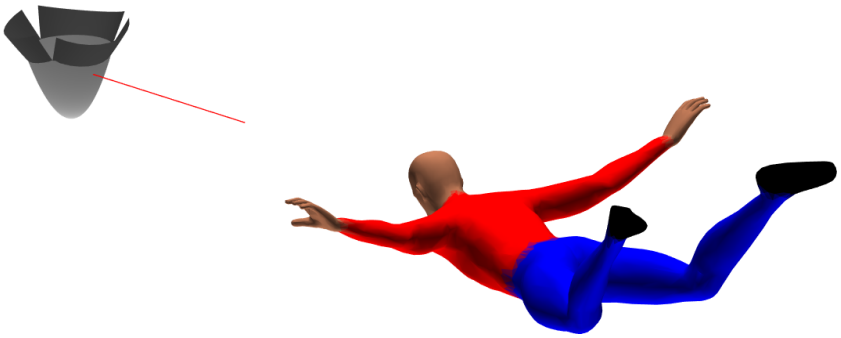


Skydiver following camera UAV

Model and animation

Jörg J. Buchholz*
Karl Stol†

25 March 2017

*Professor, Department of Mechanical Engineering, Hochschule Bremen, Germany; buchholz@hs-bremen.de
†Senior Lecturer, Department of Mechanical Engineering, University of Auckland, New Zealand; k.stol@auckland.ac.nz

Contents

Nomenclature

α	Angle of attack (`al`)
χ	Flight path azimuth (`ch`)
η	Elevator angle (`et`)
γ	Angle of climb (`ga`)
μ	Aerodynamic yaw angle (`mu`)
ρ	Air density (`rh`)
Φ	Bank angle (`ph`)
Ψ	Azimuth angle (`ps`)
Θ	Pitch angle (`th`)
$\boldsymbol{\Omega}$	Rotational velocity vector (`_Om`)
$\boldsymbol{\Phi}$	Euler angle vector (`_Ph`)
$\boldsymbol{C_Q}$	Moment coefficient vector (`_C_Q`)
$\boldsymbol{C_R}$	Force coefficient vector (`_C_R`)
$\boldsymbol{g}$	Gravitational acceleration vector (`_g`)
$\boldsymbol{I}$	Tensor of the moment of inertia (`_I`)
$\boldsymbol{M_{\Phi f}}$	Transformation matrix from the body-fixed frame to the Euler angle frame (`m_phf`)
$\boldsymbol{M_{af}}$	Transformation matrix from the body-fixed frame to the aerodynamic frame (`m_af`)
$\boldsymbol{M_{fa}}$	Transformation matrix from the aerodynamic frame to the body-fixed frame (`m_fa`)
$\boldsymbol{M_{fg}}$	Transformation matrix from the inertial frame to the body-fixed frame (`m_fg`)

$\boldsymbol{M_{gf}}$	Transformation matrix from the body-fixed frame to the inertial frame (`m_gf`)
$\boldsymbol{Q}$	Moment vector (`_Q`)
$\boldsymbol{R}$	Force vector (`_R`)
$\boldsymbol{s}$	Position vector (`_s`)
$\boldsymbol{V}$	Translational velocity vector (`_V`)
ζ	Rudder angle (`ze`)
C_D	Drag coefficient (`C_D`)
C_L	Lift coefficient (`C_L`)
C_m	Pitching moment coefficient (`C_m`)
C_{D0}	Drag coefficient for angle of attack = 0 (`C_D0`)
$C_{D\alpha 2}$	Drag coefficient with respect to square of angle of attack (`C_D_al2`)
$C_{D\alpha}$	Drag coefficient with respect to angle of attack (`C_D_al`)
$C_{D\eta}$	Drag coefficient with respect to elevator (`C_D_et`)
$C_{L\alpha}$	Lift coefficient with respect to angle of attack (`C_L_al`)
$C_{L\eta}$	Lift coefficient with respect to elevator (`C_L_et`)
$C_{l\eta}$	Rolling moment coefficient with respect to elevator (`C_l_et`)
C_{lp}	Rolling moment coefficient with respect to roll rate (`C_l_p`)
$C_{m\alpha}$	Pitching moment coefficient with respect to angle of attack (`C_m_al`)
$C_{m\eta}$	Pitching moment coefficient with respect to elevator (`C_m_et`)
C_{mq}	Pitching moment coefficient with respect to pitch rate (`C_m_q`)
$C_{n\eta}$	Yawing moment coefficient with respect to elevator (`C_n_et`)
C_{nr}	Yawing moment coefficient with respect to yaw rate (`C_n_r`)
$C_{S\eta}$	Side force coefficient with respect to elevator (`C_S_et`)
E	Aerodynamic force unit (`E`)
F	Force (`F`)

k	Shadowing factor (`k`)
l_μ	Reference length (`l_mu`)
m	Mass (`m`)
p	Rolling velocity (`p`)
q	Pitching velocity (`q`)
r	Yawing velocity (`r`)
S	Reference area (`S`)
u	Forward speed (`u`)
v	Side speed (`v`)
w	Sink speed (`w`)
x	x-direction (to front/north ...) (`x`)
y	y-direction (to right/east ...) (`y`)
z	z-direction (down) (`z`)
Index A	Aerodynamic (`_A`)
Index a	Aerodynamic frame (`_a`)
Index c	Command, control, setpoint (`_c`)
Index f	Body-fixed frame (`_f`)
Index g	Inertial frame (`_g`)
Index K	Flight path (`_K`)
Index W	Wind (`_W`)

1. Simulink model

The purpose of the UAV is provided in [1].

1.1. General overview

The main Simulink window including all subsystems is shown in figure 1.1.

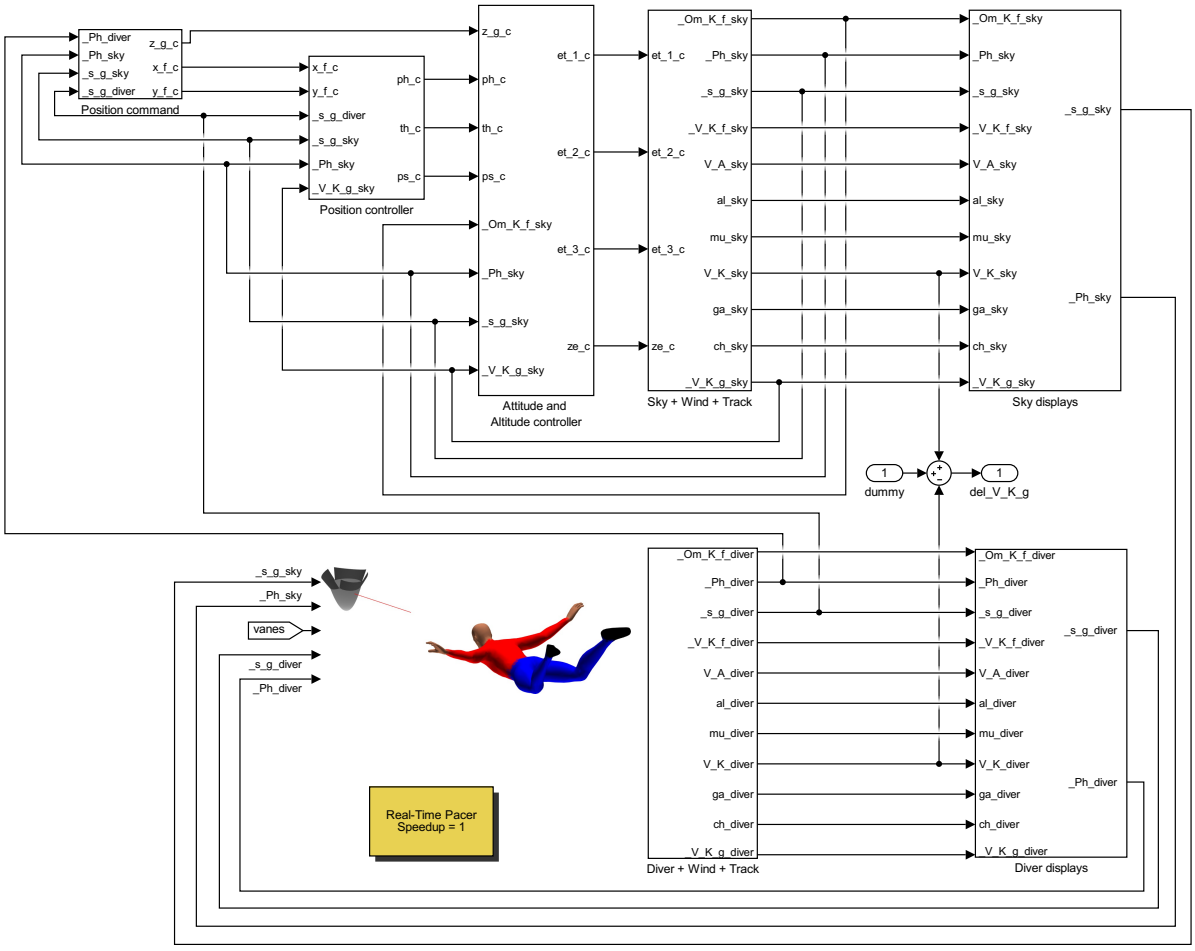


Figure 1.1.: General overview

The top central subsystem Sky[1] + Wind + Track (section 1.2) holds the model of the UAV, its own wind process and a block computing the spherical components of the UAV's flight path vector. The bottom central subsystem Diver + Wind + Track contains the corresponding models of the skydiver.

Scopes for the main signals can be found in Sky displays and Diver displays (section 1.29). An S-Function in the lower left of figure 1.1 is used to display an animation of the UAV and the diver [2] during the simulation. The "yellow" Real-Time Pacer block makes sure that the simulation runs in real-time if the hardware is fast enough.

While the skydiver is uncontrolled and reacts only to its own wind inputs, the UAV is controlled by a cascade control system. The inner (secondary, slave) Attitude and altitude controller measures the attitude and the altitude of the UAV and uses the actuators to maintain their values. The outer (primary, master) Position controller measures the position of the UAV and uses the inner loop as its actuators by commanding an attitude setpoint to the inner controller. The position setpoint for the outer controller and for the altitude is generated in the Position command block.

Additionally, figure 1.1 contains a dummy input, a sum and an output block that are only used during the trim process.

1.2. UAV + wind + track

The top level UAV block (figure 1.2) contains its mathematical model (section 1.3) including actuator dynamics, aerodynamics, and kinetics, its wind process (section 1.16), and the computation of the spherical components of its flight path vector (section 1.20).

[1]The name Sky is used as a synonym for the UAV including the camera throughout this paper.

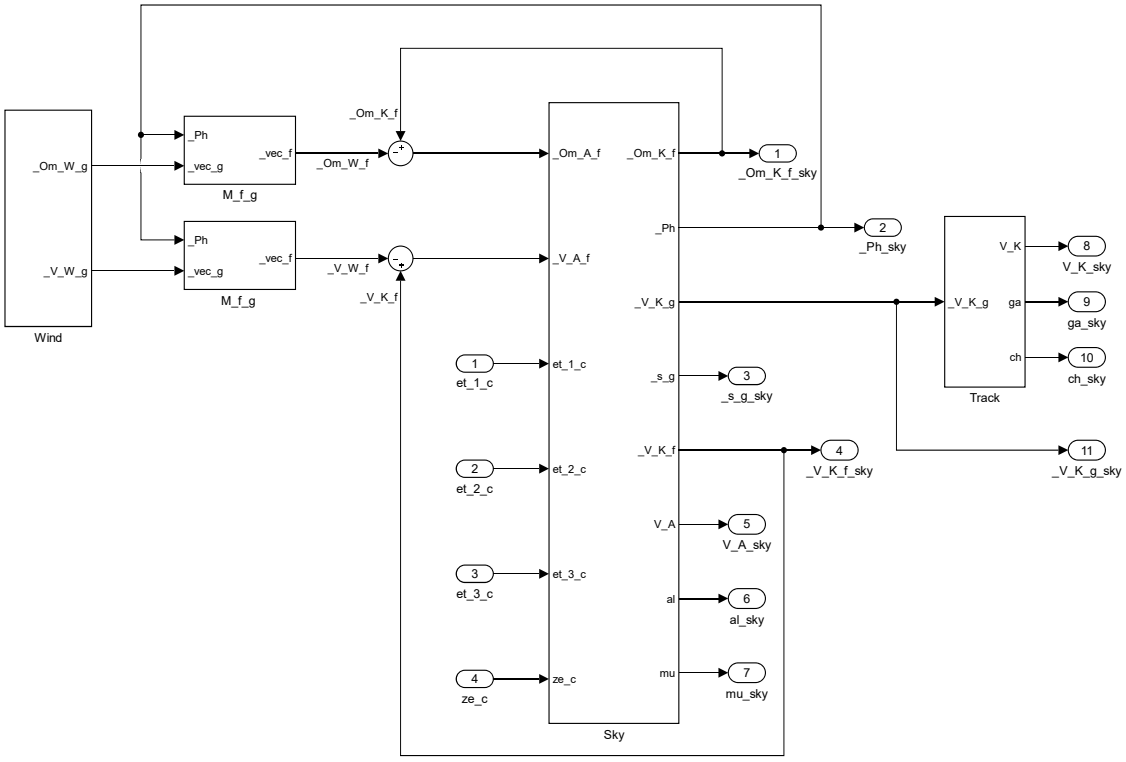


Figure 1.2.: UAV + wind + track

The translational flight path velocity vector $\boldsymbol{V_K}$ (i. e. the velocity of the UAV with respect to the ground) is the sum of the aerodynamic velocity or airspeed vector $\boldsymbol{V_A}$ (i. e. the velocity of the UAV with respect to the air) and the $\boldsymbol{V_W}$ (i. e. the velocity of the air with respect to the ground):

$$\boldsymbol{V_K} = \boldsymbol{V_A} + \boldsymbol{V_W}$$

The same is true for the rotational velocity vectors:

$$\boldsymbol{\Omega_K} = \boldsymbol{\Omega_A} + \boldsymbol{\Omega_W}$$

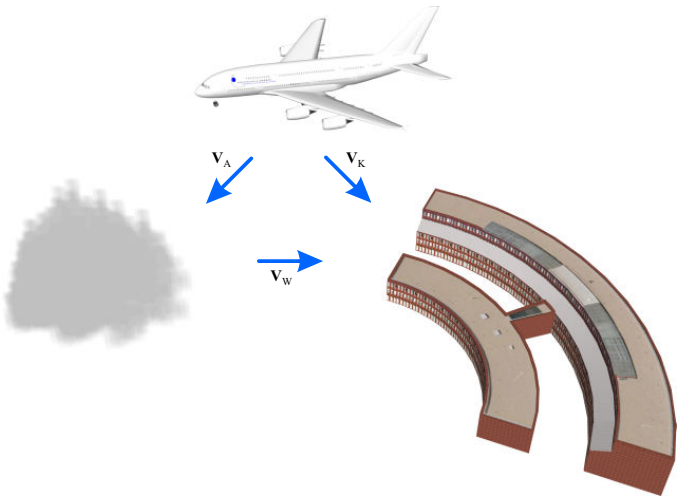


Figure 1.3.: Relation between $\boldsymbol{V_K}$, $\boldsymbol{V_A}$, and $\boldsymbol{V_W}$ [3]

Since we need the airspeed vectors in the body-fixed frame, we have to transform the wind from the inertial frame (index g) to the body-fixed frame (index f) with the help of a transformation block (section 1.26):

$$\begin{aligned}\boldsymbol{V_{Af}} =& \boldsymbol{V_{Kf}} - \boldsymbol{V_{Wf}} \\ =& \boldsymbol{V_{Kf}} - \boldsymbol{M_{fg}} \cdot \boldsymbol{V_{Wg}}\end{aligned}$$

The transformation matrix $\boldsymbol{M_{fg}}$ utilizes the Euler angle vector (attitude)

$$\boldsymbol{\Phi} = \begin{bmatrix} \Phi \\ \Theta \\ \Psi \end{bmatrix}$$

which therefore has to be fed back into the transformation blocks as well.

1.3. UAV overview

The UAV subsystem (figure 1.4) contains an Actuator dynamics block that saturates and rate limits the actuators, an Aerodynamics block that computes the aerodynamic forces $\boldsymbol{R_{Af}}$ and moments $\boldsymbol{Q_{Af}}$, and a Kinetics block that integrates the forces and moments into motion, i. e. the rotational flight path velocity vector in the body-fixed frame $\boldsymbol{\Omega_{Kf}}$, the Euler angle vector (attitude) $\boldsymbol{\Phi}$, the position vector in the inertial frame $\boldsymbol{s_g}$, and the translational flight path velocity vector in the body-fixed frame $\boldsymbol{V_{Kf}}$. Additionally, the flight path velocity vector is returned in the inertial frame $\boldsymbol{V_{Kg}}$. The limited actuator deflections are tunneled to the animation block in General overview via a Goto-block.

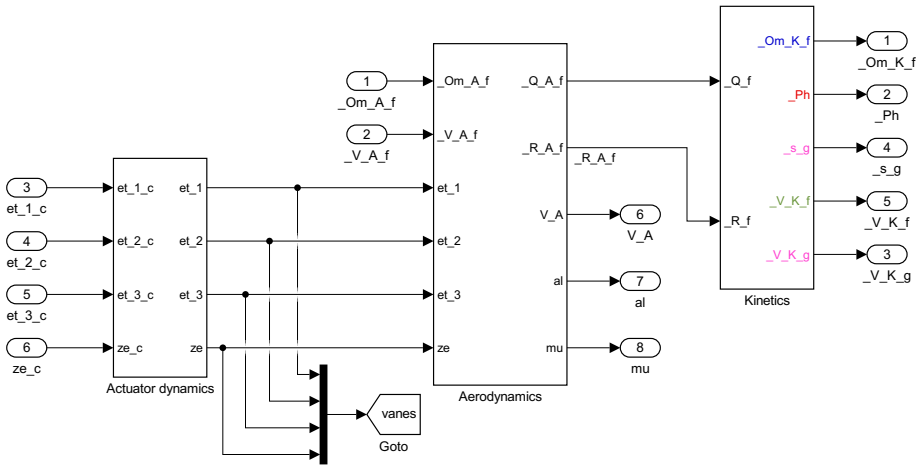


Figure 1.4.: UAV overview

1.4. Actuator dynamics

The UAV has three vanes (figure 1.5) that can be deflected individually (elevators η_1, η_2, and η_3) about a horizontal axis to produce a pitching (or rolling) moment and collectively (rudder ζ) about a vertical axis to produce a yawing moment.

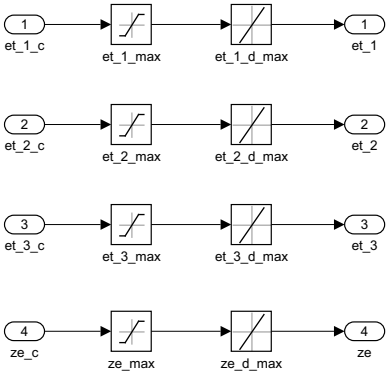


Figure 1.5.: Actuator dynamics [3]

Each elevator deflection is positive, saturated

$$0 \leq \eta \leq \eta_{max}$$

and rate limited:

$$-\dot{\eta}_{max} \leq \dot{\eta} \leq \dot{\eta}_{max}$$

The rudder has symmetrical limits:

$$-\zeta_{max} \leq \zeta \leq \zeta_{max}$$
$$-\dot{\zeta}_{max} \leq \dot{\zeta} \leq \dot{\zeta}_{max}$$

1.5. Aerodynamics

The aerodynamic force vector in the body-fixed frame $\boldsymbol{R_{Af}}$ is computed as the product of the corresponding aerodynamic force coefficient vector $\boldsymbol{C_{RAf}}$ and the Aerodynamic force unit E in figure 1.6:

$$\boldsymbol{R_{Af}} = E \cdot \boldsymbol{C_{RAf}}$$

For the moments we have to multiply the force by a reference length (mean aerodynamic cord) l_μ:

$$\boldsymbol{Q_{Af}} = l_\mu \cdot E \cdot \boldsymbol{C_{QAf}}$$

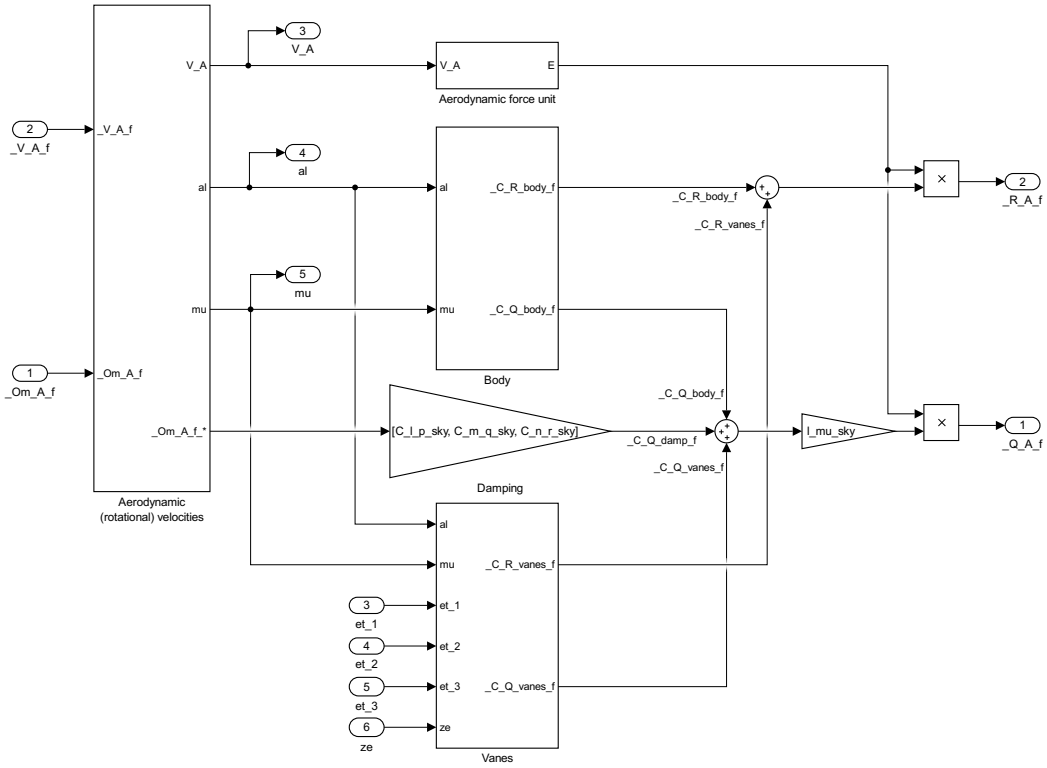


Figure 1.6.: Aerodynamics

In the subsystem Aerodynamic velocities we compute the spherical components (V_A, α, and μ) from the airspeed vector in the body-fixed frame $\boldsymbol{V_{Af}}$ and make the rotational aerodynamic velocity vector $\boldsymbol{\Omega_{Af}}$ dimensionless. The aerodynamic coefficients are computed as the sum of the coefficients of the Body, the Vanes and a linear damping:

$$\boldsymbol{C_{RAf}} = \boldsymbol{C_{RAf_{body}}} + \boldsymbol{C_{RAf_{vanes}}}$$
$$\boldsymbol{C_{QAf}} = \boldsymbol{C_{QAf_{body}}} + \boldsymbol{C_{QAf_{vanes}}} + \boldsymbol{C_{QAf_{damp}}}$$

where the linear damping coefficient $\boldsymbol{C_{QAf_{damp}}}$ is the product of the diagonal damping matrix and the dimensionless rotational aerodynamic velocity vector $\boldsymbol{\Omega^*_{Af}}$:

$$\boldsymbol{C_{QAf_{damp}}} = \begin{bmatrix} C_{lp} & 0 & 0 \\ 0 & C_{mq} & 0 \\ 0 & 0 & C_{nr} \end{bmatrix} \boldsymbol{\Omega^*_{Af}}$$

1.6. Aerodynamic velocities

In the subsystem depicted in figure 1.7 we compute the spherical components (V_A,α, and μ) of the flight path vector from its Cartesian representation $\boldsymbol{V_{Af}}$ according to equation (A.7) and equation (A.8).

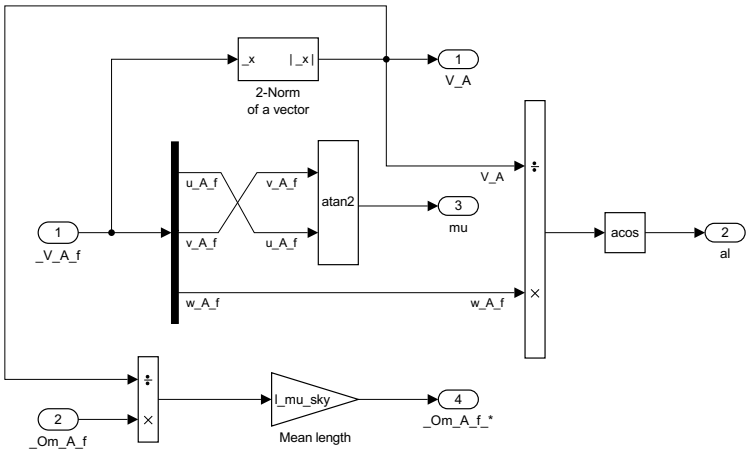


Figure 1.7.: Aerodynamic velocities [3]

Additionally, we normalize the rotational velocity vector by a "time unit" in order to make it dimensionless:

$$\boldsymbol{\Omega^*_{Af}} = \frac{l_\mu}{V_A} \cdot \boldsymbol{\Omega_{Af}}$$

1.7. Aerodynamic force unit

The aerodynamic force unit (figure 1.8) represents a force that is proportional to the air density ρ, the square of the air speed V_A and a reference area S:

$$E = \frac{\rho}{2} \cdot V_A^2 \cdot S$$

Figure 1.8.: Aerodynamic force unit [3]

It is used in section 1.5 to compute the aerodynamic forces and moments from the dimensionless coefficients.

1.8. Body

In figure 1.9 we compute the aerodynamic force and moment coefficients of the UAV's body with respect to the angle of attack α and the aerodynamic yaw angle μ. According to appendix A the aerodynamic forces $\boldsymbol{C_{Ra_{body}}}$ and moments $\boldsymbol{C_{Qa_{body}}}$ of a symmetrical body only depend on the angle of attack in the aerodynamic frame. The aerodynamic yaw angle μ is then used (together with α) to transform the forces and moments to the body-fixed frame via equation (A.4).

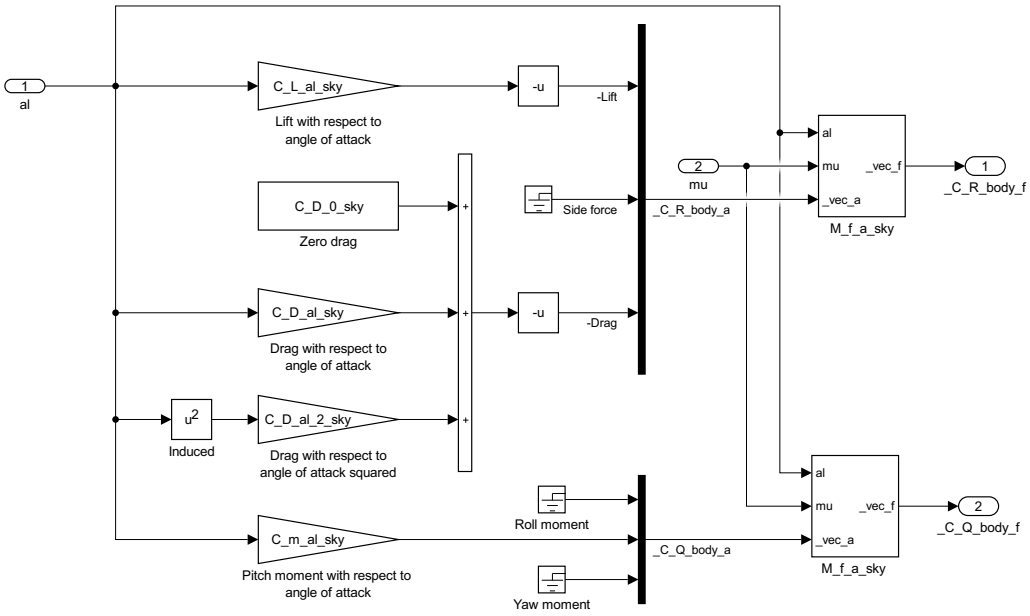


Figure 1.9.: Aerodynamics body

In this simple linear derivative aerodynamics, the lift coefficient $C_{L_{body}}$ (pointing towards the negative x-axis of the aerodynamic frame) is proportional to the angle of attack with a constant lift slope derivative $C_{L\alpha}$. Due to the axial symmetry of the body there is no lift for $\alpha = 0$. Also, the pitch moment coefficient C_m only depends on the angle of attack, with the pitch stability derivative $C_{m\alpha}$ as a proportionality factor:

$$C_m = C_{m\alpha} \cdot \alpha$$

With the drag coefficient $C_{D_{body}}$ (pointing towards the negative z-axis of the aerodynamic frame) we have to be a little bit more flexible; Not only do we have to consider a drag coefficient C_{D0} for $\alpha = 0$ because of the face area, but we also have to take into account that the induced drag is proportional to the square of the lift. Therefore, we assume a full quadratic dependence of the drag coefficient C_D on the angle of attack:

$$C_D = C_{D0} + C_{D\alpha} \cdot \alpha + C_{D\alpha 2} \cdot \alpha^2$$

In the aerodynamic frame defined in appendix A there is no aerodynamic side force, roll moment, or yaw moment.

1.9. Vanes

While the aerodynamics of the body is modeled in the aerodynamic frame and later transformed into the body-fixed frame, we compute the aerodynamic forces and moments (coefficients) of the vanes directly in the body-fixed frame (figure 1.10).

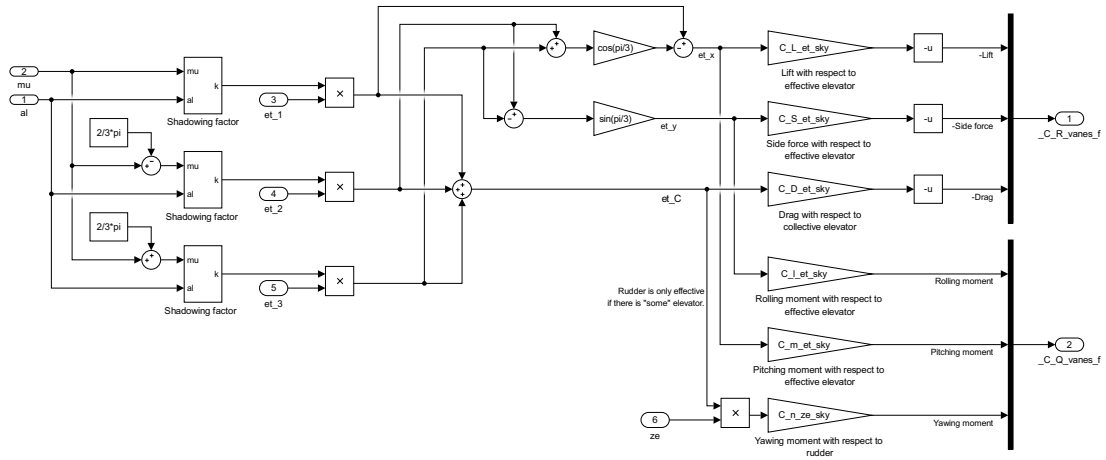


Figure 1.10.: Aerodynamic vanes

The arrangement of the three vanes of the UAV are depicted in figure 1.11 looking down in positive z_f-direction.

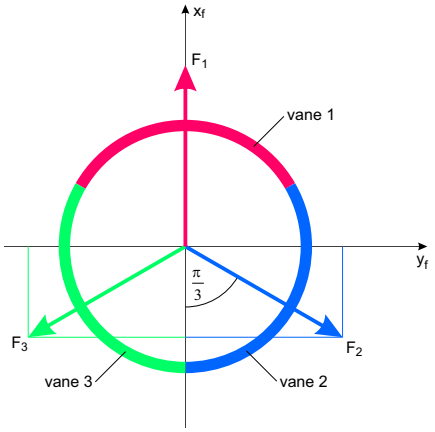


Figure 1.11.: Vanes and forces as seen from above

When deflected by its "elevator" deflection angle (η_1, η_2, η_3) each vane generates a force (F_1, F_2, F_3) in its deflection direction[2] and a moment about the perpendicular axis. Additionally, all vanes generate drag forces in (negative) z_f-direction.

In order to use the minimum number of derivatives we decompose the vane deflection angles into their effective angles in x_f- and y_f-direction.[3]

The computation of the effective angles according to figure 1.11 is done in the upper middle part of figure 1.10:

$$\eta_x = \eta_1 - \cos\left(\frac{\pi}{3}\right)(\eta_2 + \eta_3) \tag{1.1}$$

$$\eta_y = \sin\left(\frac{\pi}{3}\right)(\eta_2 - \eta_3) \tag{1.2}$$

Additionally, we compute an effective collective angle for the drag:

$$\eta_C = \eta_1 + \eta_2 + \eta_3 \tag{1.3}$$

Now we can compose the aerodynamic force coefficient vector in the body-fixed frame:

$$\boldsymbol{C_{Rf}} = \begin{bmatrix} -C_{L\eta} \cdot \eta_x \\ -C_{S\eta} \cdot \eta_y \\ -C_{D\eta} \cdot \eta_C \end{bmatrix} \tag{1.4}$$

The moments generated by the vanes are computed accordingly: the effective x-vane deflection generates a pitching moment about the y_f-axis while an effective y-vane deflection results in a rolling moment about the x_f-axis. The yawing moment about the

[2]The force can be negative. Its sign is consequently taken into account when the coefficients are sorted into the corresponding vector $\boldsymbol{C_{Rf}}$ in equation (1.4).

[3]Since the aerodynamic forces and moments linearly depend on the vane deflections in this simple model we can superpose the angles instead of the forces.

z_f-axis is caused by the concurrent deflection of all vanes about their own z-axis with the "rudder" deflection angle ζ.

If the vane has an elevator deflection angle of $\eta = 0$ a rudder deflection ζ does not have any effect. With increasing elevator the rudder becomes more effective. In a quick-and-dirty[4] implementation we implement this behavior by a product of both deflections (bottom middle in figure 1.10). We can then compose the aerodynamic moment coefficient vector in the body-fixed frame:

$$\boldsymbol{C_{Qf}} = \begin{bmatrix} C_{l\eta} \cdot \eta_y \\ C_{m\eta} \cdot \eta_x \\ C_{n\zeta} \cdot \zeta \cdot \eta_D \end{bmatrix}$$

1.10. Shadowing factor of the aerodynamic vanes

If the aerodynamic yaw angle $\mu = \pm\pi$ and the angle of attack $\alpha > 0$ (i. e. the apparent wind is coming from the "back") the first ("front") vane is shadowed by the body of the UAV and therefore less effective. We model this effectiveness reduction by a vane shadowing factor k (of the first vane) in the left part of figure 1.10 in the subsystem depicted in figure 1.12.

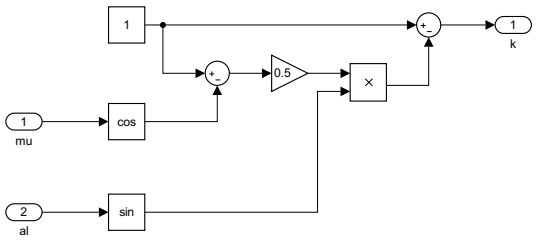


Figure 1.12.: Aerodynamic vanes shadowing factor

The shadowing factor k depends on α and μ:

$$k = 1 - \frac{1 - \cos(\mu)}{2} \cdot \sin(\alpha) \tag{1.5}$$

If $\alpha = 0$ the apparent wind is flowing directly in (negative) z_f-direction and no vane effectiveness reduction should be present. Therefore, $k = 1$ according to equation (1.5). The same is true for $\mu = 0$, i. e. the apparent wind is directly hitting the first vane. The trigonometric functions used in equation (1.5) ensure a steady effectiveness reduction down to no effectiveness at $\alpha = \frac{\pi}{2} \wedge \mu = \pm\pi$.

[4] One of the first optimizations of the aerodynamic model should start here to model the relationship between yawing moment, elevator and rudder in a more realistic way.

The surface plot in figure 1.13 illustrates the relations between α, μ, and k.

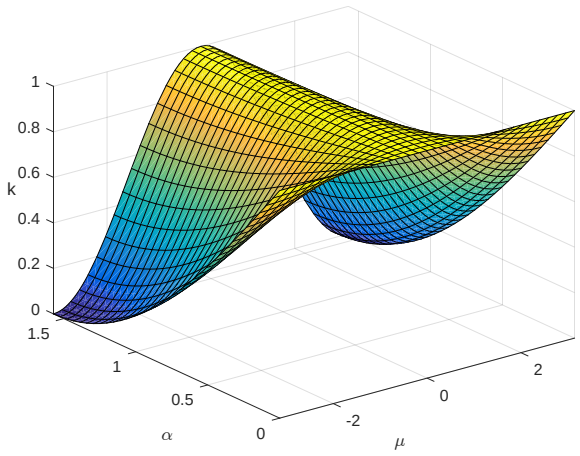


Figure 1.13.: Shadowing factor surface plot

The considerations above are valid for the first vane. For the other vanes we add $\pm\frac{2}{3}\pi$ to the aerodynamic yaw angle in figure 1.12 before computing their shadowing factor.

1.11. Kinetics

The kinetics subsystem in figure 1.14 computes the following motion state vectors:

- translational velocity vector in the body-fixed frame $\boldsymbol{V_{Kf}}$
- position vector in the inertial frame $\boldsymbol{s_g}$
- rotational velocity vector in the body-fixed frame $\boldsymbol{\Omega_{Kf}}$
- attitude vector $\boldsymbol{\Phi}$

of the UAV from the forces $\boldsymbol{R_f}$ and moments $\boldsymbol{Q_f}$ acting on the mass. It contains four (three-dimensional) vector integrators and four subsystems in which we model the non-linear 6-DOF vector differential equations (Translational velocity differential equation, Position differential equation, Rotational velocity differential equation, and Attitude differential equation).

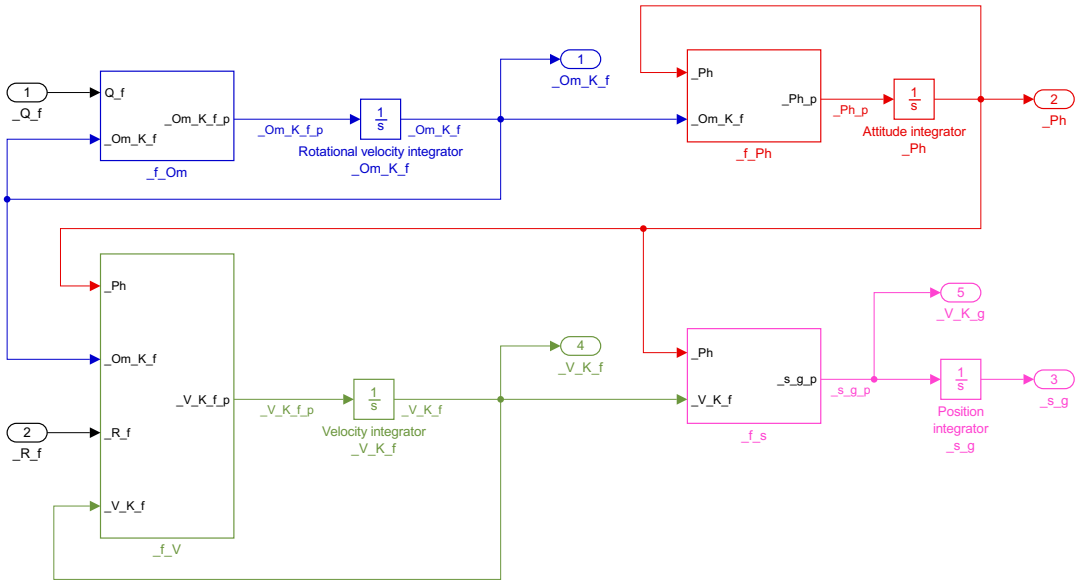


Figure 1.14.: Kinetics [3]

1.12. Translational velocity differential equation

The translational velocity vector differential equation expressed in the body-fixed frame

$$\dot{\boldsymbol{V}}_{\boldsymbol{Kf}} = \frac{\boldsymbol{R_f}}{m} + \boldsymbol{M_{fg}} \cdot \boldsymbol{g_g} - \boldsymbol{\Omega_{Kf}} \times \boldsymbol{V_{Kf}}$$

modeled in figure 1.15 computes the translational acceleration vector $\dot{\boldsymbol{V}}_{\boldsymbol{Kf}}$ from the force vector $\boldsymbol{R_f}$.

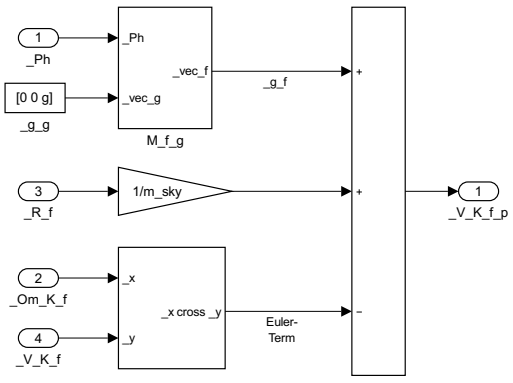


Figure 1.15.: Velocity differential equation [3]

1.13. Position differential equation

The position vector differential equation expressed in the inertial frame

$$\dot{\boldsymbol{s}}_{\boldsymbol{g}} = \boldsymbol{V}_{\boldsymbol{Kg}} = \boldsymbol{M}_{\boldsymbol{gf}} \cdot \boldsymbol{V}_{\boldsymbol{Kf}}$$

modeled in figure 1.16 computes the derivative of the position vector in the inertial frame $\dot{\boldsymbol{s}}_{\boldsymbol{g}}$ from the velocity vector in the body-fixed frame $\boldsymbol{V}_{\boldsymbol{Kf}}$.

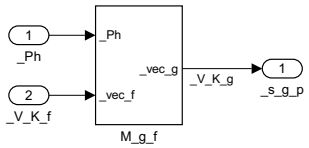


Figure 1.16.: Position differential equation [3]

1.14. Rotational velocity differential equation

The translational velocity vector differential equation expressed in the body-fixed frame

$$\dot{\boldsymbol{\Omega}}_{\boldsymbol{Kf}} = \boldsymbol{I}_{\boldsymbol{f}}^{-\mathbf{1}} \cdot (\boldsymbol{Q}_{\boldsymbol{f}} - \boldsymbol{\Omega}_{\boldsymbol{Kf}} \times (\boldsymbol{I}_{\boldsymbol{f}} \cdot \boldsymbol{\Omega}_{\boldsymbol{Kf}}))$$

modeled in figure 1.17 computes the rotational acceleration vector $\dot{\boldsymbol{\Omega}}_{\boldsymbol{Kf}}$ from the moment vector $\boldsymbol{Q}_{\boldsymbol{f}}$.

$\boldsymbol{I}_{\boldsymbol{f}}$ is the tensor of the moment of inertia in the body-fixed frame.

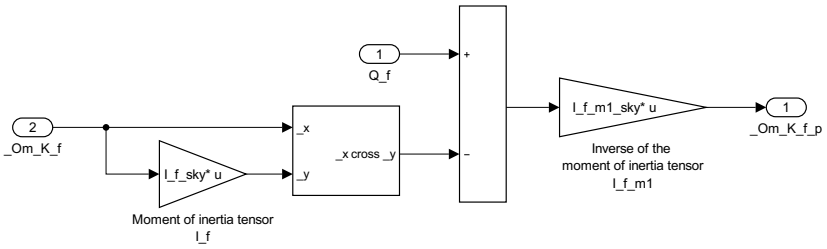


Figure 1.17.: Rotational velocity differential equation [3]

1.15. Attitude differential equation

The attitude vector differential equation

$$\dot{\boldsymbol{\Phi}} = \begin{bmatrix} \dot{\Phi} \\ \dot{\Theta} \\ \dot{\Psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin\Phi\tan\Theta & \cos\Phi\tan\Theta \\ 0 & \cos\Phi & -\sin\Phi \\ 0 & \sin\Phi/\cos\Theta & \cos\Phi/\cos\Theta \end{bmatrix} \boldsymbol{\Omega}_{\boldsymbol{Kf}} = \boldsymbol{M}_{\boldsymbol{\Phi f}} \cdot \boldsymbol{\Omega}_{\boldsymbol{Kf}} \tag{1.6}$$

modeled in figure 1.18 computes the derivative of the attitude (Euler angle) vector $\dot{\boldsymbol{\Phi}}_{\boldsymbol{Kf}}$ from the rotational velocity vector in the body-fixed frame $\boldsymbol{\Omega}_{\boldsymbol{Kf}}$.

The non-orthogonal transformation matrix $\boldsymbol{M}_{\boldsymbol{\Phi f}}$ (section 1.27) contains quotients that can result in "divide-by-zero" if $\Theta = \pm\frac{\pi}{2}$ or "undefined" gimbal lock situations if additionally $\Phi = \pm\frac{\pi}{2}$ or $\Phi = 0$.

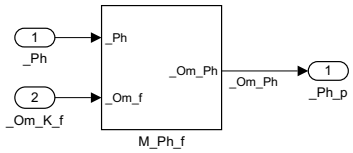


Figure 1.18.: Attitude differential equation [3]

1.16. Wind

The simple wind model in figure 1.19 generates Translational wind, Turbulence, and Rotational wind (hurricane).

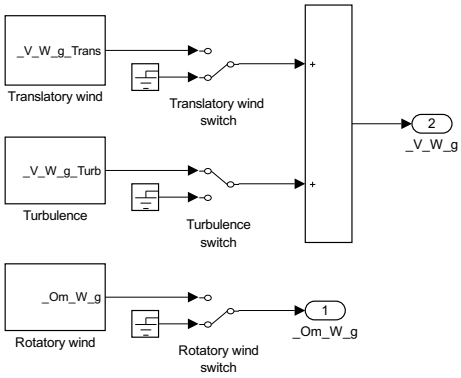


Figure 1.19.: Wind [3]

All three wind vectors can be switched on or off separately.

1.17. Translational wind

The translational wind in figure 1.20 is just a three-dimensional constant.

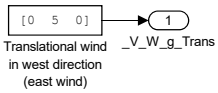


Figure 1.20.: Translational wind [3]

A north blowing south wind of $5\,\frac{\mathrm{m}}{\mathrm{s}}$ would therefore just be $\begin{bmatrix}5 & 0 & 0\end{bmatrix}$; or we could model a thermal lift of $10\,\frac{\mathrm{m}}{\mathrm{s}}$ by $\begin{bmatrix}0 & 0 & -10\end{bmatrix}$.

1.18. Turbulence

The simple turbulence model in figure 1.21 uses three Band-Limited White Noise blocks to generate random numbers for all three components of the turbulence vector and three first order low pass filters to shape the spectrum of the turbulence to be more realistic.

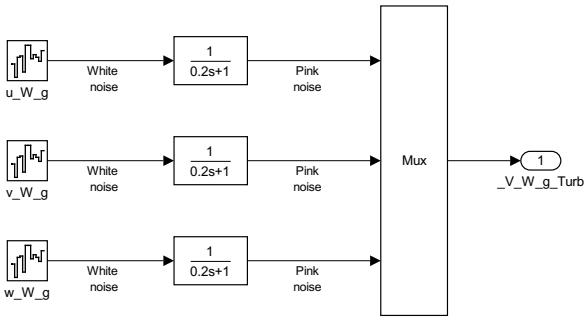


Figure 1.21.: Turbulence [3]

1.19. Rotational wind

We can use the three-dimensional constant vector in figure 1.22 e. g. to model a clockwise rotating cyclone by $\begin{bmatrix}0 & 0 & 50\end{bmatrix}$ or a single wake vortex in an easterly direction by $\begin{bmatrix}0 & 15 & 0\end{bmatrix}$.

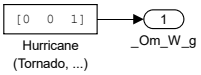


Figure 1.22.: Rotational wind [3]

1.20. Track

The subsystem in figure 1.23 is used to compute the spherical components of the flight path vector:

- absolute value of the flight path velocity vector (ground speed) V_K
- angle of climb γ
- flight path azimuth χ

from its Cartesian coordinates according to:

$$V_K = \sqrt{u_{Kg}^2 + v_{Kg}^2 + w_{Kg}^2}$$

$$\gamma = -\arcsin\left(\frac{w_{Kg}}{V_K}\right)$$

$$\chi = \arctan\left(\frac{v_{Kg}}{u_{Kg}}\right)$$

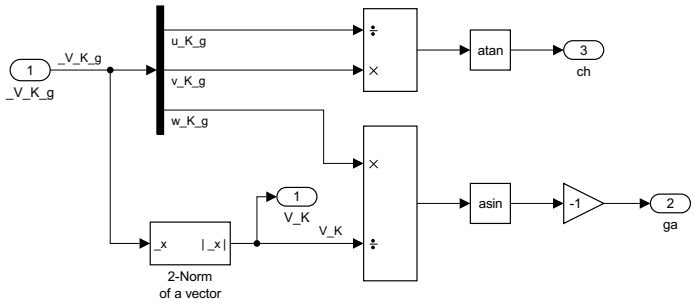


Figure 1.23.: Track [3]

1.21. Attitude and altitude controller

The attitude and altitude controller in figure 1.24 is the inner loop controller of the UAV's cascade control system.

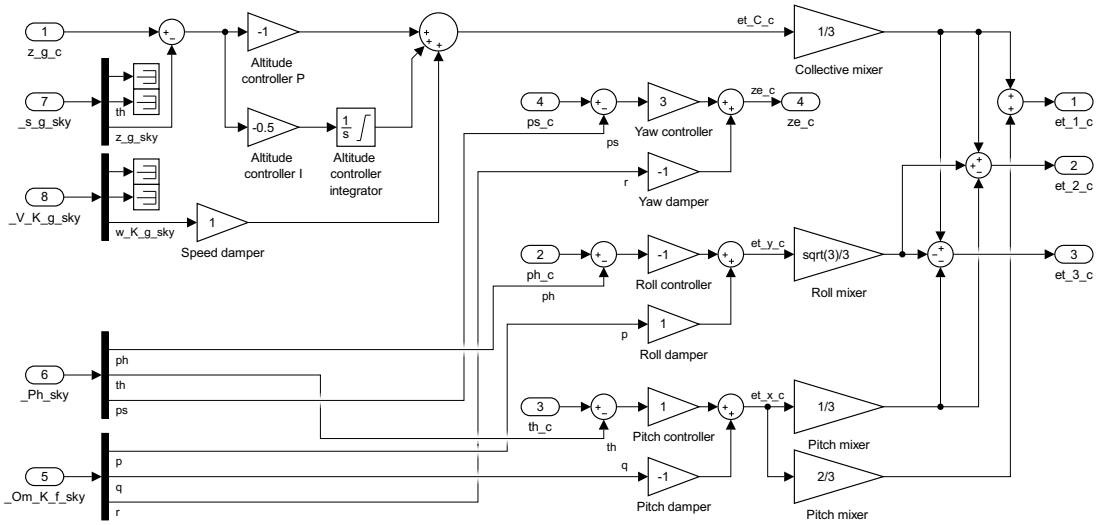


Figure 1.24.: Attitude and altitude controller

It contains the following controllers:

- yaw controller to control the azimuth Ψ, with feedback of the yawing velocity r in the yaw damper;
- roll controller to control the bank angle Φ, with feedback of the rolling velocity p in the roll damper;
- pitch controller to control the pitch angle Θ, with feedback of the pitching velocity q in the pitch damper;
- altitude controller to control the altitude $-z_g$, with feedback of the sink rate w_{Kg} in the speed damper.

The attitude controllers are pure proportional controllers; for inner controllers we can live with a reasonable steady-state control error. The yaw controller even holds its azimuth setpoint with a zero rudder deflection because of the integral behavior of the plant. All attitude controllers implement a feedback of the corresponding rotational velocities for additional damping purposes.

For the altitude controller we use a PI controller with a limited integrator to prevent wind-up for large control errors. Additionally, we feed back the sink rate to dampen overshooting of the altitude. Since the sink rate of the UAV is not zero in the steady-state case, the speed damper could be implemented as a high pass filter; in our implementation the integrator in the PI controller compensates the steady-state error introduced by the speed damper.

The altitude controller uses the elevator deflection angles $\eta_{1,2,3}$ of all three vanes concurrently to control the drag and therefore the vertical speed and altitude of the UAV (right part of figure 1.24) in order to keep the UAV at the skydiver's altitude. We assume here that the altitude controller does not know the current vertical speed of the skydiver.

The yaw controller uses the concurrent rudder deflection angles ζ of all vanes to control the azimuth of the UAV (e. g. in order to point the camera towards the skydiver).

The pitch, roll, and collective mixer distribute the effective pitch, roll and collective elevator commands η_x, η_y, and η_D to the vanes η_1, η_2, and η_3. We can use the relations between both groups of elevator commands according to equations (1.1 - 1.3) to define

```
syms et_x et_y et_C et_1 et_2 et_3

g1 = et_x == et_1 - cos (pi/3)*(et_2 + et_3)
g2 = et_y == sin (pi/3)*(et_2 - et_3)
g3 = et_C == et_1 + et_2 + et_3
```

and solve the equation system for η_1, η_2, and η_3 with the help of the symbolic[5] toolbox:

```
[et_1, et_2, et_3] = solve (g1, g2, g3, et_1, et_2, et_3)
```

[5] In the end, we are talking about a simple linear equation system that we could easily solve numerically (or even by hand); conveniently, the symbolic toolbox automatically returns the coefficients in a nice symbolic form.

```
et_1 = simplify (et_1)
et_2 = simplify (et_2)
et_3 = simplify (et_3)
```

The result

```
et_1 =

et_C/3 + (2*et_x)/3


et_2 =

et_C/3 - et_x/3 + (3^(1/2)*et_y)/3


et_3 =

et_C/3 - et_x/3 - (3^(1/2)*et_y)/3
```

looks reasonable: the collective command η_C is distributed equally to all three vanes. The pitch command η_x goes to the first (front) vane with a factor of $\frac{2}{3}$ and to both other (side-back) vanes with a factor of $\frac{1}{3}$ and an opposite sign. The roll command η_y does not affect the front vane but only the right vane with a positive sign and the left vane with the same magnitude[6] but opposite sign.

1.22. Position controller

The position controller in figure 1.25 is the outer loop controller of the UAV's cascade control system.

[6]The overall magnitude of the coefficients is not really important since it can always be compensated by the controller gains; it's their relations that count.

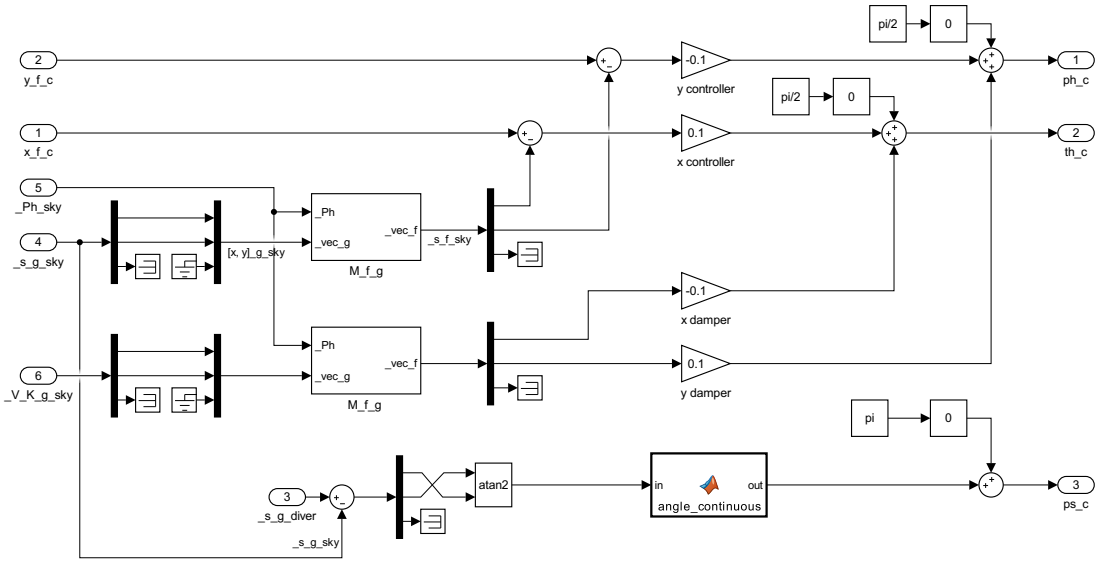


Figure 1.25.: Position controller

It contains the following controllers:

- x controller using the pitch angle command Θ_c as an input to the underlying attitude controller to control the x-position of the UAV, with feedback of the forward velocity u_{kf} in the x damper;
- y controller using the bank angle command Φ_c as an input to the underlying attitude controller to control the y-position of the UAV, with feedback of the forward velocity v_{kf} in the y damper;
- setpoint generator using the yaw angle command Ψ_c as an input to the underlying attitude controller to control the azimuth of the UAV to look towards the skydiver.

Both position controllers are pure proportional controllers; the integral nature of the underlying inner loop makes the outer position control loop steady-state error free. Feedback of the corresponding velocities is used for additional damping purposes. Since the position controllers command pitch and roll setpoints to the inner attitude controllers we have to control the position in the body-fixed frame. Therefore, we have to transform the actual position $\boldsymbol{s_g}$ to the position in the body-fixed frame $\boldsymbol{s_f}$ using the transformation block $\boldsymbol{M_{fg}}$ in figure 1.25. It is very important to set the z-component of the position to zero **before** the transformation. We only want to control the position in the horizontal inertial plane; the altitude control is done in the inner control loop. The same is true for the dampers: we extract the horizontal projection of the flight path velocity vector onto the horizontal inertial plane by zeroing its z-component and then transform the projection into the body-fixed frame.

We want the camera inside the UAV to "look" at the skydiver. Since the camera is aligned with the x_f-direction of the UAV, we have to align its x_f-axis with the vector from the UAV to the skydiver ($\boldsymbol{\Delta_{SU}}$ in figure 1.26). Therefore, we have to command a

yaw angle Ψ_c to the inner azimuth control loop that is computed as the arctangent of the ratio of the second (y_{SU}) and the first component (x_{SU}) of the vector from the UAV to the skydiver:

$$\Psi_c = \arctan\left(\frac{y_{SU}}{x_{SU}}\right)$$

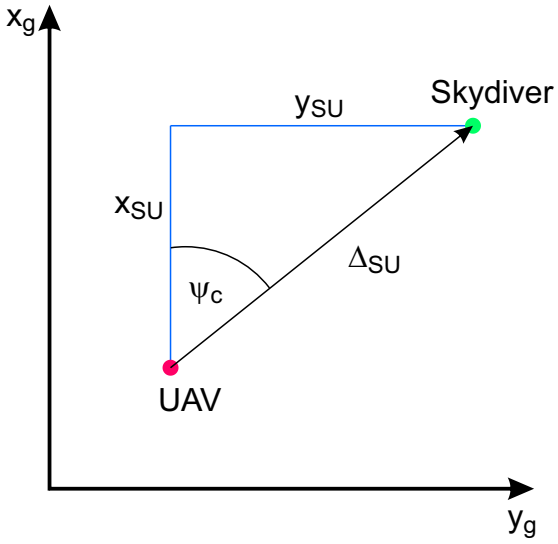


Figure 1.26.: Yaw angle command

Since the yaw angle command is computed as a four quadrant arctangent (`atan2`) its absolute value cannot exceed π. Therefore, the angle would "jump" from a value just less than π to a value just greater than $-\pi$ and vice versa. This would result in unexpected and unwanted rotations of the UAV if the unwrapped yaw angle exceeded $\pm\pi$. To solve this problem we use a tiny MATLAB function (`angle_continuous`) in figure 1.25

```
function out = angle_continuous (in)
```

that defines two persistent variables (`old` is the previous input value, `store` is the $\pm 2\pi$ offset counter)

```
persistent old store
```

that we have to initialize in the first call of the function:

```
if isempty (old)

  old = 0;

end

if isempty (store)
```

```
  store = 0;

end
```

We compute the difference between the current and the precious input value

```
delta = in - old;
```

and if there is a step with a size of more than 5 (in one simulation interval) we assume that this does not have physical reasons but is a result of the periodicity of the arctangent. If we detect a jump from $-\pi$ to $+\pi$

```
if delta > 5
```

we decrement the persistent offset counter `store` by 2π:

```
  store = store - 2*pi;
```

In case of a negative jump

```
elseif delta < -5
```

we increment the counter:

```
  store = store + 2*pi;

end
```

Finally, we save the current input value in the persistent variable `old` to be used as the previous value of the next call

```
old = in;
```

and return the current input value corrected by the accumulated multiples of 2π:

```
out = in + store;
```

1.23. Position command

The subsystem in figure 1.27 generates the position commands to be used in the Position controller and in the Attitude and altitude controller

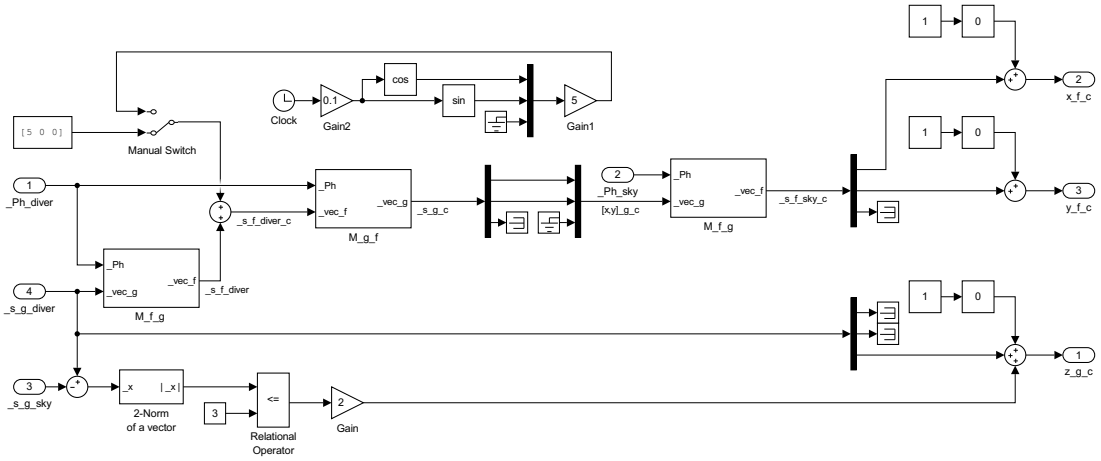


Figure 1.27.: Position command

The general idea is to keep the UAV at the same altitude as the skydiver and position it five meters in front of the skydivers head.[7] For that purpose, we

- transform the position of the skydiver from the inertial frame to the skydivers body-fixed frame,
- add five meters to the x_f-component,
- transform the position back to the inertial frame,
- get rid of the z_g-component because we want to control the position in the horizontal x_g-y_g-plane only,
- transform to the body-fixed frame again because we need the setpoints x_{fc} and y_{fc} in body-fixed coordinates.

Alternatively, there is a switch in the upper left of figure 1.27 that activates a demo setpoint generator that sends the UAV on a circular path around the skydiver.

Usually, the UAV should stay at the skydiver's altitude. Additionally, a very simple collision detection and prevention algorithm in the bottom of figure 1.27 sends the UAV two meters below the skydiver as soon as the distance between UAV and skydiver becomes less than three meters.

1.24. 2-norm of a vector

The subsystem in figure 1.28 is just a helper function and computes the 2-norm of a vector:

$$x = |\boldsymbol{x}| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} = \sqrt{\boldsymbol{x} \cdot \boldsymbol{x}}$$

[7] If the skydiver himself yaws at a high rate, the UAV might not be able to follow his head on a 5 m-circle with a high circumferential speed. It might therefore be more appropriate for the UAV to just keep any position at a predefined horizontal distance from the skydiver's center of mass.

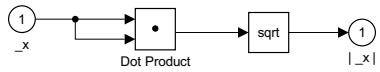


Figure 1.28.: 2-norm of a vector

1.25. Rotation about x-axis

The transformation matrix $\boldsymbol{M_x}$ of a frame rotating about an x-axis with an angle of w_x reads:

$$\boldsymbol{M_x} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos w_x & \sin w_x \\ 0 & -\sin w_x & \cos w_x \end{bmatrix}$$

It can be generated via figure 1.29.

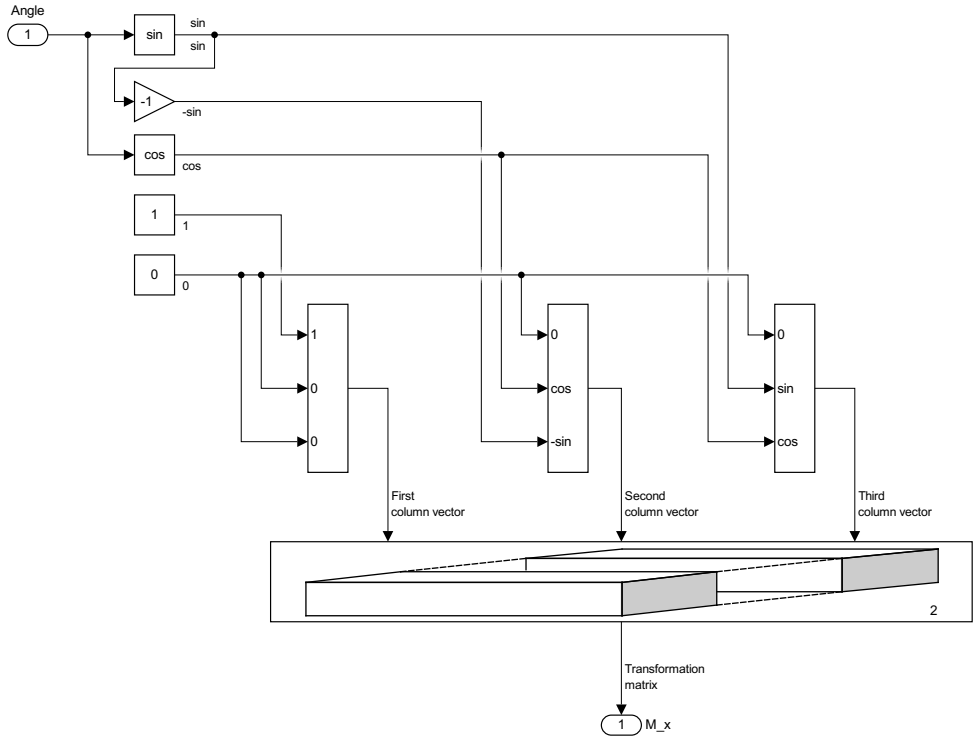


Figure 1.29.: Rotation about x-axis [3]

The corresponding transformation matrices for the rotations about a y- and a z-axis are:

$$\boldsymbol{M_y} = \begin{bmatrix} \cos w_y & 0 & -\sin w_y \\ 0 & 1 & 0 \\ \sin w_y & 0 & \cos w_y \end{bmatrix}$$

$$\boldsymbol{M_z} = \begin{bmatrix} \cos w_z & \sin w_z & 0 \\ -\sin w_z & \cos w_z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

1.26. Transformation from inertial frame to body-fixed frame

Using the single transformation matrices defined in section 1.25, the total transformation matrix including all three transformations in the order $w_z \rightarrow w_y \rightarrow w_x$ is:

$$\begin{aligned} \boldsymbol{M_{tot}} &= \boldsymbol{M_x} \cdot \boldsymbol{M_y} \cdot \boldsymbol{M_z} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos w_x & \sin w_x \\ 0 & -\sin w_x & \cos w_x \end{bmatrix} \begin{bmatrix} \cos w_y & 0 & -\sin w_y \\ 0 & 1 & 0 \\ \sin w_y & 0 & \cos w_y \end{bmatrix} \begin{bmatrix} \cos w_z & \sin w_z & 0 \\ -\sin w_z & \cos w_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{1.7}$$

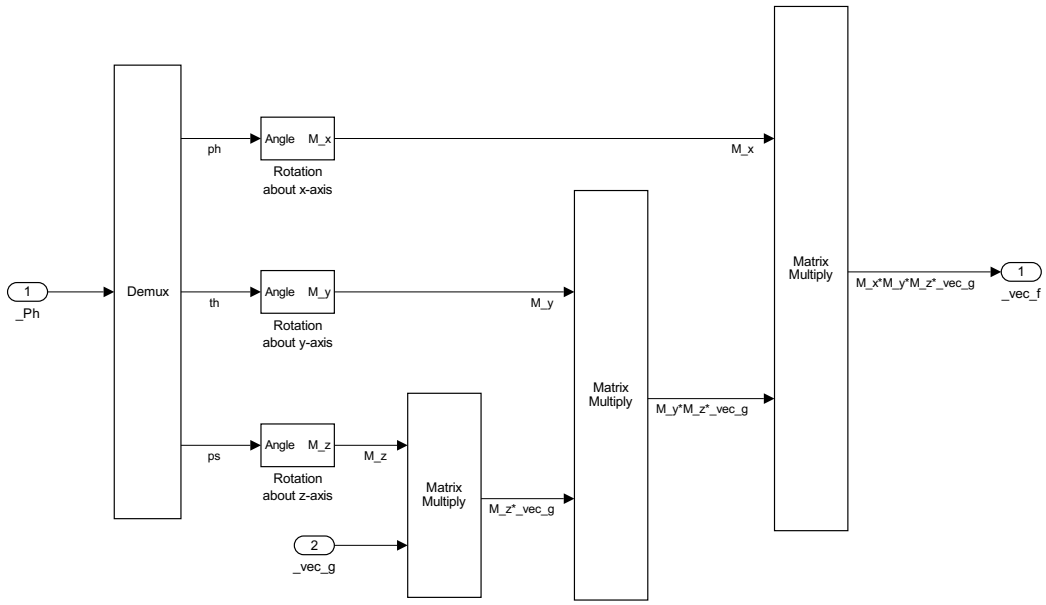


Figure 1.30.: Transformation from inertial frame to body-fixed frame [3]

We can then use equation (1.7) and the Euler angles Ψ, Θ, and Φ in figure 1.30

$$\begin{aligned} \boldsymbol{M_{fg}} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\Phi & \sin\Phi \\ 0 & -\sin\Phi & \cos\Phi \end{bmatrix} \begin{bmatrix} \cos\Theta & 0 & -\sin\Theta \\ 0 & 1 & 0 \\ \sin\Theta & 0 & \cos\Theta \end{bmatrix} \begin{bmatrix} \cos\Psi & \sin\Psi & 0 \\ -\sin\Psi & \cos\Psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos\Theta\cos\Psi & \cos\Theta\sin\Psi & -\sin\Theta \\ \sin\Phi\sin\Theta\cos\Psi - \cos\Phi\sin\Psi & \sin\Phi\sin\Theta\sin\Psi + \cos\Phi\cos\Psi & \sin\Phi\cos\Theta \\ \cos\Phi\sin\Theta\cos\Psi + \sin\Phi\sin\Psi & \cos\Phi\sin\Theta\sin\Psi - \sin\Phi\cos\Psi & \cos\Phi\cos\Theta \end{bmatrix} \end{aligned}$$

to transform any vector $\boldsymbol{v_g}$ from the inertial frame to its representation in the body-fixed frame $\boldsymbol{v_f}$ according to figure 1.31:

$$\boldsymbol{v_f} = \boldsymbol{M_{fg}} \cdot \boldsymbol{v_g}$$

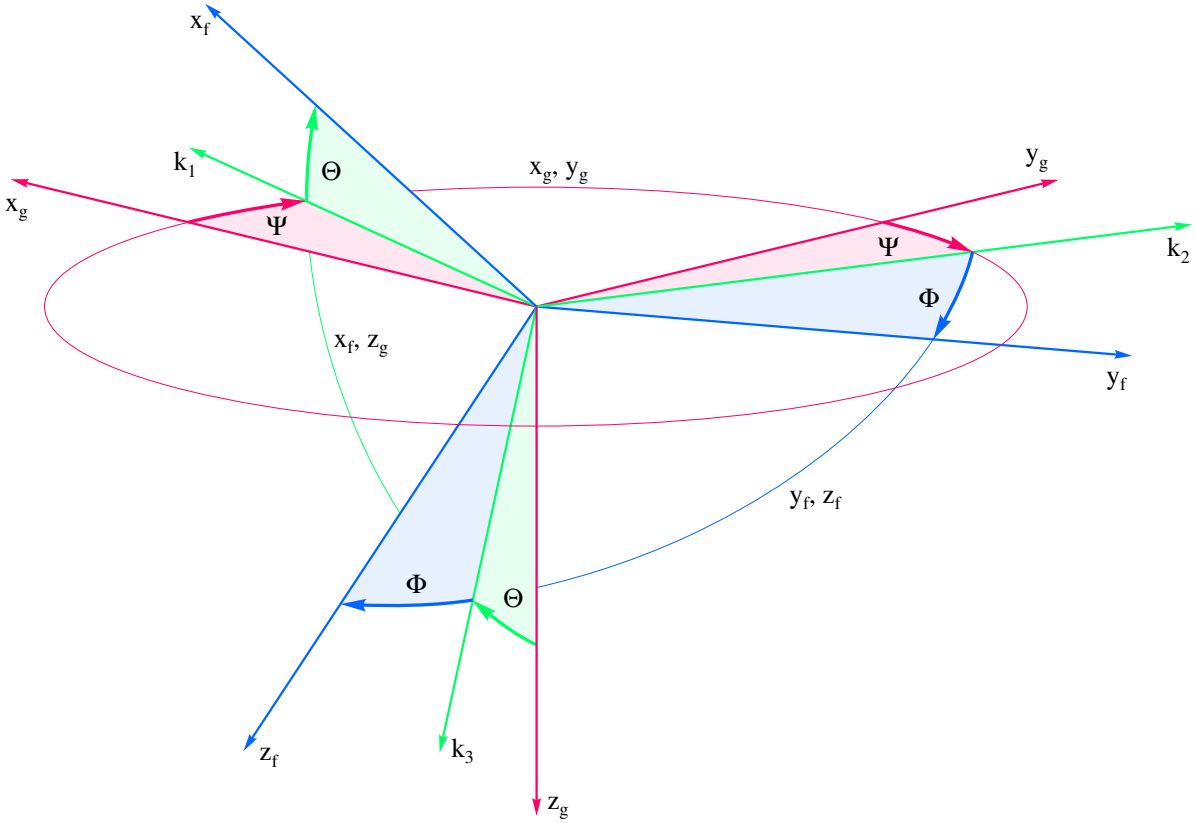


Figure 1.31.: Rotation of the body-fixed frame with respect to the inertial frame [3]

1.27. Transformation from Euler frame to body-fixed frame

The transformation from the Euler frame to the body-fixed frame depicted in figure 1.32

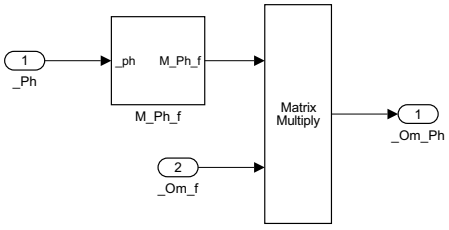


Figure 1.32.: Transformation from Euler frame to body-fixed frame [3]

cannot be defined according to equation (1.7) because the axes about which the Euler angles rotate do not stand rectangular on each other. Therefore, we have to transform the single derivatives of the Euler angles into the body-fixed frame separately:

$$
\begin{aligned}
\left(\frac{\mathrm{d}\boldsymbol{\Phi}}{\mathrm{d}t}\right)_f &= \begin{bmatrix} p_{Kf} \\ q_{Kf} \\ r_{Kf} \end{bmatrix} \\
&= \begin{bmatrix} \dot{\Phi} \\ 0 \\ 0 \end{bmatrix} \\
&+ \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\Phi & \sin\Phi \\ 0 & -\sin\Phi & \cos\Phi \end{bmatrix} \begin{bmatrix} 0 \\ \dot{\Theta} \\ 0 \end{bmatrix} \\
&+ \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\Phi & \sin\Phi \\ 0 & -\sin\Phi & \cos\Phi \end{bmatrix} \begin{bmatrix} \cos\Theta & 0 & -\sin\Theta \\ 0 & 1 & 0 \\ \sin\Theta & 0 & \cos\Theta \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ \dot{\Psi} \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & -\sin\Theta \\ 0 & \cos\Phi & \sin\Phi\cos\Theta \\ 0 & -\sin\Phi & \cos\Phi\cos\Theta \end{bmatrix} \begin{bmatrix} \dot{\Phi} \\ \dot{\Theta} \\ \dot{\Psi} \end{bmatrix} \\
&= \boldsymbol{M_{f\Phi}} \cdot \dot{\boldsymbol{\Phi}}
\end{aligned}
\tag{1.8}
$$

To solve equation (1.8) for the vector of the Euler angle derivatives $\dot{\boldsymbol{\Phi}}$ we have to invert the non-orthogonal matrix $\boldsymbol{M_{f\Phi}}$ which finally results in equation (1.6).

1.28. Euler frame to body frame transformation matrix

In order to model the transformation matrix $\boldsymbol{M_{\Phi f}}$ used in equation (1.6) and figure 1.32 we use the subsystem depicted in figure 1.33

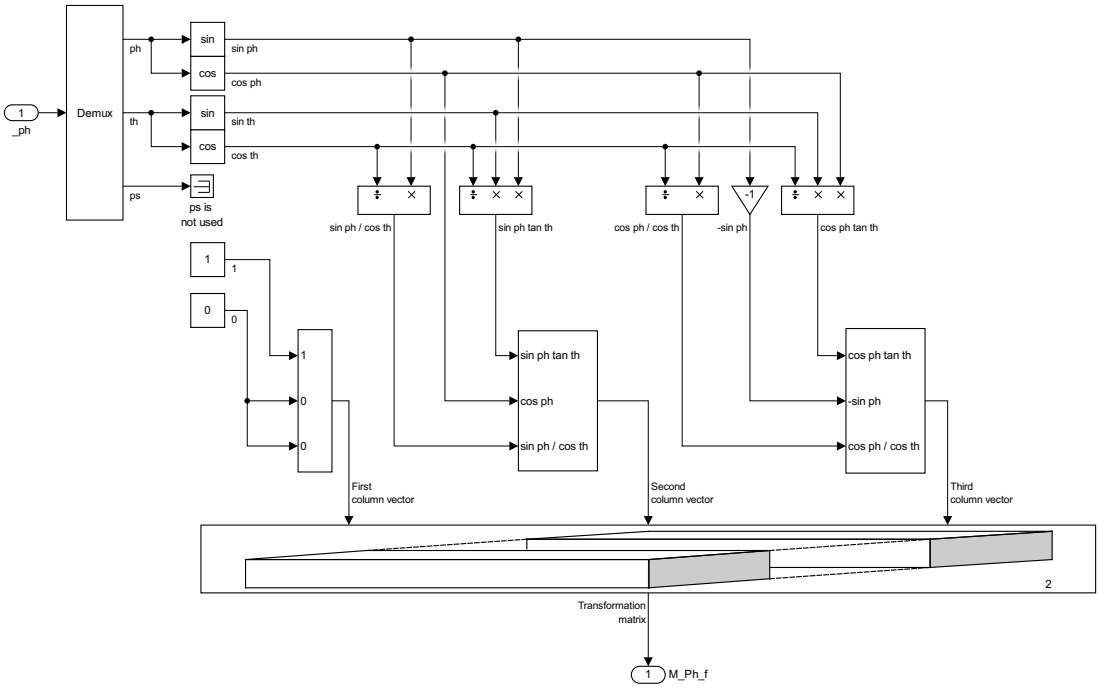


Figure 1.33.: Euler frame to body frame transformation matrix $\boldsymbol{M_{\Phi f}}$ [3]

1.29. Displays

Figure 1.34 shows part of the upper right subsystem in figure 1.1. It is just a collection of scopes to display all relevant flight mechanical vectors and scalars.

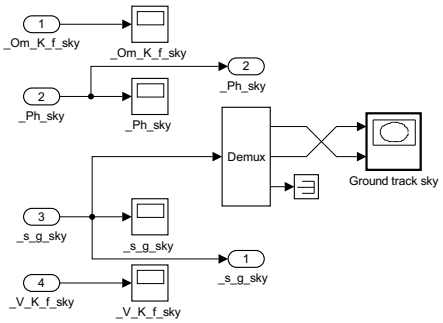


Figure 1.34.: Displays

1.30. Diver

The skydiver subsystem in the bottom center and right of figure 1.1 is very similar to the corresponding UAV subsystem. Basically, there is just one main difference: the free-

falling diver does not have any control inputs. Therefore, we do not need any actuator dynamics in figure 1.35

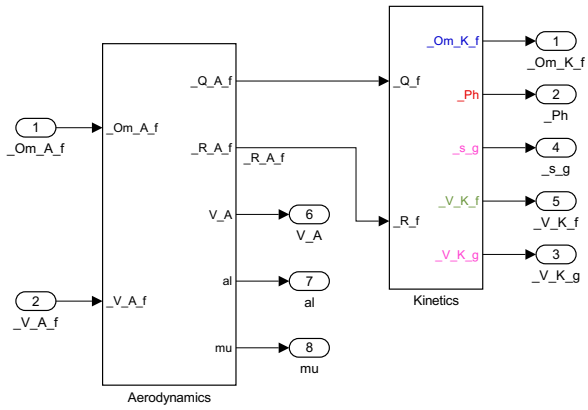


Figure 1.35.: Diver overview

and no vane aerodynamics in figure 1.36.

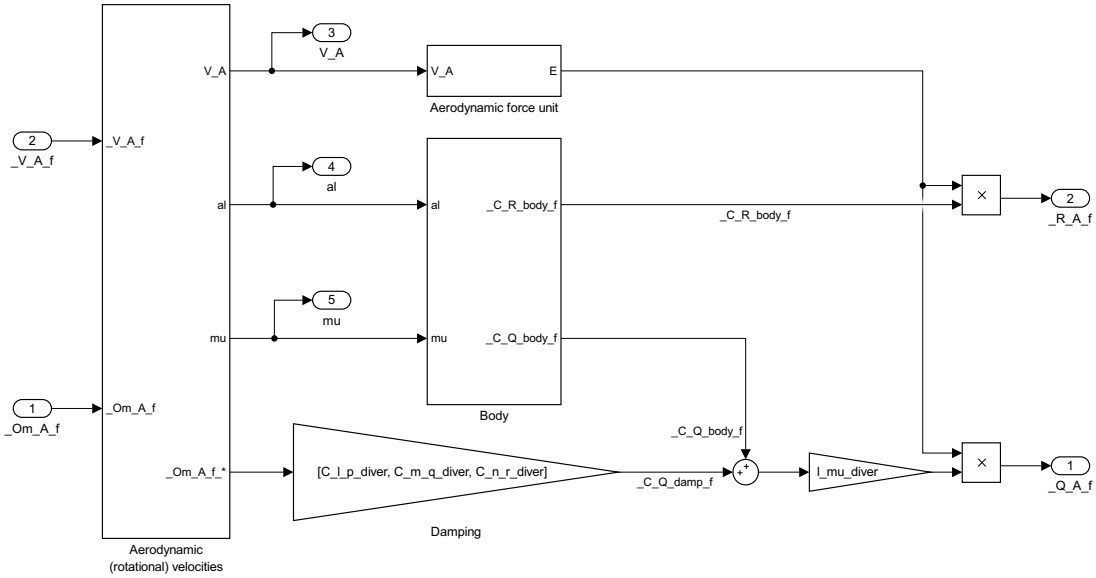


Figure 1.36.: Diver aerodynamics

1.31. Real-Time Pacer

We downloaded the real-time pacer block in the bottom left of figure 1.1 from [2]. It forces a simulation to run in real (wall clock) time if the host computer is fast enough.

1.32. sky_diver_dat.m

One of the nice features of Simulink is the true separation of model and data.[8]

The following Matlab code should be run before the start of the simulation. After an initialization in which we clear all variables and the command window

```
clear all
clc
```

we define[9] the aerodynamic parameters[10] of the UAV

```
rh = 0.413;

l_mu_sky = 1;
S_sky = 0.01;

C_L_al_sky = 3;
C_L_et_sky = 0.1;

C_S_et_sky = C_L_et_sky;

C_D_0_sky = 0.5;
C_D_al_sky = 1;
C_D_al_2_sky = 1;
C_D_et_sky = 1;

C_m_al_sky = 0.5;
C_m_q_sky = -1;
C_m_et_sky = 1;

C_l_p_sky = C_m_q_sky;
C_l_et_sky = -1;

C_n_r_sky = -1;
C_n_ze_sky = 0.2;

turbulence = 1;
```

[8] We can e. g. build a generic aircraft model with named parameters and define all aircraft specific parameters in the corresponding Matlab scripts. By running a specific .m data file before the simulation it is then very easy to switch between different aircraft.

[9] At the time of this documentation, some aerodynamic parameters could only be roughly estimated; future wind tunnel experiments and flight test based parameter identifications will come up with more precise parameter values.

[10] Please refer to the nomenclature for the names and meanings of the parameters. The general idea is to append a trailing `_sky` to all UAV parameters and a trailing `_diver` to all skydiver parameters.

the vane limits[11] of the UAV

```
et_min_sky = 0;
et_max_sky = 0.87;
et_d_max_sky = 3;

ze_min_sky = -0.87;
ze_max_sky = 0.87;
ze_d_max_sky = 3;
```

the kinetic parameters of the UAV

```
m_sky = 1;
g = 9.81;

I_x_sky = 0.1;
I_y_sky = 0.1;
I_z_sky = 0.1;
I_x_z_sky = 0;

I_f_sky = [ ...
       I_x_sky          0   -I_x_z_sky; ...
                                        ...
             0   I_y_sky             0; ...
                                        ...
    -I_x_z_sky          0      I_z_sky];

I_f_m1_sky = inv (I_f_sky);
```

the aerodynamic parameters of the skydiver

```
l_mu_diver = 1;
S_diver = 1;

C_L_al_diver = 3;

C_D_0_diver = 1;
C_D_al_diver = 1;
C_D_al_2_diver = 1;

C_m_al_diver = 0.5;
C_m_q_diver = -1;

C_l_p_diver = C_m_q_diver;

C_n_r_diver = -1;
```

[11] The rate limitations of the elevator vanes are quite overestimated to allow for a more efficient position control; the elevators of the current UAV prototype are about five times slower than assumed.

and the kinetic parameters of the skydiver:

```
m_diver = 60;

I_x_diver = 3;
I_y_diver = 10;
I_z_diver = 10;
I_x_z_diver = 0;

I_f_diver = [ ...
       I_x_diver            0   -I_x_z_diver; ...
                                             ...
                0   I_y_diver               0; ...
                                             ...
    -I_x_z_diver            0      I_z_diver];

I_f_m1_diver = inv (I_f_diver);
```

At the end we inform the user that the script has successfully been completed:

```
disp ('sky_diver_dat.m done.');
```

2. Animation

2.1. Animation overview

The level-2 s-function (`sd_sfun.m`) in the lower left of figure 1.1 draws the skydiver and the UAV in a new figure before the simulation starts and is then called in every simulation interval to redraw both objects with their current attitudes and positions.

The mask of the s-function is depicted in figure 2.1.

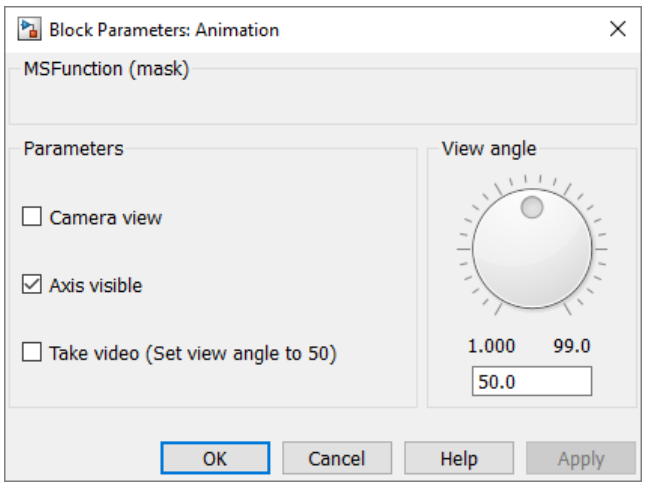


Figure 2.1.: Animation mask

The mask allows the user to choose a normal or a camera view, visible or invisible axes, the viewing angle of the camera view, and whether or not a video of the animation is recorded.

If the user selects the normal view with visible axes (figure 2.2) the axis limits are automatically calculated to always include the UAV and the skydiver.

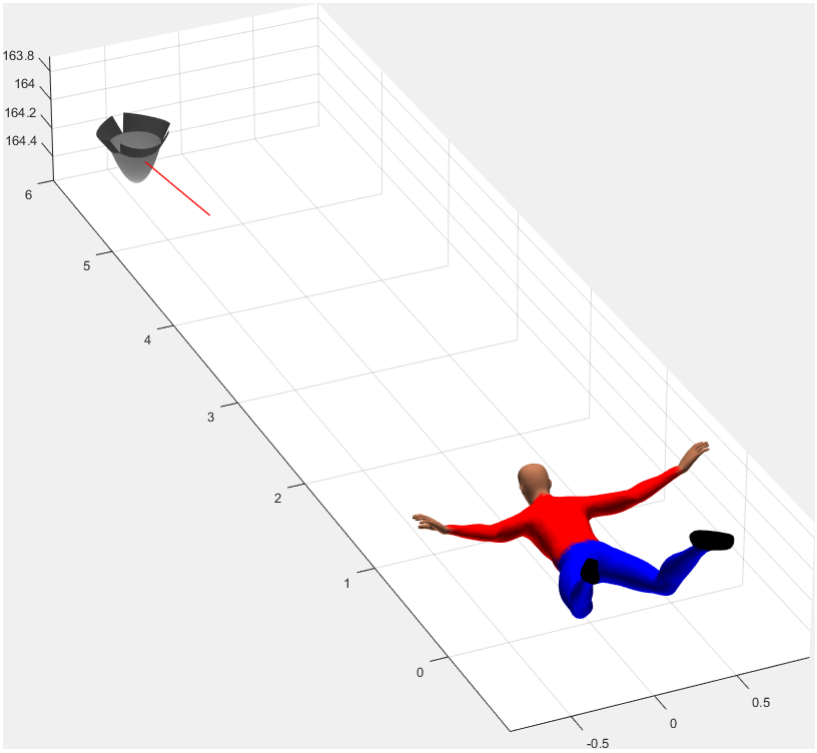


Figure 2.2.: Animation with normal view and with axes

If the user deselects the axes (figure 2.3), it becomes invisible but its limits are still automatically computed.

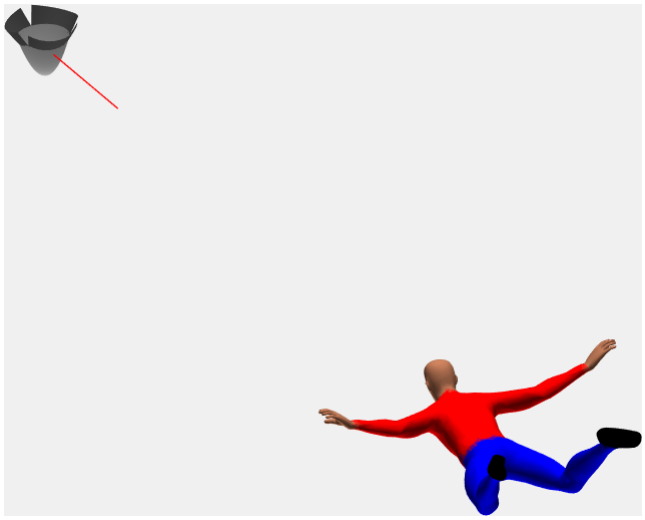


Figure 2.3.: Animation with normal view and without axes

In camera view mode (figure 2.4) with deselected axes we display the actual view as seen from the camera on-board the UAV.

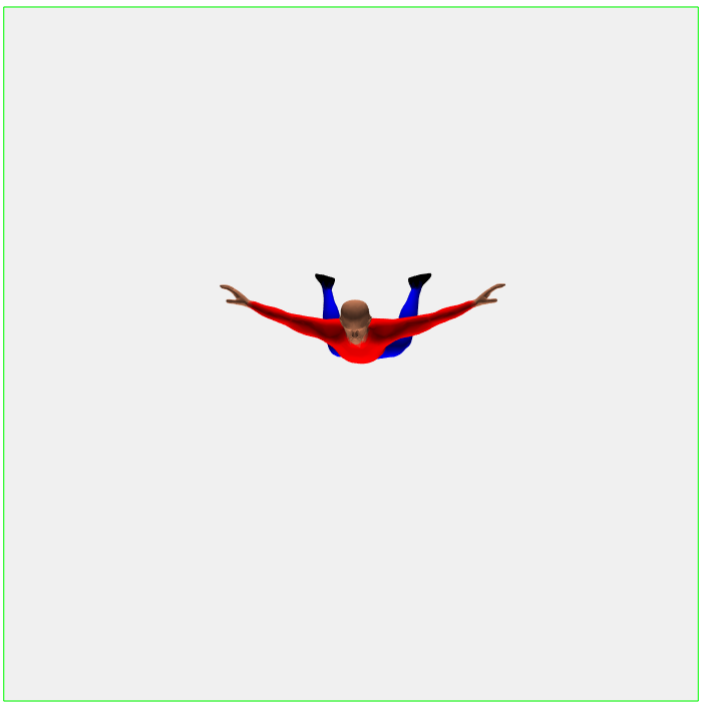


Figure 2.4.: Animation with camera view but without axes

If we make the axes visible, the green square in figure 2.5 indicates the current view of the camera.

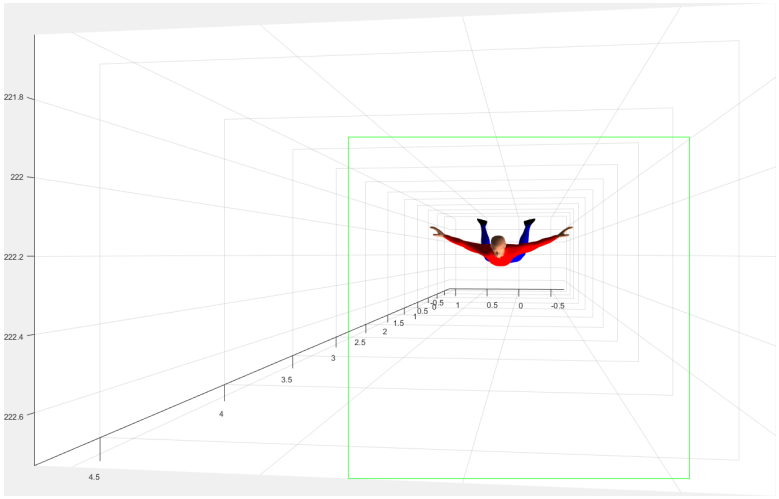


Figure 2.5.: Animation with camera view and with axes

The default camera view angle (field of view) is assumed to be 50° (figure 2.1). If we decrease the field of view to e. g. 20° (figure 2.6) the skydiver appears much bigger and cannot be displayed in the green camera view square in its entirety.

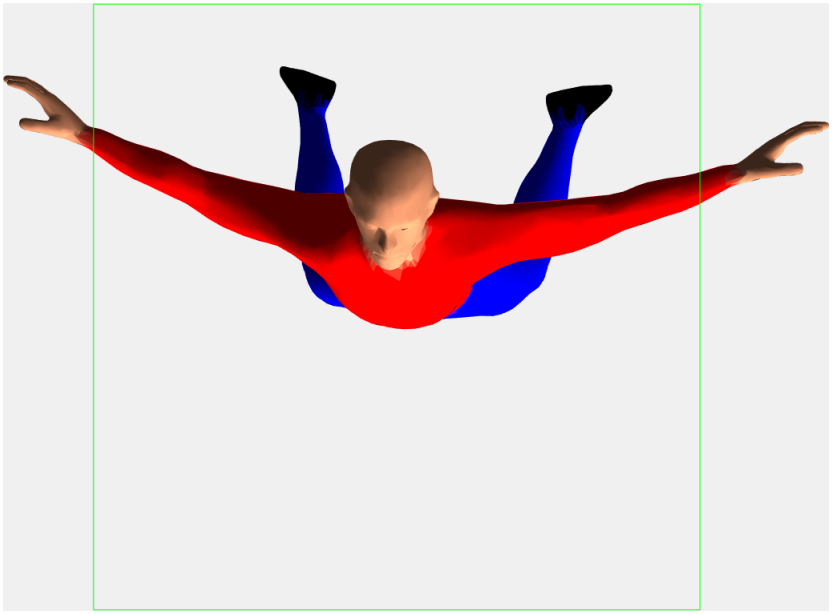


Figure 2.6.: Camera view angle decreased from 50° to 20°

2.2. sd_sfun.m

The level-2 s-function

```
function sd_sfun (block)
```

is responsible for the animation. Its functions are called by the Simulink run-time system at the beginning, during, and at the end of the simulation. All communication between the run-time system and the animation functions is done via a `block` object (section 2.2.2.9). The main function just calls its `setup` function:

```
setup (block);
```

2.2.1. setup

In the initialization function

```
function setup (block)
```

we define a number of variables the run-time system needs to know. At first, we specify the five input and zero output ports of the s-function

```
block.NumInputPorts = 5;
block.NumOutputPorts = 0;
```

indicate that they inherit their compiled properties from the model

```
block.SetPreCompInpPortInfoToDynamic;
block.SetPreCompOutPortInfoToDynamic;
```

and override some properties of each input port

```
block.InputPort(1).Dimensions = 3;
block.InputPort(1).DatatypeID = 0;  % double
block.InputPort(1).Complexity = 'Real';
block.InputPort(1).DirectFeedthrough = false;

block.InputPort(2).Dimensions = 3;
block.InputPort(2).DatatypeID = 0;  % double
block.InputPort(2).Complexity = 'Real';
block.InputPort(2).DirectFeedthrough = false;

block.InputPort(3).Dimensions = 4;
block.InputPort(3).DatatypeID = 0;  % double
block.InputPort(3).Complexity = 'Real';
block.InputPort(3).DirectFeedthrough = false;

block.InputPort(4).Dimensions = 3;
block.InputPort(4).DatatypeID = 0;  % double
block.InputPort(4).Complexity = 'Real';
block.InputPort(4).DirectFeedthrough = false;

block.InputPort(5).Dimensions = 3;
block.InputPort(5).DatatypeID = 0;  % double
block.InputPort(5).Complexity = 'Real';
block.InputPort(5).DirectFeedthrough = false;
```

We declare that the mask of the s-function has four dialog parameters

```
block.NumDialogPrms = 4;
```

tell the run-time system that we only want the animation to take place in major integration steps

```
block.SampleTimes = [0 1];
```

and that it does not have to save and restore any model simulation states:

```
block.SimStateCompliance = 'HasNoSimState';
```

Finally, we register the `start`, `update`, and `terminate` functions with the run-time system:

```
block.RegBlockMethod ('Start', @Start);
block.RegBlockMethod ('Update', @Update);
block.RegBlockMethod ('Terminate', @Terminate);
```

2.2.2. `start`

The initialization function

```
function Start (block)
```

is called by the run-time system once before the start of the simulation.

2.2.2.1. Figure and Axes

We clear the command window

```
clc
```

and check whether there is a figure window from a previous simulation:

```
existing_figure = ...
  findobj ('Type','Figure','Name','Sky animation');
```

If this is the case

```
if ~isempty (existing_figure)
```

we close the old window

```
  close (existing_figure)

end
```

and open a new one

```
sd.h_figure = figure ( ...
  'NumberTitle', 'off', ...
  'Name', 'Sky animation', ...
  'BackingStore','off', ...
  'MenuBar', 'figure', ...
  'Position', [0 0 1024 768], ...
  'Clipping', 'off', ...
  'renderer', 'opengl');
```

In the figure we open an axes with perspective projection and inverted x- and z-axes

```
sd.h_axes = axes ( ...
    'Projection', 'perspective', ...
    'XDir', 'reverse', ...
    'ZDir', 'reverse');
```

switch on the grid

```
grid on
```

set the viewing angle

```
view (66, 30)
```

use the same units along each axis and fit the axes box tightly around the objects

```
axis image
```

and give the user the opportunity to immediately use the mouse to rotate the axes:

```
rotate3d
```

2.2.2.2. Hull of the UAV

For the UAV we create a paraboloid of revolution by revolving a parabola around its axis. We define the level of detail as a positive integer

```
m = 5;
```

from which we derive the number of revolution steps as a multiple of six[1]:

```
n = 6*m;
```

We define the maximum radius of the paraboloid

```
r_max = 2;
```

and a factor that scales the integer (radius) values towards realistic metric dimensions:

```
scaling_factor = 0.1;
```

In order to come up with a greater mesh density at the tip of the paraboloid we define a linearly spaced radius vector

```
r = linspace (0, r_max, n);
```

and compute[2] the corresponding z-values:

```
z = -r.^2 + r_max;
```

Finally, we use `revolve.m`[3] to rotate the parabola about its axis

```
[hull_x, hull_y, hull_z] = ...
  revolve (z*scaling_factor, r*scaling_factor, n);
```

and invert the x-direction so that the x-axis of the body-fixed frame points towards the middle of one of the vanes (figure 1.11):

```
hull_x = -hull_x;
```

[1] Three vanes with at least two faces.

[2] The negative sign opens the paraboloid towards the negative z-axis, which points upwards in the flight mechanical frame. The `r_max`-offset moves the origin of the UAV into its "center" at $z = r_{max}$.

[3] We are not limited to a paraboloid here. We can use `revolve.m` to rotate arbitrary profiles about the z-axis.

2.2.2.3. Vanes

To model the vanes we divide the circumference of the paraboloid in three 120°-sections (figure 1.11) and use the m upper (outer) vertices (`n - m : n`). The second vane starts at the x-axis and then stretches one third of a full circle clockwise (`1 : n/3 + 1`):

```
vane_2_x = hull_x(n - m : n, 1 : n/3 + 1);
vane_2_y = hull_y(n - m : n, 1 : n/3 + 1);
vane_2_z = hull_z(n - m : n, 1 : n/3 + 1);
```

In order to move the vertices more easily during the simulation we transform the surface into (a patch with) vertices and faces

```
[vane_2_f, vane_2_v] = surf2patch (vane_2_x, vane_2_y, vane_2_z);
```

and create[4] the patch:

```
sd.vane_2.handle = patch ( ...
  'Faces', vane_2_f, ...
  'Vertices', vane_2_v, ...
  'FaceVertexCData', [0.3 0.3 0.3], ...
  'FaceColor', 'flat', ...
  'FaceLighting', 'gouraud', ...
  'AmbientStrength', 0.95, ...
  'EdgeColor', 'none');
```

Since the coordinates of the vertices will change during the simulation we save the original vertices

```
sd.vane_2.vertices = vane_2_v;
```

and the surface coordinates in the communication structure too:

```
sd.vane_2.x = vane_2_x;
sd.vane_2.y = vane_2_y;
sd.vane_2.z = vane_2_z;
```

The code for the first (and the third) vane is more or less identical; the section (`n/3 + 1 : 2*n/3 + 1` and `2*n/3 + 1 : n + 1` respectively) on the circumference being the only real difference:

```
vane_1_x = hull_x(n - m : n, n/3 + 1 : 2*n/3 + 1);
vane_1_y = hull_y(n - m : n, n/3 + 1 : 2*n/3 + 1);
vane_1_z = hull_z(n - m : n, n/3 + 1 : 2*n/3 + 1);

[vane_1_f, vane_1_v] = surf2patch (vane_1_x, vane_1_y, vane_1_z);
```

[4] We save the patch handle in the communication structure `sd` (which is saved in the `BlockHandle UserData` in turn, section 2.2.2.9) in order to transfer information between the functions of the s-function.

```
sd.vane_1.handle = patch ( ...
    'Faces', vane_1_f, ...
    'Vertices', vane_1_v, ...
    'FaceVertexCData', [0.3 0.3 0.3], ...
    'FaceColor', 'flat', ...
    'FaceLighting', 'gouraud',     ...
    'AmbientStrength', 0.95, ...
    'EdgeColor', 'none');

sd.vane_1.vertices = vane_1_v;

sd.vane_1.x = vane_1_x;
sd.vane_1.y = vane_1_y;
sd.vane_1.z = vane_1_z;


vane_3_x = hull_x(n - m : n, 2*n/3 + 1 : n + 1);
vane_3_y = hull_y(n - m : n, 2*n/3 + 1 : n + 1);
vane_3_z = hull_z(n - m : n, 2*n/3 + 1 : n + 1);

[vane_3_f, vane_3_v] = surf2patch (vane_3_x, vane_3_y, vane_3_z);

sd.vane_3.handle = patch ( ...
    'Faces', vane_3_f, ...
    'Vertices', vane_3_v, ...
    'FaceVertexCData', [0.3 0.3 0.3], ...
    'FaceColor', 'flat', ...
    'FaceLighting', 'gouraud',     ...
    'AmbientStrength', 0.95, ...
    'EdgeColor', 'none');

sd.vane_3.vertices = vane_3_v;

sd.vane_3.x = vane_3_x;
sd.vane_3.y = vane_3_y;
sd.vane_3.z = vane_3_z;
```

2.2.2.4. Hull display

After we have created the vanes as separate objects we delete the corresponding faces of the hull:

```
hull_x(n - m + 1 : n, :) = [];
hull_y(n - m + 1 : n, :) = [];
hull_z(n - m + 1 : n, :) = [];
```

Finally, we transform the hull surface into a patch with faces and vertices

```
[hull_f, hull_v] = surf2patch (hull_x, hull_y, hull_z);
```

display the hull patch

```
sd.hull.handle = patch ( ...
  'Faces', hull_f, ...
  'Vertices', hull_v, ...
  'FaceVertexCData', [0.8 0.8 0.8], ...
  'FaceColor', 'flat', ...
  'FaceLighting', 'gouraud',     ...
  'AmbientStrength', 0.95, ...
  'EdgeColor', 'none');
```

and save the original hull vertices in the communication structure (section 2.2.2.9)

```
sd.hull.vertices = hull_v;
```

2.2.2.5. x-axis line

A one meter long red line (figure 2.2)

```
sd.x_axis_line.handle = ...
  line ([0 1], [0 0], [0 0], 'LineWidth', 1, 'Color', 'red');
```

indicates the body-fixed x-axis (and by that the first vane) of the UAV.

As with the hull and the vanes, we save the original coordinates of the x-axis in the communication structure for position manipulation during the simulation:

```
sd.x_axis_line.xdata = get (sd.x_axis_line.handle, 'XData');

sd.x_axis_line.ydata = get (sd.x_axis_line.handle, 'YData');

sd.x_axis_line.zdata = get (sd.x_axis_line.handle, 'ZData');
```

2.2.2.6. Field of view

In camera view mode (figure 2.5) we display a green square to indicate the camera field of view. The coordinates of this square are updated in every simulation step; right now we just initialize the polygon as a single green line:

```
sd.fov.handle = ...
  line ([0 0], [0 0], [0 0], 'LineWidth', 0.1, 'Color', 'green');
```

2.2.2.7. Skydiver

In order to display the skydiver we load the 3D model that we previously created in section 2.3

```
load diver.mat
```

save its original faces and vertices in the communication structure

```
sd.diver.faces = diver_faces;

sd.diver.vertices = diver_vertices;
```

and display the skydiver as a single patch with colored faces:

```
sd.diver.handle = patch ( ...
  'Faces', sd.diver.faces, ...
  'Vertices', sd.diver.vertices, ...
  'FaceVertexCData', color_faces, ...
  'FaceColor', 'flat', ...
  'FaceLighting', 'gouraud',     ...
  'AmbientStrength', 0.95, ...
  'EdgeColor', 'none');
```

The default light environment is a bit dark; therefore, we create two light objects, one "above" and one "below" the scene:

```
light ('Position', [0 0 100000])

light ('Position', [0 0 -100000])
```

The default surface reflectance properties give the skydiver an unnatural glossy appearance. Especially his skin[5] looks much more natural with the appropriate material property:

```
material dull;
```

2.2.2.8. Video

If the user has checked the corresponding box in figure 2.1 indicating that he wants to record a video during the simulation

```
if block.DialogPrm(3).Data
```

we create a video writer object that will write into the file `sky_diver.mp4` during the simulation

```
  sd.video = VideoWriter('sky_diver.mp4', 'MPEG-4');
```

[5]Unfortunately, we cannot easily set different material properties for different faces or objects.

set the frame rate to 100 Hz[6]

```
  sd.video.FrameRate = 100;
```

and open the connection from the writer to the file:

```
  open (sd.video);

end
```

2.2.2.9. Communication structure

In the `start` function we have created graphical objects that we want to manipulate (e. g. move) in the `update` function that is called by the run-time system in every simulation interval.

Unfortunately, transferring information (graphics handles, vertices, ...) between two functions of an s-function is a bit tricky. The most elegant way according to [4] is to bundle all the information in a structure and save this structure in the `UserData` property of the `block` object that is automatically handed over to every function as a parameter by the run-time system. Therefore, we finally save[7] the communication structure `sd` in the mentioned property:

```
set_param (block.BlockHandle, 'UserData', sd);
```

2.2.3. `revolve.m`

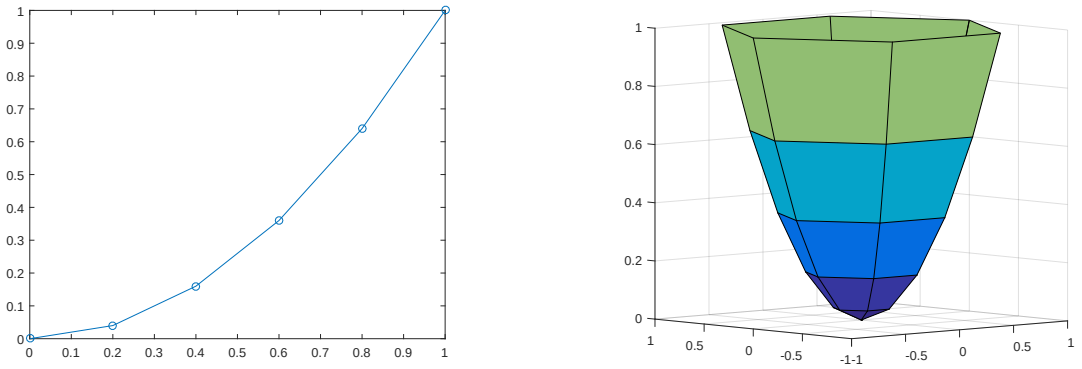
The function

```
function [xx, yy, zz] = revolve (z, r, n)
```

creates a solid of revolution (`xx`, `yy`, `zz`) from a given polygon (radius vector `r`, height vector `z`). The number of vertices on the circumference of the solid can be defined by the third parameter `n` (figure 2.7).

[6] We have to set the simulation sample time (Simulation/Model Configuration Parameters/Solver/Fixed-step size) to `0.01` manually. On some computers we will lose real-time during the simulation; nevertheless, the video will be in real-time. We can choose any other video frame rate (together with the corresponding sample time).

[7] In the other function (section 2.2.4, ...), we can later on retrieve the communication structure via: `sd = get_param (block.BlockHandle, 'UserData');`



(a) Parabola (b) Paraboloid

Figure 2.7.: `revolve` creates a solid of revolution from a given polygon.

Initially, we make sure that the radius and height vectors are column vectors:

```
r = r(:);

z = z(:);
```

Next, we create an angle vector for the `n` vertices[8] on the circumference as a row vector:

```
theta = linspace (0, 2*pi, n + 1);
```

Multiplying the radius column vector with the corresponding row vectors (`cos` and `sin` of the angle vector) now automatically creates the `xx` and `yy` domain mesh matrices used for the `surf` command:

```
xx = r*cos (theta);

yy = r*sin (theta);
```

Since all vertices on a specific circumference of the paraboloid have the same z-component the corresponding `zz` height matrix simply consists of replicas of the height vector:

```
zz = z*ones (1, n + 1);
```

2.2.4. `update`

The update function

```
function Update (block)
```

[8]Start and end vertices are identical.

is called by the run-time system in every (major) simulation step. Its main task is to move the vertices of the objects in the scene. Therefore, our first step is to read the communication structure data (handles, coordinates, ...) back into the local variable `sd`:

```
sd = get_param (block.BlockHandle, 'UserData');
```

2.2.4.1. Hull

Using the communication structure we can extract the original hull vertex matrix:

```
hull_v = sd.hull.vertices;
```

During the simulation an object is animated by rotating and translating every single of its vertices in every simulation step. The general idea is to move (translate) the object into the origin, then rotate the object about the origin using the current Euler angles and then move the rotated object to the current position.

Since we know the vertices of the original hull, our first step is to rotate all vertex vectors by the (inverse[9]) transformation matrix (DCM) computed in `m_fg` from the current Euler angle vector fed into the s-function via the second input port:

```
hull_v = (m_fg (block.InputPort(2).Data)'*hull_v')';
```

Then we can do the translation by adding the current position (incoming via the first s-function input port) to the vertex matrix:

```
hull_v = ...
  hull_v + ...
  repmat (block.InputPort(1).Data', size (hull_v, 1), 1);
```

Finally, we set all vertices to their new positions:

```
set ( ...
  sd.hull.handle, ...
  'Vertices', ...
  hull_v);
```

2.2.4.2. Vanes

We animate all three vanes by calling the separate function `vane_rotate` using the original vane vertices and the current vane angles (η_i, ζ) from the third s-function input port as parameters:

[9] Orthogonal transformation matrices have the nice property that their inverses can easily be computed by their transposes: $M_{ortho}^{-1} = M_{ortho}^{T}$

```
vane_rotate (block, sd.vane_1, ...
  block.InputPort(3).Data(1), ...
  block.InputPort(3).Data(4))

vane_rotate (block, sd.vane_2, ...
  block.InputPort(3).Data(2), ...
  block.InputPort(3).Data(4))

vane_rotate (block, sd.vane_3, ...
  block.InputPort(3).Data(3), ...
  block.InputPort(3).Data(4))
```

2.2.4.3. x-axis line

The rotation and motion of the x-axis line is done just like the transformation of the hull. We combine the (two) vertices in a matrix

```
x_axis_line_data = [ ...
  sd.x_axis_line.xdata', ...
  sd.x_axis_line.ydata', ...
  sd.x_axis_line.zdata'];
```

rotate the vectors via `m_fg` and the second input port

```
x_axis_line_data = ...
  (m_fg (block.InputPort(2).Data)'*x_axis_line_data')';
```

and translate them to the current position:

```
x_axis_line_data = ...
  x_axis_line_data + ...
  repmat (block.InputPort(1).Data', ...
  size(x_axis_line_data, 1), 1);
```

Finally, we use the rotated and translated data to update the line object:

```
set ( ...
    sd.x_axis_line.handle, ...
    'XData', x_axis_line_data(:,1), ...
    'YData', x_axis_line_data(:,2), ...
    'ZData', x_axis_line_data(:,3));
```

2.2.4.4. Skydiver

The exact same approach (vertices collection, rotation, motion, and data updating) is used for the update of the skydiver:

```
diver_v = sd.diver.vertices;

diver_v = (m_fg (block.InputPort(5).Data)'*diver_v')';

diver_v = ...
  diver_v + ...
  repmat (block.InputPort(4).Data', size (diver_v, 1), 1);

set ( ...
    sd.diver.handle, ...
    'Vertices', ...
    diver_v);
```

2.2.4.5. Camera

Depending on the choice of the user in figure 2.1

```
if block.DialogPrm(2).Data
```

we make the axes visible

```
  axis on
```

or invisible:

```
else

  axis off

end
```

If the user has checked camera view in figure 2.1

```
if block.DialogPrm(1).Data
```

we position the camera in the current center of the UAV

```
  cam_pos = block.InputPort(1).Data';

  campos (cam_pos);
```

and define the direction[10] the camera is looking at to be the "end point" of the x-axis line:

```
  cam_target = x_axis_line_data(2,:);
```

Since the camera is gyro stabilized in two axes (roll and pitch) we additionally assume that the target vector lies in the x_g-y_g-plane:

[10] Since we used a perspective projection in section 2.2.2.1 only the direction of the target vector is utilized; its length is irrelevant [5].

```
cam_target(3) = cam_pos(3);

camtarget (cam_target)
```

The user can define the camera viewing angle in figure 2.1:

```
va = block.DialogPrm(4).Data;

camva (va);
```

2.2.4.6. Field of view (FOV)

In order to generate the green FOV[11] square in figure 2.5 we define the unit[12] vector `d` from the camera position to the camera target:

```
d = cam_target - cam_pos;
```

Additionally, we compute the half-width of the FOV square in 1 m distance from the camera position (i. e. at the camera target) according to section 2.2.5:

```
fov_hw = atan (va/2*pi/180);
```

We define the FOV vertical unit vector (down)

```
fov_vv = [0 0 1];
```

and compute the resulting horizontal unit vector (to the left):

```
fov_hv = cross (d, fov_vv)
```

Now, we can compute all four vertices of the FOV square using half-width and unit vectors:

```
fov_v = [...
  cam_target + fov_hw*(fov_vv + fov_hv); ...
  cam_target + fov_hw*(fov_vv - fov_hv); ...
  cam_target + fov_hw*(-fov_vv - fov_hv); ...
  cam_target + fov_hw*(-fov_vv + fov_hv); ...
  cam_target + fov_hw*(fov_vv + fov_hv)];
```

and display the square

```
set (sd.fov.handle, ...
  'XData', fov_v(:,1), ...
  'YData', fov_v(:,2), ...
  'ZData', fov_v(:,3))
```

[11] See `fov_test.m` for a better understanding on how the field of view is defined and manipulated in Matlab.

[12] Since we defined the camera target at the end of the 1 m long x-axis line, `d` is a unit vector.

2.2.4.7. Visibility

In camera mode, the UAV itself should not be visible

```
set ([ ...
  sd.hull.handle, ...
  sd.vane_1.handle, ...
  sd.vane_2.handle, ...
  sd.vane_3.handle, ...
  sd.x_axis_line.handle, ...
  ], ...
  'visible', ...
  'off')
```

but the FOV square is visible:

```
set ([ ...
  sd.fov.handle, ...
  ], ...
  'visible', ...
  'on')
```

If he user has deselected camera mode in figure 2.1

```
else
```

we let Matlab choose appropriate values for the position, the target, and the viewing angle of the camera:

```
campos ('auto');

camtarget ('auto');

camva ('auto');
```

In normal view (non-camera mode) we make the UAV visible

```
set ([ ...
  sd.hull.handle, ...
  sd.vane_1.handle, ...
  sd.vane_2.handle, ...
  sd.vane_3.handle, ...
  sd.x_axis_line.handle, ...
  ], ...
  'visible', ...
  'on')
```

and the FOV square invisible

```
  set ([ ...
    sd.fov.handle, ...
    ], ...
    'visible', ...
    'off')
end
```

Before we can see the current frame we have to tell the run-time system to actually display all objects in the scene

```
drawnow limitrate
```

Using the `limitrate` parameter can speed up the simulation significantly.

2.2.4.8. Video

If the user has chosen to record a video

```
if block.DialogPrm(3).Data
```

we save a screenshot[13] of the current frame in the video file defined in section 2.2.2.8:

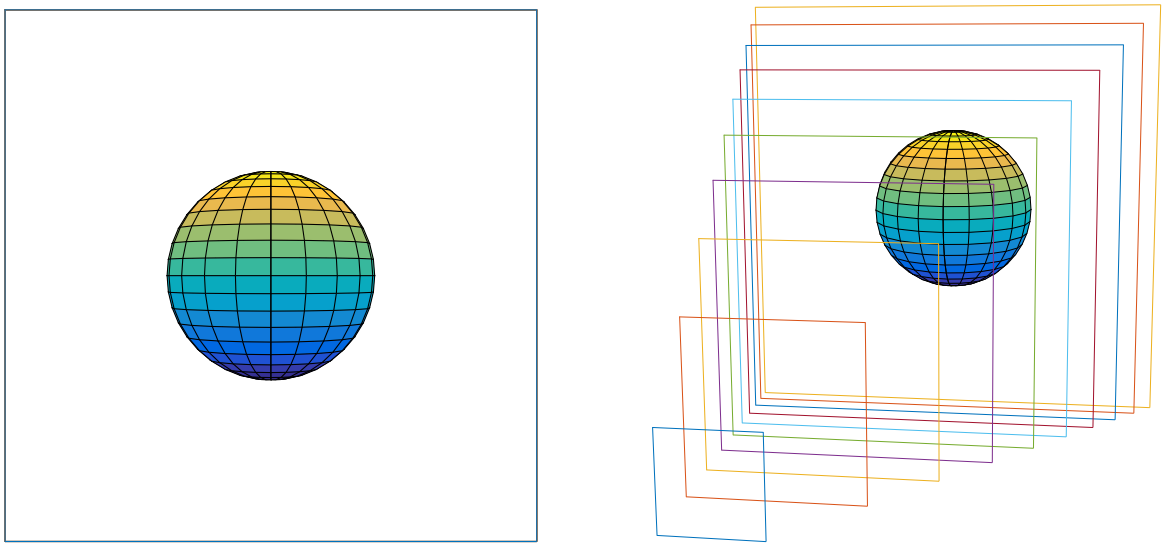
```
  writeVideo (sd.video, ...
    getframe (sd.h_figure, [252 118 556 556]));

end
```

2.2.5. `fov_test.m`

We wrote this small external[14] script `fov_test.m` to shed some light on how Matlab's camera field of view (FOV) is computed and displayed. For that purpose, we created a number of parallel squares on the boundaries of the FOV pyramid according to [5]. As long as the camera is kept in the correct position, the camera target vector lies on the line linking the camera and the target, and we use perspective projection instead of the default orthographic projection, all FOV squares seem to exist in the same place (figure 2.8a). As soon as we move the camera is becomes obvious that this is not the case (figure 2.8b).

[13] By defining an empirically determined rectangle as the second parameter of the `getframe` command we take a video of only the actual FOV. This works precisely only if the viewing angle is set to 50°.
[14] `fov_test` is not needed for the simulation and animation of the UAV and the skydiver.



(a) All FOV squares match.

(b) FOV squares are distinguishable.

Figure 2.8.: Only in perspective projection mode all FOV squares appear as one (2.8a) if the camera position and the camera target vector fit.

In order to create figure 2.8 we define a sphere around the origin

```
sphere
```

make all axes equidistant

```
axis equal
```

and invisible

```
axis off
```

and switch on perspective projection mode:

```
set (gca, 'Projection', 'perspective')
```

We position the camera at $x = -10$

```
campos ([-10, 0, 0])
```

let it look at[15] the sphere

```
camtarget ([0, 0, 0])
```

and set the camera viewing angle[16] to a moderate 30°:

[15] In perspective projection mode the camera target vector could have an arbitrary x-component of $x > -10$. We could e. g. have used `camtarget ([-3, 0, 0])`.

[16] Any other valid viewing angle would come up with an equivalent result.

```
va = 30;

camva (va)
```

We want to draw all squares in the already open axes

```
hold on
```

and start a loop over ten squares:

```
for x = 1 : 10
```

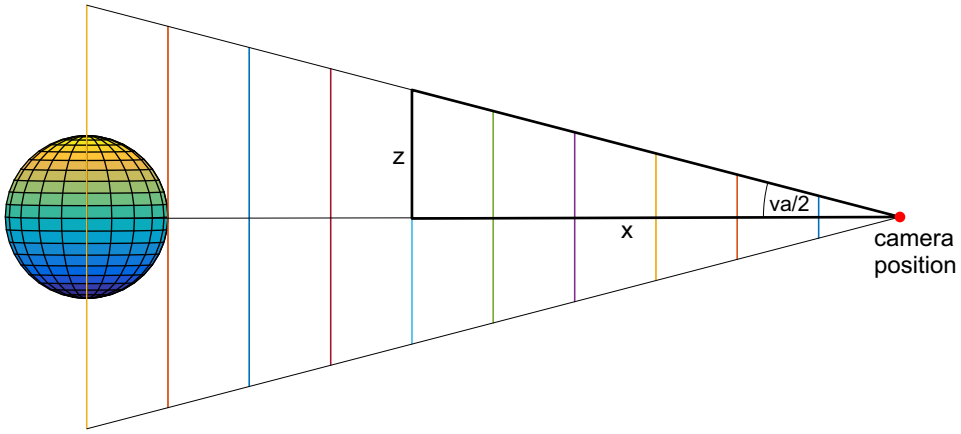


Figure 2.9.: Size of the FOV squares depending on the viewing angle va and the distance from the camera x.

According to figure 2.9, we can calculate the half-width z of a square in the distance of x from the camera position via the arctangent of half of the viewing angle:

$$\arctan\left(\frac{va}{2}\right) = \frac{z}{x}$$

```
  z = x*atan(va/2*pi/180);
```

Using the distance of the current square from the origin

```
  xx = -10 + x;
```

we can finally draw the current square

```
  plot3 ( ...
    [xx, xx, xx, xx, xx], ...
    [-z, z, z, -z, -z], ...
    [-z, -z, z, z, -z])

end
```

2.2.6. terminate

The terminate function

```
function Terminate (block)
```

is called by the run-time system at the end of the simulation. We use it to clean up the video recording. If the user has decided to record a video

```
if block.DialogPrm(3).Data
```

we extract the communication structure

```
  sd = get_param (block.BlockHandle, 'UserData');
```

and finalize the video writer object we created in section 2.2.2.8:

```
  close (sd.video);

end
```

2.2.7. vane_rotate

We outsourced the extensive computation of the vane deflections from section 2.2.4.2 into:

```
function vane_rotate (block, vane, roll_angle, yaw_angle)
```

Once again, the basic idea is to translate (move) the vane into the origin with one of its vertices, do the rotation about a specific axis in the origin, and translate (the rotation axis of) the vane back to its old position. Since there is a roll and a yaw rotation we have to do both deflections one after the other.

2.2.7.1. Roll

For the roll deflection (η) we define the lowest outer vane vertices as axis points

```
vane_roll_axis_point_1 = [ ...
    vane.x(1,1), ...
    vane.y(1,1), ...
    vane.z(1,1)];

vane_roll_axis_point_2 = [ ...
    vane.x(1,end), ...
    vane.y(1,end), ...
    vane.z(1,end)];
```

and the roll axis to go through these two vertices:

```
vane_roll_axis = vane_roll_axis_point_2 - vane_roll_axis_point_1;
```

After buffering the vane vertices

```
vane_v = vane.vertices;
```

we can translate the vane with its first roll axis vertex into the origin:

```
vane_v = vane_v - repmat ( ...
  vane_roll_axis_point_1, size(vane_v, 1), 1);
```

We can now roll all vane vertices about the roll axis in the orig:

```
vane_v = (rotation_about_arbitrary_axis ( ...
  vane_roll_axis, roll_angle)*vane_v')';
```

Finally, we have to translate the first roll axis vertex from the origin back to its previous position:

```
vane_v = vane_v + repmat ( ...
  vane_roll_axis_point_1, size(vane_v, 1), 1);
```

2.2.7.2. Yaw

The yawing axis is defined by the lowest and the highest middle vertex of the vane:

```
vane_yaw_axis_point_1 = [ ...
    vane.x(1,(end + 1)/2), ...
    vane.y(1,(end + 1)/2), ...
    vane.z(1,(end + 1)/2)];

vane_yaw_axis_point_2 = [ ...
    vane.x(end,(end + 1)/2), ...
    vane.y(end,(end + 1)/2), ...
    vane.z(end,(end + 1)/2)];
```

Unfortunately, we have to roll the yaw axis before we can yaw. Therefore, we merge both vertices into a matrix

```
vane_yaw_axis_points = [ ...
  vane_yaw_axis_point_1; vane_yaw_axis_point_2];
```

move the yaw axis with the first roll axis point into the origin

```
vane_yaw_axis_points = vane_yaw_axis_points - repmat ( ...
  vane_roll_axis_point_1, 2, 1);
```

rotate the yaw axis about the roll axis in the origin

```
vane_yaw_axis_points = ...
  (rotation_about_arbitrary_axis ( ...
  vane_roll_axis, roll_angle)*vane_yaw_axis_points')';
```

and move the yaw axis back up again into its previous position

```
vane_yaw_axis_points = vane_yaw_axis_points + repmat ( ...
  vane_roll_axis_point_1, 2, 1);
```

Now, we can read the rolled yaw axis vertices from the matrix

```
vane_yaw_axis_point_1 = vane_yaw_axis_points(1,:);
vane_yaw_axis_point_2 = vane_yaw_axis_points(2,:);
```

and define the final yaw axis

```
vane_yaw_axis = vane_yaw_axis_point_2 - vane_yaw_axis_point_1;
```

The rest is just the usual procedure: Move the vane down with its first yaw axis vertex into the origin:

```
vane_v = vane_v - repmat ( ...
  vane_yaw_axis_point_1, size(vane_v, 1), 1);
```

rotate (ζ) the vane around its yaw axis in the origin

```
vane_v = (rotation_about_arbitrary_axis ( ...
  vane_yaw_axis, yaw_angle)*vane_v')';
```

and translate the vane back up again with its first yaw axis vertex into its previous position:

```
vane_v = vane_v + repmat ( ...
  vane_yaw_axis_point_1, size(vane_v, 1), 1);
```

Finally, we have to rotate

```
vane_v = (m_fg(block.InputPort(2).Data)'*vane_v')';
```

and translate the vane together with the hull:

```
vane_v = vane_v + repmat ( ...
  block.InputPort(1).Data', size(vane_v, 1), 1);
```

and update all vane vertices:

```
set ( ...
  vane.handle, ...
  'Vertices', ...
  vane_v);
```

2.2.8. m_fg

The helper function

```
function mfg = m_fg (ph_th_ps)
```

computes the (direction cosine) transformation matrix from the inertial to the body-fixed frame (figure 1.31) using trigonometric functions of the Euler angles (Φ, Θ, and Ψ). Since the sine and the cosine of the Euler angles are used more than once we buffer them in extra variables:

```
cos_ps = cos (ph_th_ps(3));
sin_ps = sin (ph_th_ps(3));
cos_th = cos (ph_th_ps(2));
sin_th = sin (ph_th_ps(2));
cos_ph = cos (ph_th_ps(1));
sin_ph = sin (ph_th_ps(1));
```

Now we can compute the single transformation matrices for the rotation about the z-axis

```
m_ps = [ ...
   cos_ps, sin_ps, 0; ...
  -sin_ps, cos_ps, 0; ...
        0,      0, 1];
```

the y-axis

```
m_th = [ ...
  cos_th, 0, -sin_th; ...
       0, 1,       0; ...
  sin_th, 0,  cos_th];
```

and the x-axis:

```
m_ph = [ ...
  1,       0,      0; ...
  0,  cos_ph, sin_ph; ...
  0, -sin_ph, cos_ph];
```

Finally, we combine all three rotations by multiplying the single matrices:

```
mfg = m_ph * m_th * m_ps;
```

Alternatively, we could have multiplied the matrices symbolically

```
mfg = [ ...
                 cps*cth,                cth*sps,    -sth; ...
  cps*sph*sth - cph*sps, cph*cps + sph*sps*sth, cth*sph; ...
  sph*sps + cph*cps*sth, cph*sps*sth - cps*sph, cph*cth];
```

in order to save a little bit of computation time.

2.2.9. rotation_about_arbitrary_axis

For a transformation (rotation) about an arbitrary axis we wrote a helper function according to [6]:

```
function m = rotation_about_arbitrary_axis (axis, angle)
```

In the first step we make sure the rotation axis vector is a unit vector:

```
n = axis/norm(axis);
```

We buffer the components of the (unit) axis vector

```
n1 = n(1);

n2 = n(2);

n3 = n(3);
```

the sine and cosine of the rotation angle

```
c = cos (angle);

s = sin (angle);
```

and the cosine unit complement in extra variables:

```
c1 = 1 - c;
```

Finally, we can compute the rotation matrix

```
m = [ ...
  n1*n1*c1 +    c, n1*n2*c1 - n3*s, n1*n3*c1 + n2*s; ...
  n2*n1*c1 + n3*s, n2*n2*c1 +    c, n2*n3*c1 - n1*s; ...
  n3*n1*c1 - n2*s, n3*n2*c1 + n1*s, n3*n3*c1 +    c];
```

2.3. Skydiver model

Fortunately, we found an extremely detailed (297,984 vertices) 3D model (figure 2.10) of a male skydiver at [7].

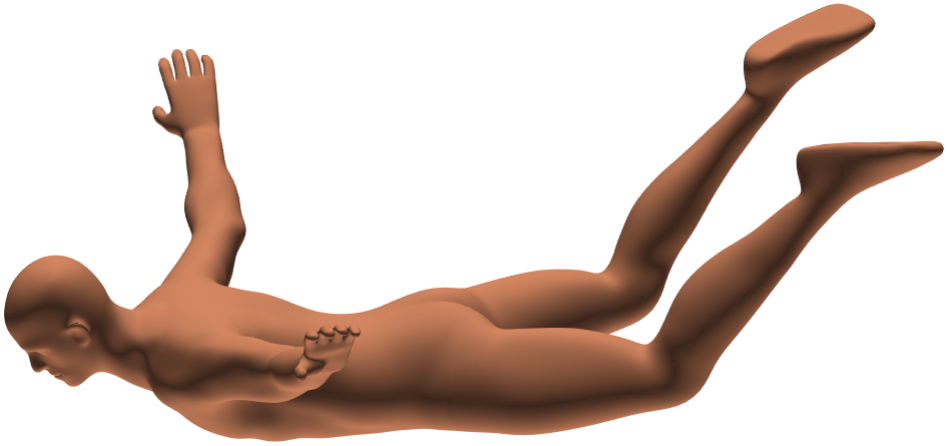


Figure 2.10.: 3D model of a male skydiver from [7]

Eric Johnson has written a nice little function [8] to import binary STL files into Matlab and Matlab itself offers its `reducepatch` command to decrease the number of vertices of the model to an amount that can be displayed by a contemporary computer in real-time. Additionally, we want to create the illusion of a decent skydiver by painting parts of his body with appropriate colors (figure 2.11).

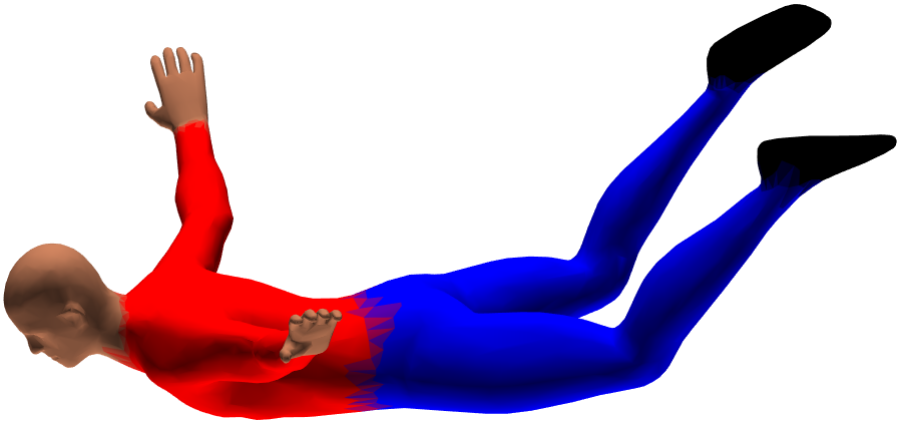


Figure 2.11.: Skydiver with "clothes on"

In order to create a reasonable Matlab model of a skydiver we use the following steps in an external script called `diver_stl2mat.m`:

After a soft reset

```
clear all
close all
clc
```

we define that our final model should only have about 5000 vertices:

```
detail_factor = 0.1;
```

We read [8] the STL file into a Matlab structure

```
diver_struct = stlread ('diver_slow.stl');
```

and reduce the complexity of the model significantly:

```
[diver_faces, diver_vertices] = ...
  reducepatch (diver_struct, detail_factor);
```

Since the flight mechanical inertial system will have its z-axis pointing "down" we have to invert the diver's z-axis

```
diver_vertices(:,3) = -diver_vertices(:,3);
```

and rotate (exchange with one negative sign) its x- and y-axes:

```
buffer = diver_vertices(:,1);

diver_vertices(:,1) = -diver_vertices(:,2);

diver_vertices(:,2) = buffer;
```

In order to come up with a reasonably sized human ($\approx 1.80\,\mathrm{m}$) we scale the model:

```
diver_vertices = diver_vertices/90;
```

We move the origin of the coordinate system into the mean of all vertices, which is not exactly the skydiver's center of mass but comes sufficiently close:

```
diver_vertices = ...
  diver_vertices - ...
  repmat (mean (diver_vertices), ...
  size (diver_vertices, 1), 1);
```

We define colors for different parts of the body

```
color_skin = [192, 122, 88]/255;
color_shirt = [255, 0, 0]/255;
color_pants = [0, 0, 255]/255;
color_shoes = [0, 0, 0]/255;
```

determine the number of vertices and faces of the reduced model

```
n_vertices = size (diver_vertices, 1);
n_faces = size (diver_faces, 1);
```

and initialize the arrays that will later hold the colored vertices and faces[17]:

```
color_vertices = zeros (n_vertices, 3);
color_faces = zeros (n_faces, 3);
```

[17]We know that all faces of this model are triangles and therefore have exactly three vertices.

In a loop over all vertices

```
for i_vertex = 1 : n_vertices
```

we define certain areas (e. g. the pants)

```
  if ...
    diver_vertices(i_vertex, 1) < -0.03 && ...
    diver_vertices(i_vertex, 3) > -0.25
```

in which the vertices (and later the faces) have a certain color:

```
    color_vertices(i_vertex, :) = color_pants;
```

This coloring based on the location of the vertices is then done for the vertices of the other body parts too:

```
  elseif ...
    diver_vertices(i_vertex, 3) <= -0.25

    color_vertices(i_vertex, :) = color_shoes;

  elseif ...
    diver_vertices(i_vertex, 1) >= -0.03 && ...
    diver_vertices(i_vertex, 1) < 0.47 && ...
    diver_vertices(i_vertex, 2) > -0.71 && ...
    diver_vertices(i_vertex, 2) < 0.71

    color_vertices(i_vertex, :) = color_shirt;

  else

    color_vertices(i_vertex, :) = color_skin;

  end

end
```

In a loop over every face

```
for i_face = 1 : n_faces
```

we take the mean[18] of the colors of all three vertices of a face to define the color of that face:

[18] The random location of border vertices makes the "borderline" between two differently colored body parts look pretty ragged. Computing the mean of all three vertex colors of a border face makes the transition between body parts a little bit smoother. All three vertices of non-border faces have the same color; the mean does not change that.

```
  color_faces(i_face, :) = ...
    mean (color_vertices(diver_faces(i_face, :), :));

end
```

To verify that everything has worked out as expected, we can draw (figure 2.11) the skydiver in a realistic environment:

```
patch (...
  'Faces', diver_faces, ...
  'Vertices', diver_vertices, ...
  'FaceVertexCData', color_faces, ...
  'FaceColor', 'flat', ...
  'FaceLighting', 'gouraud',     ...
  'AmbientStrength', 0.95, ...
  'EdgeColor', 'none');

light ('Position', [0 0 1000])
light ('Position', [0 0 -1000])

material dull;

axis equal

rotate3d
```

Finally, we save the vertices, faces, and face colors in a binary Matlab file

```
save diver.mat diver_vertices diver_faces color_faces
```

that is read by section 2.2.2.7 during the initialization of the animation.

3. Skydiver flight data

Instead of simulating the skydiver we can also use flight data that have been recorded during various skydives. The data are preprocessed and enter the simulation via the From Workspace Block in figure 3.1.

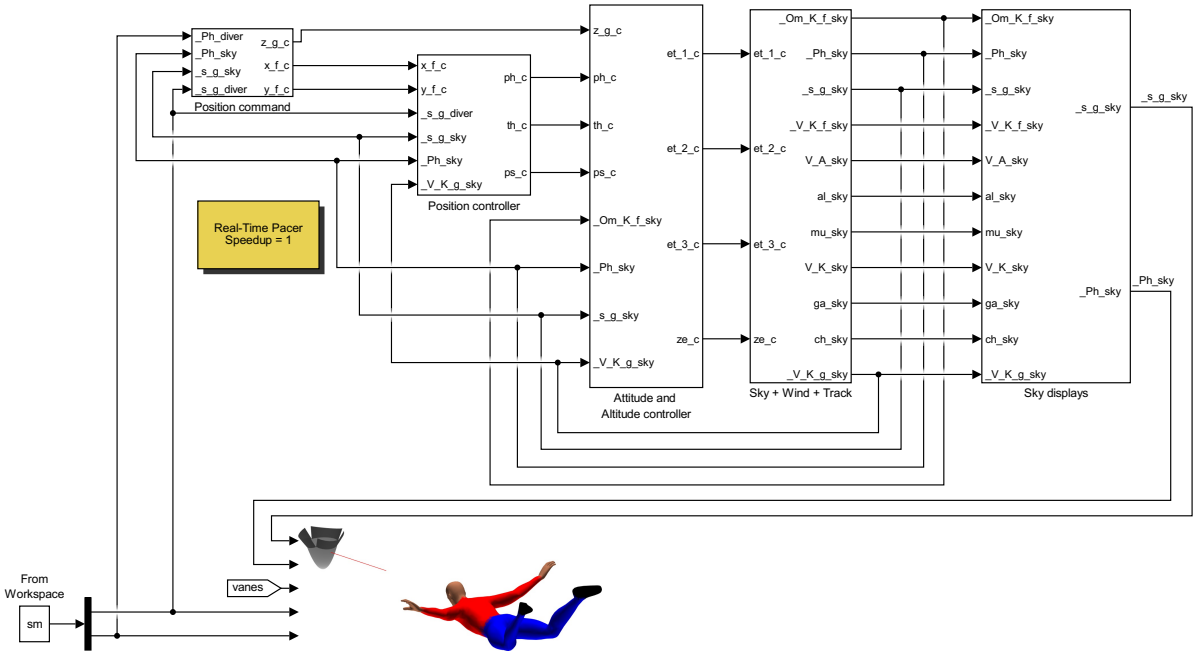


Figure 3.1.: Using real-world skydive data

3.1. `sky_diver_dat.m`

The preprocessing of the skydive data is done in same m-file (`sky_diver_dat.m`) that defines the parameters of the UAV and the skydiver.

First, we load the raw data from one of the flight data MAT-files:

```
load Skydiver_1238_Tandem
```

We skip the first frames until the velocity and attitude of the skydiver have stabilized:

```
skydiver_motion = skydiver_motion(100 : end, :);
```

The first column is the time. It should start with 0.0:

```
t = skydiver_motion(:, 1) -  skydiver_motion(1, 1);
```

The frame rate of the data acquisition system is 5 Hz, which leads to unrealistically disjointed and jumpy motions of the skydiver in some parts of the simulation. Therefore, we resample the data with a frame rate of 100 Hz and a spline interpolation (figure 3.2).

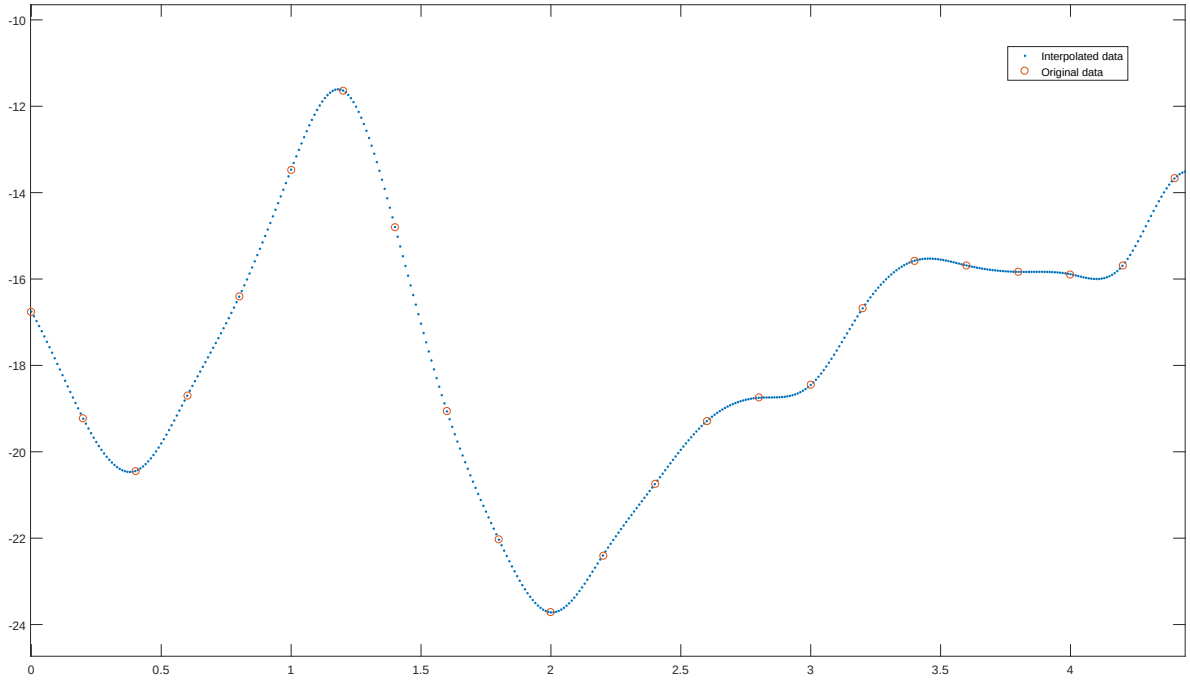


Figure 3.2.: Spline interpolation of the bank angle

We define the finer step size

```
dt = 0.01;
```

and compute the corresponding time vector:

```
sm(:, 1) = 0 : dt : t(end);
```

We move the starting position into the origin by subtracting the first values and use the new time vector for the spline interpolations of the translational components:

```
sm(:, 2) = ...
  spline (t, skydiver_motion(:, 3) - ...
  skydiver_motion(1, 3), sm(:, 1));

sm(:, 3) = ...
  spline (t, skydiver_motion(:, 2) - ...
  skydiver_motion(1, 2), sm(:, 1));

sm(:, 4) = ...
```

```
  spline (t, -skydiver_motion(:, 4) + ...
  skydiver_motion(1, 4), sm(:, 1));
```

The Euler angles have to be converted from degrees to radians; we use the `unwrap` command to take care of 2π-jumps (especially in the azimuth angle); the `hampel` command effectively eliminates outliers, before we finally do the spline resampling:

```
sm(:, 5) = ...
  spline (t, ...
  hampel (unwrap ...
  (pi/180*skydiver_motion(:, 5))), sm(:, 1));

sm(:, 6) = ...
  spline (t, ...
  hampel (unwrap ...
  (pi/180*skydiver_motion(:, 6))), sm(:, 1));

sm(:, 7) = ...
  spline (t, ...
  hampel (unwrap ...
  (pi/180*(skydiver_motion(:, 7) + 19.5))), sm(:, 1));
```

4. trimmod

We used `trimmod` from [9] to find an unaccelerated equilibrium (trim point) where the skydiver and the UAV both have the same initial velocity vector.

4.1. Documentation

From the documentation of `trimmod`:

4.1.1. Syntax

```
trimmod

h = trimmod
```

4.1.2. Description

`trimmod` finds the trim point (equilibrium) of a Simulink system. When invoked without left-hand arguments, `trimmod` opens a new figure with a graphical user interface (figure 4.1).

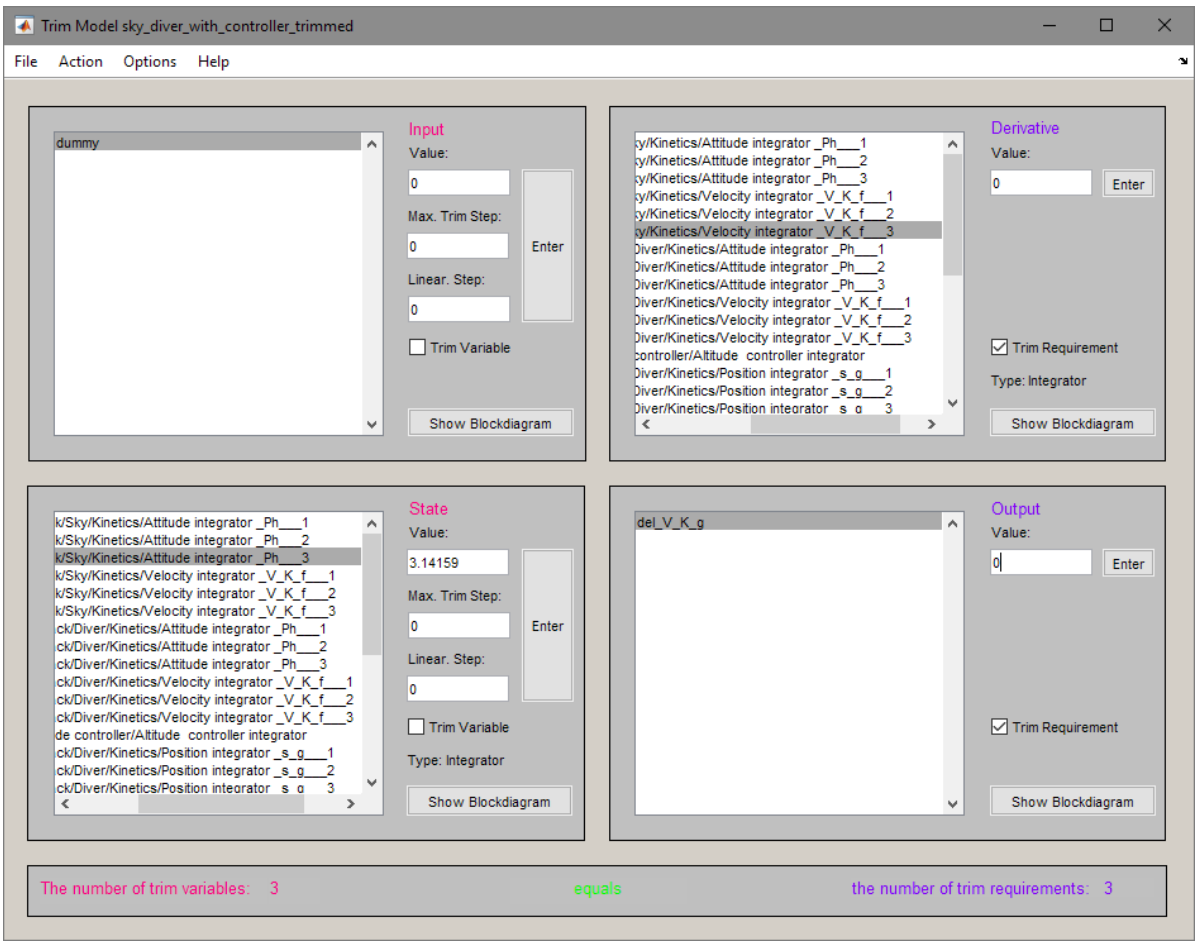


Figure 4.1.: Trimmod graphical user interface

The user can load a Simulink system (`.mdl` or `.slx`), define certain trim point requirements and ask `trimmod` to calculate the corresponding trim point variables that are necessary to satisfy the requirements. This trim point is then automatically transferred to the Simulation; Model Configuration Parameters; Data Import/Export; Load from workspace dialog box (Input and Initial state) of the corresponding Simulink system.

When invoked with a left-hand argument,

```
h = trimmod
```

opens the gui and additionally returns the handle of the figure.

4.1.3. Arguments

`trimmod` does not need any input arguments. The optional output argument is the handle of the newly opened figure.

4.1.4. Example

- Invoke the graphical user interface: `trimmod`
- Open the Simulink system named `trimtest.mdl`: File; Open Model; trimtest.mdl
- Load the trim point from the file `trimtest.mat`: File; Load Trim Point; trimtest.mat
- Check everything in one view: Action; Overview
- Trim the system: Action; Trim
- Simulate the system using Simulink
- Modify the trim point using the graphical user interface
- Save the new trim point: File; Save Trim point in trimtest.mat
- Trim again, simulate again, ...

4.1.5. Menu

Table 4.1 describes the effect of `trimmod`'s menu entries.

Table 4.1.: Trimmod menu

Menu command	**Action**
File; Open Model	Open a Simulink system via file select dialog box.
File; Load Trim Point	Load a trim point that has been previously saved, via file select dialog box.
File; Save Trim Point in ...	Save current trim point in a .mat-file whose name is the name of the current Simulink system. It might be useful to save a newly defined trim point before calling the trim algorithm because trim requirements and trim variables are modified by the trim algorithm. (see Action; Untrim)
File; Save Trim Point as	Save current trim point via file select dialog box.
File; Exit TrimMod	Game over. Ask user if he wants to save current trim point.
Action; Overview	Display an overview over all inputs, states, state derivatives, and outputs along with their pre- and post-trim values and the information, whether they are trim variable or trim requirements.
Action; Trim	Trim current Simulink system using current trim requirements and trim variables. It might be useful to save a newly defined trim point before calling the trim algorithm (see File; Save Trim Point in ...).
Action; Untrim	Countermand the effects of the previous trim. If trimming was not successful (because of bad starting guesses, unrealizable trim requirements, or linear dependencies), the trim algorithm aborts and the values in the gui represent the current (possibly totally wrong) state of the algorithm. This error state might be very useful for the analysis of trim problems, but a reload of the original trim point (via Load Trim Point or Untrim) might be necessary prior to the next trim cycle.
Options; Show Tooltips	Tooltips are very useful for the inexperienced user, but can become quite annoying after a while. Therefore, they can be switched off via a check button.
Help; Help on TrimMod	This manual
Help; About TrimMod	The usual "about"-information: version, copyright, author, ...

4.1.6. Algorithm

A nonlinear time invariant system can be described via its differential equation system

$$d = f(x, u)$$

and its output equation system

$$y = g(x, u)$$

where u is the input vector, x is the state vector, $d = \dot{x}$ is the time derivative of the state vector, y is the output vector, and f and g are nonlinear vector functions, evaluated every simulation time step. State vector x and input vector u are the independent variables on the right-hand side of the equations. Both vectors can be combined into a generalized input vector

$$xu = \begin{bmatrix} x \\ u \end{bmatrix}$$

Derivative vector d and output vector y are the left-hand side results of the function evaluations. They can be combined into the generalized output vector

$$dy = \begin{bmatrix} d \\ y \end{bmatrix}$$

Both equation systems can then be combined into

$$dy = h(xu) \tag{4.1}$$

where $h = \begin{bmatrix} f \\ g \end{bmatrix}$ is the generalized system vector function.

To start a simulation, all elements of the generalized input vector xu (the complete x and u vectors) have to be known for the first evaluations of equation (4.1). Unfortunately, the trim point is often defined as a mixture of u, x, d, and y: The initial speed (x) of a car might be known, but not the corresponding engine power or the accelerator angle (u) for no acceleration (d). The radius of the curve might be predefined, but not the corresponding turning wheel angle, ... Usually the user initially defines some elements of the generalized output vector dy that have to be satisfied, and some elements of the generalized input vector xu that are known a priori. The other (unknown) elements of the generalized input vector xu have to be found by the trim algorithm. The unknown elements of the generalized output vector dy can then easily be calculated via equation (4.1) if xu is completely known.

Both generalized vectors can therefore be split up into a known (subscript k) and an unknown (subscript n) part:

$$dy = \begin{bmatrix} dy_k \\ dy_n \end{bmatrix}$$

$$xu = \begin{bmatrix} xu_k \\ xu_n \end{bmatrix}$$

Accordingly, equation (4.1) too can be split up into two (vector) equations, one for the predefined elements of dy and one for the unknowns:

$$dy_k = h_k(xu) = h_k\left(\begin{bmatrix} xu_k \\ xu_n \end{bmatrix}\right) \tag{4.2}$$

$$dy_n = h_n(xu) = h_n\left(\begin{bmatrix} xu_k \\ xu_n \end{bmatrix}\right)$$

The trim algorithms now has to solve the nonlinear equation system (4.2) with respect to the unknown vector xu_n (called the *trim variables* vector), while the vector dy_k is called the *trim requirements* vector.

Trim requirements dy_k Those (known) elements of the generalized output vector dy that have to be satisfied

Trim variables xu_n Those (unknown) elements of the generalized input vector xu that the trim algorithm is free to vary

For a unique solution of equation (4.2) the number of (unknown) trim variables (`length (xu_n)`) has to equal the number of equations, given by the number of trim requirements (`length (dy_k)`).

If this prerequisite is fulfilled, `trimmod` (the graphical user interface) calls `jj_trim` (the actual trim algorithm).

As shown in figure 4.2, the first step of `jj_trim` is to put in the initial guess of the trim variable vector $xu_{n_{old}}$ on the right hand side of equation (4.2) and to check whether the trim requirement vector $dy_{k_{trim}}$ is already met by $dy_{k_{old}}$. As this is usually not the case, a modified multidimensional Newton-Raphson-algorithm is used to iteratively find new trim variable vectors $xu_{n_{new}}$ that - hopefully - finally approach the sought $xu_{n_{trim}}$.

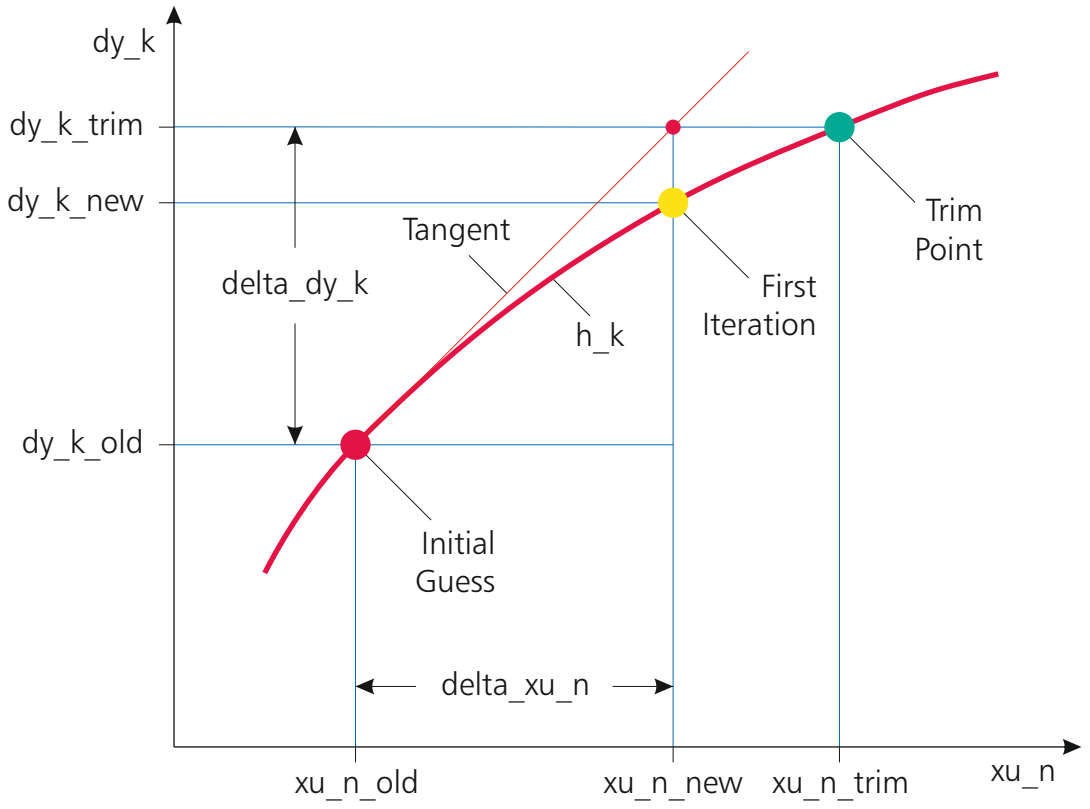


Figure 4.2.: One-dimensional Newton-Raphson step

Newton-Raphson relies on the local derivatives which can graphically be represented as a tangent hyperplane in the multidimensional case. The linearization routine `jj_lin` finds the gradients of this tangent hyperplane at $xu_{n_{old}}$ and returns a sensitivity matrix (Jacobean matrix) $jaco$, which represents the linear relation

$$\Delta dy_k = jaco \cdot \Delta xu_n \tag{4.3}$$

of the trim requirement error

$$\Delta dy_k = dy_{k_{trim}} - dy_{k_{old}}$$

with respect to the trim variable correction

$$\Delta xu_n = xu_{n_{new}} - xu_{n_{old}} \tag{4.4}$$

A singular system decomposition (singular values and singular vectors) of the sensitivity matrix $jaco$ is done, in order to find trim variables that have no influence on any trim requirement, trim requirements that cannot be influenced by any trim variable, and linear dependencies of trim variables or trim requirements. One or more singular values of zero indicate a wrong choice of trim requirements and/or trim variables. The corresponding

singular vectors clearly show which trim requirements and trim variables are responsible for the rank deficiency. This detailed information can then be used to chose those trim requirements and trim variables that correctly describe the desired trim state.

If the sensitivity matrix $jaco$ has full rank (is non-singular), the linear equation system (4.3) can be solved:

$$\Delta xu_n = jaco \backslash \Delta dy_k$$

and equation (4.4) can be used to find the next solution vector:

$$xu_{n_{new}} = xu_{n_{old}} + \Delta xu_n$$

4.2. Trimmod overview for UAV and skydiver

Figure 4.3 shows the pre-trim[1] and post-trim values for the trim variables

- sink velocity of the UAV
- sink velocity of the skydiver
- altitude controller integrator

and the trim requirements

- no sink accelerations of the UAV
- no sink accelerations of the skydiver
- no difference between the sink velocities of UAV and skydiver

Additionally, we define that

- the UAV's initial azimuth should be π in order to have it "look" towards the skydiver
- the UAV should have an initial distance of 5 m (in positive x-direction) from the skydiver

[1] We choose very realistic initial values in order to ease the work for the trim algorithm and prevent it from iterating into saturations.

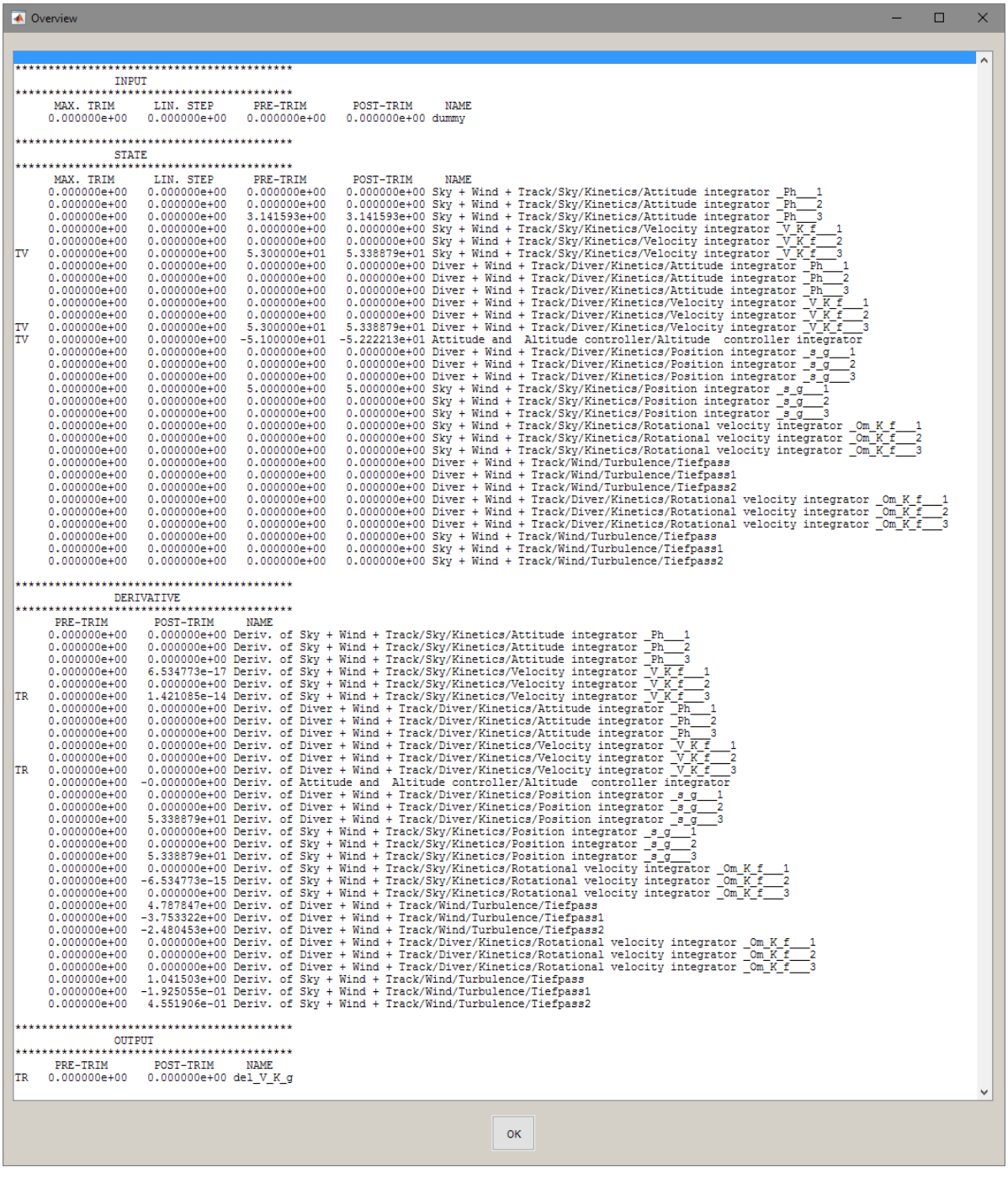


Figure 4.3.: Trimmod overview

Bibliography

[1] D. How, B. Barbarich-Bacher, and K. Stol, "Design and Analysis of a UAV for Skydiving." IEEE Int. Conf. on Unmanned Aircraft Systems (ICUAS'15), 2015.

[2] G. Vallabha. (2016) Real-Time Pacer for Simulink. [Online]. Available: https://de.mathworks.com/matlabcentral/fileexchange/29107-real-time-pacer-for-simulink

[3] J. J. Buchholz. (2016) Skript Regelungstechnik und Flugregler. [Online]. Available: http://prof.red/rtfr/skript/skript10.pdf

[4] ——. (2016) Transfer graphics handle between methods in Level-2 MATLAB S-Function. [Online]. Available: https://de.mathworks.com/matlabcentral/answers/313422-transfer-graphics-handle-between-methods-in-level-2-matlab-s-function

[5] The Mathworks. (2016) Camera Graphics Terminology. [Online]. Available: https://de.mathworks.com/help/matlab/visualize/defining-scenes-with-camera-graphics.html

[6] C. Tornau. (2016) Drehungen um beliebige Achsen mit Hilfe der Rotationsmatrix. [Online]. Available: http://www.informatikseite.de/animation/node14.php

[7] Autodesk Premium. (2016) Male Skydiving - Splayed. [Online]. Available: http://www.123dapp.com/123C-3D-Model/Male-Skydiving--Splayed/659997

[8] E. Johnson. (2016) Stl file reader. [Online]. Available: https://de.mathworks.com/matlabcentral/fileexchange/22409-stl-file-reader/content/STLRead/stlread.m

[9] J. J. Buchholz. (2000) trimmod. [Online]. Available: https://de.mathworks.com/matlabcentral/fileexchange/268-trimmod

A. Aerodynamic frame of an axially symmetric body falling down

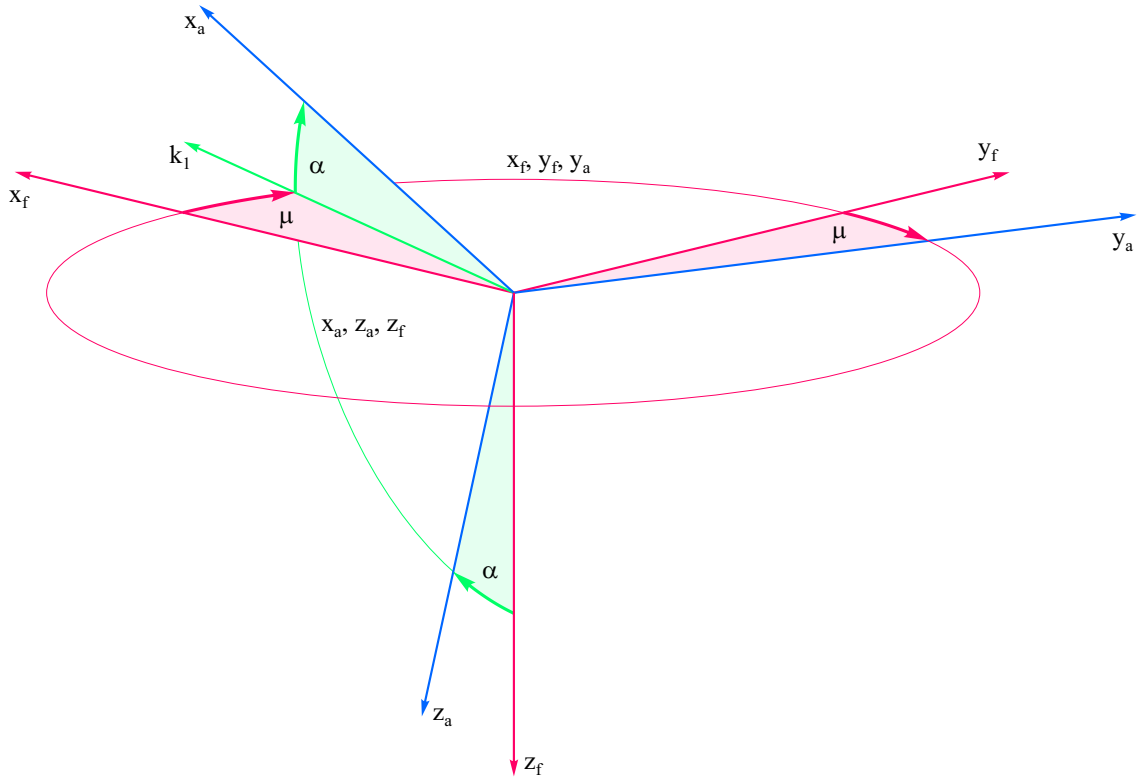


Figure A.1.: Transformation from the body-fixed frame (index f) to the aerodynamic frame (index a)

- The body falls down in a general z-direction.
- The body is axially symmetric with respect to its z_f-axis.
- The apparent airflow (and therefore the drag) point towards the negative z_a-axis.
- The x_a-axis lies in the z_a-z_f-plane by definition. As a consequence, the y_a-axis lies in the x_f-y_f-plane.
- The lift points towards the negative x_a-axis.

- There is no sideslip angle, no side force, no roll moment, and no yawing moment in the aerodynamic frame. Lift, drag and pitching moment only depend on α (s. below).
- The direction of rotation (transformation), as shown in figure A.1, is from the body-fixed frame (index f) to the aerodynamic frame (index a), which is the opposite direction as used in the norm.
- The first rotation is about the z_f-axis in the x_f-y_f-plane with the (aerodynamic yaw or azimuth) angle μ which can rotate for a full circle: $-\pi < \mu \leq \pi$.
- The second rotation is about the y_a-axis in the x_a-z_a-plane with the angle[1] (of attack) α which can only rotate for half a circle: $0 \leq \alpha \leq \pi$.

Therefore, the transformation matrix $\boldsymbol{M_{af}}$ from the f-system to the a-system is a transformation about a z-axis with the angle μ, followed (as read from the right to the left) by a transformation about a y-axis with the angle α:

$$\begin{aligned}\boldsymbol{M_{af}} &= \boldsymbol{M_y}(\alpha) \cdot \boldsymbol{M_z}(\mu) \\ &= \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{bmatrix} \cdot \begin{bmatrix} \cos\mu & \sin\mu & 0 \\ -\sin\mu & \cos\mu & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos\alpha\cdot\cos\mu & \cos\alpha\cdot\sin\mu & -\sin\alpha \\ -\sin\mu & \cos\mu & 0 \\ \sin\alpha\cdot\cos\mu & \sin\alpha\cdot\sin\mu & \cos\alpha \end{bmatrix}\end{aligned}$$

The opposite transformation matrix $\boldsymbol{M_{fa}}$(from the a-frame to the f-frame) is the inverse (and fortunately also the transpose, because we are dealing with orthogonal matrices) of $\boldsymbol{M_{af}}$:

$$\begin{aligned}\boldsymbol{M_{fa}} &= \boldsymbol{M_{af}^{-1}} = \boldsymbol{M_{af}^{T}} \\ &= \begin{bmatrix} \cos\alpha\cdot\cos\mu & -\sin\mu & \sin\alpha\cdot\cos\mu \\ \cos\alpha\cdot\sin\mu & \cos\mu & \sin\alpha\cdot\sin\mu \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix}\end{aligned} \tag{A.1}$$

We now want to express the spherical components of the aerodynamic velocity vector (V_A, α, μ) with respect to its Cartesian components (u_{Af}, v_{Af}, w_{Af}) in the body-fixed frame.

The first spherical component V_A is just the norm of the velocity vector:

$$V_A = \sqrt{u_{Af}^2 + v_{Af}^2 + w_{Af}^2} \tag{A.2}$$

[1] We could have defined α in the opposite direction to make it point more towards the x_f-axis instead of pointing towards the x_a-axis, which would be closer to the norm. On the other hand, phrases like "from above" or "from below" lose their significance with an axially symmetric body, our definition follows the right hand rule, and in the end it's just a question of definition and consequently applying this definition.

The aerodynamic velocity vector expressed in Cartesian coordinates in the body-fixed frame

$$\boldsymbol{V_{Af}} = \begin{bmatrix} u_A \\ v_A \\ w_A \end{bmatrix}_f = \begin{bmatrix} u_{Af} \\ v_{Af} \\ w_{Af} \end{bmatrix} \tag{A.3}$$

equals the aerodynamic velocity vector expressed in the aerodynamic frame $\boldsymbol{V_{Aa}}$ transformed from the aerodynamic frame to the body-fixed frame:

$$\boldsymbol{V_{Af}} = \boldsymbol{M_{fa}} \cdot \boldsymbol{V_{Aa}} \tag{A.4}$$

Respecting the fact, that the velocity vector only has a z-component in the aerodynamic frame

$$\boldsymbol{V_{Aa}} = \begin{bmatrix} 0 \\ 0 \\ V_A \end{bmatrix} \tag{A.5}$$

we can use equation (A.1), equation (A.3), and equation (A.5) in equation (A.4):

$$\begin{aligned} \begin{bmatrix} u_{Af} \\ v_{Af} \\ w_{Af} \end{bmatrix} &= \begin{bmatrix} \cos\alpha \cdot \cos\mu & -\sin\mu & \sin\alpha \cdot \cos\mu \\ \cos\alpha \cdot \sin\mu & \cos\mu & \sin\alpha \cdot \sin\mu \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ V_A \end{bmatrix} \\ &= \begin{bmatrix} V_A \cdot \sin\alpha \cdot \cos\mu \\ V_A \cdot \sin\alpha \cdot \sin\mu \\ V_A \cdot \cos\alpha \end{bmatrix} \end{aligned} \tag{A.6}$$

The third vector component of equation (A.6)

$$w_{Af} = V_A \cdot \cos\alpha$$

returns the equation for the angle of attack:

$$\alpha = \arccos\left(\frac{w_{Af}}{V_A}\right) \tag{A.7}$$

Dividing the second by the first vector component of equation (A.6)

$$\begin{aligned} \frac{v_{Af}}{u_{Af}} &= \frac{V_A \cdot \sin\alpha \cdot \sin\mu}{V_A \cdot \sin\alpha \cdot \cos\mu} \\ &= \tan\mu \end{aligned}$$

returns the equation for the aerodynamic yaw angle:

$$\mu = \arctan\left(\frac{v_{Af}}{u_{Af}}\right) \tag{A.8}$$

A quick consistency check of equation (A.7) shows that α is indeed zero if the airflow only has a z-component in the body-fixed frame ($w_{Af} = V_A$) and that the yaw angle μ is not defined[2] in that case ($\frac{0}{0}$) according to equation (A.8).

A zero yaw angle corresponds to a v_{Af} of zero according to equation (A.8), while $v_{Af} = u_{Af}$ mathematically and physically results in a yaw angle of $\frac{\pi}{4}$. If u_{Af} is zero the argument of the arc tangent becomes infinity and the yaw angle is correctly computed as $\pm\frac{\pi}{2}$ (sign depending on the sign of v_{Af}). As usual, we have to use the `atan2`-function with two arguments to come up with yaw angles of a magnitude greater than $\frac{\pi}{2}$.

[2]Matlab defines `atan2 (0, 0) = 0`, which is a bit unconventional but quite helpful in our case.