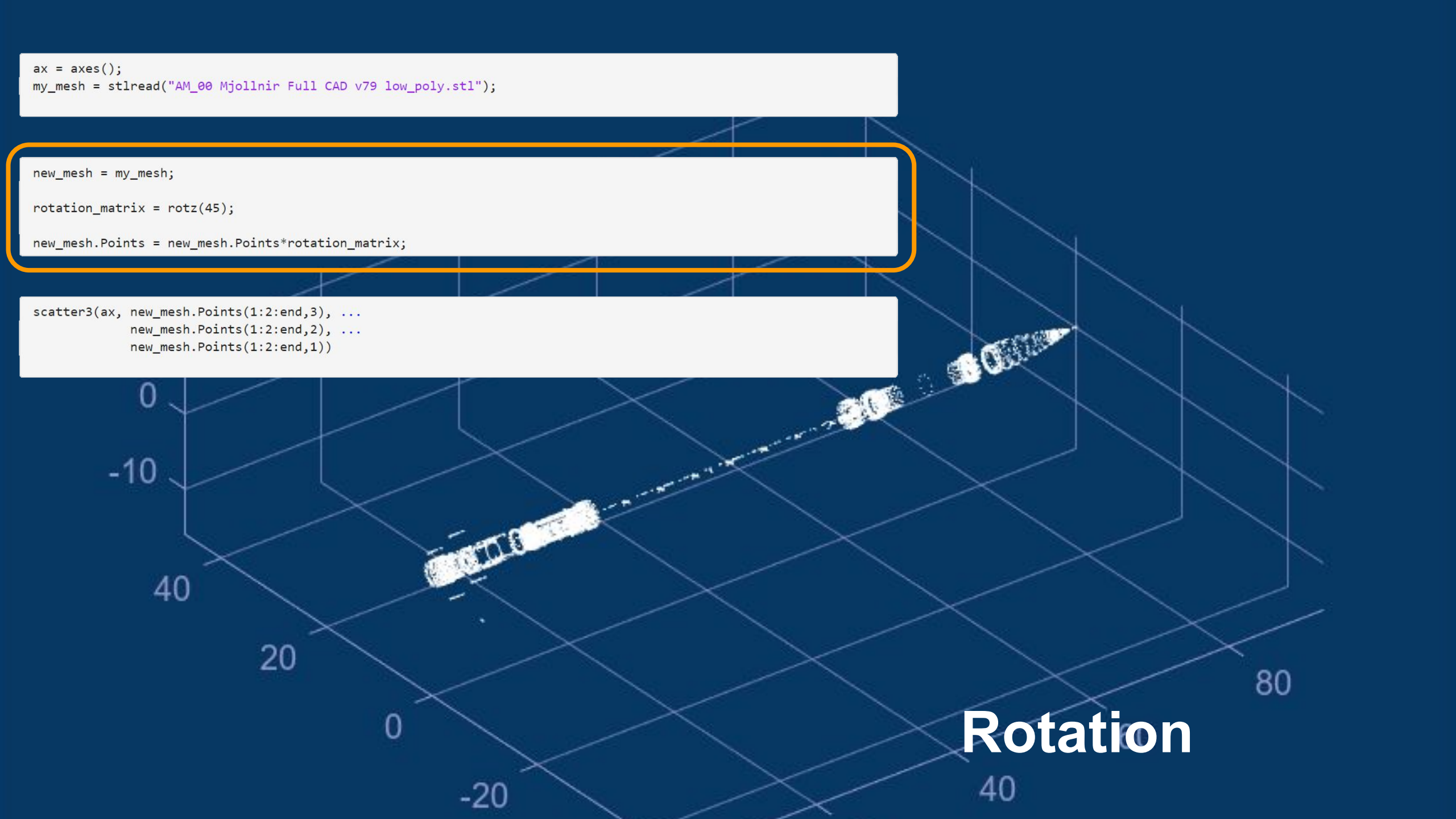


Translation

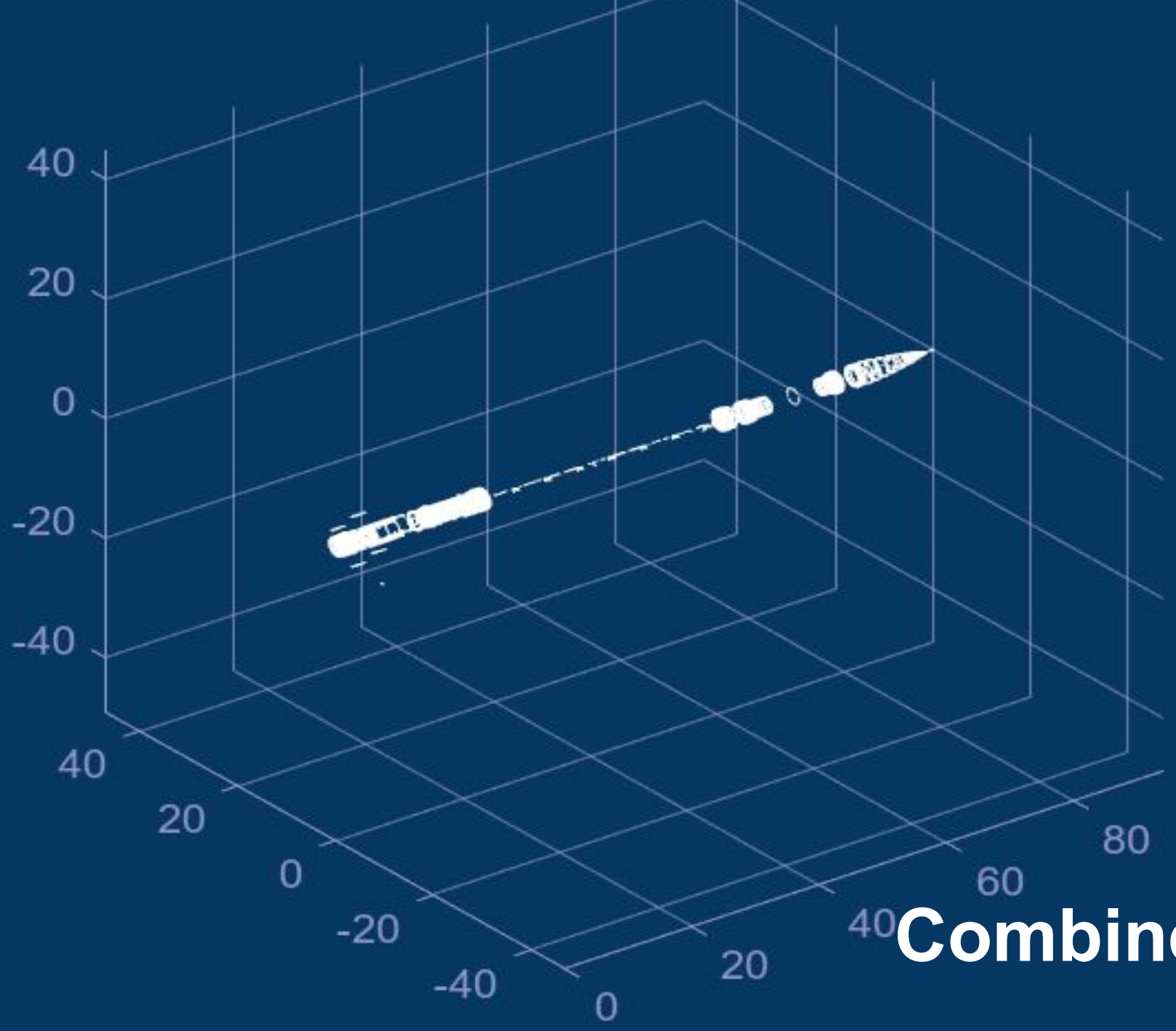
```
ax = axes();
my_mesh = stlread("AM_00 Mjollnir Full CAD v79 low_poly.stl");
```

```
new_mesh = my_mesh;

rotation_matrix = rotz(45);

new_mesh.Points = new_mesh.Points*rotation_matrix;
```

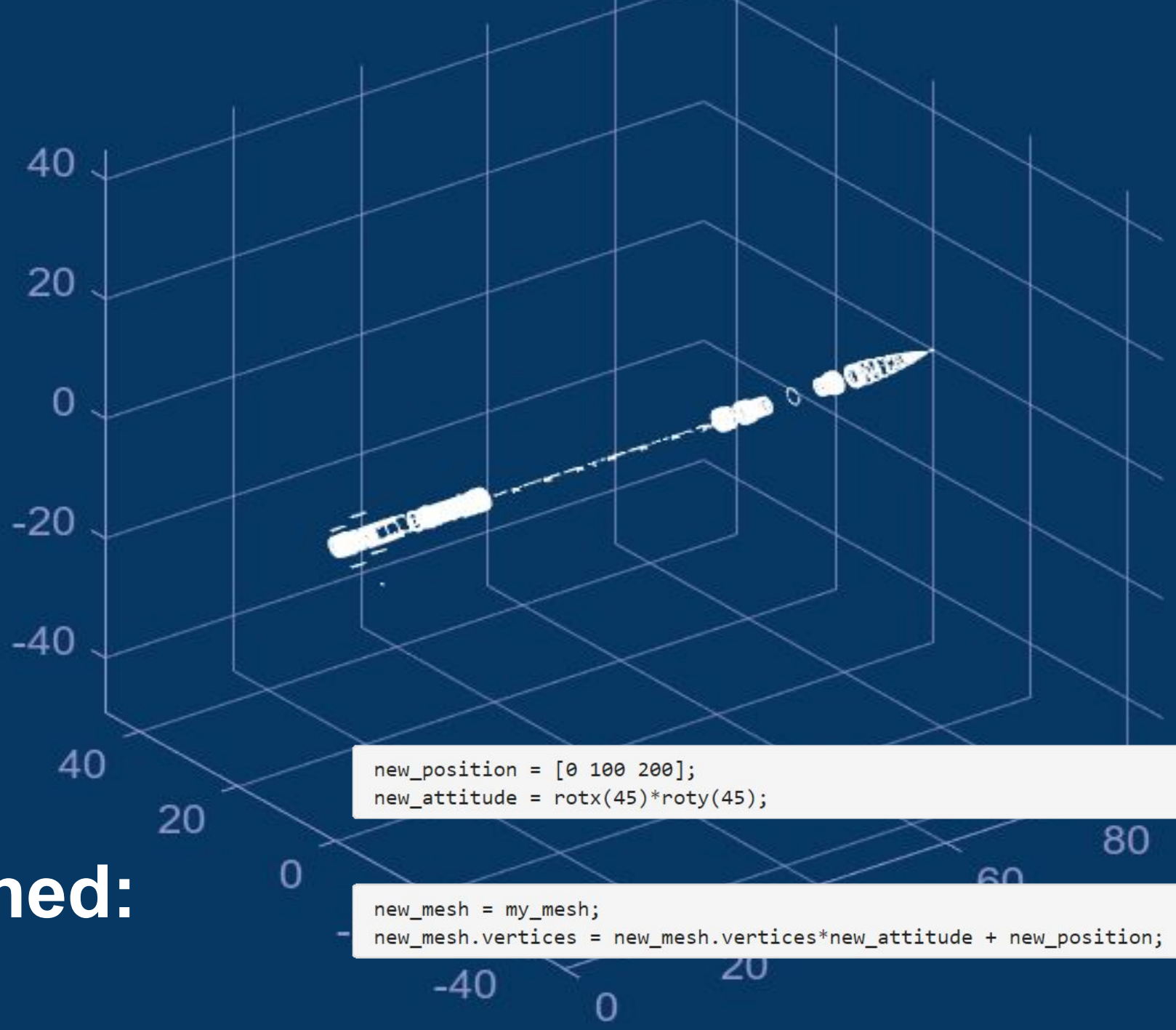
```
scatter3(ax, new_mesh.Points(1:2:end,3), ...
             new_mesh.Points(1:2:end,2), ...
             new_mesh.Points(1:2:end,1))
```



Rotation



Combined:

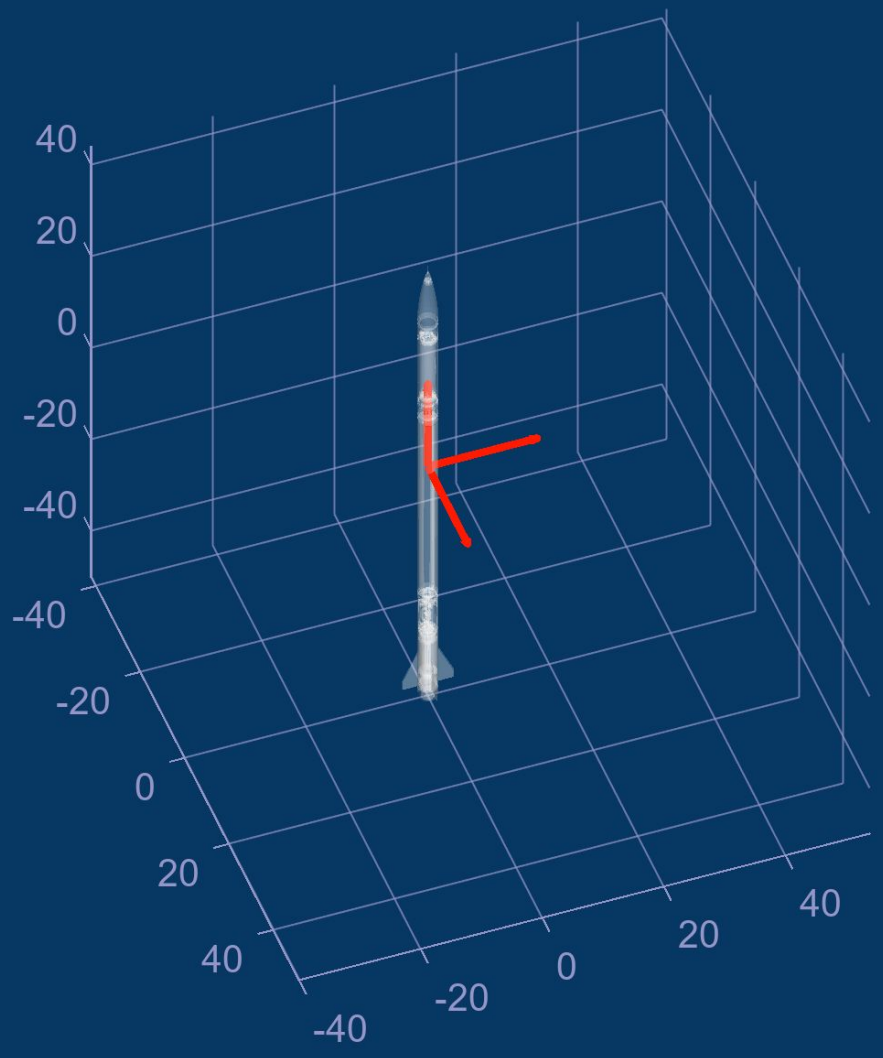


Combined:

```
new_position = [0 100 200];
new_attitude = rotx(45)*roty(45);
```

```
new_mesh = my_mesh;
new_mesh.vertices = new_mesh.vertices*new_attitude + new_position;
```

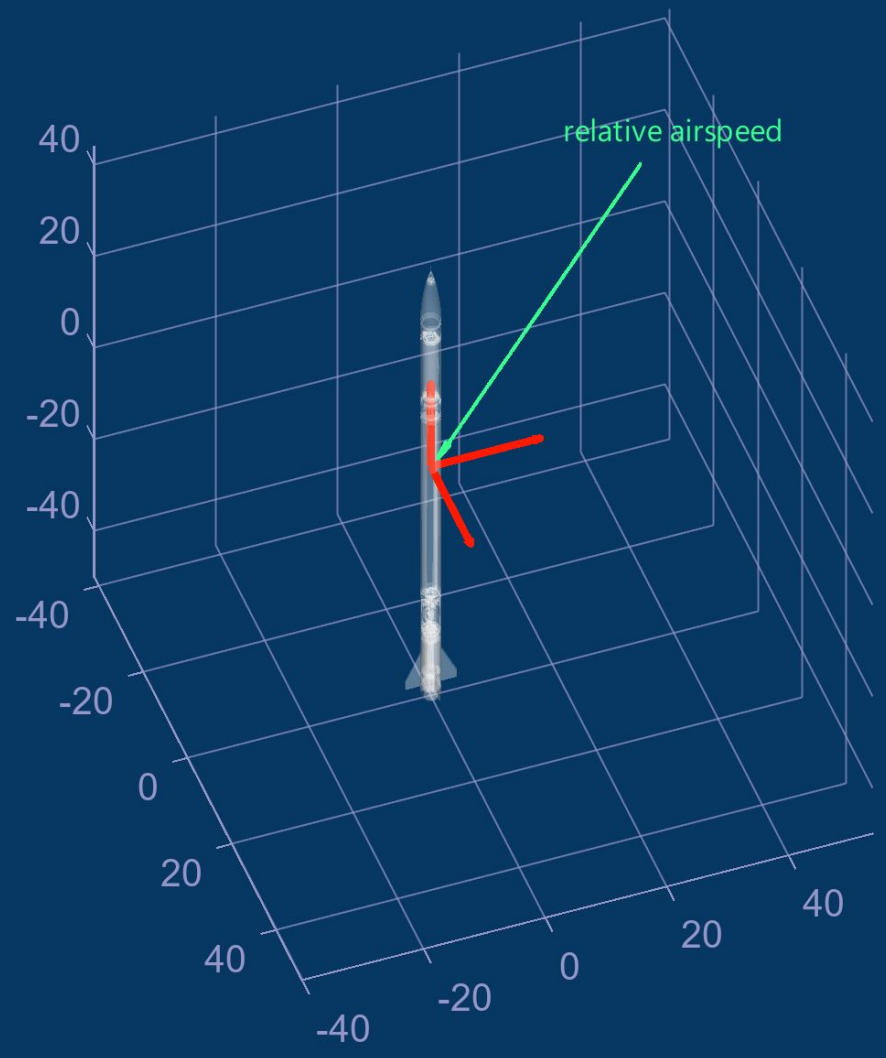
Projection and basis-changes



Assumptions:

- Lift-forces act normal to the plane
- Drag-forces act in the direction of airflow

Projection and basis-changes

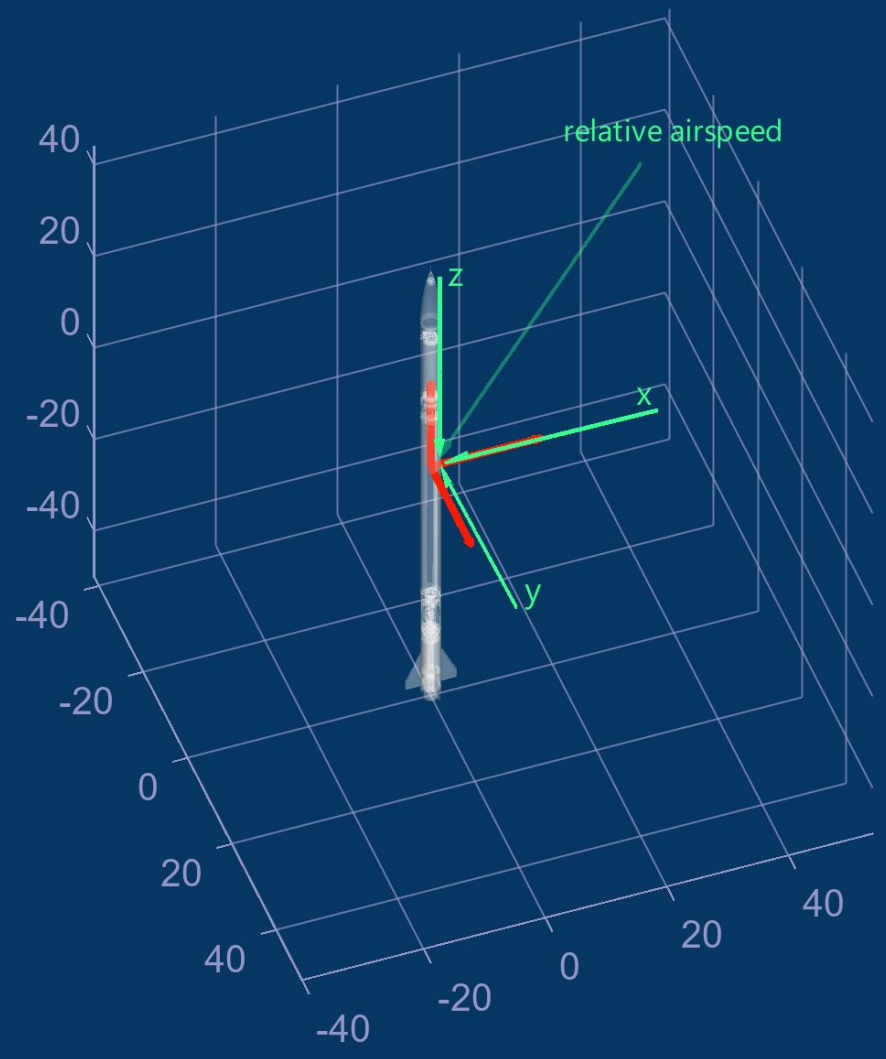


Assumptions:

- Lift-forces act normal to the plane
- Drag-forces act in the direction of airflow

$$\vec{F}_{lift} = \hat{n}\frac{1}{2}C\rho A v^2 sign(v)$$

Projection and basis-changes

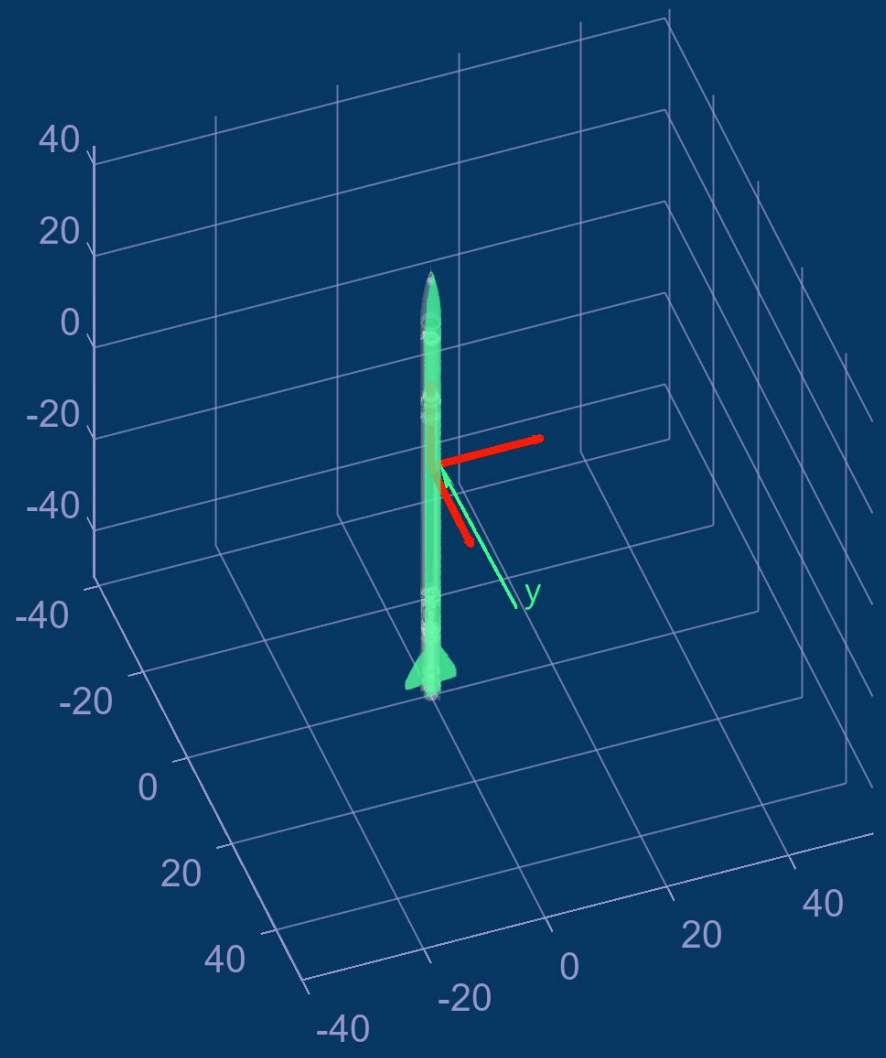


Assumptions:

- Lift-forces act normal to the plane
- Drag-forces act in the direction of airflow

$$\vec{F}_{lift} = \hat{n}\frac{1}{2}C\rho Av^2 sign(v)$$

Projection and basis-changes

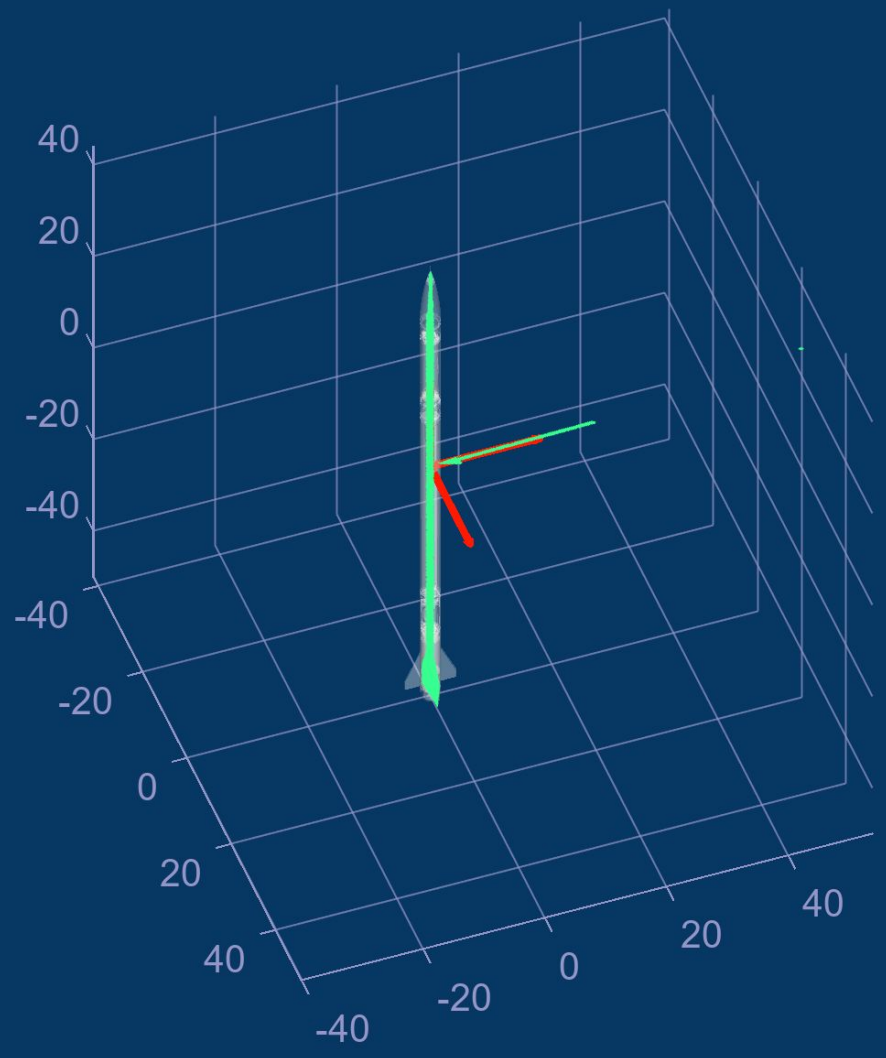


Assumptions:

- Lift-forces act normal to the plane
- Drag-forces act in the direction of airflow

$$\vec{F}_{lift} = \hat{n}\frac{1}{2}C\rho A v^2 sign(v)$$

Projection and basis-changes

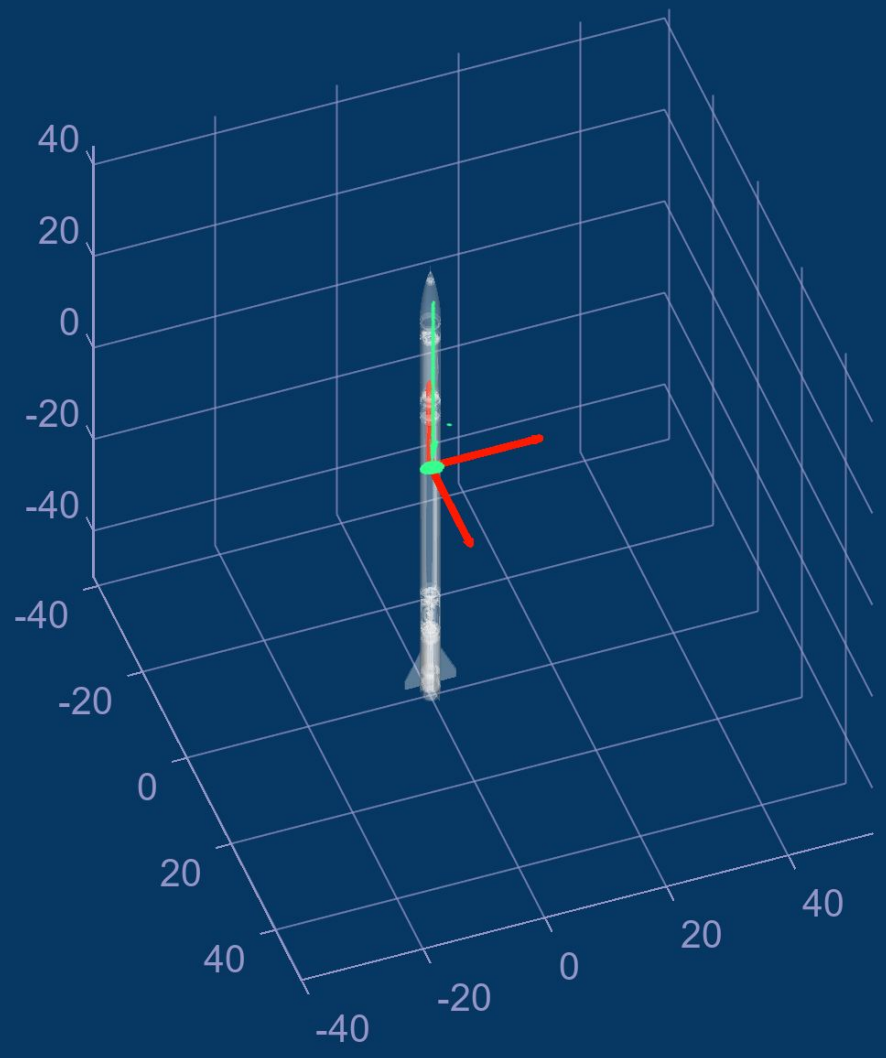


Assumptions:

- Lift-forces act normal to the plane
- Drag-forces act in the direction of airflow

$$\vec{F}_{lift} = \hat{n}\frac{1}{2}C\rho A v^2 sign(v)$$

Projection and basis-changes

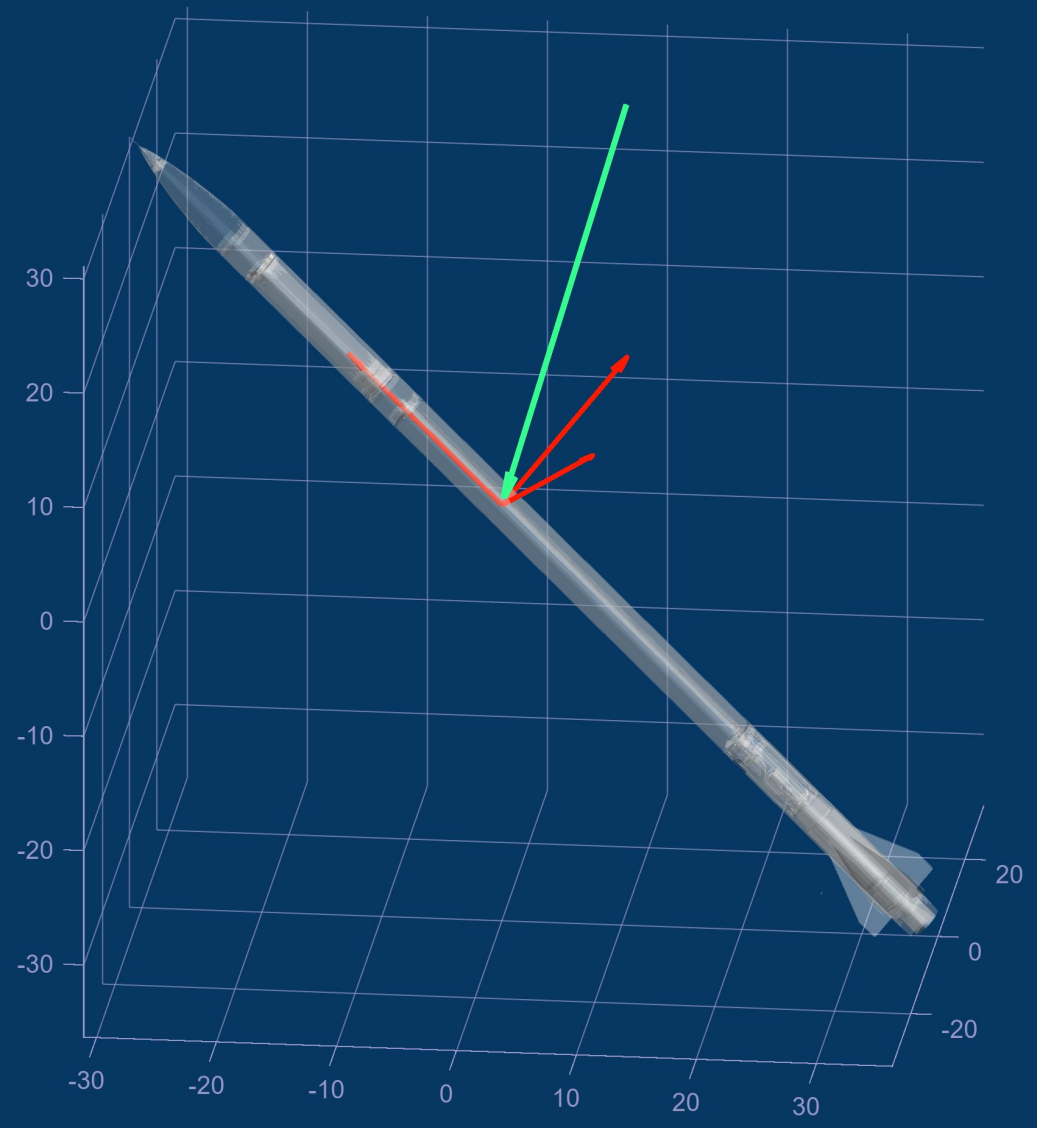


Assumptions:

- Lift-forces act normal to the plane
- Drag-forces act in the direction of airflow

$$\vec{F}_{lift} = \hat{n}\frac{1}{2}C\rho A v^2 sign(v)$$

Projection and basis-changes

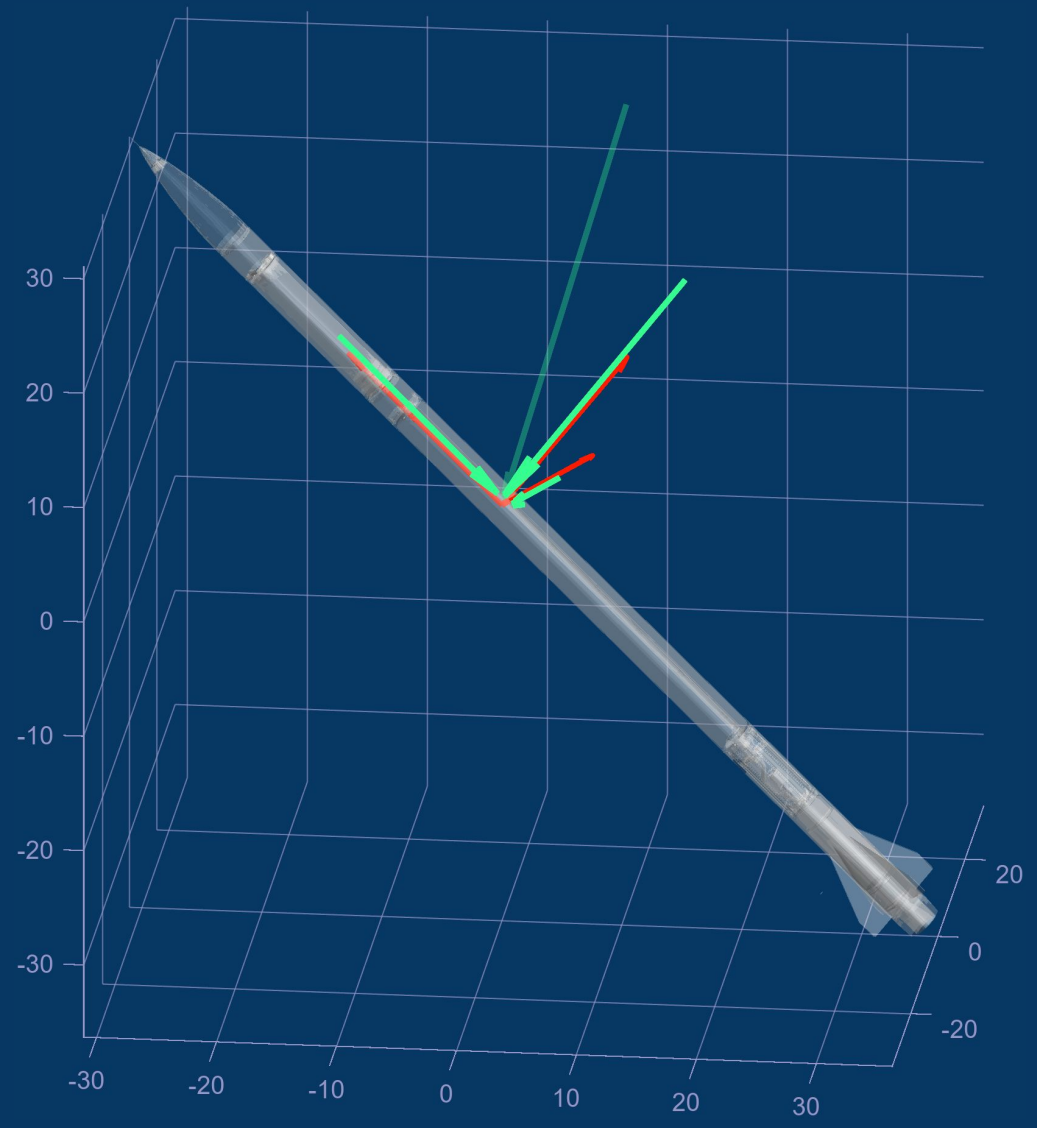


Assumptions:

- Lift-forces act normal to the plane
- Drag-forces act in the direction of airflow

$$\vec{F}_{lift} = \hat{n}\frac{1}{2}C\rho A v^2 sign(v)$$

Projection and basis-changes

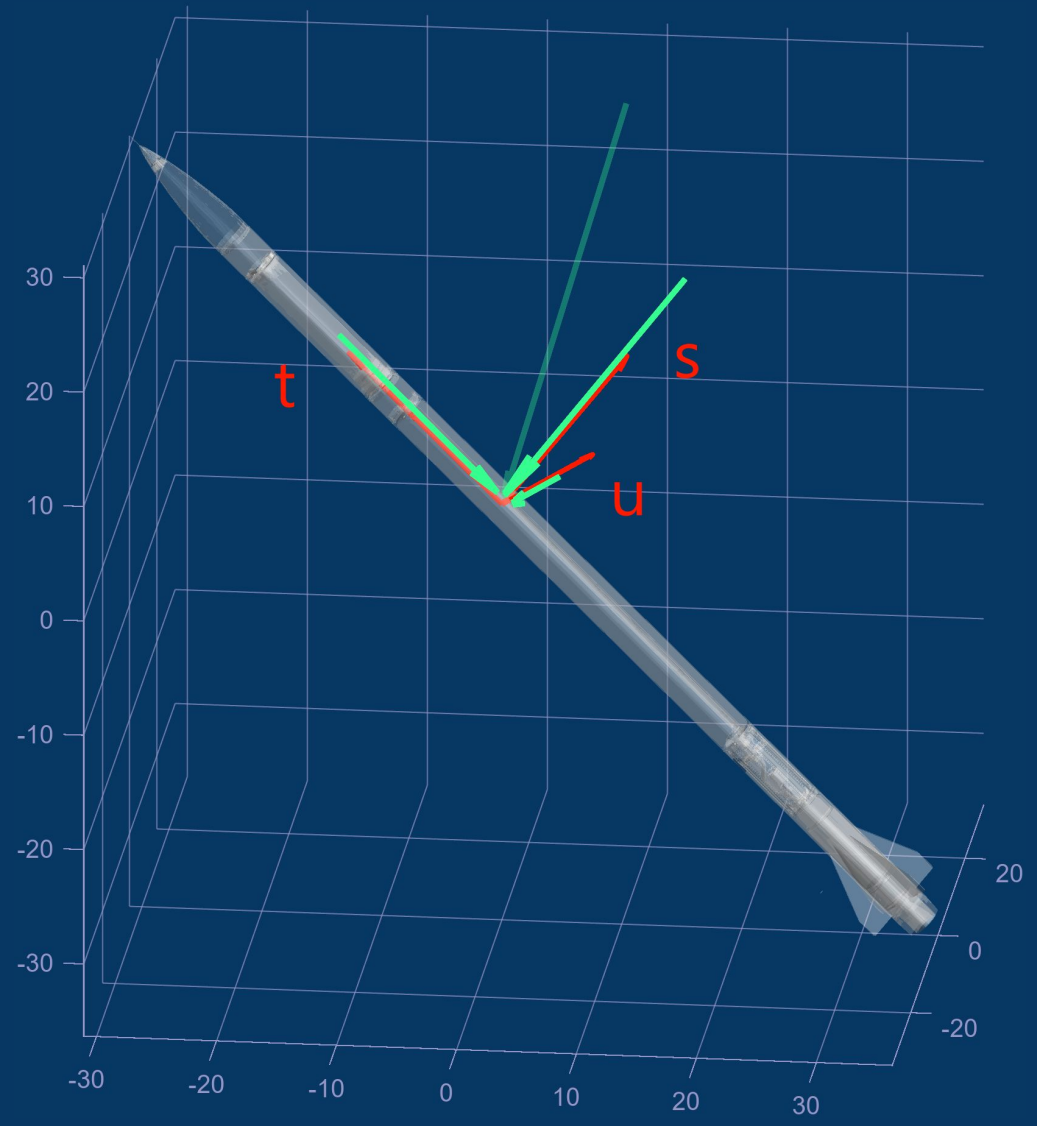


Assumptions:

- Lift-forces act normal to the plane
- Drag-forces act in the direction of airflow

$$\vec{F}_{lift} = \hat{n}\frac{1}{2}C\rho A v^2 sign(v)$$

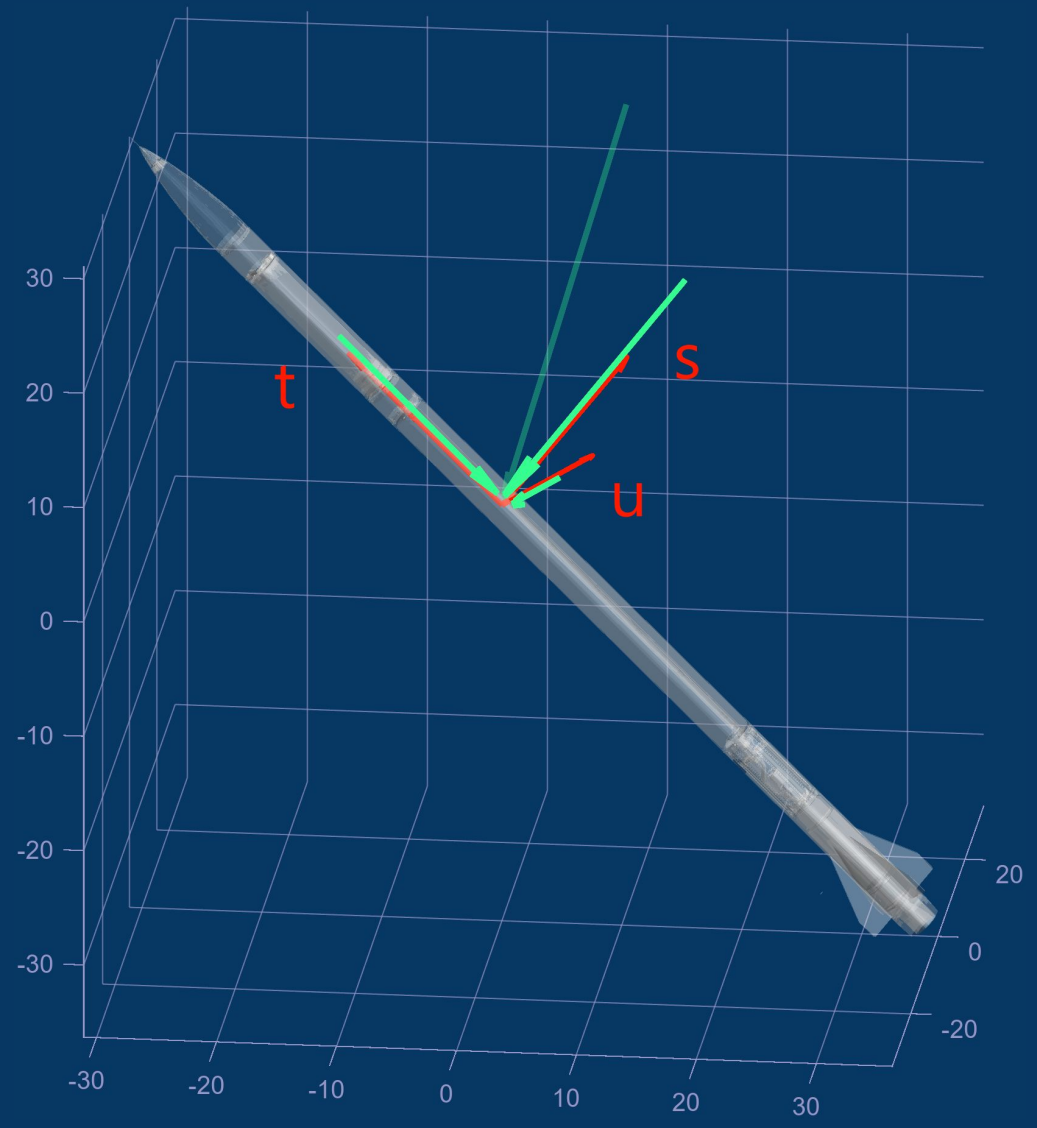
Projection and basis-changes



$$[\hat{t}\ \hat{u}\ \hat{s}] = \begin{bmatrix} t_x & u_x & s_x \\ t_y & u_y & s_y \\ t_z & u_z & s_z \end{bmatrix}$$

$$\vec{v}_{\hat{t},\hat{u},\hat{s}} \longrightarrow \vec{v}_{\hat{x},\hat{y},\hat{z}}$$

Projection and basis-changes



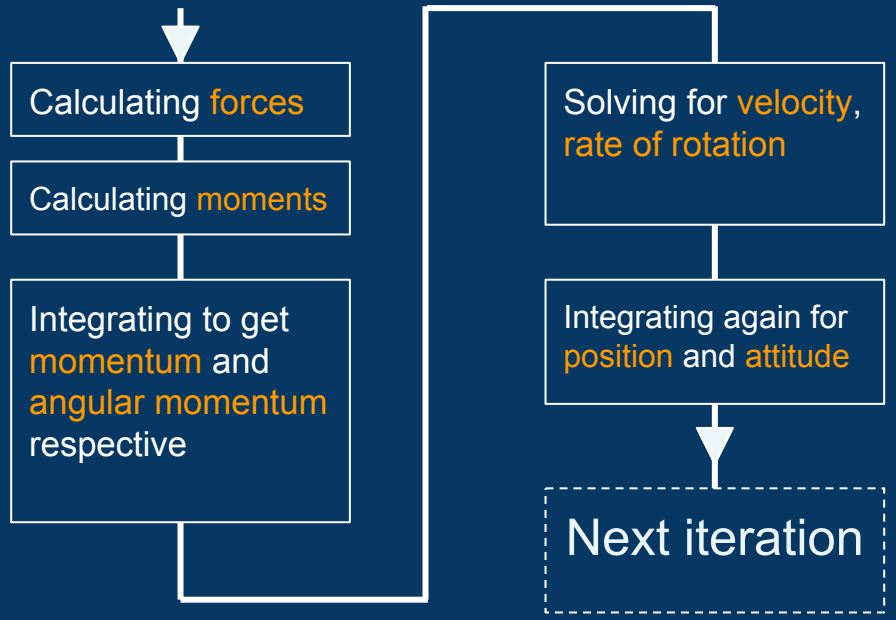
$$\begin{bmatrix} t_x & u_x & s_x \\ t_y & u_y & s_y \\ t_z & u_z & s_z \end{bmatrix}^{-1}$$

$$\vec{v}_{\hat{t},\hat{u},\hat{s}} \longleftarrow \vec{v}_{\hat{x},\hat{y},\hat{z}}$$

Now!

Now we have a bunch of tools: 🔧🛠️

- We have a rough idea of the simulation-pipeline
- We can have a lot of different variables and properties without it causing a headache (encapsulation)
- We can move and manipulate meshes and other vectors in 3D-space
- We can change basis from the vehicle's own basis to the world-basis, and the other way around



Now to put it all together!

Assigning variables:

```
%% Setup:

my_rocket.mass                  = 100;
my_rocket.center_of_mass        = [0;0;0.5];
my_rocket.center_of_pressure    = [0;0;-0.5];
my_rocket.mesh                  = stlread(mypath+"AM_00 Mjollnir Full CAD v79 low_poly 0.03.stl");
my_rocket.mesh.vertices         = my_rocket.mesh.vertices*0.2;
my_rocket.mesh.vertices         = my_rocket.mesh.vertices - 0.5*[max(my_rocket.mesh.vertices(:,1))+min(my_rocket.mesh.vertices(:,1));
                                                                 max(my_rocket.mesh.vertices(:,2))+min(my_rocket.mesh.vertices(:,2));
                                                                 max(my_rocket.mesh.vertices(:,3))+min(my_rocket.mesh.vertices(:,3))]';
my_rocket                       = data_from_mesh(my_rocket);
my_rocket.mesh.vertices         = 1.5*my_rocket.mesh.vertices/my_rocket.length_scale(3);
my_rocket.area                  = [0.1;0.1;0.1^2];
my_rocket.moment_of_inertia     = eye(3)*(my_rocket.mass*my_rocket.length_scale(3).^2)/12;
my_rocket.moment_of_inertia(3,3) = (my_rocket.mass*my_rocket.length_scale(1).^2)/2;

my_rocket.friction_coefficient  = [1;1;1]*0.2;
my_rocket.attitude              = roty(5)*my_rocket.attitude;

my_rocket.forces("gravity")     = force(g*my_rocket.mass*[0;0;-1], my_rocket.center_of_mass);
my_rocket.position              = [0;0;1];
```

Main simulation pipeline:

```
%% Main loop

iteration = 1;
while true


my_rocket = apply_aerodynamics(my_rocket);
my_rocket = add_thrust(my_rocket, t);
my_rocket = step_sim(my_rocket);


if mod(iteration, 2) == 0
draw_component(ax, my_rocket);

end



if my_rocket.position(3) < 0; break; end

iteration = iteration +1;
t = t + dt;
end
```

Main simulation pipeline:

```
%% Main loop

iteration = 1;
while true


my_rocket = apply_aerodynamics(my_rocket);
my_rocket = add_thrust(my_rocket, t);
my_rocket = step_sim(my_rocket);


if mod(iteration, 2) == 0
draw_component(ax, my_rocket);

end



if my_rocket.position(3) < 0; break; end

iteration = iteration +1;
t = t + dt;
end
```

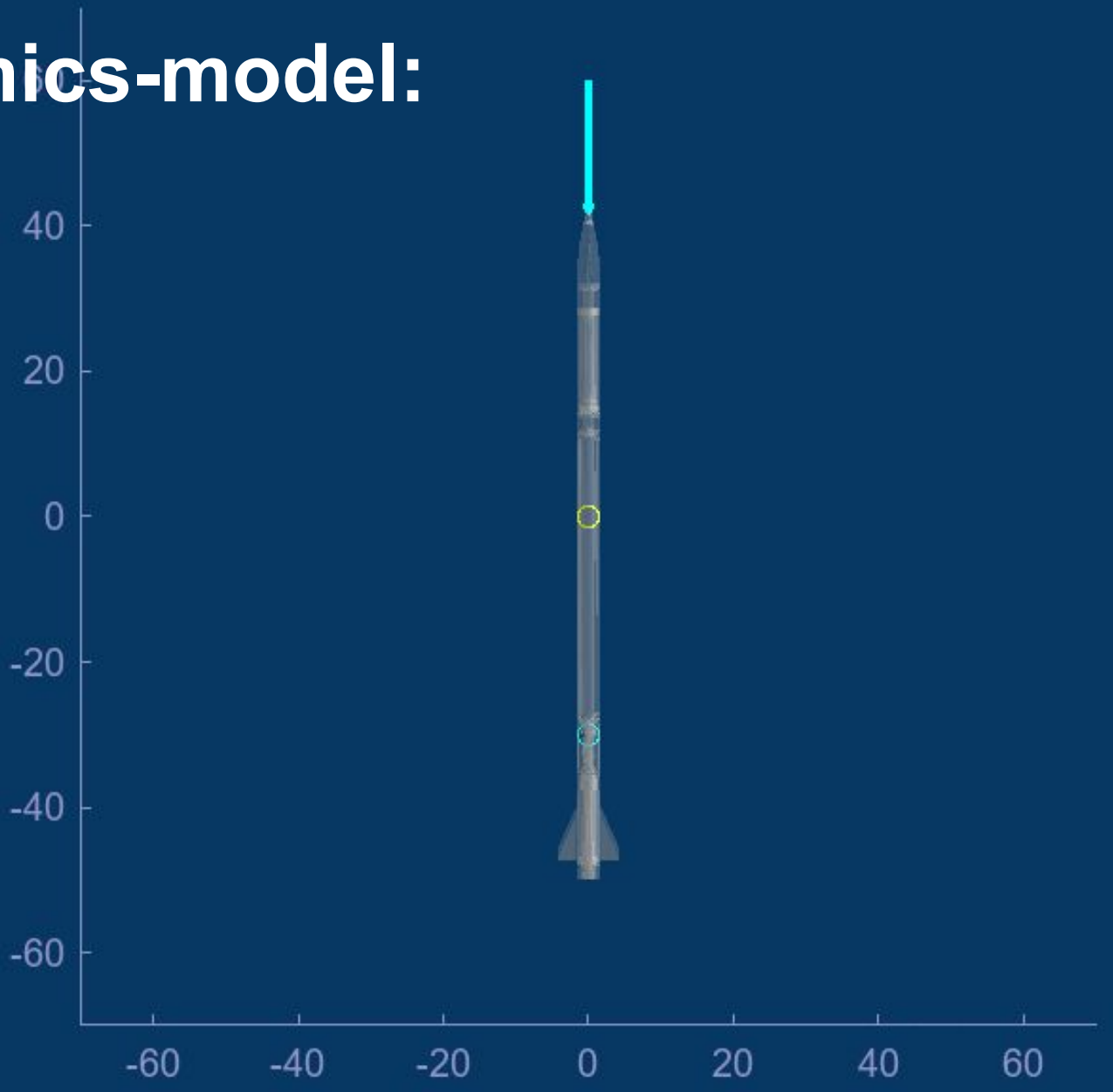
Aerodynamics-model:

```
function comp = apply_aerodynamics(comp)


wind_velocity = evalin("base", "wind_velocity");
air_density   = evalin("base", "air_density");



relative_velocity               = wind_velocity - comp.velocity - cross(comp.rotation_rate, comp.attitude*(comp.center_of_pressure - comp.center_of_mass));
relative_velocity_comp_basis    = (comp.attitude')*relative_velocity;

parallel_velocity_magnitude     = sqrt(norm(relative_velocity)^2 - relative_velocity_comp_basis*norm(relative_velocity));




%% Forces:

lift_force = comp.attitude*(comp.pressure_coefficient*(comp.area.*sign(relative_velocity_comp_basis).*(relative_velocity_comp_basis.^2))*air_density);
% factor 1/2 dissapears as we have two surfaces per plane
drag_force = normalize(relative_velocity)*sum(comp.friction_coefficient.*comp.area.*parallel_velocity_magnitude.^2)*air_density;

comp.forces("drag force") = force(drag_force, comp.center_of_pressure);
comp.forces("lift force") = force(lift_force, comp.center_of_pressure);




end
```

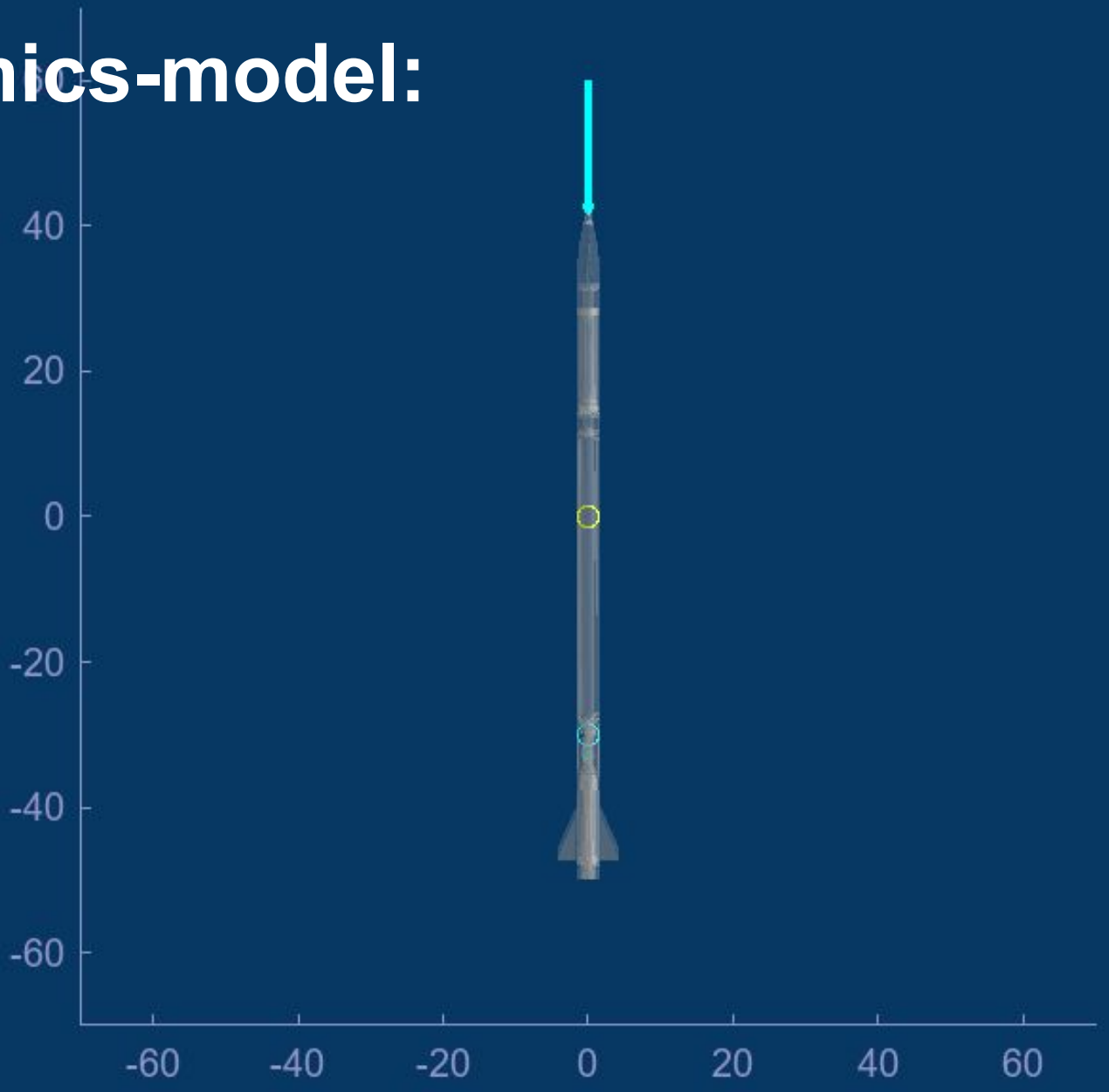
Aerodynamics-model:

```
function comp = apply_aerodynamics(comp)


wind_velocity = evalin("base", "wind_velocity");
air_density   = evalin("base", "air_density");



relative_velocity               = wind_velocity - comp.velocity - cross(comp.rotation_rate, comp.attitude*(comp.center_of_pressure - comp.center_of_mass));
relative_velocity_comp_basis    = (comp.attitude')*relative_velocity;

parallel_velocity_magnitude     = sqrt(norm(relative_velocity)^2 - relative_velocity_comp_basis*norm(relative_velocity));




%% Forces:

lift_force = comp.attitude*(comp.pressure_coefficient*(comp.area.*sign(relative_velocity_comp_basis).*(relative_velocity_comp_basis.^2))*air_density);
% factor 1/2 dissapears as we have two surfaces per plane
drag_force = normalize(relative_velocity)*sum(comp.friction_coefficient.*comp.area.*parallel_velocity_magnitude.^2)*air_density;

comp.forces("drag force") = force(drag_force, comp.center_of_pressure);
comp.forces("lift force") = force(lift_force, comp.center_of_pressure);




end
```

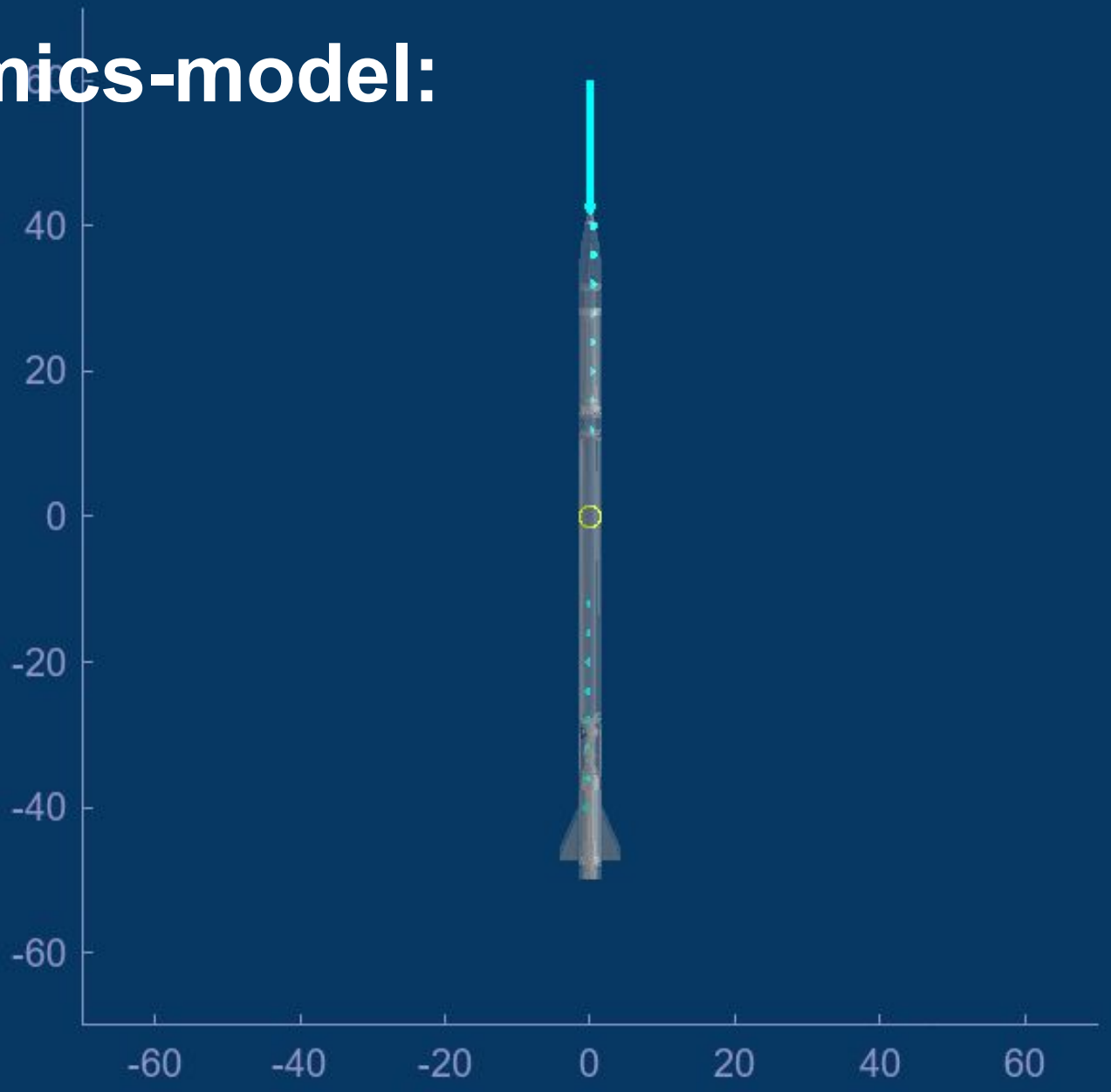
Aerodynamics-model:



Aerodynamics-model:



Aerodynamics-model:



Aerodynamics-model:

```
function comp = apply_aerodynamics(comp)


wind_velocity = evalin("base", "wind_velocity");
air_density   = evalin("base", "air_density");



relative_velocity              = wind_velocity - comp.velocity - cross(comp.rotation_rate, comp.attitude*(comp.
relative_velocity_comp_basis   = (comp.attitude')*relative_velocity;

parallel_velocity_magnitude    = sqrt(norm(relative_velocity)^2 - relative_velocity_comp_basis*norm(relative_ve



%% Forces:

lift_force = comp.attitude*(comp.pressure_coefficient*(comp.area.*sign(relative_velocity_comp_basis).*(relative_
% factor 1/2 dissapears as we have two surfaces per plane
drag_force = normalize(relative_velocity)*sum(comp.friction_coefficient.*comp.area.*parallel_velocity_magnitude.^2)*air_density;

comp.forces("drag force") = force(drag_force, comp.center_of_pressure);
comp.forces("lift force") = force(lift_force, comp.center_of_pressure);



end
```

$$\begin{bmatrix} t_x & u_x & s_x \\ t_y & u_y & s_y \\ t_z & u_z & s_z \end{bmatrix}^{-1}$$

$$\vec{v}_{\hat{t},\hat{u},\hat{s}} \longleftarrow \vec{v}_{\hat{x},\hat{y},\hat{z}}$$

Aerodynamics-model:

```
function comp = apply_aerodynamics(comp)


wind_velocity = evalin("base", "wind_velocity");
air_density   = evalin("base", "air_density");



relative_velocity               = wind_velocity - comp.velocity - cross(comp.rotation_rate, comp.attitude*(comp.center
relative_velocity_comp_basis    = (comp.attitude')*relative_velocity;

parallel_velocity_magnitude     = sqrt(norm(relative_velocity)^2 - relative_velocity_comp_basis*norm(relative_velocity



%% Forces:

lift_force = comp.attitude*(comp.pressure_coefficient*(comp.area.*sign(relative_velocity_comp_basis).*(relative_velocity_comp_basis.^2))*air_density);
% factor 1/2 dissapears as we have two surfaces per plane
drag_force = normalize(relative_velocity)*sum(comp.friction_coefficient.*comp.area.*parallel_velocity_magnitude.^2)*air_density;

comp.forces("drag force") = force(drag_force, comp.center_of_pressure);
comp.forces("lift force") = force(lift_force, comp.center_of_pressure);




end
```

$$\vec{F}_{lift} = \hat{n}\frac{1}{2}C\rho Av^2 sign(v)$$

Aerodynamics-model:

```
function comp = apply_aerodynamics(comp)


wind_velocity = evalin("base", "wind_velocity");
air_density   = evalin("base", "air_density");



relative_velocity               = wind_velocity - comp.velocity - cross(comp.rotation_rate, comp.attitude*(comp.center_
relative_velocity_comp_basis    = (comp.attitude')*relative_velocity;

parallel_velocity_magnitude     = sqrt(norm(relative_velocity)^2 - relative_velocity_comp_basis*norm(relative_velocity))



%% Forces:

lift_force = comp.attitude*(comp.pressure_coefficient*(comp.area.*sign(relative_velocity_comp_basis).*(relative_velocity_comp_basis.^2))*air_density);
% factor 1/2 dissapears as we have two surfaces per plane
drag_force = normalize(relative_velocity)*sum(comp.friction_coefficient.*comp.area.*parallel_velocity_magnitude.^2)*air_density;

comp.forces("drag force") = force(drag_force, comp.center_of_pressure);
comp.forces("lift force") = force(lift_force, comp.center_of_pressure);



end
```

$$\vec{F}_{drag} = \hat{v}\frac{1}{2}C\rho Av^2$$

Thrust-model:

```
%% Main loop

iteration = 1;
while true


my_rocket = apply_aerodynamics(my_rocket);
my_rocket = add_thrust(my_rocket, t);
my_rocket = step_sim(my_rocket);


if mod(iteration, 2) == 0
draw_component(ax, my_rocket);

end



if my_rocket.position(3) < 0; break; end

iteration = iteration +1;
t = t + dt;
end
```

Thrust-model:

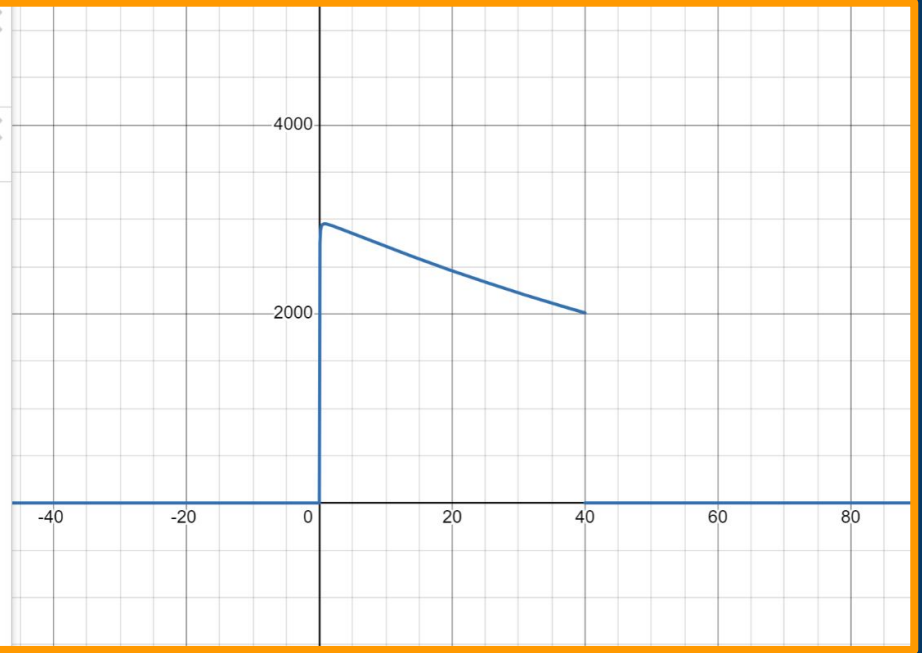
```
function comp = add_thrust(comp, t)

thrust_force = comp.attitude*[0;0;1]*3000*(2/pi)*atan(100*t)*exp(-t/100)*(t < 20);

comp.forces("thrust force") = force(thrust_force, [0;0;-0.9]);

end
```

Thrust-model:

```
function comp = add_thrust(comp, t)

thrust_force = comp.attitude*[0;0;1]*3000*(2/pi)*atan(100*t)*exp(-t/100)*(t < 20);

comp.forces("thrust force") = force(thrust_force, [0;0;-0.9]);

end
```

$$F(t) = 3000 \cdot \frac{2}{\pi} \arctan(100 \cdot t) e^{-\frac{t}{100}} \cdot H(40 - t) \cdot H(t)$$

$$H(t) = \{t < 0 : 0, t \geq 0 : 1\}$$



Integration and solving:

```
%% Main loop

iteration = 1;
while true


my_rocket = apply_aerodynamics(my_rocket);
my_rocket = add_thrust(my_rocket, t);
my_rocket = step_sim(my_rocket);


if mod(iteration, 2) == 0
draw_component(ax, my_rocket);

end



if my_rocket.position(3) < 0; break; end

iteration = iteration +1;
t = t + dt;
end
```

Integration and solving:

```
function comp=step_sim(comp)
dt = evalin("base", "dt");



%sum goes brrrrrrrrrrrrrrrr

force_sum  = cellsum(cellfun(@(force)  force.vec ,                                                    values(comp.forces , "cell"), "UniformOutput",false));
moment_sum = cellsum(cellfun(@(moment) moment.vec ,                                                   values(comp.moments, "cell"), "UniformOutput",false)) + ...
             cellsum(cellfun(@(force)  cross(comp.attitude*(force.pos - comp.center_of_mass), force.vec),   values(comp.forces,  "cell"), "UniformOutput",false));



% Stepping velocity
comp.velocity = comp.velocity + dt*force_sum/comp.mass;
comp.position = comp.position + comp.velocity*dt;



% Stepping angular stuff
comp.angular_momentum = comp.angular_momentum + moment_sum*dt;
comp.rotation_rate = (comp.attitude*comp.moment_of_inertia)\comp.angular_momentum;

if norm(comp.rotation_rate) ~= 0
% Gram-Schmidt orthogonalization:
rotation_rate_basis = rand(3);
rotation_rate_basis(:,1) = comp.rotation_rate/norm(comp.rotation_rate);
rotation_rate_basis(:,2) = cross(rotation_rate_basis(:,1), rand(3,1));
rotation_rate_basis(:,2) = rotation_rate_basis(:,2)/norm(rotation_rate_basis(:,2));
rotation_rate_basis(:,3) = cross(rotation_rate_basis(:,1), rotation_rate_basis(:,2));

dtheta        = norm(comp.rotation_rate)*dt*2*pi;
comp.attitude = rotation_rate_basis*rotx(dtheta*360/(2*pi))*(rotation_rate_basis\comp.attitude);
end

end
```

Integration and solving:

```
function comp=step_sim(comp)
dt = evalin("base", "dt");



%sum goes brrrrrrrrrrrrrrrr

force_sum  = cellsum(cellfun(@(force)  force.vec ,                                                  values(comp.forces , "cell"), "UniformOutput",false));
moment_sum = cellsum(cellfun(@(moment) moment.vec ,                                                 values(comp.moments, "cell"), "UniformOutput",false)) + ...
             cellsum(cellfun(@(force)  cross(comp.attitude*(force.pos - comp.center_of_mass), force.vec), values(comp.forces,  "cell"), "UniformOutput",false));



% Stepping velocity
comp.velocity = comp.velocity + dt*force_sum/comp.mass;
comp.position = comp.position + comp.velocity*dt;



% Stepping angular stuff
comp.angular_momentum = comp.angular_momentum + moment_sum*dt;
comp.rotation_rate = (comp.attitude*comp.moment_of_inertia)\comp.angular_momentum;

if norm(comp.rotation_rate) ~= 0
% Gram-Schmidt orthogonalization:
rotation_rate_basis = rand(3);
rotation_rate_basis(:,1) = comp.rotation_rate/norm(comp.rotation_rate);
rotation_rate_basis(:,2) = cross(rotation_rate_basis(:,1), rand(3,1));
rotation_rate_basis(:,2) = rotation_rate_basis(:,2)/norm(rotation_rate_basis(:,2));
rotation_rate_basis(:,3) = cross(rotation_rate_basis(:,1), rotation_rate_basis(:,2));

dtheta        = norm(comp.rotation_rate)*dt*2*pi;
comp.attitude = rotation_rate_basis*rotx(dtheta*360/(2*pi))*(rotation_rate_basis\comp.attitude);
end

end
```

Integration and solving:

```
function comp=step_sim(comp)
dt = evalin("base", "dt");



%sum goes brrrrrrrrrrrrrrrr

force_sum  = cellsum(cellfun(@(force)  force.vec ,                                                    values(comp.forces , "cell"), "UniformOutput",false));
moment_sum = cellsum(cellfun(@(moment) moment.vec ,                                                   values(comp.moments, "cell"), "UniformOutput",false)) + ...
             cellsum(cellfun(@(force)  cross(comp.attitude*(force.pos - comp.center_of_mass), force.vec),  values(comp.forces,  "cell"), "UniformOutput",false));



% Stepping velocity
comp.velocity = comp.velocity + dt*force_sum/comp.mass;
comp.position = comp.position + comp.velocity*dt;



% Stepping angular stuff
comp.angular_momentum = comp.angular_momentum + moment_sum*dt;
comp.rotation_rate = (comp.attitude*comp.moment_of_inertia)\comp.angular_momentum;

if norm(comp.rotation_rate) ~= 0
% Gram-Schmidt orthogonalization:
rotation_rate_basis = rand(3);
rotation_rate_basis(:,1) = comp.rotation_rate/norm(comp.rotation_rate);
rotation_rate_basis(:,2) = cross(rotation_rate_basis(:,1), rand(3,1));
rotation_rate_basis(:,2) = rotation_rate_basis(:,2)/norm(rotation_rate_basis(:,2));
rotation_rate_basis(:,3) = cross(rotation_rate_basis(:,1), rotation_rate_basis(:,2));

dtheta        = norm(comp.rotation_rate)*dt*2*pi;
comp.attitude = rotation_rate_basis*rotx(dtheta*360/(2*pi))*(rotation_rate_basis\comp.attitude);
end

end
```

Integration and solving:

```
function comp=step_sim(comp)
dt = evalin("base", "dt");



%sum goes brrrrrrrrrrrrrrrr

force_sum  = cellsum(cellfun(@(force)  force.vec ,                                                    values(comp.forces , "cell"), "UniformOutput",false));
moment_sum = cellsum(cellfun(@(moment) moment.vec ,                                                   values(comp.moments, "cell"), "UniformOutput",false)) + ...
             cellsum(cellfun(@(force)  cross(comp.attitude*(force.pos - comp.center_of_mass), force.vec), values(comp.forces,  "cell"), "UniformOutput",false));



% Stepping velocity
comp.velocity = comp.velocity + dt*force_sum/comp.mass;
comp.position = comp.position + comp.velocity*dt;



% Stepping angular stuff
comp.angular_momentum = comp.angular_momentum + moment_sum*dt;
comp.rotation_rate = (comp.attitude*comp.moment_of_inertia)\comp.angular_momentum;

if norm(comp.rotation_rate) ~= 0
% Gram-Schmidt orthogonalization:
rotation_rate_basis = rand(3);
rotation_rate_basis(:,1) = comp.rotation_rate/norm(comp.rotation_rate);
rotation_rate_basis(:,2) = cross(rotation_rate_basis(:,1), rand(3,1));
rotation_rate_basis(:,2) = rotation_rate_basis(:,2)/norm(rotation_rate_basis(:,2));
rotation_rate_basis(:,3) = cross(rotation_rate_basis(:,1), rotation_rate_basis(:,2));

dtheta        = norm(comp.rotation_rate)*dt*2*pi;
comp.attitude = rotation_rate_basis*rotx(dtheta*360/(2*pi))*(rotation_rate_basis\comp.attitude);
end

end
```

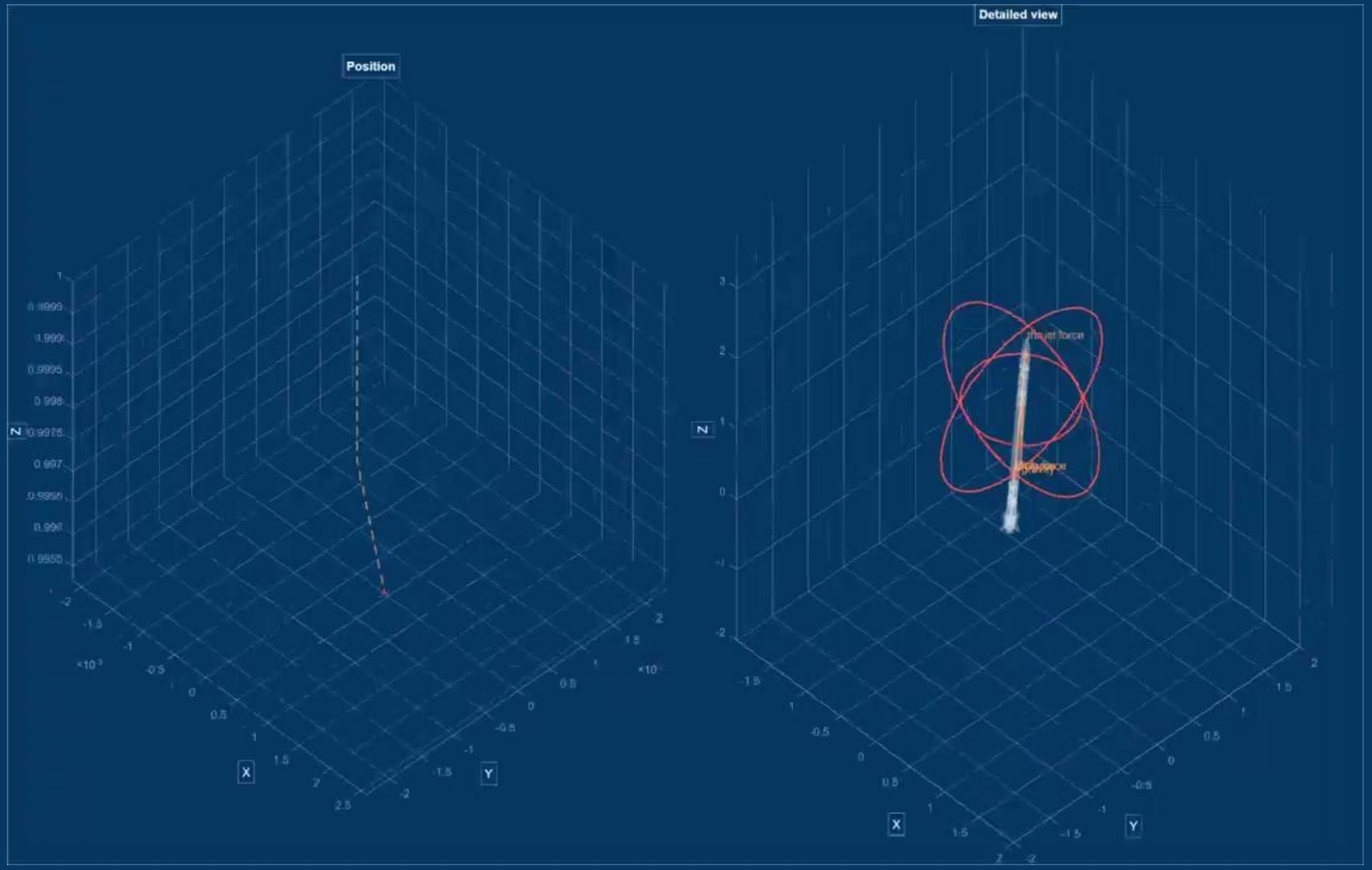
Integration and solving:

```
function comp=step_sim(comp)
dt = evalin("base", "dt");



%sum goes brrrrrrrrrrrrrrrr

force_sum  = cellsum(cellfun(@(force)  force.vec ,                                                    values(comp.forces , "cell"), "UniformOutput",false));
moment_sum = cellsum(cellfun(@(moment) moment.vec ,                                                   values(comp.moments, "cell"), "UniformOutput",false)) + ...
             cellsum(cellfun(@(force)  cross(comp.attitude*(force.pos - comp.center_of_mass), force.vec), values(comp.forces,  "cell"), "UniformOutput",false));



% Stepping velocity
comp.velocity = comp.velocity + dt*force_sum/comp.mass;
comp.position = comp.position + comp.velocity*dt;



% Stepping angular stuff
comp.angular_momentum = comp.angular_momentum + moment_sum*dt;
comp.rotation_rate = (comp.attitude*comp.moment_of_inertia)\comp.angular_momentum;

if norm(comp.rotation_rate) ~= 0
% Gram-Schmidt orthogonalization:
rotation_rate_basis = rand(3);
rotation_rate_basis(:,1) = comp.rotation_rate/norm(comp.rotation_rate);
rotation_rate_basis(:,2) = cross(rotation_rate_basis(:,1), rand(3,1));
rotation_rate_basis(:,2) = rotation_rate_basis(:,2)/norm(rotation_rate_basis(:,2));
rotation_rate_basis(:,3) = cross(rotation_rate_basis(:,1), rotation_rate_basis(:,2));

dtheta        = norm(comp.rotation_rate)*dt*2*pi;
comp.attitude = rotation_rate_basis*rotx(dtheta*360/(2*pi))*(rotation_rate_basis\comp.attitude);
end

end
```

Integration and solving:

```
function comp=step_sim(comp)
dt = evalin("base", "dt");



%sum goes brrrrrrrrrrrrrrrr

force_sum  = cellsum(cellfun(@(force)  force.vec ,                                                    values(comp.forces , "cell"), "UniformOutput",false));
moment_sum = cellsum(cellfun(@(moment) moment.vec ,                                                   values(comp.moments, "cell"), "UniformOutput",false)) + ...
             cellsum(cellfun(@(force)  cross(comp.attitude*(force.pos - comp.center_of_mass), force.vec), values(comp.forces,  "cell"), "UniformOutput",false));



% Stepping velocity
comp.velocity = comp.velocity + dt*force_sum/comp.mass;
comp.position = comp.position + comp.velocity*dt;



% Stepping angular stuff
comp.angular_momentum = comp.angular_momentum + moment_sum*dt;
comp.rotation_rate = (comp.attitude*comp.moment_of_inertia)\comp.angular_momentum;

if norm(comp.rotation_rate) ~= 0
% Gram-Schmidt orthogonalization:
rotation_rate_basis = rand(3);
rotation_rate_basis(:,1) = comp.rotation_rate/norm(comp.rotation_rate);
rotation_rate_basis(:,2) = cross(rotation_rate_basis(:,1), rand(3,1));
rotation_rate_basis(:,2) = rotation_rate_basis(:,2)/norm(rotation_rate_basis(:,2));
rotation_rate_basis(:,3) = cross(rotation_rate_basis(:,1), rotation_rate_basis(:,2));

dtheta        = norm(comp.rotation_rate)*dt*2*pi;
comp.attitude = rotation_rate_basis*rotx(dtheta*360/(2*pi))*(rotation_rate_basis\comp.attitude);
end

end
```

Putting it all together:



Thank you so much for your attention!

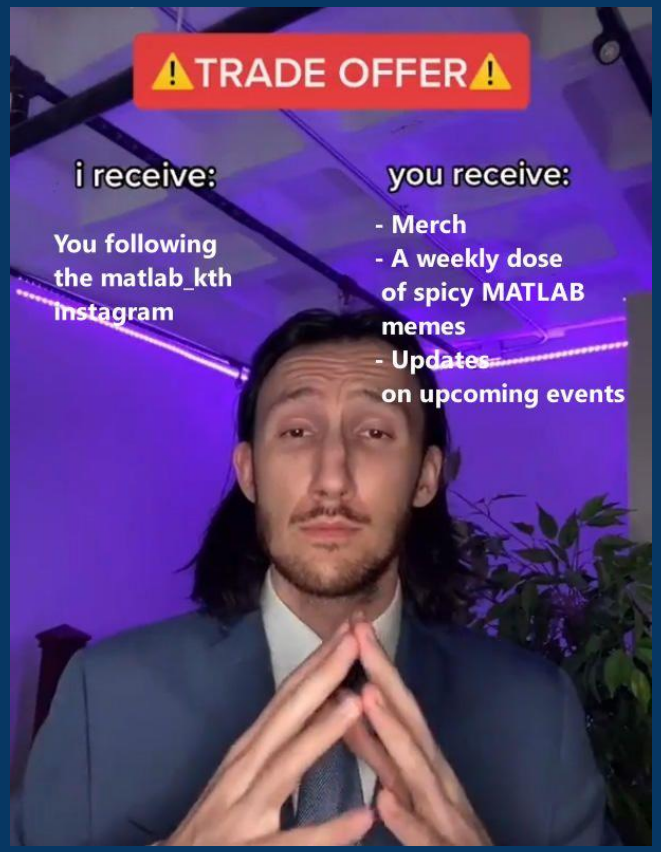
If you have an idea for an event, or are working on something interesting, be it a thesis, PhD topic or just a cool side-hobby that uses or is adjacent to MATLAB or Simulink, please reach out!

You can find me via the matlab_kth facebook-group or DM matlab_kth on instagram!

Merch giveaway requirements:

- Follow matlab_kth on instagram or join the facebook-group!

matlab_kth
facebook
group



SCAN ME

matlab_kth
instagram