version will be available by September 2017. Please visit this page again.

Online Textbook

The Drones Demystified program is also accompanied with an **Online Textbook** with the hope that this corresponds to a comprehensive educational material. The contents of the online textbook are the following:

1. **Modeling and Dynamics Formulation**
    1. Frame Rotations and Representations
    2. Dynamics of a Multirotor Micro Aerial Vehicle
    3. Dynamics of a Fixed-Wing Unmanned Aerial Vehicle
2. **State Estimation**
    1. Inertial Sensors
    2. The Kalman Filter
    3. Inertial Navigation Systems
3. **Flight Controls**
    1. PID Control
    2. LQR Control
    3. Linear Model Predictive Control
    4. An Autopilot Solution
4. **Motion Planning**
    1. Holonomic Vehicle Boundary

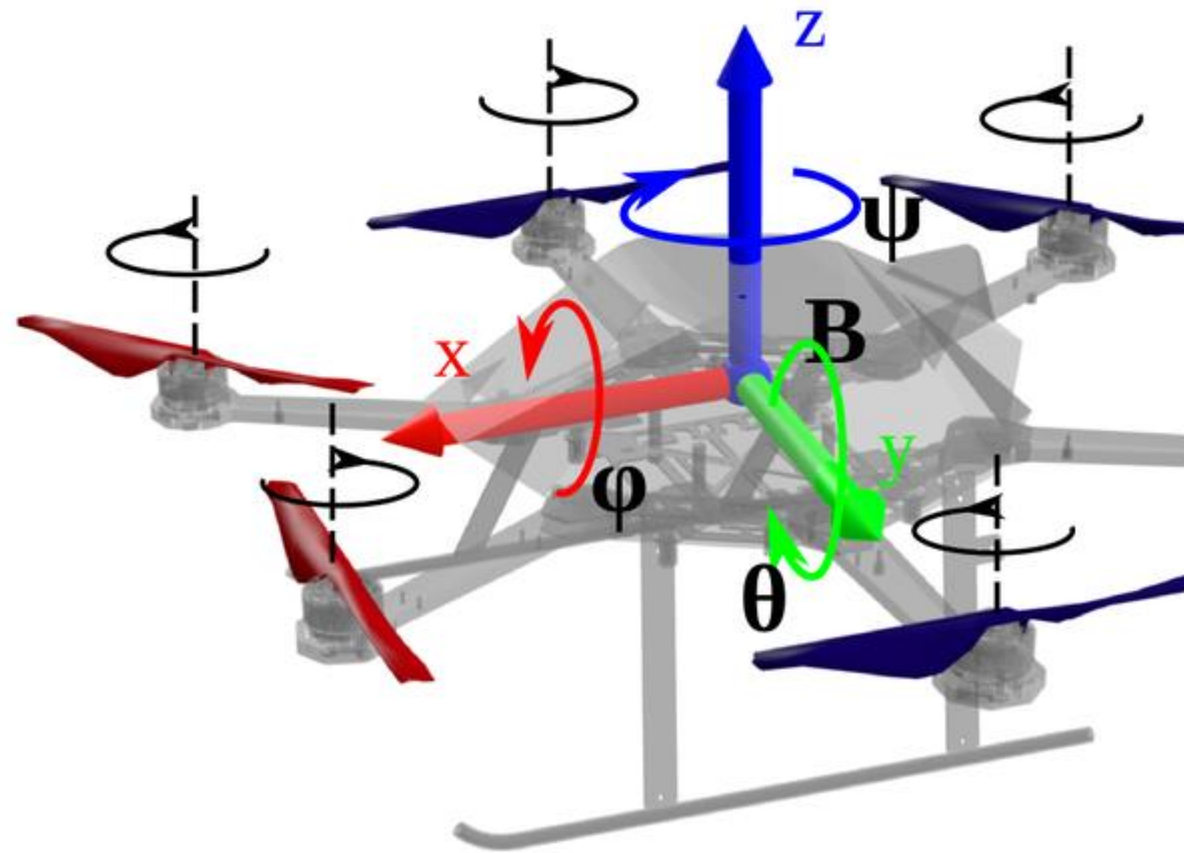
Value Solver
2. Dubins Airplane model Boundary Value Solver
3. Collision-free Navigation
4. Structural Inspection Path Planning
5. **Simulation Tools**
1. Simulations with SimPy
2. The RotorS Integrated Simulation Solution

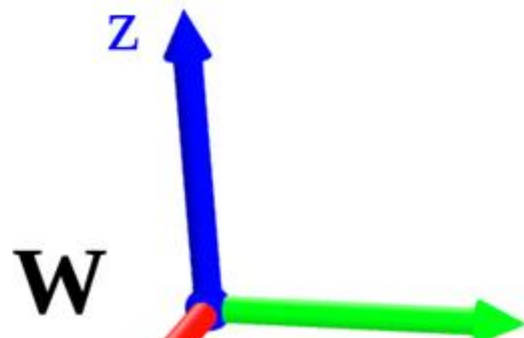
Visit this page also later in time as it will be continuously updated.

Frame Rotations and Representations

Euler Angles

Euler angles are a method to determine and represent the rotation of a body as expressed in a given coordinate frame. They are defined as three (chained) rotations relative to the three major axes of the coordinate frame. Euler angles are typically representes as phi (φ) for x-axis rotation, theta (θ) for y-axis rotation, and psi (ψ) for z-axis rotation. Any orientation can be described through a combination of these angles. Figure 1 represents the Euler angles for a multirotor aerial robot.





These elemental

As seen there are many ways to do this set of rotations - with the variations be based on the order of rotations. All would be formally acceptable, but some are much more commonly used than others.

Among them, one that is particularly widely used is the following: start with the body fixed-frame (attached on the vehicle) (x,y,z) aligned with the inertial frame (X,Y,Z), and then perform 3 rotations to re-orient the body frame.

1. Rotate by ψ about Z : x', y', z'
2. Rotate by θ about y' : x'', y'', z''
3. Rotate by φ about x'' '' x, y, z

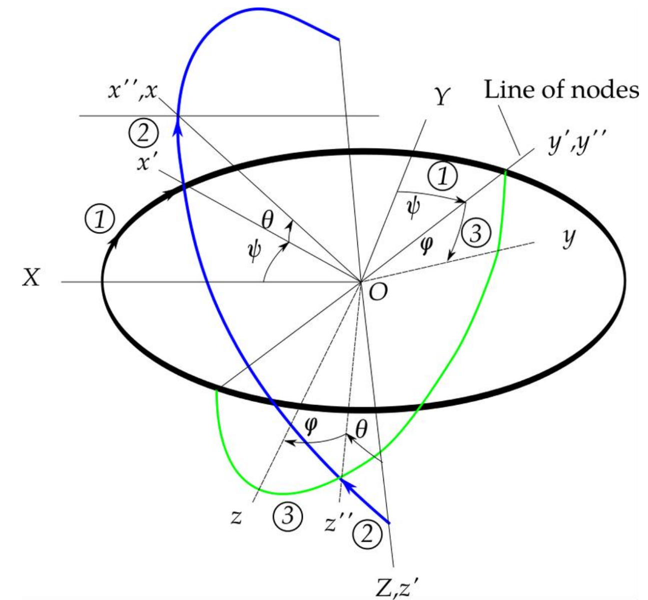


Figure 2: Representation of the Euler Angles.

Euler angles:

- ψ : Yaw (heading)
- θ : Pitch
- φ : Roll

To learn more about different conventions, please visit:

- https://en.wikipedia.org/wiki/Euler_angles
- http://www.mathworks.com/discovery/euler-angles.html
- http://mathworld.wolfram.com/EulerAngles.html

We can write the aforementioned set of rotations as:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \mathbf{R}_\psi \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

$$\begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \mathbf{R}_\theta \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix} = \mathbf{R}_\phi \begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix}$$

which combines to give:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \mathbf{R}_\phi \mathbf{R}_\theta \mathbf{R}_\psi \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} \cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta \\ -\cos\psi\sin\psi + \sin\phi\sin\theta\cos\psi & \cos\phi\cos\psi + \sin\phi\sin\theta\sin\psi & \sin\phi\cos\theta \\ \sin\phi\sin\psi + \cos\phi\sin\theta\cos\psi & -\sin\phi\cos\psi + \cos\phi\sin\theta\sin\psi & \cos\phi\cos\theta \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

To get the angular velocity, we will have to include three terms, namely the time derivative of ψ around Z, the time derivative of θ around y' and the time derivative of φ around x''. These three terms are combined to give the vector of angular velocities ω. The final result takes the form:

$$\omega_x = \dot\phi - \dot\psi \sin\theta$$
$$\omega_y = \dot\theta \cos\phi + \dot\psi \cos\theta \sin\phi$$
$$\omega_z = -\dot\theta \sin\phi + \dot\psi \cos\theta \cos\phi$$

And in inverse form:

$$\dot\phi = \omega_x + [\omega_y \sin\phi + \omega_z \cos\phi] \tan\theta$$
$$\dot\theta = \omega_y \cos\phi - \omega_z \sin\phi$$
$$\dot\psi = [\omega_y \sin\phi + \omega_z \cos\phi] \sec\theta$$

Note that one has to take care for singularities such as the pitch angle at plus/minus 90 degrees. A typical set of Euler Angles considers the following set of limitations:

$$\begin{bmatrix} 0 \\ -\frac{\pi}{2} \\ 0 \end{bmatrix} \leq \begin{bmatrix} \psi \\ \theta \\ \phi \end{bmatrix} \begin{bmatrix} 2\pi \\ \frac{\pi}{2} \\ 2\pi \end{bmatrix}$$

Quaternions

Theorem by Euler states that any given sequence of rotations can be represented as a signle rotation about a single fixed axis. The concept of Quaternions provides a convenient parametrization of this effective axis and the rotation angle:

$$\ddot{b} \quad \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \quad \begin{bmatrix} \bar{E}\sin\zeta/2 \\ \cos\zeta/2 \end{bmatrix}$$

where $\dot{E}$ is a unit vector ζ is a positive rotation around E

Notes:

- $||\bar{b}||=1$ abd thus there are only 3 degrees of freedom in this formulation, and
- if $\bar{b}$ represents the rotational transformation from the reference frame (α) to the frame b, the frame α is aligned with frame b if frame α is rotated by ζ about $\bar{E}$

This representation is connected with the Euler angles form, according to the following expression:

$$\begin{aligned} \sin\theta & \quad -2(b_2 b_4 + b_1 b_3) \\ \psi & = \arctan 2[2(b_2 b_3 - b_1 b_4), 1 - 2(b_1^2 + b_2^2)]) \\ \psi & \quad \arctan 2[2(b_1 b_2 - b_3 b_4), 1 - 2(b_2^2 + b_3^3)] \end{aligned}$$

This representation has the great advantage of being:

- Singularity-free and
- Computationally efficient to do state propagation (typically within an Extended Kalman Filter)

On the other hand, it has one main disadvantage, namely being far less intuitive.

Euler to-and-from Quaternions Python Implementation

File: **QuatEulerMain.py**

```
#       __QUATEULERMAIN__
#       This main file demonstrates functions for handling
#       and manipulating quaternios and Euler Angles
#
#       Authors:
#       Kostas Alexis (kalexis@unr.edu)
```

```
from numpy import *
import numpy as np
from QuatEulerFunctions import *

# demo values
q_ = np.array([0.25,0.5,0.1,0.2])

print 'Quaternion: '
print q_

rpy_ = quat2rpy(q_)
print 'Euler angles'
print rpy_

quat_ = rpy2quat(rpy_)
print 'Recovered quaternion:'
print quat_

rot_ = quat2r(quat_)
print 'Recovered rotation matrix:'
print rot_

rpy_rec_ = r2rpy(rot_)
print 'Recovered Euler'
print rpy_rec_

q_norm_ = normalized(q_)
print 'Normalized quaternion:'
print q_norm_
```

File: **QuatEulerFunctions.py**

```
#       __QUATEULERFUNCTIONS__
#       This file implements functions for handling
#       and manipulating quaternios and Euler Angles
```

```python
#
#       Authors:
#       Kostas Alexis (kalexis@unr.edu)

from numpy import *
import numpy as np

def quat2r(q_AB=None):
    C_AB = np.zeros((3,3))
    C_AB[0,0] = q_AB[0]*q_AB[0] - q_AB[1]*q_AB[1] - q_AB[2]*q_AB[2] + q_AB[3]*q_AB[3]
    C_AB[0,1] = q_AB[0]*q_AB[1]*2.0 + q_AB[2]*q_AB[3]*2.0
    C_AB[0,2] = q_AB[0]*q_AB[2]*2.0 - q_AB[1]*q_AB[3]*2.0

    C_AB[1,0] = q_AB[0]*q_AB[1]*2.0 - q_AB[2]*q_AB[3]*2.0
    C_AB[1,1] = -q_AB[0]*q_AB[0] + q_AB[1]*q_AB[1] - q_AB[2]*q_AB[2] + q_AB[3]*q_AB[3]
    C_AB[1,2] = q_AB[0]*q_AB[3]*2.0 - q_AB[1]*q_AB[2]*2.0

    C_AB[2,0] = q_AB[0]*q_AB[2]*2.0 + q_AB[1]*q_AB[3]*2.0
    C_AB[2,1] = q_AB[0]*q_AB[3]*(-2.0) + q_AB[1]*q_AB[2]*2.0
    C_AB[2,2] = -q_AB[0]*q_AB[0] - q_AB[1]*q_AB[1] + q_AB[2]*q_AB[2] + q_AB[3]*q_AB[3]
    return C_AB

def quat2rpy(q_AB=None):
    C = quat2r(q_AB)
    theta = np.arcsin(-C[2,0])
    phi = np.arctan2(C[2,1],C[2,2])
    psi = np.arctan2(C[1,0],C[0,0])
    rpy = np.zeros((3,1))
    rpy[0] = phi
    rpy[1] = theta
    rpy[2] = psi
    return rpy

def rpy2quat(rpy=None):
```

```
    r = rpy[0]
    p = rpy[1]
    y = rpy[2]
    cRh = np.cos(r/2)
    sRh = np.sin(r/2)
    cPh = np.cos(p/2)
    sPh = np.sin(p/2)
    cYh = np.cos(y/2)
    sYh = np.sin(y/2)
    qs_cmpl = np.array([ -(np.multiply(np.multiply(sRh,cPh),cYh) - np.multiply(np.multiply(cRh,sPh),sYh)),
                         -(np.multiply(np.multiply(cRh,sPh),cYh) + np.multiply(np.multiply(sRh,cPh),sYh)),
                         -(np.multiply(np.multiply(cRh,cPh),sYh) - np.multiply(np.multiply(sRh,sPh),cYh)),
                         np.multiply(np.multiply(cRh,cPh),cYh) + np.multiply(np.multiply(sRh,sPh),sYh)])
    qs = np.real(qs_cmpl)
    return qs

def r2rpy(C=None):
    theta = np.arcsin(-C[2,0])
    phi = np.arctan2(C[2,1],C[2,2])
    psi = np.arctan2(C[1,0],C[0,0])
    rpy = np.zeros((3,1))
    rpy[0] = phi
    rpy[1] = theta
    rpy[2] = psi
    return rpy

def normalized(x=None):
    y=x/np.sqrt(np.dot(x,x))
    return y
```

Euler to-and-from Quaternions MATLAB Implementation

File: **QuatEulerMain.m**

```
%         __QUATEULERMAIN__
%         This main file demonstrates functions for handling
```

```
%         and manipulating quaternios and Euler Angles
%
%         Authors:
%         Shehryar Khattak (shehryar@nevada.unr.edu)

%Clear Workspace and Command Window
clear;clc;

%Demo Quaternion value
q = [0.25 0.5 0.1 0.2];

%Display Quaternion
fprintf('\n --- Quaternion --- \n')
disp(q');

%Normalize Quaternion
q_norm=q/norm(q);
fprintf('\n --- Normailized Quaternion --- \n');
disp(q_norm');

%Euler Angles
eul_ang=quat2rpy(q);
fprintf('\n --- Euler Angles --- \n')
fprintf('Roll=%f\nPitch=%f\nYaw=%f\n',eul_ang(1),eul_ang(2),eul_ang(3));

%Recovered Quaternion
q_rec=rpy2quat(eul_ang);
fprintf('\n --- Recovered Quaternion --- \n')
disp(q_rec');

%Rotation Matrix from Quaternion
rot_m=quat2r(q_rec);
fprintf('\n --- Rotation Matrix from Quaternion --- \n')
disp(rot_m);
```

```
%Recoverd Euler from Rotation Matrix
rpy_rec=r2rpy(rot_m);
fprintf('\n --- Recovered Euler Angles from Rotation Matrix --- \n');
fprintf('Roll=%f\nPitch=%f\nYaw=%f\n',rpy_rec(1),rpy_rec(2),rpy_rec(3));
```

File: **quat2r.m**

```
%         __Quaternion to Rotation Matrix__
%
%         Authors:
%         Shehryar Khattak (shehryar@nevada.unr.edu)
%         Reference: http://www.euclideanspace.com/maths/geometry/rotations/conversions/quaternionToMatrix/

%Convert Quaternion to Rotation Matrix
function [r] = quat2r(q)

    %Check if vector and if have it has enough input values
    if ~isvector(q) || (length(q))~=4
        error('Input to quat2r must be a vector and must have 4 values')
    end

    %Initialize
    r=zeros(3,3);

    %Normailize Quaternion before using
    q=q/norm(q);

    %Quaternion components
    qw=q(1);
    qx=q(2);
    qy=q(3);
    qz=q(4);

    %Rotation Matrix Elements
    r(1,1) = 1 - 2*qy^2 - 2*qz^2;
```

```
    r(1,2) = 2*qx*qy - 2*qz*qw;
    r(1,3) = 2*qx*qz + 2*qy*qw;

    r(2,1) = 2*qx*qy + 2*qz*qw;
    r(2,2) = 1 - 2*qx^2 - 2*qz^2;
    r(2,3) = 2*qy*qz - 2*qx*qw;

    r(3,1) = 2*qx*qz - 2*qy*qw;
    r(3,2) = 2*qy*qz + 2*qx*qw;
    r(3,3) = 1 - 2*qx^2 - 2*qy^2;

    %Equivalent built-in MATLAB function: quat2rotm
end
```

File: **quat2rpy.m**

```
%        __Quaternion to Euler Angles__
%
%        Authors:
%        Shehryar Khattak (shehryar@nevada.unr.edu)
%        Reference: Beard, Randal W., and Timothy W. McLain. Small unmanned aircraft: Theory and practice. Princeton
University Press, 2012. Appendix B, Page 259

%Convert Quaternion to Euler
function [eul_ang] = quat2rpy(q)

    %Check if vector and if have it has enough input values
    if ~isvector(q) || (length(q))~=4
        error('Input to quat2rpy must be a vector and must have 4 values')
    end

    %Normailize Quaternion before using
    q=q/norm(q);

    %Roll
    eul_ang(1)      =   atan2(   (2*(q(1)*q(2)+(q(3)*q(4)))) , (q(1)^2+q(4)^2-q(2)^2-q(3)^2) );
```

```
    %Pitch
    eul_ang(2)      =   asin(    2*((q(1)*q(3))-(q(2)*q(4)))  );
    %Yaw
    eul_ang(3)      =   atan2(    (2*(q(1)*q(4)+(q(2)*q(3))))  ,  (q(1)^2+q(2)^2-q(3)^2-q(4)^2)  );

    %Equivalent built-in MATLAB function: quat2eul
end
```

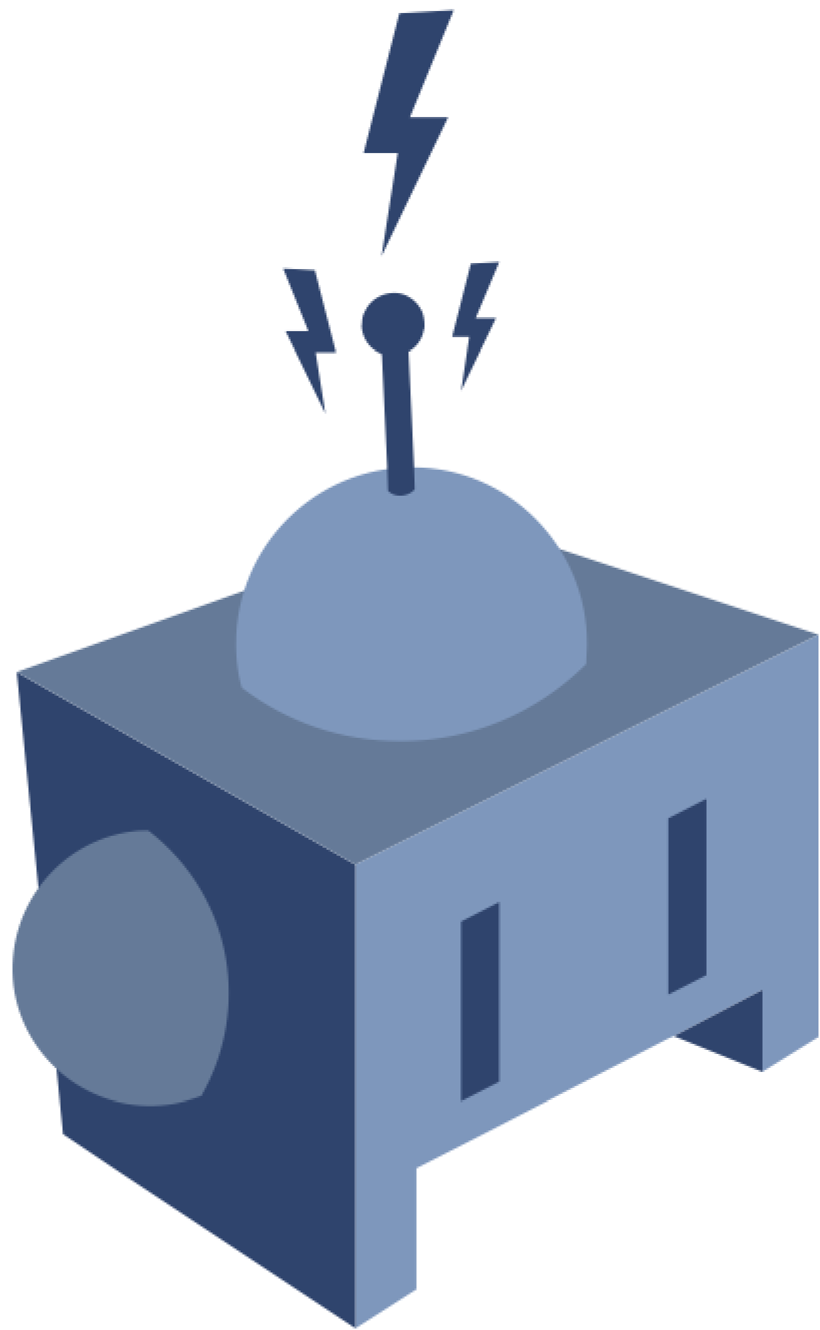
File: **r2rpy.m**

```
%        __Rotation Matrix to Euler Angles__
%
%        Authors:
%        Shehryar Khattak (shehryar@nevada.unr.edu)
%        Reference: http://nghiaho.com/?page_id=846


%Convert Rotation Matrix to Euler Angles
function [eul_ang]=r2rpy(rot_m)

    %Check if matrix and if have it has enough input values
    if ~ismatrix(rot_m) || numel(rot_m)~=9
        error('Input to r2rpy must be a 3x3 matrix')
    end

    eul_ang(1)=atan2(rot_m(3,2),rot_m(3,3));    %ro - roll
    eul_ang(2)=atan2(-rot_m(3,1),sqrt(rot_m(3,2)^2+rot_m(3,3)^2)); %theta - pitch
    eul_ang(3)=atan2(rot_m(2,1),rot_m(1,1));    %psi - yaw

    %Equivalent built-in MATLAB function: eul2rotm
end
```

File: **rpy2quat.m**

```
%        __Euler Angles to Quaternion__
%
%        Authors:
```

```
%        Shehryar Khattak (shehryar@nevada.unr.edu)
%        Reference: Beard, Randal W., and Timothy W. McLain. Small unmanned aircraft: Theory and practice. Princeton
University Press, 2012. Appendix B, Page 259

%Convert Euler anngles to Quaternion
function [q]=rpy2quat(eul_ang)

    %Check if vector and if have it has enough input values
    if ~isvector(eul_ang) || (length(eul_ang))~=3
        error('Input to eul_ang must be a vector and must have 3 values')
    end

    r=eul_ang(1)/2;
    p=eul_ang(2)/2;
    y=eul_ang(3)/2;

    q(1)= (cos(y)*cos(p)*cos(r)) + (sin(y)*sin(p)*sin(r)) ;
    q(2)= (cos(y)*cos(p)*sin(r)) - (sin(y)*sin(p)*cos(r)) ;
    q(3)= (cos(y)*sin(p)*cos(r)) + (sin(y)*cos(p)*sin(r)) ;
    q(4)= (sin(y)*cos(p)*cos(r)) - (cos(y)*sin(p)*sin(r)) ;

    %Equivalent built-in MATLAB function: eul2quat
end
```



AUTONO
ROBOTS
LAB

Dynamics of a Multirotor Micro Aerial Vehicle

Multirotor Micro Aerial Vehicles (MAVs), systems with a diameter of less than 1m and weight no more than a few kg, are among the prevailing aerial robotic configurations of our times. The reasons behind this outstanding success are related with the mechanical simplicity and robustness, flight performance and agility, payload capabilities and more.

Multirotor MAVs are also relatively easy to model mathematically. A great deal of complex aerodynamic phenomena that one has to account during conventional helicopter modeling are negligible in small multirotors. In fact, Multirotor MAVs can be (at least for control purposes) modeled as simply rigid body systems on which the lift and drag forces of their propellers act. Rigid-body dynamics studies the movement of systems of interconnected bodies under the action of external forces. The assumption that the bodies are rigid, which means that they do not deform under the action of applied forces, simplifies the analysis by reducing the parameters that describe the configuration of the system to the translation and rotation of reference frames attached to each body.

Video: Performance of a Nonlinear Model Predictive Controller during a propeller loss, designed using the presented model of the hexacopter MAV.

The two basic coordinate systems utilized to describe the vehicle's motion are: a) the **B** Body-Fixed-Frame (BFF), which is attached onto the Unmanned Aerial Vehicle's (UAV) Center-of-Mass (CoM) and rotates & translates along with its motion, and b) the **W** World-Fixed-Frame (WFF) which is commonly assumed as a flat earth frame. Those are visualized in Figure 1 below.

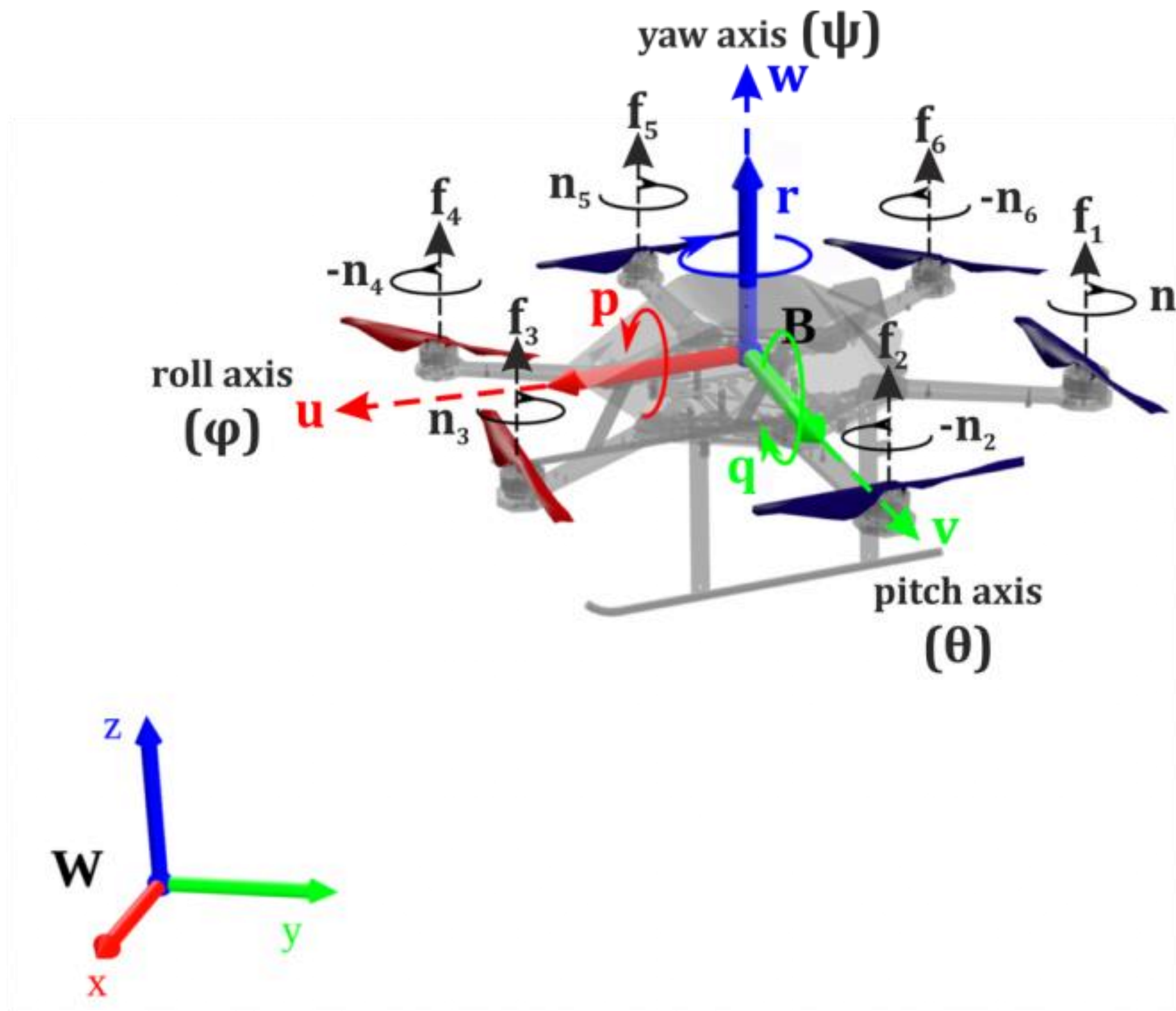


Figure 1: Coordinate frames used for the modeling of the Micro Aerial Vehicle and forces acting on it.

The baseline state variables for the multirotor MAV are:

- $\omega=[p,q,r]^T$ marks the BFF-based angular rotation rates
- $\mathbf{\Theta}=[\phi,\theta,\psi]^T$ marks the WFF-based rotation angles
- $\mathbf{U}=[u,v,w]^T$ marks the BFF-based velocities
- $\mathbf{p}=[x,y,z]^T$ marks the WFF-based position
- $\{n_i,f_i,M_i\},i\in[1,6]$ marks the i-th rotor's rotation speed, thrust force, and moment respectively, and specifically for the hexarotor MAV shown in Figure 1, the generated thrust and moment are given as:

$$\begin{aligned} f_i &= k_n\ n_i^2 \\ M_i &= (-1)^{i-1}\ k_m\ f_i \end{aligned}$$

The rigid-body [1] approach is typically followed to derive the dynamics of multirotors: The aerial vehicle's structure (as well as the propellers') is supposed to be rigid, and the BFF-origin is selected such that it coincides with the CoM. The active forces and moments (either by external sources such as gravity and aerodynamic drag, or generated by the multirotors' actuation authorities) are then appended onto this rigid-body of total mass and moment of inertia [2] matrix $\boldsymbol{J}$.

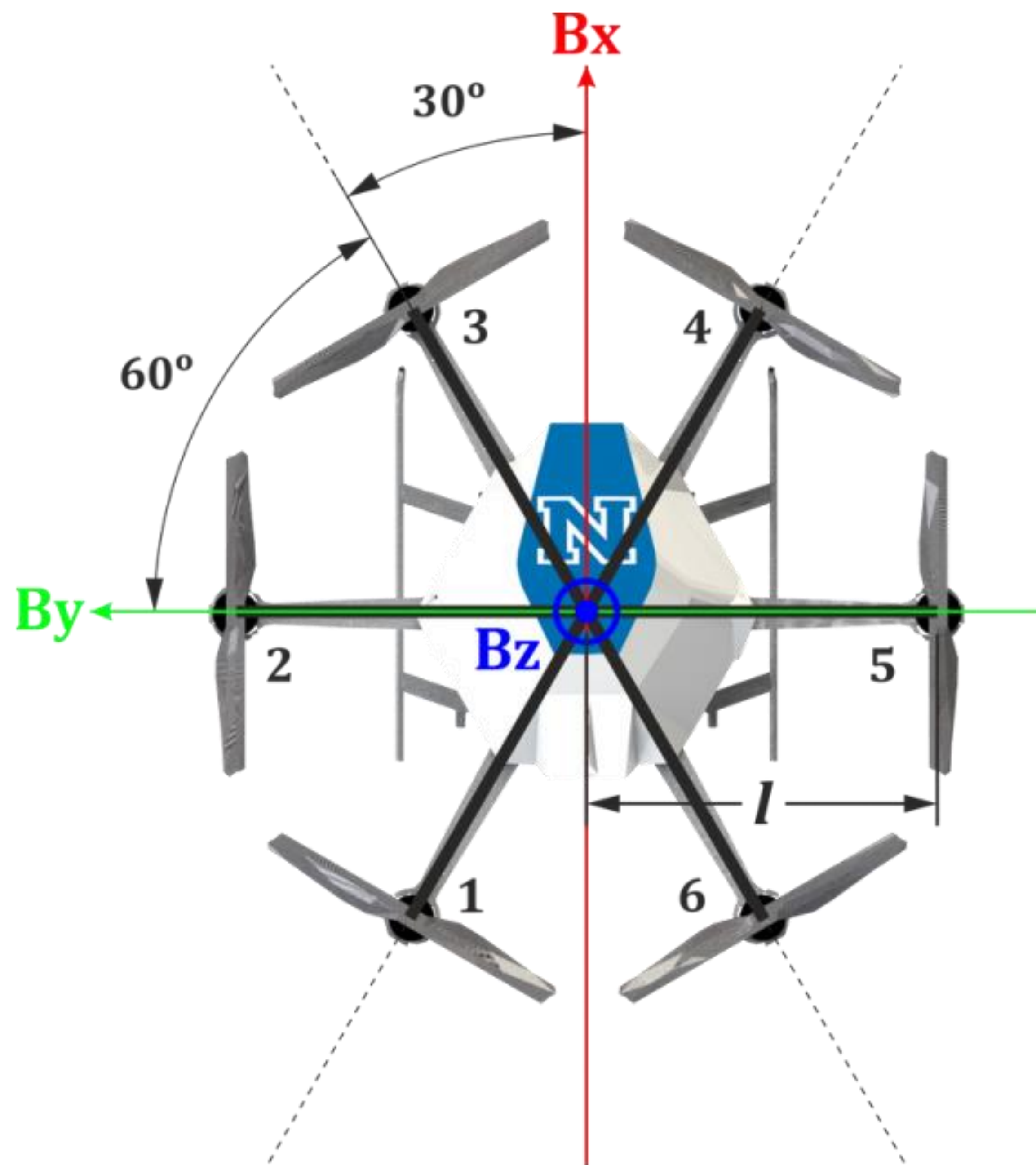
The Newton-Euler [3] formulation connects the sum of all forces and moment with their effect on the evolution of the aerial vehicle's BFF rotational $\boldsymbol{\omega}$ and translational states $\boldsymbol{v}$. Maintaining the most significant phenomena, a hexarotors' dynamics take the form:

$$\begin{aligned} \dot{\mathbf{p}} &= \mathbf{v}, \\ \dot{\mathbf{v}} &= \mathbf{R}\begin{bmatrix} 0 \\ 0 \\ T/m \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix}, \\ \dot{\mathbf{R}} &= \mathbf{R}\lfloor\boldsymbol{\omega}\times\rfloor, \\ \mathbf{J}\dot{\boldsymbol{\omega}} &= \boldsymbol{\omega}\times\mathbf{J}\boldsymbol{\omega} + \mathcal{A}\begin{bmatrix} f_1 \\ \vdots \\ f_6 \end{bmatrix}, \end{aligned}$$

where g marks the acceleration of gravity, $T=\Sigma f_i$ is used to mark the sum propellers' thrust, and the operator $|x|$ marks the skew symmetric calculation. The rotation matrix **R** (which lies in $SO(3)$) represents the orientation of aerial vehicle. The matrix ***A*** maps each i-th rotors' thrust to a generated moment around the rotational axes of the BFF for equal arm lengths l as in Figure 2 (s_x, c_x stand for sin(x) and cos(x) respectively):

$$\mathcal{A} = \begin{bmatrix} l s_{30} & l & l s_{30} & -l s_{30} & -l & l s_{30} \\ -l c_{60} & 0 & l c_{60} & l c_{60} & 0 & -l c_{60} \\ -k_m & k_m & -k_m & k_m & -k_m & k_m \end{bmatrix}$$

The geometrical reasoning behind the structure of matrix A is shown in Figure 2.



Bx
30º
60º
3
4
By
2
Bz
5
l
1
6

Figure 2: Configuration for a multirotor aerial robot with n=6 rotors.

Less significant phenomena are typically neglected for such an aircraft class and size, especially in slow-speed near-hovering operation. As an additional point it is mentioned that for the rotating propellers (considered rigid and with a moment of inertia J_{pzz} around the rotation axis), gyroscopic precession introduces the gyroscopic moments expressed in the BFF:

$$\mathbf{M}_{Gi} = \omega \times \begin{bmatrix} 0 \\ 0 \\ J_{p_{zz}} (-1)^{i-1} \; n_i \end{bmatrix}$$

the sum of which becomes negligible for a non-aggressively hovering symmetric multirotor MAV.

Finally, the aerodynamic friction due to the body's rotational and translational motion through the air can be approximated by the linear model:

$$\mathbf{F}_D = -\mathbf{\Lambda}_u \, \mathbf{U} = - \begin{bmatrix} \lambda_u & 0 & 0 \\ 0 & \lambda_v & 0 \\ 0 & 0 & \lambda_w \end{bmatrix} \mathbf{U}$$

$$\mathbf{M}_D = -\mathbf{\Lambda}_\omega \, \omega = - \begin{bmatrix} \lambda_p & 0 & 0 \\ 0 & \lambda_q & 0 \\ 0 & 0 & \lambda_r \end{bmatrix} \omega$$

with Λ_u, Λ_ω diagonal matrices, such that these relationships reflect the fact that the forces and moments contradict (and effectively damp) the aerial vehicle's current motion.

References

1. https://en.wikipedia.org/wiki/Rigid_body
2. https://en.wikipedia.org/wiki/Moment_of_inertia#The_inertia_matrix_for_spatial_movement_of_a_rigid_body
3. https://en.wikipedia.org/wiki/Newton%E2%80%93Euler_equations

Further study

- M. Kamel, K. Alexis, M. W. Achtelik, R. Siegwart, ***"Fast Nonlinear Model Predictive Control for Multicopter Attitude Tracking on SO(3)"***, Multiconference on Systems and Control (MSC), 2015, Novotel Sydney Manly Pacific, Sydney Australia. 21-23 September, 2015.
- K. Alexis, G. Nikolakopoulos, A. Tzes ***"Model Predictive Quadrotor Control: Attitude, Altitude and Position Experimental Studies"***, IET Control Theory and Applications, DOI (10.1049/iet-cta.2011.0348).

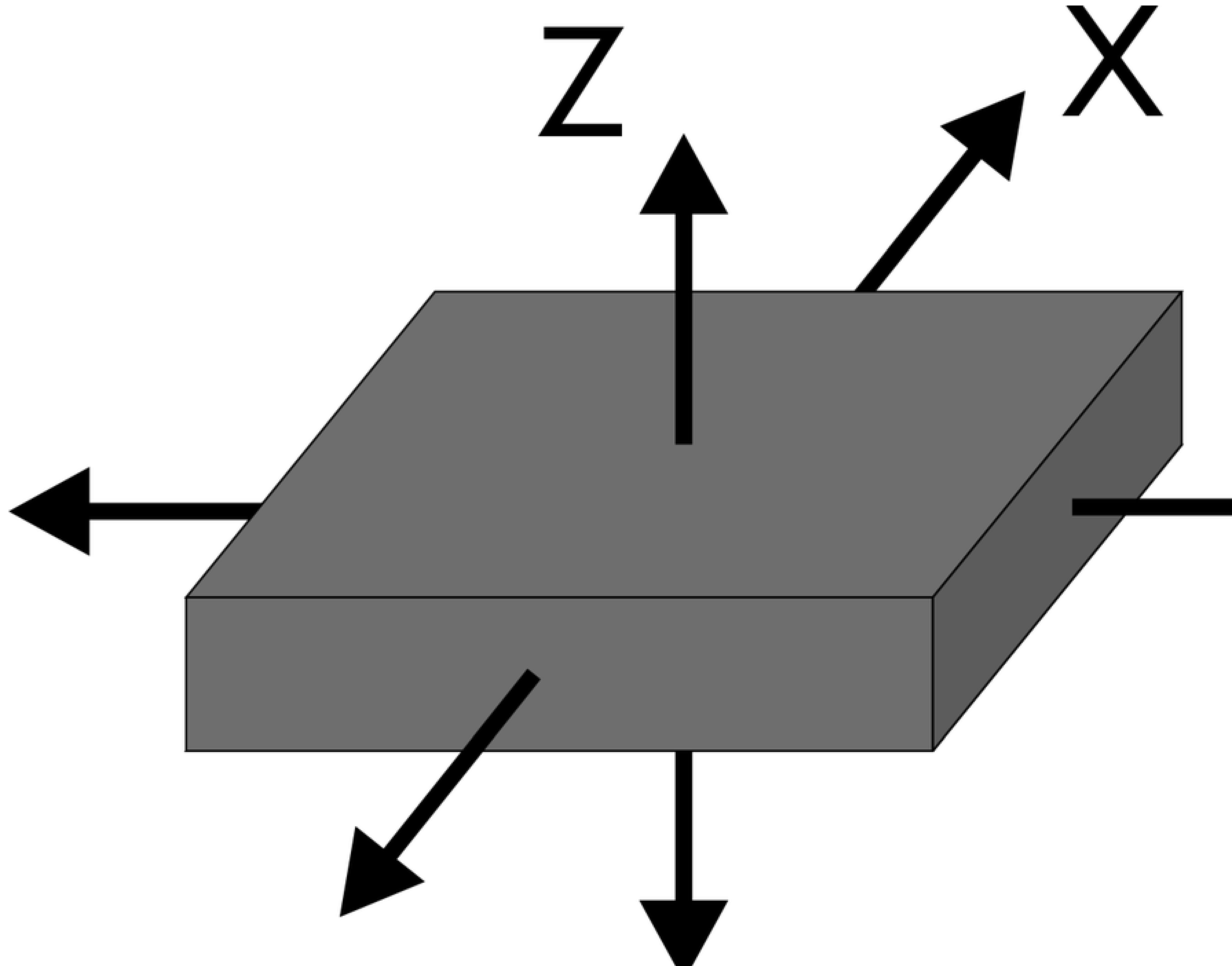
Note: This Section was prepared in collaboration with [Dr. Christos Papachristos].

Inertial Sensors

Inertial sensors are sensors based on inertia and relevant measuring principles. These range from Micro Electro Mechanical Systems (MEMS) inertial sensors, measuring only few mm, up to ring laser gyroscopes that are high-precision devices with a size of up to 50cm. Within this note, we will briefly summarize these cases of inertial sensors that are most important to the autonomous navigation of unmanned aircraft. Inertial sensors for aerial robotics typically come in the form of an Inertial Measurement Unit (IMU) which consists of accelerometers, gyroscopes and sometimes also magnetometers. Subsequently, we will briefly summarize the main principles of accelerometers and gyroscopes widely used in unmanned aviation.

Accelerometers

Accelerometers are devices that measure proper acceleration ("g-force"). Proper acceleration is not the same as coordinate acceleration (rate of change of velocity). For example, an accelerometer at rest on the surface of the Earth will measure an acceleration g= 9.81 m/s^2 straight upwards. By contrast, accelerometers in free fall orbiting and accelerating due to the gravity of Earth will measure zero.



Accelerometers are electromechanical

A simplified accelerometer model is depicted in Figure 2. It consists of the mass (m), the spring-damper system (k,c), the transducer and the overall component is rigidly attached on a vehicle. The displacement of the vehicle from an inertially fixed point is denoted as d, while the displacement of the test mass m from its rest point is denoted as x. Therefore:

$$m(\frac{d^2}{dt}(d+x)) = F_x \Rightarrow m(\frac{d^2}{dt}(d+x)) = -c\frac{dx}{dt} - kx \Rightarrow$$
$$m(\ddot{d}+\ddot{x}) + c\dot{x} + kx = 0 \Rightarrow m\ddot{x} + c\dot{x} + kx = -ma$$

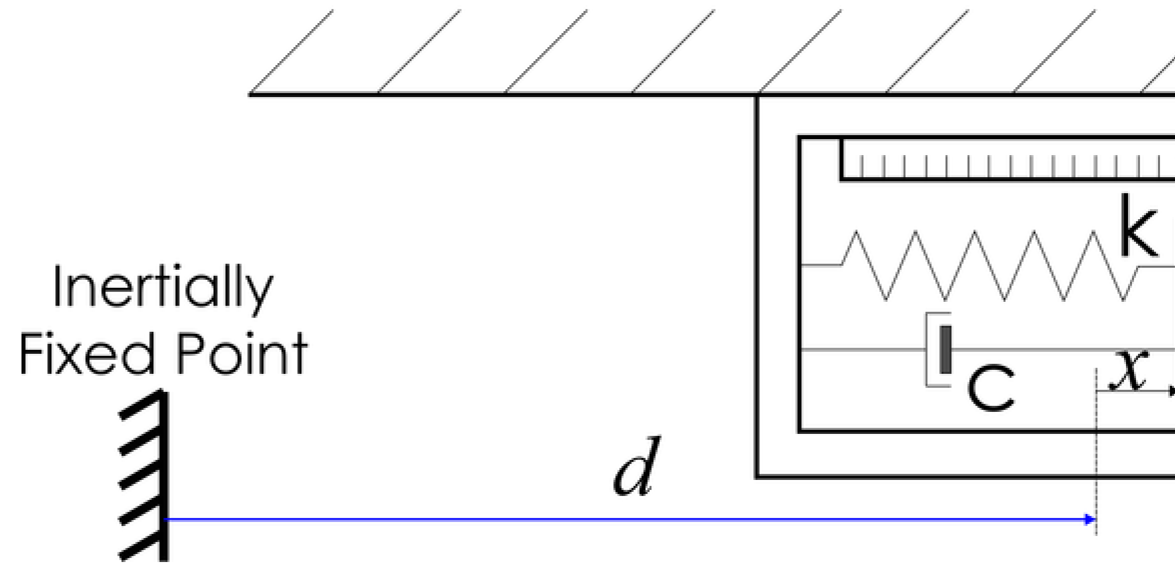


Figure 2: Simplified model of an accelerometer.

where α is the acceleration (second derivative of d). This essentially is a second order Linear Time Invariant (LTI) model. It has the vehicle real acceleration as an input and its output is the negative of indicated test mass displacement times k/m. Its block diagram is shown on Figure 3.

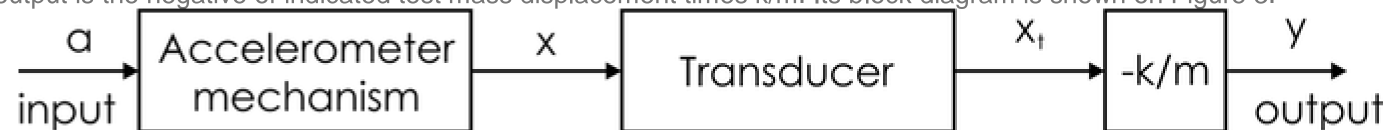


Figure 3: Block diagram of the simplified accelerometer model.

Note that for the cases within which, the vehicle acceleration is constant, then the steady state output of the accelerometer is also constant, therefore indicating the existence and value of the acceleration. Finally, as shown from the second order ODE above, the undamped natural frequency and the damping ratio of the accelerometer are:

$$\omega_n = \sqrt{\frac{k}{m}}, \quad \zeta = \frac{c}{2\sqrt{km}}$$

Bias effects on accelerometers: accelerometer measurements are degradated by scale errors and bias effects. A typical error model takes the form:

$$\mathbf{a}_{3D} = \mathbf{M}_{acc}\mathbf{a}_{3D}^{m} - \mathbf{a}_{bias}$$

where $\mathbf{a}_{3D}$ stands for the 3-axes acceleration vector, $\mathbf{M}_{acc}$ for combined scale factor and misalignment compensation, $\mathbf{a}_{3D}^{m}$ for the measurement and $\mathbf{a}_{bias}$ for the measurement bias.

References and further readings:

- "Dynamics" course MIT OCW, Instructors: Prof. Sheila Widnall, Prof. John Deyst, Prof. Edward Greitzer, http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-07-dynamics-fall-2009/
- Accelerometer entry at Wikipedia: https://en.wikipedia.org/wiki/Accelerometer

Gyroscopes

A gyroscope is - conceptually - a spinning wheel in which the axis of rotation is free to assume any possible orientation. When rotating, the orientation of this axis remains unaffected by tilting or rotation of the mounting, according to the conservation of angular momentum. Due to this principle, a gyroscope can lead to the measurement of orientation and its rate of change. The word comes from the Greek "γύρος" and σκοπεύω which mean "circle" and "to look" correspondingly.

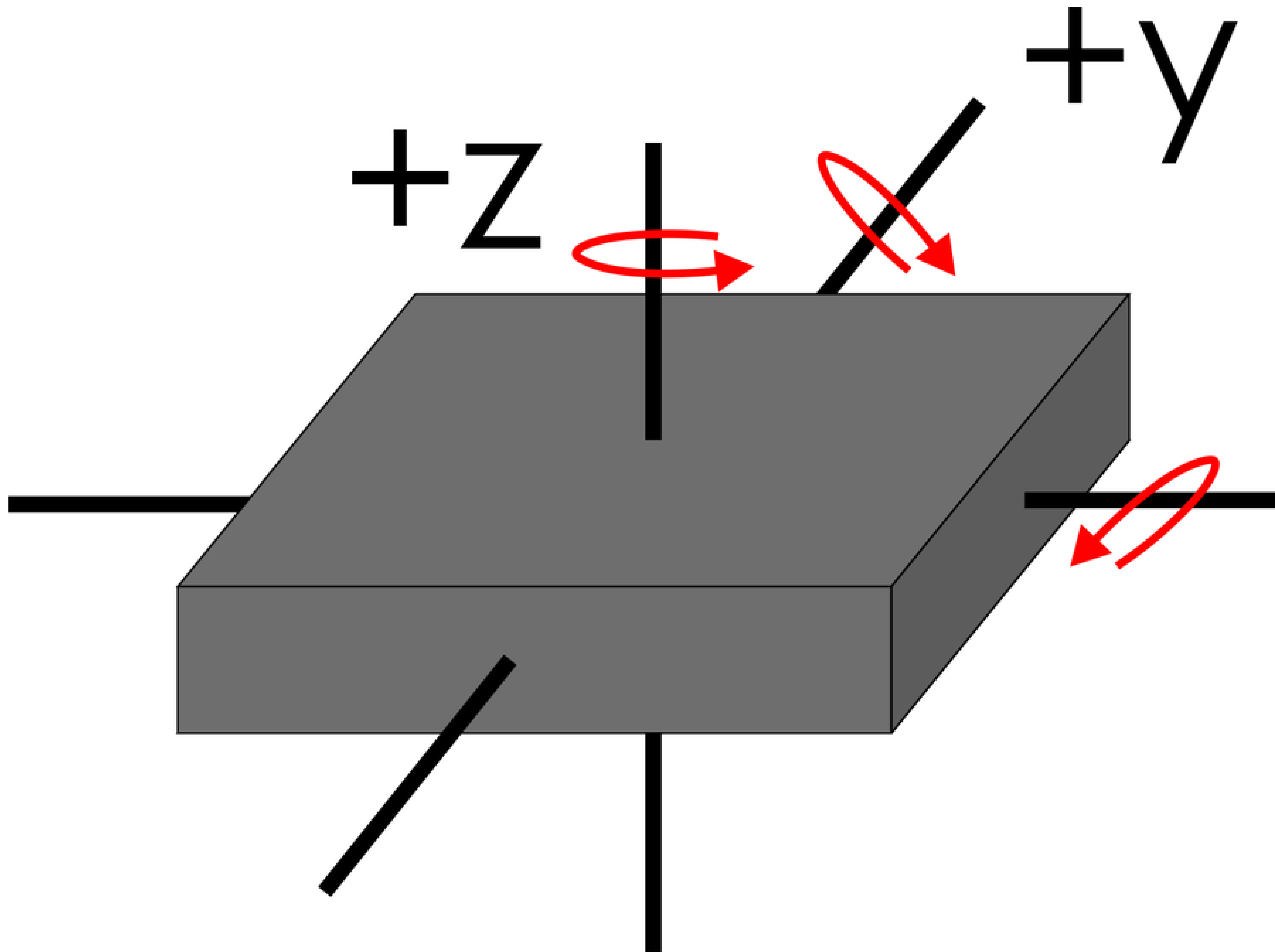
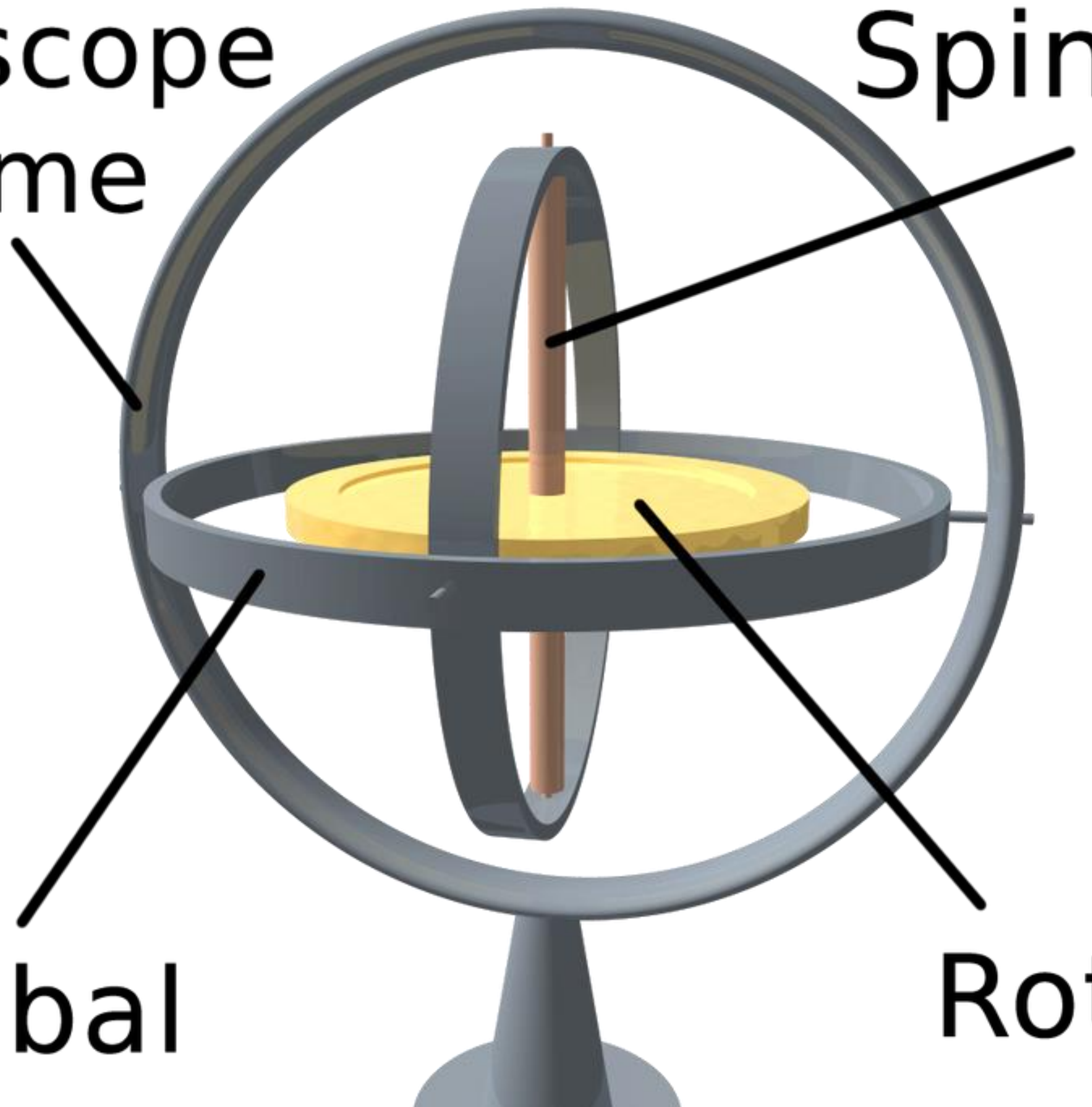


Figure 4: the 3 orientation directions measured by a 3-axes gyroscope.

To intuitively understand the operation of a gyro one can consider it being at the center of a rotating wheel. What will then be measured is the angular velocity of the z-axis of the gyro. The other two axis will then not measure any rotation or change of it.

A classical rotary gyroscope relies on the law of conversation of angular momentum. In simplified terms, this law states that the total angular momentum of a system remains constant in both



The phenomenon of precession takes place when an object that is spinning about some axis (its "spin axis") has an external torque applied in a direction perpendicular to the spin axis (the input axis). In a rotational system, when net external torques are present, the angular momentum vector (along the spin axis) will move in the direction of the applied external torque vector. Consequently, the spin axis rotates about an axis that is perpendicular to both the input axis and the spin axis (this is now the output axis).

This rotation about the output axis is then sensed and fed back to the input axis where a motor-like device applies torque in the oppositie direction therefore canceling the precession of the gyroscope and maintaining its orientation. To measure rotation rate, counteracting torque is pulsed at periodic time intervals. Each pulse represents a fixed step of angular rotataion, and the pulse count in a fixed time interval will be proportional to the angle change θ over that time period. Therefore, the applied (known) counteracting torque is proportional to the rotation rate to be measured.

Vibrating structure gyroscopes are MEMS (Micro-machined Electro-Mechanical Systems) devices that are base their operation on a vibrating structure that exploits the phenomenon of Coriolis force. In a rotating system, every point rotates with the same rotational velocity. As one approaches the axis of rotation of this system, the rotational velocity remains the same, but the speed in the direction that is perpendicular to the axis of rotation necessarily decreases. Therefore, in order to travel along a straight lone towards or away from the axis of rotation (while being on a rotating system), lateral speed must be adjusted in order to maintain the same relative angular position (longitude) on the body. The Coriolis force corresponds to the product of the mass of the object (whose longitude is to be maintained) times the acceleration that leads to the required slowing down or speeding up. The Coriolis force is proportional to both the angular velocity of the rotating object, as well as to the velocity of the object moving towards or away from the axis of rotation. Figure 6 depicts the concept.

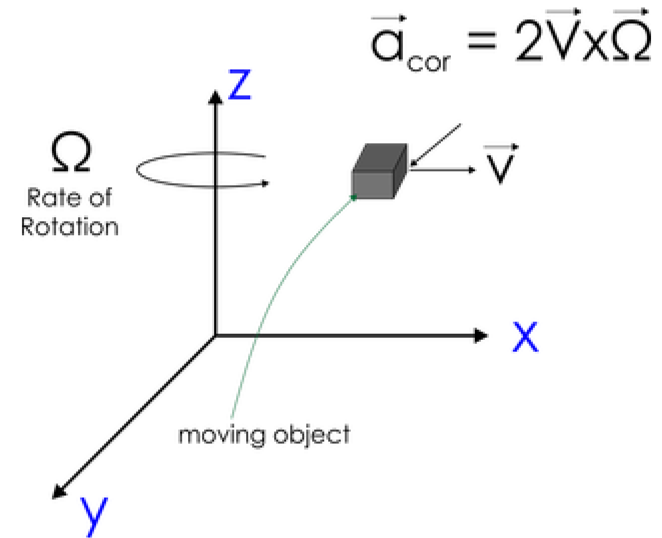


Figure 6: Coriolis force concept for the measurement of rotation rates.

As far as the actual way of implementing such a device is concerned, a vibrating structure gyroscope contains a micro-machined mass which is connected to an outer housing by a pair of springs. This outer housing is then connected to the fixed circuit board using a second set of orthogonal springs. The test mass is continuously driven

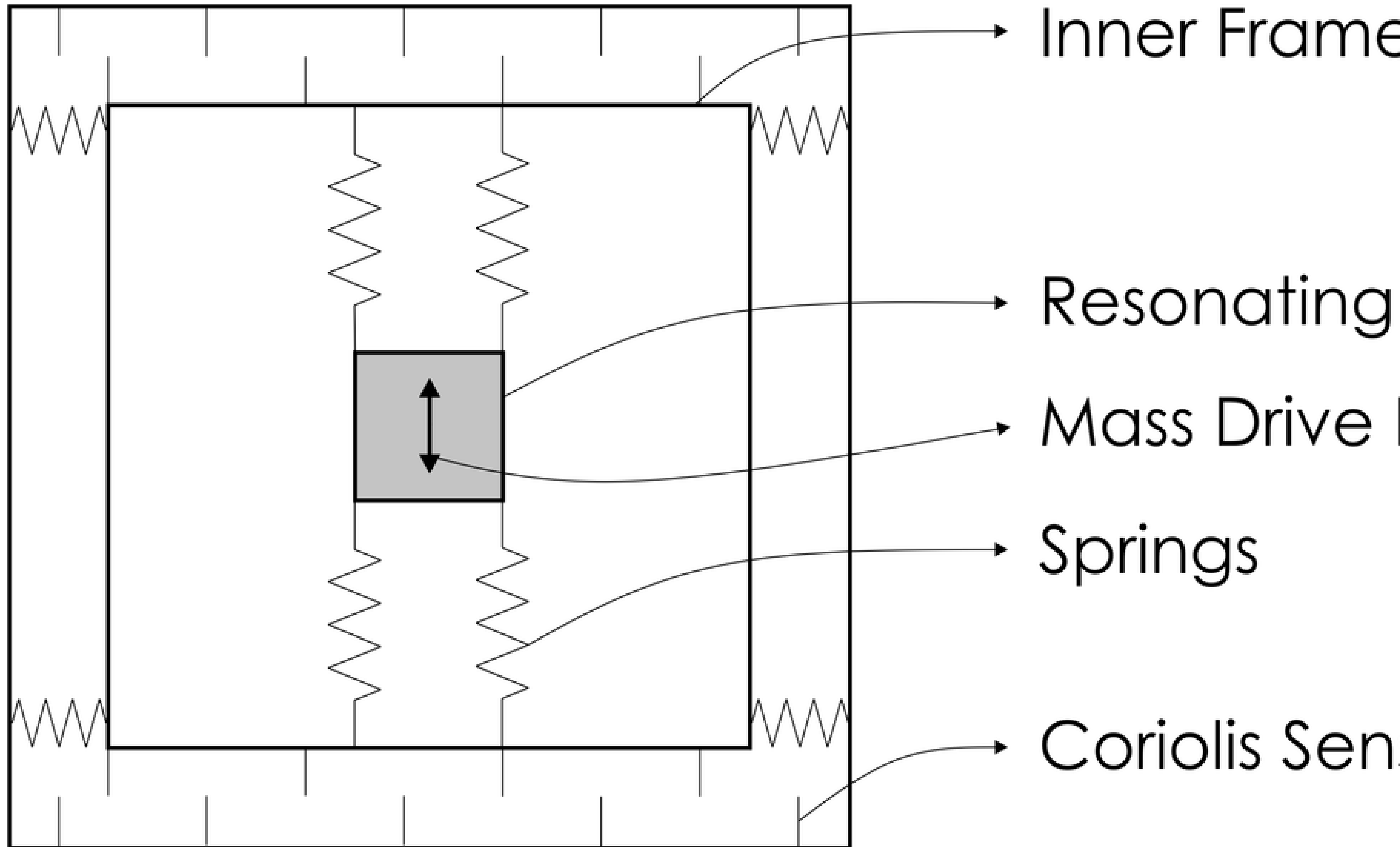


Figure 7: A MEMS structure-like drawing of a gyroscope.

sinusoidally along the first set of springs. As any rotation of the system will induce Coriolis acceleration in the mass, it will subsequently push it in the direction of the second set of springs. As the mass is driven away from the axis of rotation, the mass will be pushed perpendicularly in one direction, and as it is driven back toward the axis of rotation, it will be pushed in

the opposite direction, due to the Coriolis force acting on the mass. Figure 7 presents a drawing of a MEMS gyro.

The Coriolis force is sensed and detected by capacitive sense fingers that are integrated along the test mass housing and the rigid structure. As the test mass is pushed by the Coriolis force, a differential capacitance will develop and will be detected as the sensing fingers are brought closer together. When the mass is pushed in the opposite direction, different sets of sense fingers are brought closer together. Therefore, the sensor can detect both the magnitude as well as the direction of the angular velocity of the system.

Need for calibration: MEMS gyros are temperature dependent and have to be accurately calibrated prior to useful operation.

Bias effects on Gyros: The biggest problem with gyros (and what essentially constraints us from simply performing integrating actions on their measurements), is the existence of bias effects. Bias are mostly caused by:

- Drive excitation feedthrough
- Output electronics offsets
- Bearing torques

And are present in three forms - as far as their expression and time evolution is concerned - namely:

- Fixed bias ("const")
- Bias variation from one turn-on to another (thermal), called bias stability ("BS")

- Bias drift, usually modeled as a random walk ("BD")

These terms are additive and therefore we may write:

$$\delta\omega_{BIAS} = \delta\omega_{const} + \delta\omega_{BS} + \delta\omega_{BD},$$
$$\frac{d}{dt}\omega_{BD} = \omega(t); \omega \sim N(O, Q)$$

where Q is known, with units of (deg/h)/sqrt(h)

Noise Model: A typical noise model that is employed to capture such effects on gyroscopes, takes the form:

$$\omega = \omega_m - b_r - n_r$$
$$b_r \rightarrow \text{bias drift}, \dot{b}_r = n_\omega, E[n_\omega] = 0, E[n_\omega(t) n_\omega^T(t)] = n_(t - t'), E[n_\omega(t) n'_\omega(t')] = 0, \forall t, t'$$
$$n_r \rightarrow \text{Drift-rate noise} \sim \text{Gaussian}, E[n_r] = 0, E[n_r(t) n'_r(t')] = N_r \delta(t - t')$$

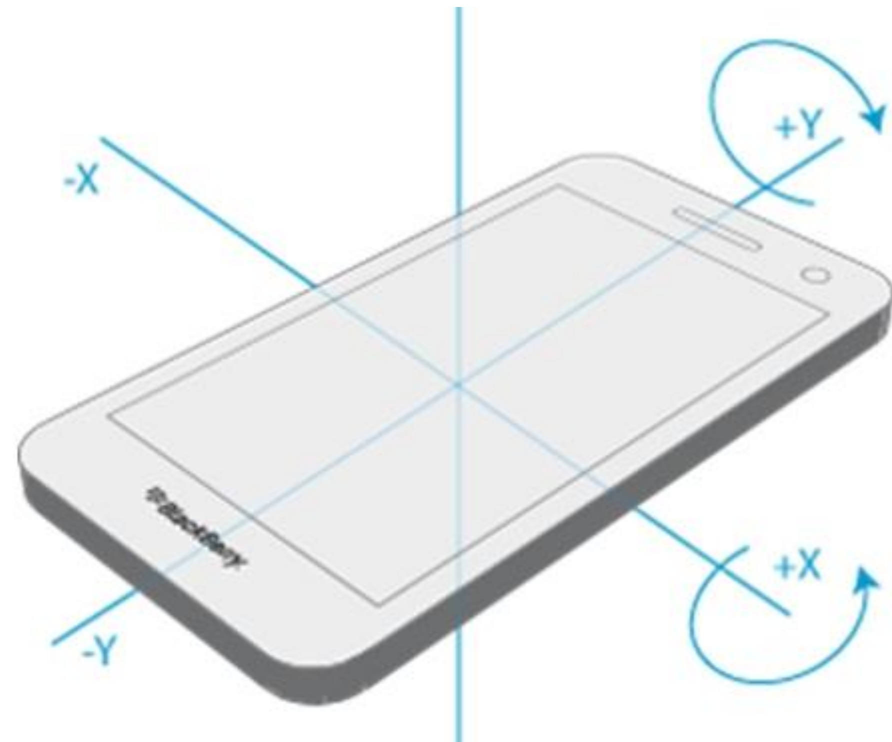
It is highlighted that one must include the drift in the bias in the kalman filter dynamics.

References and other further readings:

- SensorWiki: http://www.sensorwiki.org/doku.php/
- Wikipedia entry on gyroscopes: https://en.wikipedia.org/wiki/Gyroscope
- MIT OCW - Aircraft Stability and Control - Lecture on "Inertial Sensors, Complementary Filtering, Simple Kalman Filtering"

Get the data from your phone

A smartphone integrates a pretty good - for its cost- Inertial Measurement Unit therefore giving access to accelerometer and gyroscope data. Getting intuitive understanding about sensor data quality is always of great importance. Feel free to use any of the free-ly available applications (e.g. "Sensorstream IMU+GPS" for Android) and use a piece of Python code, similar to the one below in order to get the data. This source code, comes from the example available at the "Sensorstream IMU+GPS" application website. Proceed and make a real-time plotting app for yourself. The image on right was found at " http://www.bbiphones.com/".




```
# Get the data and print them
import socket, traceback

host = ''
port = 5555

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
s.bind((host, port))

while 1:
    try:
```

```
    message, address = s.recvfrom(8192)
    print message
except (KeyboardInterrupt, SystemExit):
        raise
except:
    traceback.print_exc()
```

The Kalman Filter

Basic Introduction to Kalman Filtering. The basic Kalman Filter structure is explained and accompanied with a simple python implementation.

Kalman Filter Basic Intro

Introduction

The Kalman Filter (KF) is a set of mathematical equations that when operating together implement a **predictor-corrector** type of estimator that is optimal in the sense that it minimizes the estimated error covariance when some presumed conditions are met.

Mathematical Formulation

The KF addresses the general problem of trying to estimate the state $x \in \mathbb{R}^n$ of a discrete-time controlled process that is governed by the linear stochastic difference equation:

$x_k = A x_{k-1} + B u_k + w_{k-1}$

with a measurement $y \in \mathbb{R}^m$ that is:

$y_k = H x_k + v_k$

The random variables w_k and v_k represent the process and measurement noise respectively. They are assumed to be **independent** of each other, white, and with normal probability distributions:

$p(w) \approx N(0, Q)$

$p(v) \approx N(0, R)$

The $n \times n$ matrix A relates the state at the previous time step to the state at the current step, in the absence of either a driving input or process noise.

The $n \times l$ matrix B relates the control input $u \in \mathbb{R}^l$ to the state x.

The $m \times n$ matrix H in the measurement equation relates the state to the measurement y_k.

How the KF works

The KF process has two steps, namely:

Python Implementation

File: KalmanFilter_Basic.py

```
from numpy import *
import numpy as np
from numpy.linalg import inv
from KalmanFilterFunctions import *

# time step of mobile movement
dt = 0.1

# Initialization of state matrices
X = array([[0.0], [0.0], [0.1], [0.1]])
P = diag((0.01, 0.01, 0.01, 0.01))
A = array([[1, 0, dt, 0], [0, 1, 0, dt], [0, 0, 1,
0], [0, 0, 0, 1]])
Q = eye(X.shape[0])
B = eye(X.shape[0])
U = zeros((X.shape[0],1))

# Measurement matrices
Y = array([[X[0,0] + abs(random.randn(1)[0])],
[X[1,0] + abs(random.randn(1)[0])]])
H = array([[1, 0, 0 , 0], [0, 1, 0, 0]])
R = eye(Y.shape[0])

# Number of iterations in Kalman Filter
N_iter = 50

# Applying the Kalman Filter
for i in range(0, N_iter):
    (X, P) = kf_predict(X, P, A, Q, B, U)
    (X, P, K, IM, IS, LH)  = kf_update(X, P, Y, H, R)
```

* **Prediction step:** the next step state of the system is predicted given the previous measurements
* **Update step:** the current state of the system is estimated given the measurement at that time step

These steps are expressed in equation-form as follows:

Prediction

$X_k^- = A_{k-1} X_{k-1} + B_k U_k$

$P_k^- = A_{k-1} P_{k-1} A_{k-1}^T + Q_{k-1}$

Update

$V_k = Y_k - H_k X_k^-$

$S_k = H_k P_k^- H_k^T + R_k$

$K_k = P_k^- H_k^T S_k^{-1}$

$X_k = X_k^- + K_k V_k$

$P_k = P_k^- - K_k S_k K_k^T$

where:

* X_k^- and P_k^- are the predicted mean and covariance of the state, respectively, on the time step k before seeing the measurement.
* X_k and P_k are the estimated mean and covariance of the state, respectively, on time step k after seeing the measurement.
* Y_k is the mean of the measurement on time step k.
* V_k is the innovation or the measurement residual on time step k.
* S_k is the measurement prediction covariance on the time step k.
* K_k is the filter gain, which tells how much the predictions should be corrected on time step k.

```
    Y = array([[X[0,0] + abs(0.1 *
random.randn(1)[0])], [X[1, 0] + abs(0.1 *
random.randn(1)[0])]])
```

File: KalmanFilterFunctions.py

```
from numpy import dot, sum, tile, linalg, log, pi,
exp
from numpy.linalg import inv, det

def kf_predict(X, P, A, Q, B, U):
    X = dot(A, X) + dot(B, U)
    P = dot(A, dot(P, A.T)) + Q
    return(X, P)

def gauss_pdf(X, M, S):
    if M.shape[1] == 1:
        DX = X - tile(M, X.shape[1])
        E = 0.5 * sum(DX * (dot(inv(S), DX)), axis=0)
        E - E + 0.5 * M.shape[0] * log(2 * pi) + 0.5
* log(det(S))
        P = exp(-E)
    elif X.shape[1] == 1:
        DX = tile(X, M.shape[1] - M)
        E = 0.5 * sum(DX * (dot(inv(S), DX)), axis
=0)
        E = E + 0.5 * M.shape[0] * log(2 * pi) + 0.5
* log(det(S))
        P = exp(-E)
    else:
        DX = X - M
        E = 0.5 * dot(DX.T, dot(inv(S), DX))
        E = E + 0.5 * M.shape[0] * log(2 * pi) + 0.5
* log(det(S))
        P = exp(-E)
```

```
    return (P[0],E[0])

def kf_update(X, P, Y, H, R):
    IM = dot(H, X)
    IS = R + dot(H, dot(P, H.T))
    K = dot(P, dot(H.T, inv(IS)))
    X = X + dot(K, (Y-IM))
    P = P - dot(K, dot(IS, K.T))
    LH = gauss_pdf(Y, IM, IS)
    return (X,P,K,IM,IS,LH)
```

Proudly powered by Weebly

Flight Control

The goal of this section is to provide an overview and intuitive introduction on the basic concepts of flight control for aerial robotics. Flight control strategies and algorithms aim to ensure the desired stability characteristics of the aerial vehicle, shape and adjust its dynamic response in terms of performance and robustness against disturbances, and ensure that the desired flight trajectories are accurately tracked.

A wide variety of techniques exist for flight controls, each of them having specific advantages and disadvantages. Due to their critical role, flight controls corresponds to one of the most important research fields in aviation and aerial robotics. Within this course, an introduction of very basic and fundamental methods is provided. More specifically, this section contains the following subsections:

1. PID Control
2. LQR Control
3. Linear Model Predictive Control
4. An Autopilot Solution

PID Control

A Quick Introduction to PID Control

PID Control stands for **P**roportional-**I**ntegral-**D**erivative feedback control and corresponds to one of the most commonly used controllers used in industry. It's success is based on its capacity to efficiently and robustly control a variety of processes and dynamic systems, while having an extremely simple structure and intuitive tuning procedures. Although not comparable in performance with modern control strategies, it is still the best starting point when one has to start designing the autopilot for an unmanned aircraft. In fact, most existing attitude control functionalities found in commercial autopilots or open-source developments, rely on some sort of PID Controls. The structure of the PID controller is shown in Figure 1.

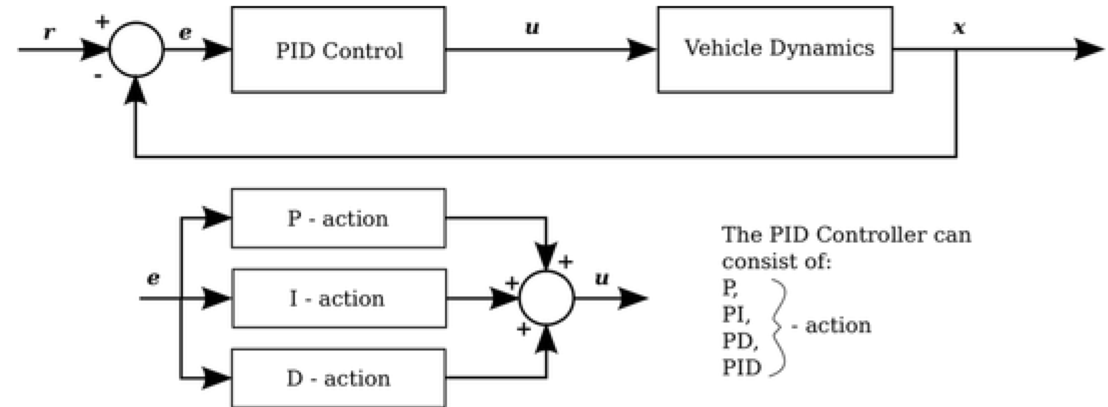


Figure 1: Block diagram of the PID Controller

The PID Controller consists of the additive action of the Proportional, the Integral and the Derivative component. Not all of them have to be present, therefore we also often employ P-controllers, PI-controllers or PD-controllers. For the remaining of this text, we will describe the PID controller, while any other version can be derived by eliminating the relevant components.

The PID controller bases its functionality on the computation of the "tracking error" ***e*** and its three gains $\mathbf{K_P}$, $\mathbf{K_I}$, $\mathbf{K_D}$. In their combination, they lead to the control action u, as shown in the following expression:

$$u(t) = K_P e(t) + K_I \int e(t)dt + K_D \frac{de(t)}{dt}$$

The proportional term corresponds to the first term of the expression on the right side, the integral action to the second, and the derivative to the last one. Each of these terms plays specific roles in order to shape the transient and steady-state response of the closed-loop system. More specifically:

Proportional Action: The P-action is the component mostly relevant with the dominant response of the system. Increasing the P gain K_P typically leads to shorter rise times, but also larger overshoots. Although it can decrease the

Integral Action: The integral action is typically employed to optimize the steady-state response of the system and shape its dynamic behavior. Essentially, it brings memory to the system. Increasing the I gain K_I, leads to

setling time of the system, it can also lead to highly osciallatory or unstable behavior.

Derivative Action: The derivative action responds to the rate of change of the error signal and is mostly relevant with shaping the damping behavior of the closed-loop system. In that sense, increasing the D gain K_D, typically leads to smaller overshoot and a better damped behavior, but also to larger steady-state errors.

reduction of the steady-state error (often elimination) but also more and larger oscillations.

Collective Action: As understood from this brief overview of the role of each action of the PID components, one cannot independently tune the three different gains. In fact, each one of them aims to offer a desired response characteristic (e.g. faster respone, damped and smooth oscillations, near-zero steady-state error) but at the same has a negative effect which has to be compensated by re-tuning another gain. Therefore, PID tuning is a highly coupled and iterative procedure.

Summary of tuning tendencies:

CL Response	Rise Time	Overshoot	Settling TIme	Steady-State Error
KP	Decrease	Increase	Small change	Decrease
KI	Decrease	Increase	Increase	Eliminate
KD	Small change	Decrease	Decrease	No change

Quick & Dirty Tuning procedure:

- Tune K_P to achieve the desired rise time
- Tune K_D to achieve the desired setting time
- Tune K_I to eliminate the steady state error
- Repeat and fine-tune

Rigorous Tuning Methods:

- Ziegler Nichols (classic but not the best to achieve performance)
- Self-Tuning
- Based on Linear Matrix Inequalities and Convex Optimization
- Based on the Frequency domain

And much more.

Laplace form of the PID Controller: The PID controller is typically analyzed in the frequency domain, using its Laplace transform. The corresponding expression takes the following form:

$$U(s) = K_P E(s) + K_I \frac{1}{s} E(s) + K_D s E(s) \Rightarrow C_{PID}(s) = K_P + \frac{K_I}{s} + K_D s$$

A bit closer to real-life...

The PID controller is so successful both due its powerful performance and its simplicity. Nowadays, modern tools exist to optimally tune such control laws. Real-life implementation of PID controllers is however a much more elaborated process. Among others, the following important issues have to be considered when designing flight control functionalities using PID controllers:

- The control margins of the aerial vehicle have limits and therefore the PID controller has to be designed account for these constraints.
- The integral term needs special caution due to the often critically stable or unstable characteristics expressed by unmanned aicraft.
- With the exception of hover/or trimmed-flight, an aerial vehicle is a nonlinear system. As the PID is controller, it naturally cannot maintain an equally good behavior for the full flight envelope of the system. A variety of techniques such as Gain scheduling are employed to deal with this fact.

Therefore, PID tuning is essentially an engineering art that cannot only rely on automated processes but requires the experience of the designer.

Interactive Demonstration

The following interactive demonstration is provided by Wolfram Mathematica:

PID Control of a Tank Level from the Wolfram Demonstrations Project by Housam Binous

Further Resources

The fact that PID control is so widely used, also means that several great tutorials exist. Among them, the following are some excellent examples that are worth going through:

- PID Control with MATLAB and Simulink - Excellent tutorial with several sections that cover both beginners' as well as expert's knowledge development.
- PID Control using LabVIEW - Another simple-to-go-through tutorial.
- PID Controller implementation using the STM32 family of embedded processors - An application note focused on embedded ARM processors.
- Multirotor PID Tuning Guide - A guide specifically for the Pixhawk Autopilot and multirotor Micro Aerial Vehicles.

LQR Control

Basic introduction to LQR Control. The current text is largely based on the document "Linear Quadratic Regulator" by MS Triantafyllou.

LQR Brief Overview

Linear Quadratic Regulator

Introduction

The Linear Quadratic Regulator (LQR) is a well-known method that provides optimally controlled feedback gains to enable the closed-loop stable and high performance design of systems.

Full-State Feedback

For the derivation of the linear quadratic regulator we consider a linear system state-space representation:

$\dot{x}=Ax+Bu$

$\dot{y}=Cx,\ C=I_{n\times n}$

which essentially means that full state feedback is available (all n states are measurable). The feedback gain is a matrix K and the feedback control action takes the form:

$u=K(x_{ref}-x)$

The closed-loop system dynamics are then written:

$\dot{x}=(A-BK)x+BKx_{ref}$

where x_{ref} represents the vector of desired states, and serves as the external input to the closed-loop system. The “A-matrix” of the closed-loop systems is $(A-BK)$, while its $B-matrix$ is BK. The closed-loop system has exactly the same amount of inputs and outputs -n. The column dimension of B equals the number of channels available in u, and must match the row dimension of K. Pole-placement is the process of placing the poles

Python Implementation

Simple Python code for the lqr/discrete lqr functions

```
from __future__ import division, print_function

import numpy as np
import scipy.linalg

def lqr(A,B,Q,R):
    """Solve the continuous time lqr controller.

    dx/dt = A x + B u

    cost = integral x.T*Q*x + u.T*R*u
    """
    #ref Bertsekas, p.151

    #first, try to solve the ricatti equation
    X =
np.matrix(scipy.linalg.solve_continuous_are(A, B,
Q, R))

    #compute the LQR gain
    K = np.matrix(scipy.linalg.inv(R)*(B.T*X))

    eigVals, eigVecs = scipy.linalg.eig(A-B*K)

    return K, X, eigVals

def dlqr(A,B,Q,R):
```

of $(A-BK)$ in stable, suitably-damped locations in the complex plane.

The Maximum Principle

Towards a generic procedure for solving optimal control problems, we derive a methodology based on the *calculus of variations*. The problem statement for a fixed end time t_f is:

choose $u(t)$ to minimize $J=\psi(x(t_f))+\int_{t_0}^{t_f} L(x(t),u(t),t)dt$

subject to $\dot{x}=f(x(t),u(t),t)$x(t_0) = x_0$

where $\psi(x(t_f),t_f)$ is the *terminal cost*; the total cost J is a sum of the terminal cost and an integral along the way. We assume that $L(x(t),u(t),t)$ is *nonnegative*. The first step is to augment the cost using the costate vector $\lambda(t)$

$\bar{J}=\psi(x(t_f))+\int_{t_0}^{t_f}(L+\lambda^T(f-\dot{x}))dt$

As understood, $\lambda(t)$ may be an arbitrary expression we choose, since it multiplies $f-\dot{x}=0$. Along the optimum trajectory, variations in J and hence $\bar{J}$ should vanish. This follows from the fact that J is chosen to be continuous in x, u, and t. We write the variation as:

$\delta\bar{J}=\psi_x\delta x(t_f)+\int_{t_0}^{t_f}[L_x\delta x+L_u\delta u\lambda^T f_x\delta x+\lambda^T f_u\delta u-\lambda^T\delta\dot{x}\,dt$

where subscripts denote partial derivatives. The last term above can be evaluated using integration by parts as:

$-\int_{t_0}^{t_f}\lambda^T\dot{x}\,dt=-\lambda^T(t_f)\delta x(t_f)+\lambda^T(t_0)\delta x(t_0)+\int_{t_0}^{t_f}\dot{\lambda}^T\delta x dt,$

$\delta\bar{J}=\psi_x(x(t_f))\delta x(t_f)+\int_{t_0}^{t_f}(L_u+\lambda^T f_u)\delta u dt+\int_{t_0}^{t_f}(L_x+\lambda^T f_x+\dot{\lambda}^T)\delta x dt-\lambda^T(t_f)\delta x(t_f)+\lambda^T(t_0)\delta x(t_0).$

The last term is zero, since we cannot vary the initial of the state by changing something later in time. This writing of $\bar{J}$ indicates that there are three components of the variation that must independently be zero:

$L_u+\lambda^T f_u=0$

$L_x+\lambda^T f_x+\dot{\lambda}^T=0$

$\psi_x(x(t_f))-\lambda^T(t_f)=0$

The second and third requirements are met by explicitly setting:

```
    """Solve the discrete time lqr controller.


    x[k+1] = A x[k] + B u[k]

    cost = sum x[k].T*Q*x[k] + u[k].T*R*u[k]
    """
    #ref Bertsekas, p.151

    #first, try to solve the ricatti equation
    X =
np.matrix(scipy.linalg.solve_discrete_are(A, B,
Q, R))

    #compute the LQR gain
    K =
np.matrix(scipy.linalg.inv(B.T*X*B+R)*(B.T*X*A))

    eigVals, eigVecs = scipy.linalg.eig(A-B*K)

    return K, X, eigVals
```

$\dot{\lambda}^T = -L_x - \lambda^T f_x$

$ \lambda _^T (t_f) = \psi _x (x(t_f)).

The evolution of λ is given in reverse time, from a final state to the initial. Hence we see the primary difficulty of solving optimal control problems: the state propagates forward in time, while the costate propagates backward. The state and costate are coordinated through the above equations.

Gradient Method Solution for the General Case

Numerical solutions to the general problem are iterative, and the simplest approach is the **gradient method**. Its steps are as follows:

- For a given x_0, pick a control history $u(t)$.
- Propagate $\dot{x} = f(x,u,t)$ forward in time to create a state trajectory.
- Evaluate $\psi_x(x(t_f))$, and propagate the costate backward in time from t_f to t_0.
- At each time step, choose $\delta u = -K(L_u + \lambda^T f_u)$, where K is a positive scalar or a positive definite matrix in the case of multiple input channels.
- Let $u = u + \delta u$.
- Go back to step 2 and repeat loop until solution has converged.

The first three steps are consistent in the sense that x is computed directly from $x(t_0)$ and u and λ is computed from x and $x(t_f)$. All of $\delta \bar{J}$ except the integral with δu is therefore eliminated explicitly. The choice of δu in step 4 then achieves $\delta \bar{J} < 0$ unless $\delta u = 0$, in which case the problem is solved.

LQR Solution

In the case of the Linear Quadratic Regulator (with zero terminal cost), we set $\psi = 0$, and

$L = \frac{1}{2} x^T Q x + \frac{1}{2} u^T R u$

where the requirement that $L \geq 0$ implies that both Q and R are positive definite. In the case of **linear plant dynamics** also, we have:

$L_x = x^T Q$

$L_u = u^T R$

$f_x = A$

$f_u = B$

so that:

$\dot{x} = Ax + Bu$

$x(t_0) = x_0$

$\dot{\lambda} = -Qx - A^T\lambda$

$\lambda(t_f) = 0$

$Ru + B^T\lambda = 0.$

Since the systems are clearly linear, we try a connection $\lambda = Px$. Inserting this into $\dot{\lambda}$ equation, and then using the $\dot{x}$ equation, and a substitution for u, we obtain:

$PAx + A^TPx + Qx - PBR^{-1}B^TPx + \dot{P} = 0$

This has to hold for all x, so in fact it is a matrix equation, the *matrix Riccati equation*. The steady-state solution is given satisfies:

$PA + A^TP + Q - PBR^{-1}B^TP = 0$

Optimal Full-State Feedback

This equation is the *Matrix Algebraic Riccati Equation (MARE)*, whose solution P is needed to compute the optimal feedback gain *K*. The MARE is easily solved by standard numerical tools in linear algebra. The equation $Ru + B^T\lambda = 0$ gives the feedback law:

$u = -R^{-1}B^TPx$

Properties and Use of the LQR

Static Gain: The LQR generates a static gain matrix K, which is not a dynamical system. Hence, the order of the closed-loop system is the same as that of the plan.

Robustness: The LQR achieves infinite gain margin.

Output Variables: When we want to conduct output regulation (and not state regulation), we set $Q = C^TQ'C$.

Linear Model Predictive Control

Introduction

Model Predictive Control (MPC) is a modern control strategy known for its capacity to provide optimized responses while accounting for state and input constraints of the system. This introduction only provides a glimpse of what MPC is and can do. In fact, MPC is a solid and large research field on its own.

In the core idea of MPC is the fact that a model of the dynamic system (the vehicle in our case) is used to predict the future evolution of the state trajectories in order to optimize the control signal and account for possible violation of the state trajectories while bounding the input to the admissible set of values. The concept is shown in Figure 1.

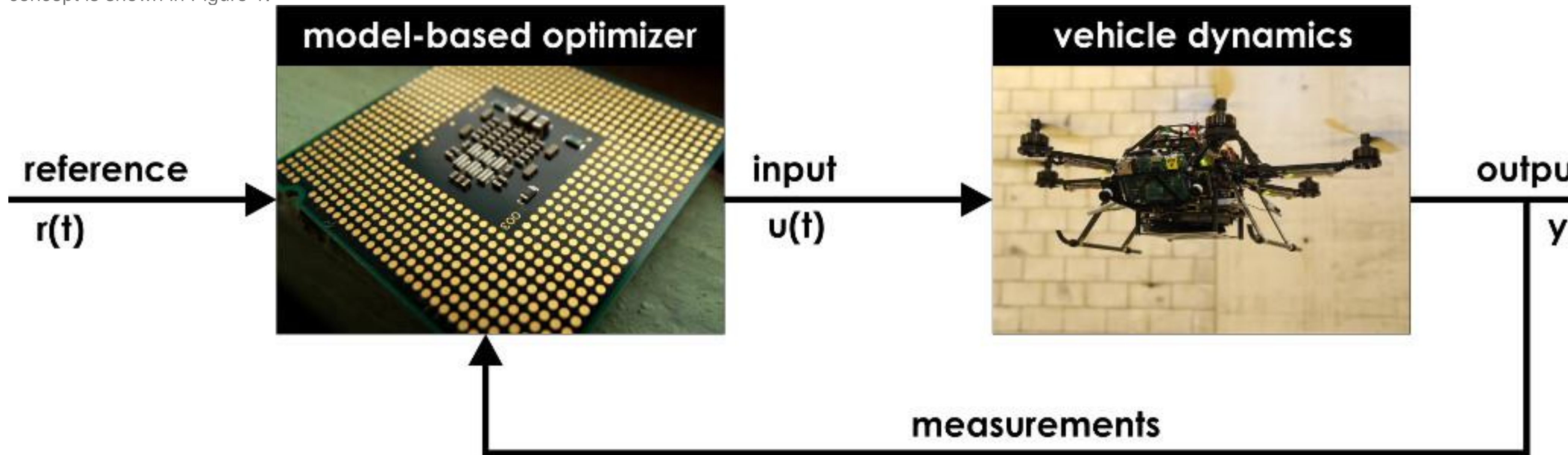


Figure 1: The basic concept of model predictive control as a model-based and optimization-based solution.

Receding Horizon Strategies

Model Predictive Control relies on the concept of receding horizon optimal control derivation. According to this approach, at time ***t*** we solve to find the optimal control sequence over a finite future horizon of ***N*** steps. The relevant formulation is shown below:

Based on the concept of receding horizon, we derive the optimal sequence over ***N*** steps but we only apply its first element - the first optimal control move-action ***u*(t).*** At time ***t+1***, we get new measurements/state estimates and repeat the optimization. Essentially, we exploit feedback to update the optimization over the time horizon selected to predict the future evolution of the system outputs.

To understand the concept of receding horizon control, one can consider the **analogy of a driver steering a car**. Within that analogy:

- Prediction model is what describes how the vehicle is expected to move on the map.
- System Constraints is the set of rules to drive on roads, respect one-ways, don't exceed mechanical capacities of the vehicle.
- Disturbances are the driver's innattention and other reasons for uncontrolled deviation from the desired trajectory.
- Set point is the desired location.
- Cost Function may be the goal of minimum time, minimum distance etc.

The receding horizon control strategy would re-plan the route of the car and the corresponding driver actions periodically in time, find the overall set of actions over a time horizon, apply the first and then re-plan for the next-step.

Deriving a Good Model for Model Predictive Control

MPC relies on the provided model for its computations. In fact, the model selection has a major role regarding the computational complexity of the algorithm, its theoretical properties (e.g. stability). At the same time, the selected objective and imposed constraints also influence and define these properties.

A good model for MPC is a model that is descriptive enough, captures the dominant and important dynamics of the system but also remains simple enough such that it allows the optimization problem to be tractable and solvable in real-time. Finding a good balance between these two requirements is a balance. A good model is simple as possible, but not simpler.

Therefore, the following questions should be answered before deriving a model to be used with Model Predictive Control methods:

- Should -for control purposes- the system be captured with Nonlinear, Linear or Hybrid dynamics?
- What is the required -for control purposes- order of the system?
- Can we decouple the system? What assumptions are required? Are they reasonable?

Linear MPC Tracking Problem

Considering a system captured using the linear dynamics:

The Optimal Constrained Control Problem of Linear MPC Reference Tracking takes the form:

$$\min_{\Delta \mathbf{U}} \sum_{k=0}^{N-1} (\mathbf{y}(k+1) - \mathbf{r}(t))^T \mathbf{Q} (\mathbf{y}(k+1) - \mathbf{r}(t)) + (\mathbf{u}(k) - \mathbf{u}(k-1))^T \mathbf{R} (\mathbf{u}(k) - \mathbf{u}(k-1))$$

$$s.t.: \mathbf{u}_{\min} \leq \mathbf{u}(k) \leq \mathbf{u}_{\max}, \ k = 0, \ldots, N-1$$

$$\Delta \mathbf{u}_{\min} \leq \mathbf{u}(k) - \mathbf{u}(k-1) \leq \Delta \mathbf{u}_{\max}, \ k = 0, \ldots, N-1$$

$$\mathbf{y}_{\min} \leq \mathbf{y}(k) \leq \mathbf{y}_{\max}, \ k = 1, \ldots, N$$

where:

This Convex Quadratic Program (QP) can be re-written in the more general form:

And the derivation of its solutions can be accomplished using modern methods for Convex Optimization.

Design and Implementation of a Linear MPC

Design, implementation and efficient execution of model predictive control is a very challenging problem that requires deep understanding of optimization methods and strong coding skills. However, the great success of the method lead to the fact that one can use advanced software tools to achieve this goal

quite seamlessly. Although deep understanding is always beneficial, implementation of a single MPC may be nothing more than writing a very brief abstract program.

An excellent tool is the "CVXGEN: Code Generation for Convex Optimization". CVXGEN generates fast custom code for small, QP-representable convex optimization problems, using an online interface with no software installation. With minimal effort, turn a mathematical problem description into a high speed solve.

As can be found in the relevant example "Example: Model Predictive Control (MPC)", the following CVX code is enough to produce auto-generated C-code for a Linear MPC:

```
dimensions
  m = 2  # inputs.
  n = 5  # states.
  T = 10  # horizon.
end

parameters
  A (n,n)  # dynamics matrix.
  B (n,m)  # transfer matrix.
  Q (n,n) psd  # state cost.
  Q_final (n,n) psd  # final state cost.
  R (m,m) psd  # input cost.
  x[0] (n)  # initial state.
  u_max nonnegative  # amplitude limit.
  S nonnegative  # slew rate limit.
end

variables
  x[t] (n), t=1..T+1  # state.
  u[t] (m), t=0..T  # input.
end
```

```
minimize
  sum[t=0..T](quad(x[t], Q) + quad(u[t], R)) + quad(x[T+1], Q_final)
subject to
  x[t+1] == A*x[t] + B*u[t], t=0..T  # dynamics constraints.
  abs(u[t]) <= u_max, t=0..T  # maximum input box constraint.
  norminf(u[t+1] - u[t]) <= S, t=0..T-1  # slew rate constraint.
end
```

Examples of use of Model Predictive Control in Aerial Robotics

The following four papers, provide indicative examples of use of Model Predictive Control methods for the problem of flight (and in one case, physical interaction) control of aerial robotics:

Linear MPC:

- K. Alexis, G. Nikolakopoulos, A. Tzes ***"Model Predictive Quadrotor Control: Attitude, Altitude and Position Experimental Studies"***, IET Control Theory and Applications, DOI (10.1049/iet-cta.2011.0348), awarded with the **IET 2014 Premium Award for Best Paper in Control Theory & Applications**. **[Download the Paper]**
- Philipp Oettershagen, Amir Melzer, Stefan Leutenegger, Kostas Alexis, Roland Y. Siegwart, ***"Explicit Model Predictive Control and L1-Navigation Strategies for Fixed–Wing UAV Path Tracking"***, Mediterranean Control Conference, 2014, Palermo, Italy, June 16-June 19, 2014, p. 1159-1165 [**Download the Paper**]

Robust Linear MPC:

- K. Alexis, C. Papachristos, R. Siegwart, A. Tzes, ***"Robust Model Predictive Flight Control of Unmanned Rotorcrafts"***, Journal of Intelligent and Robotic Systems, Springer (DOI: 10.1007/s10846-015-0238-7) [**Download the Paper**]

Hybrid MPC:

- G. Darivianakis, K. Alexis, M. Burri, R. Siegwart, "***Hybrid Predictive Control for Aerial Robotic Physical Interaction towards Inspection Operations***", IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31-June 7, 2014, p. 53-58 (**Best Automation Paper Finalist**) [**Download the Paper**]

Nolinear MPC:

- Mina Kamel, Kostas Alexis, Markus Wilhelm Achtelik, Roland Siegwart, "***Fast Nonlinear Model Predictive Control for Multicopter Attitude Tracking on SO(3)***", Multiconference on Systems and Control (MSC), 2015, Novotel Sydney Manly Pacific, Sydney Australia. 21-23 September, 2015 [**Download the Paper**]

Indicative **Video** results from the aforementioned works are presented below
https://youtu.be/cocvUrPfyfo

https://youtu.be/cMpHo0Lm8jg

https://youtu.be/6TkLvK7IR9Q

https://youtu.be/aF0jGT7UBM0

Motion Planning

The goal of this section is to provide an overview and intuitive introduction on the basic concepts of motion planning for aerial robotics. Motion planning deals with the challenge of how the trajectory/path of an aerial robot should be such that it accomplishes a desired task such as collision-free point-to-point navigation, inspection of a desired structure or exploration of an unknown environment.

A wide variety of techniques for motion planning exist, each of them having its own advantages and disadvantages. Within this course, only a small subset of the relevant methods is presented. More specifically, this section contains the following subsections:

1. Holonomic Vehicle Boundary Value Solver
2. Dubins Airplane model Boundary Value Solver
3. Collision-free Navigation
4. Structural Inspection Path Planning

Holonomic Vehicle Boundary Value Solver

Explicit solutions to the problem of point-to-point navigation of a holonomic vehicle operating within an obstacle-free world are straightforward. More specifically, a 6-degrees of freedom (DOF) vehicle that can be approximated to assume only small roll and pitch angles can be approximated using a very simple Boundary Value Solver (BVS). Considering an approximate state vector **ξ = [x,y,z,ψ]** (where x,y,z are the 3 position states and ψ the yaw angle), the path from the state configuration ξ_0 to ξ_1 is given by:

$$\xi(s) = s\xi_1 + (1 - s)\xi_0, \ \xi \in [0, 1]$$

And considering a limitation on the possible rate of change of the yaw angle \d*ψ/dt*|*max* and the maximum linear velocity u_{max}, the execution time is:

$$t_{ex} = \max(d/v_{max}, \|\psi_1 - \psi_0\|/\dot{\psi}_{\max}$$

with d used to denote the Euclidean distance.

Python Implementation

File: HoverModeMain.py

```
#       __HOVERMODEMAIN__
#       This is the main file to execute examples of the Hover mode
```

```python
#
#       Authors:
#       Kostas Alexis (kalexis@unr.edu)

from HoverFunctions import *
from PlottingTools import plot3
import numpy as np
import time
import sys

pi = np.pi
verbose_flag = 0
plot_flag = 1

point_0 = np.array([0,0,0,0])
point_1 = np.array([10,10,10,pi/4])

class ExecutionFlags(object):
    """
        Execution flags
    """
    def __init__(self, verbose_flag, plot_flag):
        self.verbose = verbose_flag
        self.plot = plot_flag

class VehicleParameters(object):
    """
        Vehicle Parameters
    """
    def __init__(self, v_max, psi_rate_max):
        self.v_max = v_max
        self.psi_rate_max = psi_rate_max

def main():
```

```
    # Example main for the Hover mode
    t0 = time.clock()
    VehiclePars = VehicleParameters( 4, pi/4,)
    ExFlags = ExecutionFlags( verbose_flag,plot_flag )

    flag_nc = 0
    fname = 'hover_solution.txt'
    HoverSolution = AssembleHoverSolution(point_0, point_1, VehiclePars.v_max, VehiclePars.psi_rate_max)
    path_hover = HoverSolution['connection']
    np.savetxt( fname, path_hover, delimiter = ',' )
    if ExFlags.plot :
            print '### Hover solution plot'
            plot3( path_hover[:,0], path_hover[:,1], path_hover[:,2], 'o', 'g' )


def testAllCases():
    main()
    print 'Press any button to continue'
    raw_input()


if __name__ == "__main__":
    testAllCases()
```

File: HoverFunctions.py

```
#       __HOVERFUNCTIONS__
#       This is the main file to execute functions of the Hover mode
#
#       Authors:
#       Kostas Alexis (kalexis@unr.edu)


from __future__ import division
import numpy as np
```

```python
from math import tan, sin, cos, atan2, fmod, acos, asin, pow, sqrt, fabs,atan
from ElementaryFunctions import max, min

pi = np.pi

HoverSolution = { }
HoverSolution['connection'] = np.zeros((2,4));
HoverSolution['distance'] = 0;
HoverSolution['t_ex'] = 0;

def computeHoverConnection(point_0=None, point_1=None):
    L = np.array([point_0, point_1])
    return L

def computeHoverConnectionDistance(point_0=None, point_1=None):
    dist_ = sqrt( pow(point_1[0]-point_0[0],2) + pow(point_1[1]-point_0[1],2) + pow(point_1[2]-point_0[2],2) )
    return dist_

def computeHoverConnectionTime(point_0=None,point_1=None,v_max=None,psi_rate_max=None):
    dist_ = computeHoverConnectionDistance(point_0, point_1)
    t_ex = max( dist_/v_max, fabs(point_1[3]-point_0[3])/psi_rate_max)
    return t_ex

def AssembleHoverSolution(point_0=None,point_1=None,v_max=None,psi_rate_max=None):
    L = computeHoverConnection(point_0, point_1)
    dist_ = computeHoverConnectionDistance(point_0, point_1)
    t_ex = computeHoverConnectionTime(point_0, point_1, v_max, psi_rate_max)
    HoverSolution['connection'] = L
    HoverSolution['distance'] = dist_
    HoverSolution['t_ex'] = t_ex
    return HoverSolution
```

File: PlottingTools.py

```
#       __PLOTTINGTOOLS__
#       Plotting functions
#
#       Authors:
#       Kostas Alexis (kalexis@unr.edu)

import matplotlib.pyplot as mpplot
from matplotlib import pyplot
import pylab
from mpl_toolkits.mplot3d import Axes3D

def plot3(a,b,c,mark="o",col="r"):
    # mimic matlab plot3
    from matplotlib import pyplot
    import pylab
    pylab.ion()
    fig = pylab.figure()
    ax = Axes3D(fig)
    ax.invert_zaxis()
    ax.invert_xaxis()
    ax.set_aspect('equal', 'datalim')
    ax.plot(a, b,c,color=col,marker=mark)
    fig.show()
```
Proudly powered by Weebly

Dubins Airplane

Dubins airplane is an extension of the classical Dubins car model for the 3D case of an airplane. The specific implementation provided here relies on the formulation presented in:

(b)Mark Owen, Randal W. Beard and Timothy W. McLain, "Implementing Dubins Airplane Paths on Fixed-Wing UAVs"

and essentially (as described in this paper) corresponds to a modification of the initial model proposed by *Lavalle et al.* so that it becomes more consistent with the kinematics of a fixed-wing aircraft. Dubins airplane paths are more complicated than Dubins car paths because of the altitude component. Based on

the difference between the altitude of the initial and final configurations, Dubins airplane paths can be classified as low, medium, or high altitude gain. While for medium and high altitude gain there are many different Dubins airplane paths, this implementation selects the pat that maximizes the average altitude throughout the maneuver.

Provided Implementations

1. Python

Run the example script “DubinsAirplaneMain.py” in which, one of the 16 supported path-cases may be found (parameter *dubins_case*):

```
python DubinsAirplaneMain.py
```

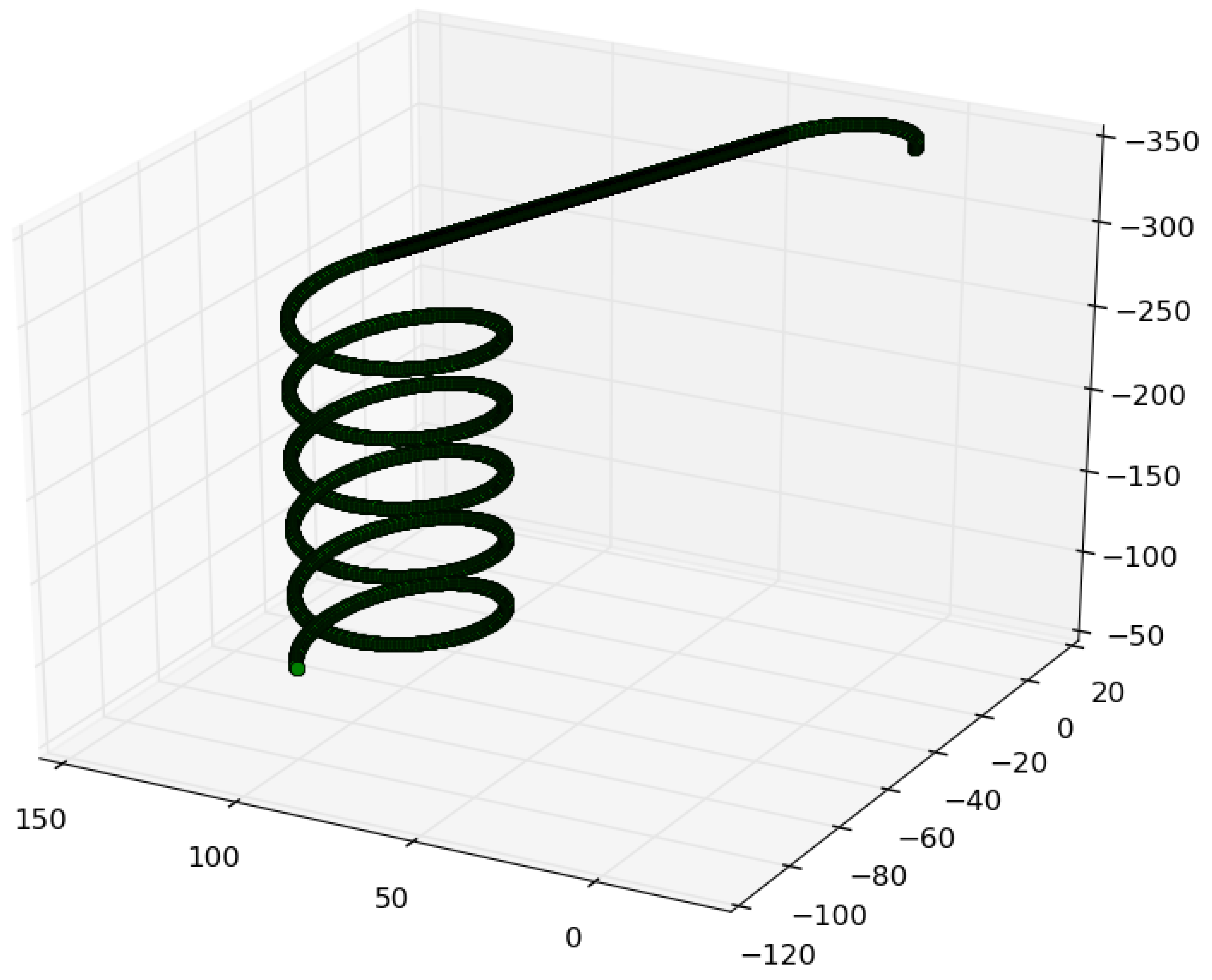
In the same file, the filename to save the path result (*fname*) can be edited while flags related with plotting (*plot_flag*) and execution verbosity (*verbose_flag*) can be set. Furthermore, the vehicle and mission-specific parameters:

* *Vairspeed_0*: nominal airspeed

* *Bank_max*: maximum banki angle

* *Gamma_max*: maximum flight path angle

should be modified.



−350
−300
−250
−200
−150
−100
−50
20
0
−20
−40
−60
−80
−100
−120
150
100
50
0

You can get the code at: https://github.com/UNR-AERIAL/DubinsAirplane or have a quick look to it below.

Dubins Airplane Implementation

Main File: **DubinsAirplaneMain.py**

```
#       __DUBINSAIRPLANEMAIN__
#       This is the main file to execute examples of the Dubins Airplane mode
#       that supports 16 cases of possible trajectories
#
#       Authors:
#       Kostas Alexis (konstantinos.alexis@mavt.ethz.ch)

from DubinsAirplaneFunctions import *
from PlottingTools import plot3
import numpy as np
import time
import sys

pi = np.pi
dubins_case = 0
verbose_flag = 0
plot_flag = 1

class ExecutionFlags(object):
    """
        Execution flags
    """
    def __init__(self, verbose_flag, plot_flag):
        self.verbose = verbose_flag
        self.plot = plot_flag

class VehicleParameters(object):
```

```
    """
        Vehicle Parameters
    """
    def __init__(self, Vairspeed_0, Bank_max, Gamma_max):
        self.Vairspeed_0 = Vairspeed_0
        self.Bank_max = Bank_max
        self.Gamma_max = Gamma_max

def main():
    # Example main for the 16 cases Dubins Airplane paths
    t0 = time.clock()
    VehiclePars = VehicleParameters( 15, pi/4, pi/6 )
    ExFlags = ExecutionFlags( verbose_flag,plot_flag )

    flag_nc = 0
    fname = 'path_dubins_solution.txt'

    if dubins_case == 1: # short climb RSR
        print '### Path Type: short climb RSR'
        start_node  = np.array( [0,   0,   -100,   0*pi/180,    VehiclePars.Vairspeed_0] )
        end_node    = np.array( [0, 200,   -125, 270*pi/180,    VehiclePars.Vairspeed_0] )
    elif dubins_case == 2: # short climb RSL
        print '### Path Type: short climb RSL'
        start_node  = np.array([ 0,   0,    -100, -70*pi/180,    VehiclePars.Vairspeed_0] )
        end_node    = np.array( [100, 100,  -125, -70*pi/180,    VehiclePars.Vairspeed_0] )
    elif dubins_case == 3: # short climb LSR
        print '### Path Type: short climb LSR'
        start_node  = np.array( [0,   0,    -100, 70*pi/180,    VehiclePars.Vairspeed_0] )
        end_node    = np.array( [100, -100, -125, 70*pi/180,    VehiclePars.Vairspeed_0] )
    elif dubins_case == 4: # short climb LSL
        print '### Path Type: short climb LSL'
        start_node  = np.array( [0,   0,    -100,  70*pi/180,   VehiclePars.Vairspeed_0] )
        end_node    = np.array( [100, -100, -125, -135*pi/180,  VehiclePars.Vairspeed_0] )
    elif dubins_case == 5: # long climb RSR
```

```
        print '### Path Type: long climb RSR'
        start_node  = np.array( [0,    0,    -100,    0*pi/180,     VehiclePars.Vairspeed_0] )
        end_node    = np.array( [0, 200,    -250, 270*pi/180,     VehiclePars.Vairspeed_0] )
    elif dubins_case == 6: # long climb RSL
        print '### Path Type: long climb RSL'
        start_node  = np.array( [0,    0,     -100, -70*pi/180,     VehiclePars.Vairspeed_0] )
        end_node    = np.array( [100, 100,  -350, -70*pi/180,     VehiclePars.Vairspeed_0] )
    elif dubins_case == 7: # long climb LSR
        print '### Path Type: long climb LSR'
        start_node  = np.array( [0,    0,     -350, 70*pi/180,     VehiclePars.Vairspeed_0] )
        end_node    = np.array( [100, -100, -100, 70*pi/180,     VehiclePars.Vairspeed_0] )
    elif dubins_case == 8: # long climb LSL
        print '### Path Type: long climb LSL'
        start_node  = np.array( [0,    0,     -350,  70*pi/180,    VehiclePars.Vairspeed_0] )
        end_node    = np.array( [100, -100, -100, -135*pi/180,  VehiclePars.Vairspeed_0] )
    elif dubins_case == 9: # intermediate climb RLSR (climb at beginning)
        print '### Path Type: intermediate climb RLSR (climb at beginning)'
        start_node  = np.array( [0,    0,    -100,    0*pi/180,      VehiclePars.Vairspeed_0] )
        end_node    = np.array( [0, 200,    -200, 270*pi/180,     VehiclePars.Vairspeed_0] )
    elif dubins_case == 10: # intermediate climb RLSL (climb at beginning)
        print '### Path Type: intermediate climb RLSL (climb at beginning)'
        start_node  = np.array( [0,    0,    -100, 0*pi/180,      VehiclePars.Vairspeed_0] )
        end_node    = np.array( [100, 100, -200, -90*pi/180,    VehiclePars.Vairspeed_0] )
    elif dubins_case == 11: # intermediate climb LRSR (climb at beginning)
        print '### Path Type: intermediate climb LRSR (climb at beginning)'
        start_node  = np.array( [0,    0,    -100, 0*pi/180,      VehiclePars.Vairspeed_0] )
        end_node    = np.array( [100, -100, -200, 90*pi/180,     VehiclePars.Vairspeed_0] )
    elif dubins_case == 12: # intermediate climb LRSL (climb at beginning)
        print '### Path Type: intermediate climb LRSL (climb at beginning)'
        start_node  = np.array( [0,    0,    -100, 0*pi/180,      VehiclePars.Vairspeed_0] )
        end_node    = np.array( [100, -100, -200, -90*pi/180,    VehiclePars.Vairspeed_0] )
    elif dubins_case == 13: # intermediate climb RSLR (descend at end)
        print '### Path Type: intermediate climb RSLR (descend at end)'
        start_node  = np.array( [0,    0,    -200, 0*pi/180,      VehiclePars.Vairspeed_0] )
```

```
    end_node    = np.array( [100, 100, -100, 90*pi/180,    VehiclePars.Vairspeed_0] )
elif dubins_case == 14: # intermediate climb RSRL (descend at end)
    print '### Path Type: intermediate climb RSRL (descend at end)'
    start_node  = np.array( [0,    0,    -200, 0*pi/180,      VehiclePars.Vairspeed_0] )
    end_node    = np.array( [100, 100, -100, -90*pi/180,   VehiclePars.Vairspeed_0] )
elif dubins_case == 15: # intermediate climb LSLR (descend at end)
    print '### Path Type: intermediate climb LSLR (descend at end)'
    start_node  = np.array( [0,    0,    -200, 70*pi/180,      VehiclePars.Vairspeed_0] )
    end_node    = np.array( [100, -100, -100, 90*pi/180,     VehiclePars.Vairspeed_0] )
elif dubins_case == 16: # intermediate climb LSRL (descend at end)
    print '### Path Type: intermediate climb LSRL (descend at end)'
    start_node  = np.array( [0,    0,    -150, 0*pi/180,      VehiclePars.Vairspeed_0] )
    end_node    = np.array( [100, -100,-100, -90*pi/180,   VehiclePars.Vairspeed_0] )
elif dubins_case == 0: # for fixing errors
    print '### Path Type: for fixing errors'
    start_node  = np.array( [0,    0,    0, 0, VehiclePars.Vairspeed_0] )
    end_node    = np.array( [40, -140,  100, 11*pi/9, VehiclePars.Vairspeed_0] ) # LSRL
    #end_node    = np.array( [40, -140,  140, 2*pi/9, VehiclePars.Vairspeed_0] ) # LSLR

    #end_node    = np.array( [40, 140,  140, 11*pi/9, VehiclePars.Vairspeed_0] ) # RSLR
    #end_node    = np.array( [40, 140,  140, 1*pi/9, VehiclePars.Vairspeed_0] ) # RSRL

    #end_node    = np.array( [40, 140,  -140, 11*pi/9, VehiclePars.Vairspeed_0] ) # RLSR
    end_node    = np.array( [60, 140,  -140, 0*pi/14, VehiclePars.Vairspeed_0] ) # RLSL
    #end_node    = np.array( [40, -140,  -100, 11*pi/9, VehiclePars.Vairspeed_0] ) # LRSL
    end_node    = np.array( [40, -140,  -100, 10*pi/180, VehiclePars.Vairspeed_0] ) # LRSR

if dubins_case > 16:
    flag_nc = 1
    print 'Not a case'

if flag_nc == 0:
    print '### Case loaded.'
    print '### computing path...'
```

```
        R_min = MinTurnRadius_DubinsAirplane( VehiclePars.Vairspeed_0, VehiclePars.Bank_max )

        # Check if start and end node are too close. Since spiral-spiral-spiral (or curve-curve-curve) paths are not
considered, the optimal path may not be found... (see Shkel, Lumelsky, 2001, Classification of the Dubins set, Prop.
5/6). Below is a conservative bound, which seems (by experiments) to assure a unproblematical computation of the
dubins path.
        if ( np.linalg.norm(end_node[0:2] - start_node[0:2],ord=2) < 6*R_min ):
            print "!!!!!!!!!!!!!!!!"
            print "Conservative condition (end_node[0:2] - start_node[0:2],ord=2) < 6*R_min) not fulfilled!"
            print "Start and end pose are close together. Path of type RLR, LRL may be optimal"
            print "May fail to compute optimal path! Aborting"
            print "!!!!!!!!!!!!!!!!"
            sys.exit()


        DubinsAirplaneSolution = DubinsAirplanePath( start_node, end_node, R_min, VehiclePars.Gamma_max )
        if ExFlags.verbose :
            PrintSolutionAgainstMATLAB( DubinsAirplaneSolution )

        path_dubins_airplane = ExtractDubinsAirplanePath( DubinsAirplaneSolution )
        path_dubins_airplane = path_dubins_airplane.T
        print '### Execution time = ' + str( time.clock() - t0 )
        np.savetxt( fname, path_dubins_airplane.T, delimiter = ',' )
        print '### Dubins airplane solution saved in ' + fname
        if ExFlags.plot :
            print '### Dubins airplane solution plot'
            plot3( path_dubins_airplane[:,0], path_dubins_airplane[:,1], path_dubins_airplane[:,2], 'o', 'g' )

        # print 'Press any button to continue'
        # raw_input()


def testAllCases():
    #for dubins_case in xrange(1, 16):
```

```
    main()
    print 'Press any button to continue'
    raw_input()


if __name__ == "__main__":
    testAllCases()
```

Dubins Airplane Functions file: **DubinsAirplaneFunctions.py**

```
#       __DUBINSAIRPLANEFUNCTIONS__
#       The functions here implement the 3D Dubins Airplane model with totally
#       16 cases of possible trajectories
#
#       Authors:
#       Kostas Alexis (konstantinos.alexis@mavt.ethz.ch)

from __future__ import division
import numpy as np
from math import tan, sin, cos, atan2, fmod, acos, asin, pow, sqrt, fabs,atan
from ElementaryFunctions import max, min

pi = np.pi

# create the results dictionary
DubinsAirplaneSolution = { }
DubinsAirplaneSolution['case'] = 0
DubinsAirplaneSolution['p_s'] = np.zeros((3,1))
DubinsAirplaneSolution['angl_s'] = 0
DubinsAirplaneSolution['p_e'] = np.zeros((3,1))
DubinsAirplaneSolution['R'] = 0
DubinsAirplaneSolution['gamma'] = 0
DubinsAirplaneSolution['L'] = 0
DubinsAirplaneSolution['c_s'] = np.zeros((3,1))
DubinsAirplaneSolution['psi_s'] = 0
```

```
DubinsAirplaneSolution['lamda_s'] = 0
DubinsAirplaneSolution['lamda_si'] = 0
DubinsAirplaneSolution['k_s'] = 0
DubinsAirplaneSolution['c_ei'] = np.zeros((3,1))
DubinsAirplaneSolution['c_si'] = np.zeros((3,1))
DubinsAirplaneSolution['psi_ei'] = 0
DubinsAirplaneSolution['lamda_ei'] = 0
DubinsAirplaneSolution['psi_si'] = 0
DubinsAirplaneSolution['k_ei'] = 0
DubinsAirplaneSolution['c_e'] = 0
DubinsAirplaneSolution['k_si'] = 0
DubinsAirplaneSolution['psi_e'] = 0
DubinsAirplaneSolution['lamda_e'] = 0
DubinsAirplaneSolution['k_e'] = 0
DubinsAirplaneSolution['w_s'] = np.zeros((3,1))
DubinsAirplaneSolution['q_s'] = np.zeros((3,1))
DubinsAirplaneSolution['w_si'] = np.zeros((3,1))
DubinsAirplaneSolution['q_si'] = np.zeros((3,1))
DubinsAirplaneSolution['w_l'] = np.zeros((3,1))
DubinsAirplaneSolution['q_l'] = np.zeros((3,1))
DubinsAirplaneSolution['w_ei'] = np.zeros((3,1))
DubinsAirplaneSolution['q_ei'] = np.zeros((3,1))
DubinsAirplaneSolution['w_e'] = np.zeros((3,1))
DubinsAirplaneSolution['q_e'] = np.zeros((3,1))

def roty(theta=None):
    # Rotation around y
    R = np.array([  [cos(theta),    0,       sin(theta)],
                    [0,             1,       0],
                    [-sin(theta),   0,       cos(theta)] ])
    return R


def rotz(theta=None):
```

```
    # Rotation around z
    R = np.array([  [cos(theta),    -sin(theta),    0],
                    [sin(theta),    cos(theta),     0],
                    [0,             0,              1]  ])
    return R


def computeDubinsRSR(R=None, crs=None, cre=None, anglstart=None,anglend=None):
    # Compute Dubins RSR
    theta = atan2( cre[1]-crs[1], cre[0]-crs[0] )
    L = np.linalg.norm( crs[0:2]-cre[0:2],ord=2 ) + R* fmod( 2*pi+fmod(theta-pi/2,2*pi )-fmod( anglstart-
pi/2,2*pi),2*pi ) + R*fmod( 2*pi+fmod(anglend-pi/2,2*pi )-fmod( theta-pi/2,2*pi),2*pi )
    return L


def computeDubinsLSL(R=None, cls=None, cle=None, anglstart=None, anglend=None):
    # Compute Dubins LSL
    theta = atan2( cle[1]-cls[1], cle[0]-cls[0] )
    L = np.linalg.norm(cls[0:2]-cle[0:2]) + R*fmod( 2*pi-fmod(theta+pi/2,2*pi)+fmod(anglstart+pi/2,2*pi),2*pi ) +
R*fmod( 2*pi-fmod(anglend+pi/2,2*pi)+fmod(theta+pi/2,2*pi),2*pi )
    return L

def computeDubinsLSR(R=None, cls=None, cre=None, anglstart=None, anglend=None):
    # Compute Dubins LSR
    ell = np.linalg.norm( cre[0:2]-cls[0:2],ord=2 )
    theta = atan2( cre[1]-cls[1],cre[0]-cls[0] )
    acos_value = 2 * R / ell
    if fabs( acos_value ) > 1:
        flag_zero = 1
    else:
        flag_zero = 0

    acos_value = max( acos_value,-1 )
    acos_value = min( acos_value,1 )
```

```python
    if ell == 0:
        theta2 = 0
    else:
        theta2 = acos( acos_value )

    if flag_zero == 1:
        theta2 = 0

    if theta2 == 0:
        L = pow(10.0,8)
    else:
        L = sqrt( pow(ell,2) - 4*pow(R,2) ) + R*fmod( 2*pi-fmod(theta+theta2,2*pi ) + fmod(
anglstart+pi/2,2*pi),2*pi ) + R*fmod( 2*pi-fmod(theta+theta2-pi,2*pi)+fmod(anglend-pi/2,2*pi),2*pi )
    return L

def computeDubinsRSL(R=None, crs=None, cle=None, anglstart=None, anglend=None):
    # Compute Dubins RSL
    ell = np.linalg.norm( cle[0:2]-crs[0:2],ord=2 )
    theta = atan2( cle[1]-crs[1],cle[0]-crs[0] )
    asin_value = 2 * R / ell
    if fabs( asin_value ) > 1:
        flag_zero = 1
    else:
        flag_zero = 0
    asin_value = max( asin_value, -1 )
    asin_value = min( asin_value, 1 )
    if ell == 0:
        theta2 = 0
    else:
        theta2 = theta - pi/2 + asin( asin_value )

    if theta2 == 0:
        L = pow( 10.0, 8 )
    else:
```

```
        L = sqrt( fabs(pow(ell,2)-4 * pow(R,2)) ) + R * fmod( 2 * pi + fmod( theta2, 2 * pi )-fmod( anglstart-pi/2,
2 * pi ), 2 * pi ) + R * fmod( 2 * pi + fmod( theta2 + pi, 2 * pi )-fmod( anglend + pi / 2, 2 * pi ), 2 * pi )
    return L

def computeOptimalRadius(zs=None, anglstart=None, ze=None, anglend=None, R_min=None, gamma_max=None, idx=None,
k=None, hdist=None):
    # Compute Optimal Radius
    R1 = R_min
    R2 = 2 * R_min
    R = ( R1 + R2 ) / 2

    if idx == 1:
        error = 1
        while fabs( error ) > 0.1:
            crs = zs + R * np.dot( rotz( pi / 2 ),np.array( [cos(anglstart), sin(anglstart), 0] ).T )
            cre = ze + R * np.dot( rotz( pi / 2 ), np.array( [cos(anglend), sin(anglend), 0] ).T )
            L = computeDubinsRSR(R, crs, cre, anglstart, anglend)
            error = ( L + 2 * pi * k * R ) - fabs( hdist ) / tan( gamma_max )
            if error > 0:
                R2 = R
            else:
                R1 = R
            R= ( R1 + R2 ) / 2
    elif idx == 2:
        error = 1
        while fabs( error ) > 0.1:
            crs = zs + R * np.dot( rotz( pi / 2 ),np.array( [cos(anglstart), sin(anglstart), 0] ).T )
            cle = ze + R * np.dot( rotz( -pi/2 ),np.array( [cos(anglend), sin(anglend), 0] ).T )
            L = computeDubinsRSL( R, crs, cle, anglstart, anglend )
            error = ( L + 2 * pi * k * R ) * tan( gamma_max ) - fabs( hdist )
            if error > 0:
                R2 = R
            else:
                R1 = R
```

```
            R = ( R1 + R2 ) / 2
    elif idx == 3:
        error = 1
        while fabs( error ) > 0.1:
            cls = zs + R * np.dot( rotz( -pi / 2 ), np.array( [cos(anglstart), sin(anglstart), 0] ).T )
            cre = ze + R * np.dot( rotz( pi / 2 ), np.array( [cos(anglend), sin(anglend), 0] ).T )
            L = computeDubinsLSR( R, cls, cre, anglstart, anglend )
            error = ( L + 2 * pi * k * R ) * tan( gamma_max ) - fabs( hdist )
            if error > 0:
                R2 = R
            else:
                R1 = R
            R = ( R1 + R2 ) / 2
    elif idx == 4:
        error = 1
        while fabs( error ) > 0.1:
            cls = zs + R * np.dot( rotz( -pi / 2 ), np.array( [cos(anglstart), sin(anglstart), 0] ).T )
            cle = ze + R * np.dot( rotz( -pi / 2 ), np.array( [cos(anglend), sin(anglend), 0] ).T )
            L = computeDubinsLSL( R, cls, cle, anglstart, anglend )
            error = ( L + 2 * pi * k * R ) * tan( gamma_max ) - fabs( hdist )
            if error > 0:
                R2 = R
            else:
                R1 = R
            R = ( R1 + R2 ) / 2
    return R



def MinTurnRadius_DubinsAirplane(V=None,phi_max=None):
    # Compute Minimum Turning Radius
    g = 9.8065
    Rmin = pow( V,2 ) / (g * tan( phi_max ) )
```

```
    return Rmin

def addSpiralBeginning(zs=None, anglstart=None, ze=None, anglend=None, R_min=None, gamma_max=None, idx=None,
hdist=None):
    # Add Spiral in the Dubins Airplane Path beginning
    cli = np.zeros((3,1))
    cri = np.zeros((3,1))
    zi = np.zeros((3,1))
    anglinter = 0
    L = 0
    ci = np.zeros((3,1))
    psii = 0
    psi1 = 0
    psi2 = 2 * pi
    psi = ( psi1 + psi2 ) / 2

    if idx == 1: # RLSR
        crs = zs + R_min * np.dot( rotz( pi/2 ), np.array( [cos(anglstart), sin(anglstart), 0] ).T )
        cre = ze + R_min * np.dot( rotz( pi/2 ), np.array( [cos(anglend), sin(anglend), 0] ).T )
        L = computeDubinsRSR( R_min, crs, cre, anglstart, anglend )
        error = L - fabs( hdist / tan( gamma_max ) )
        while fabs( error ) > 0.001:
            zi = crs + np.dot( rotz( psi ),( zs-crs ) )
            anglinter = anglstart + psi
            cli = zi + R_min * np.dot( rotz( -pi/2 ), np.array( [cos(anglinter), sin(anglinter), 0] ).T )
            L = computeDubinsLSR( R_min, cli, cre, anglinter, anglend )
            error = ( L + fabs( psi ) * R_min ) - fabs( hdist / tan( gamma_max ) )

            if error > 0:
                psi2 = (179*psi2+psi)/180
            else:
                psi1 = (179*psi1+psi)/180

            psi = ( psi1 + psi2 ) / 2
```

```
        zi = crs + np.dot( rotz( psi ), ( zs-crs ) )
        anglinter = anglstart + psi
        L = L + fabs( psi ) * R_min
        ci = cli
        psii = psi

    elif idx == 2: # RLSL
        crs = zs + R_min * np.dot( rotz( pi / 2 ), np.array( [cos(anglstart), sin(anglstart), 0] ).T )
        cle = ze + R_min * np.dot( rotz( -pi / 2 ), np.array( [cos(anglend), sin(anglend), 0] ).T )
        L = computeDubinsRSL( R_min, crs, cle, anglstart, anglend )
        error = L - fabs( hdist / tan( gamma_max ) )
        while fabs( error ) > 0.001:
            zi = crs + np.dot( rotz( psi ), ( zs-crs ) )
            anglinter = anglstart + psi
            cli = zi + R_min * np.dot( rotz( -pi / 2 ), np.array( [cos(anglinter), sin(anglinter), 0] ).T )
            L = computeDubinsLSL( R_min, cli, cle, anglinter, anglend )
            error = ( L + fabs( psi ) * R_min ) - fabs( hdist / tan( gamma_max ) )

            if error > 0:
                psi2 = (179*psi2+psi)/180
            else:
                psi1 = (179*psi1+psi)/180

            psi = ( psi1 + psi2 ) / 2

        zi   = crs + np.dot( rotz( psi ), ( zs-crs ) )
        anglinter = anglstart + psi
        L = L + fabs( psi ) * R_min
        ci = cli
        psii = psi

    elif idx == 3: # LRSR
        cls = zs + R_min * np.dot( rotz( -pi/2 ), np.array( [cos(anglstart), sin(anglstart), 0] ).T )
```

```
        cre = ze + R_min * np.dot( rotz( pi/2 ), np.array( [cos(anglend), sin(anglend), 0] ).T )
        L = computeDubinsLSR( R_min, cls, cre, anglstart, anglend )
        error = L - fabs( hdist / tan( gamma_max ) )

        while fabs( error ) > 0.001:
            zi = cls + np.dot( rotz( -psi ), ( zs-cls ) )
            anglinter = anglstart - psi
            cri = zi + R_min * np.dot( rotz( pi / 2 ), np.array( [cos(anglinter), sin(anglinter), 0] ).T )
            L = computeDubinsRSR( R_min, cri, cre, anglinter, anglend )
            error = ( L + fabs( psi ) * R_min ) - fabs( hdist / tan( gamma_max ) )

            if error > 0:
                psi2 = (179*psi2+psi)/180
            else:
                psi1 = (179*psi1+psi)/180

            psi = ( psi1 + psi2 ) / 2

        zi   = cls + np.dot( rotz( -psi ), ( zs-cls ) )
        anglinter = anglstart - psi
        L = L + fabs( psi ) * R_min
        ci = cri
        psii = psi

    elif idx == 4: # LRSL
        cls = zs + R_min * np.dot( rotz( -pi/2 ), np.array( [cos(anglstart), sin(anglstart), 0] ).T )
        cle = ze + R_min * np.dot( rotz( -pi/2 ), np.array( [cos(anglend), sin(anglend), 0] ).T )
        # above modified by liucz 2015-10-12, fix spell mistake cre -> cle
        # origin is "cre = ze + R_min * np.dot( rotz( -pi/2 ), np.array( [cos(anglend), sin(anglend), 0] ).T )"
        L = computeDubinsLSL( R_min, cls, cle, anglstart, anglend )
        error = L - fabs( hdist / tan( gamma_max ) )

        while fabs( error ) > 0.001:
            zi = cls + np.dot( rotz( -psi ), ( zs-cls ) )
```

```
            anglinter = anglstart - psi
            cri = zi + R_min * np.dot( rotz( pi / 2 ), np.array( [cos(anglinter), sin(anglinter), 0 ] ).T )
            # above is modified by licz 2015-10-12, fix written mistake np.array -> np.dot
            # origin is "cri = zi + R_min * np.array( rotz( pi / 2 ), np.array( [cos(anglinter), sin(anglinter), 0 ]
).T )"
            L = computeDubinsRSL( R_min, cri, cle, anglinter, anglend )
            error = ( L + fabs( psi ) * R_min ) - fabs( hdist / tan( gamma_max) )

            if error > 0:
                psi2 = (179*psi2+psi)/180
            else:
                psi1 = (179*psi1+psi)/180

            psi = ( psi1 + psi2 ) / 2

        zi   = cls + np.dot( rotz( -psi ), ( zs-cls ) )
        anglinter = anglstart - psi
        L = L + fabs( psi ) * R_min
        ci = cri
        psii = psi

    return zi, anglinter, L, ci, psii


def addSpiralEnd(zs=None, anglstart=None, ze=None, anglend=None, R_min=None, gamma_max=None, idx=None, hdist=None):
    # Add Spiral at the end of the Dubins Airplane path
    cli = np.zeros((3,1))
    cri = np.zeros((3,1))
    zi = np.zeros((3,1))
    anglinter = 0
    L = 0
    ci = np.zeros((3,1))
    psii = 0
    psi1 = 0
```

```
psi2 = 2 * pi
psi = ( psi1 + psi2 ) / 2

if idx == 1: # RSLR
    crs = zs + R_min * np.dot( rotz( pi / 2 ), np.array( [cos(anglstart), sin(anglstart), 0] ).T )
    cre = ze + R_min * np.dot( rotz( pi/2 ), np.array( [cos(anglend), sin(anglend), 0] ).T )
    L = computeDubinsRSR( R_min, crs, cre, anglstart, anglend )
    error = L - fabs( hdist / tan( gamma_max ) )

    while fabs( error ) > 0.001:
        zi = cre + np.dot( rotz( -psi ), ( ze-cre ) )
        anglinter = anglend - psi
        cli = zi + R_min * np.dot( rotz( -pi / 2 ), np.array( [cos(anglinter), sin(anglinter), 0] ).T )
        L = computeDubinsRSL( R_min, crs, cli, anglstart, anglinter )
        error = ( L + fabs( psi ) * R_min ) - fabs( hdist / tan( gamma_max ) )

        if error > 0:
            psi2 = (179*psi2+psi)/180
        else:
            psi1 = (179*psi1+psi)/180

        psi = ( psi1 + psi2 ) / 2

    zi   = cre + np.dot( rotz( -psi ), ( ze-cre ) )
    anglinter = anglend - psi
    L = L + abs( psi ) * R_min
    ci = cli
    psii = psi

elif idx == 2: # RSRL
    crs = zs + R_min * np.dot( rotz( pi / 2 ), np.array( [cos(anglstart), sin(anglstart), 0] ).T )
    cle = ze + R_min * np.dot( rotz( -pi/2 ), np.array( [cos(anglend), sin(anglend), 0] ).T )
    L = computeDubinsRSL( R_min, crs, cle, anglstart, anglend )
    error = L - fabs( hdist / tan( gamma_max ) )
```

```
        while fabs( error ) > 0.001:
            zi = cle + np.dot( rotz( psi ), ( ze-cle ) )
            anglinter = anglend + psi
            cri = zi + R_min * np.dot( rotz( pi / 2 ), np.array( [cos(anglinter), sin(anglinter), 0] ).T )
            L = computeDubinsRSR( R_min, crs, cri, anglstart, anglinter )
            error = ( L + fabs( psi ) * R_min ) - fabs( hdist / tan( gamma_max ) )

            if error > 0:
                psi2 = (179*psi2+psi)/180
            else:
                psi1 = (179*psi1+psi)/180

            psi = ( psi1 + psi2 ) / 2

        zi = cle + np.dot( rotz( psi ), ( ze-cle ) )
        anglinter = anglend + psi
        cri  = zi + R_min * np.dot( rotz( pi / 2 ), np.array( [cos(anglinter), sin(anglinter), 0] ).T )
        L = L + fabs( psi ) * R_min
        ci = cri
        psii = psi

    elif idx == 3: # LSLR
        cls = zs + R_min * np.dot( rotz( -pi / 2 ), np.array( [cos(anglstart), sin(anglstart), 0] ).T )
        cre = ze + R_min * np.dot( rotz( pi / 2 ), np.array( [cos(anglend), sin(anglend), 0] ).T )
        L = computeDubinsLSR( R_min, cls, cre, anglstart, anglend )
        error = L - fabs( hdist / tan( gamma_max ))

        while fabs( error ) > 0.001:
            zi = cre + np.dot( rotz( -psi ), ( ze-cre ) )
            anglinter = anglend - psi
            cli = zi + R_min * np.dot( rotz( -pi / 2 ), np.array( [cos(anglinter), sin(anglinter), 0] ).T )
            L = computeDubinsLSL( R_min, cls, cli, anglstart, anglinter )
            error = ( L + fabs( psi ) * R_min ) - fabs( hdist / tan( gamma_max ) )
```

```
            if error > 0:
                psi2 = (179*psi2+psi)/180
            else:
                psi1 = (179*psi1+psi)/180

            psi = ( psi1 + psi2 ) / 2

        zi   = cre + np.dot( rotz( -psi ), ( ze-cre ) )
        anglinter = anglend - psi
        L = L + fabs( psi ) * R_min
        ci = cli
        psii = psi

    elif idx == 4:

        cls = zs + R_min * np.dot( rotz( -pi/2 ), np.array( [cos(anglstart), sin(anglstart), 0] ).T )
        cle = ze + R_min * np.dot( rotz( -pi/2 ), np.array( [cos(anglend), sin(anglend), 0] ).T )
        L = computeDubinsLSL( R_min, cls, cle, anglstart, anglend )
        error = L - fabs( hdist / tan( gamma_max ) )

        while fabs( error ) > 0.001:
            zi = cle + np.dot( rotz( psi ), ( ze-cle ) )
            anglinter = anglend + psi
            cri = zi + R_min * np.dot( rotz( pi / 2 ), np.array( [cos(anglinter), sin(anglinter), 0] ).T )
            L = computeDubinsLSR( R_min, cls, cri, anglstart, anglinter )
            error = ( L + fabs( psi ) * R_min ) - fabs( hdist / tan( gamma_max ) )

            if error > 0:
                psi2 = (179*psi2+psi)/180
            else:
                psi1 = (179*psi1+psi)/180

            psi = ( psi1 + psi2 ) / 2
```

```
        zi = cle + np.dot( rotz( psi ), ( ze-cle ))
        anglinter = anglend + psi
        L = L + fabs( psi ) * R_min
        ci = cri
        psii = psi

    return zi, anglinter, L, ci, psii



def DubinsAirplanePath(init_conf=None, final_conf=None, R_min=None, gamma_max=None):
    # Compute the Dubins Airplane path

    zs = (init_conf[0:3]).T
    anglstart = init_conf[3]
    ze = (final_conf[0:3]).T
    anglend = final_conf[3]

    DubinsAirplaneSolution['p_s'] = zs
    DubinsAirplaneSolution['angl_s'] = anglstart
    DubinsAirplaneSolution['p_e'] = ze
    DubinsAirplaneSolution['angl_e'] = anglend

    crs = zs + R_min*np.dot(rotz(pi/2), np.array([cos(anglstart), sin(anglstart), 0]).T)
    cls = zs + R_min*np.dot(rotz(-pi/2),np.array([cos(anglstart), sin(anglstart), 0]).T)
    cre = ze + R_min*np.dot(rotz(pi/2),np.array([cos(anglend), sin(anglend), 0]).T)
    cle = ze + R_min*np.dot(rotz(-pi/2),np.array([cos(anglend), sin(anglend), 0]).T)

    # compute L1, L2, L3, L4
    L1 = computeDubinsRSR(R_min, crs, cre, anglstart, anglend)
    L2 = computeDubinsRSL(R_min, crs, cle, anglstart, anglend)
    L3 = computeDubinsLSR(R_min, cls, cre, anglstart, anglend)
    L4 = computeDubinsLSL(R_min, cls, cle, anglstart, anglend)
```

```
# L is the minimum distance
L = np.amin(np.array([L1, L2, L3, L4]))
idx = np.where(np.array([L1,L2,L3,L4])==L)[0][0] + 1

hdist = -(ze[2] - zs[2])
if fabs(hdist) <= L*tan(gamma_max):
    gam = atan(hdist/L)
    DubinsAirplaneSolution['case'] = 1
    DubinsAirplaneSolution['R'] = R_min
    DubinsAirplaneSolution['gamma'] = gam
    DubinsAirplaneSolution['L'] = L/cos(gam)
    DubinsAirplaneSolution['k_s'] = 0
    DubinsAirplaneSolution['k_e'] = 0
elif fabs(hdist) >= (L+2*pi*R_min)*tan(gamma_max):

    k = np.floor( (fabs(hdist)/tan(gamma_max) - L)/(2*pi*R_min))

    if hdist >= 0:

        DubinsAirplaneSolution['k_s'] = k
        DubinsAirplaneSolution['k_e'] = 0
    else:
        DubinsAirplaneSolution['k_s'] = 0
        DubinsAirplaneSolution['k_e'] = k

    # find optimal turning radius
    R = computeOptimalRadius(zs, anglstart, ze, anglend, R_min, gamma_max, idx, k, hdist)

    # recompute the centers of spirals and Dubins path length with new R
    crs = zs + R*np.dot(rotz(pi/2), np.array( [cos(anglstart), sin(anglstart), 0] ).T )
    cls = zs + R*np.dot(rotz(-pi/2), np.array( [cos(anglstart), sin(anglstart), 0] ).T )
    cre = ze + R*np.dot(rotz(pi/2), np.array( [cos(anglend), sin(anglend), 0] ).T )
    cle = ze + R*np.dot(rotz(-pi/2), np.array( [cos(anglend), sin(anglend), 0] ).T )
```

```
        if idx == 1:
            L = computeDubinsRSR( R, crs, cre, anglstart, anglend )
        elif idx == 2:
            L = computeDubinsRSL( R, crs, cle, anglstart, anglend )
        elif idx == 3:
            L = computeDubinsLSR( R, cls, cre, anglstart, anglend )
        elif idx == 4:
            L = computeDubinsLSL( R, cls, cle, anglstart, anglend )

        DubinsAirplaneSolution['case'] = 1
        DubinsAirplaneSolution['R'] = R
        gam = np.sign( hdist ) * gamma_max
        DubinsAirplaneSolution['gamma'] = gam
        DubinsAirplaneSolution['L'] = ( L + 2 * pi * k * R ) / cos( gamma_max )

    else:

        gam = np.sign( hdist ) * gamma_max

        if hdist > 0:
            zi, chii, L, ci, psii = addSpiralBeginning( zs, anglstart, ze, anglend, R_min, gam, idx, hdist )
            DubinsAirplaneSolution['case'] = 2
        else:
            zi, chii, L, ci, psii = addSpiralEnd( zs, anglstart, ze, anglend, R_min, gam, idx, hdist )
            DubinsAirplaneSolution['case'] = 3

        DubinsAirplaneSolution['R'] = R_min
        DubinsAirplaneSolution['gamma'] = gam
        DubinsAirplaneSolution['L'] = L / cos( gamma_max )

    e1 = np.array( [1, 0, 0] ).T
    R = DubinsAirplaneSolution['R']
```

```
    if np.isscalar(DubinsAirplaneSolution['case']):
        pass
    else:
        print '### Error'



    if DubinsAirplaneSolution['case'] == 1: # spiral-line-spiral
        if idx == 1: # right-straight-right
            theta = atan2( cre[1]-crs[1], cre[0]-crs[0])
            dist1 = R*fmod(2*pi+fmod(theta-pi/2,2*pi)-fmod(anglstart-pi/2,2*pi),2*pi) +
2*pi*R*DubinsAirplaneSolution['k_s']
            dist2 = R*fmod(2*pi+fmod(anglend-pi/2,2*pi)-fmod(theta-pi/2,2*pi),2*pi) +
2*pi*R*DubinsAirplaneSolution['k_e']
            w1 = crs + DubinsAirplaneSolution['R']*np.dot(rotz(theta-pi/2),e1.T).T + np.array([0,0,-
dist1*tan(gam)]).T
            w2 = cre + DubinsAirplaneSolution['R']*np.dot(rotz(theta-pi/2),e1.T).T - np.array([0,0,-
dist2*tan(gam)]).T
            q1 = (w2-w1)/np.linalg.norm(w2-w1,ord=2) # direction of line

            DubinsAirplaneSolution['c_s']    = crs
            DubinsAirplaneSolution['psi_s'] = anglstart-pi/2
            DubinsAirplaneSolution['lamda_s'] = 1
            # end spiral
            DubinsAirplaneSolution['c_e']    = cre-np.array([0,0,-dist2*tan(gam)])
            DubinsAirplaneSolution['psi_e'] = theta-pi/2
            DubinsAirplaneSolution['lamda_e'] = 1
            # hyperplane H_s: switch from first spiral to line
            DubinsAirplaneSolution['w_s']    = w1
            DubinsAirplaneSolution['q_s']    = q1
            # hyperplane H_l: switch from line to last spiral
            DubinsAirplaneSolution['w_l']    = w2
            DubinsAirplaneSolution['q_l']    = q1
            # hyperplane H_e: end of Dubins path
```

```
            DubinsAirplaneSolution['w_e']    = ze
            DubinsAirplaneSolution['q_e']    = np.dot(rotz(anglend), np.array([1,0,0]).T)
        elif idx == 2: # right-straight-left
            ell = np.linalg.norm(cle[0:2] - crs[0:2],ord=2)
            theta = atan2(cle[1]-crs[1], cle[0]-crs[0])
            theta2 = theta - pi/2 + asin(2*R/ell)
            dist1 = R*fmod(2*pi+fmod(theta2,2*pi)-fmod(anglstart-pi/2,2*pi),2*pi) +
2*pi*R*DubinsAirplaneSolution['k_s']
            dist2 = R*fmod(2*pi+fmod(theta2+pi,2*pi)-fmod(anglend+pi/2,2*pi),2*pi) +
2*pi*R*DubinsAirplaneSolution['k_e']
            w1 = crs + R*np.dot(rotz(theta2), e1.T).T + np.array([0, 0, -dist1*tan(gam)]).T
            w2 = cle + R*np.dot(rotz(theta2+pi),e1.T).T - np.array([0,0,-dist2*tan(gam)]).T
            q1 = (w2-w1)/np.linalg.norm(w2-w1,ord=2)
            # start spiral
            DubinsAirplaneSolution['c_s']    = crs
            DubinsAirplaneSolution['psi_s'] = anglstart-pi/2
            DubinsAirplaneSolution['lamda_s'] = 1
            # end spiral
            DubinsAirplaneSolution['c_e']    = cle - np.array([0,0,-dist2*tan(gam)]).T
            DubinsAirplaneSolution['psi_e'] = theta2+pi
            DubinsAirplaneSolution['lamda_e'] = -1
            # hyperplane H_s: switch from first spiral to line
            DubinsAirplaneSolution['w_s']    = w1
            DubinsAirplaneSolution['q_s']    = q1
            # hyperplane H_l: switch from line to end spiral
            DubinsAirplaneSolution['w_l']    = w2
            DubinsAirplaneSolution['q_l']    = q1
            # hyperplane H_e: end of Dubins path
            DubinsAirplaneSolution['w_e'] = ze
            DubinsAirplaneSolution['q_e']    = np.dot(rotz(anglend),np.array([1,0,0]).T)
        elif idx == 3: # left-straight-right
            ell = np.linalg.norm(cre[0:2]-cls[0:2],ord=2)
            theta = atan2( cre[1]-cls[1],cre[0]-cls[0])
            theta2 = acos(2*R/ell)
```

```
            dist1 = R*fmod(2*pi-fmod(theta+theta2,2*pi) + fmod(anglstart+pi/2,2*pi),2*pi) +
2*pi*R*DubinsAirplaneSolution['k_s']
            dist2 = R*fmod(2*pi-fmod(theta+theta2-pi,2*pi)+fmod(anglend-pi/2,2*pi),2*pi) +
2*pi*R*DubinsAirplaneSolution['k_e']
            w1 = cls + R*np.dot(rotz(theta+theta2),e1.T).T + np.array([0, 0, -dist1*tan(gam)]).T
            w2 = cre + R*np.dot(rotz(-pi+theta+theta2),e1.T).T - np.array([0, 0, -dist2*tan(gam)]).T
            q1 = (w2-w1)/np.linalg.norm(w2-w1,ord=2)

            # start spiral
            DubinsAirplaneSolution['c_s']    = cls
            DubinsAirplaneSolution['psi_s'] = anglstart+pi/2
            DubinsAirplaneSolution['lamda_s'] = -1
            # end spiral
            DubinsAirplaneSolution['c_e']    = cre - np.array([0,0,-dist2*tan(gam)]).T
            DubinsAirplaneSolution['psi_e'] = fmod(theta+theta2-pi,2*pi)
            DubinsAirplaneSolution['lamda_e'] = 1
            # hyperplane H_s: switch from first spiral to line
            DubinsAirplaneSolution['w_s']    = w1
            DubinsAirplaneSolution['q_s']    = q1
            # hyperplane H_l: switch from line to end spiral
            DubinsAirplaneSolution['w_l']    = w2
            DubinsAirplaneSolution['q_l']    = q1
            # hyperplane H_e: end of Dubins path
            DubinsAirplaneSolution['w_e']    = ze
            DubinsAirplaneSolution['q_e']    = np.dot(rotz(anglend), np.array([1, 0, 0]).T)
        elif idx ==4: # left-straight-left
            theta = atan2(cle[1] -cls[1], cle[0] - cls[0])
            dist1 = R*fmod(2*pi-fmod(theta+pi/2,2*pi)+fmod(anglstart+pi/2,2*pi),2*pi) +
2*pi*R*DubinsAirplaneSolution['k_s']
            dist2 = R*fmod(2*pi-fmod(anglend+pi/2,2*pi)+fmod(theta+pi/2,2*pi),2*pi) +
2*pi*R*DubinsAirplaneSolution['k_e']
            w1 = cls + DubinsAirplaneSolution['R']*np.dot(rotz(theta+pi/2),e1.T).T + np.array([0,0,-
dist1*tan(gam)]).T
```

```
            w2 = cle + DubinsAirplaneSolution['R']*np.dot(rotz(theta+pi/2),e1.T).T - np.array([0,0,-
dist2*tan(gam)]).T
            q1 = (w2-w1)/np.linalg.norm(w2-w1,ord=2)

            # start spiral
            DubinsAirplaneSolution['c_s']   = cls
            DubinsAirplaneSolution['psi_s'] = anglstart+pi/2
            DubinsAirplaneSolution['lamda_s'] = -1
            # end spiral
            DubinsAirplaneSolution['c_e']   = cle - np.array([0,0,-dist2*tan(gam)]).T
            DubinsAirplaneSolution['psi_e'] = theta+pi/2
            DubinsAirplaneSolution['lamda_e'] = -1
            # hyperplane H_s: switch from first spiral to line
            DubinsAirplaneSolution['w_s']   = w1
            DubinsAirplaneSolution['q_s']   = q1
            # hyperplane H_l: switch from line to end spiral
            DubinsAirplaneSolution['w_l']   = w2
            DubinsAirplaneSolution['q_l']   = q1
            # hyperplane H_e: end of Dubins path
            DubinsAirplaneSolution['w_e']   = ze
            DubinsAirplaneSolution['q_e']   = np.dot(rotz(anglend), np.array([1,0,0]).T)
    elif DubinsAirplaneSolution['case'] == 2:
        if idx == 1: # right-left-straight-right
            # start spiral
            DubinsAirplaneSolution['c_s'] = crs
            DubinsAirplaneSolution['psi_s'] = anglstart-pi/2
            DubinsAirplaneSolution['lamda_s'] = 1
            DubinsAirplaneSolution['k_s']   = 0
            ell = np.linalg.norm(cre[0:2]-ci[0:2],ord=2)
            theta = atan2(cre[1] - ci[1], cre[0] - ci[0])
            theta2 = acos(2*R/ell)
            dist1 = R_min*psii + R*fmod(2*pi-fmod(theta+theta2,2*pi) + fmod(chii+pi/2,2*pi),2*pi)
            dist2 = R*fmod(2*pi-fmod(theta+theta2-pi,2*pi)+fmod(anglend-pi/2,2*pi),2*pi)
            w1 = ci + R*np.dot(rotz(theta+theta2),e1.T).T + np.array([0, 0, -dist1*tan(gam)]).T
```

```
            w2 = cre + R*np.dot(rotz(-pi+theta+theta2),e1.T).T - np.array([0, 0, -dist2*tan(gam)]).T
            q1 = (w2-w1)/np.linalg.norm(w2-w1,ord=2)
            # intermediate-start spiral
            DubinsAirplaneSolution['c_si']   = ci + np.array([0, 0, -R_min*psii*tan(gam)]).T
            DubinsAirplaneSolution['psi_si'] = chii + pi/2
            DubinsAirplaneSolution['lamda_si'] = -1
            DubinsAirplaneSolution['k_si']   = 0
            # end spiral
            DubinsAirplaneSolution['c_e']   = cre - np.array([0,0,-dist2*tan(gam)]).T
            DubinsAirplaneSolution['psi_e'] = fmod(theta+theta2-pi,2*pi)
            DubinsAirplaneSolution['lamda_e'] = 1
            DubinsAirplaneSolution['k_e']   = 0
            # hyperplane H_s: switch from first to second spiral
            DubinsAirplaneSolution['w_s']   = zi - np.array([0, 0, -psii*R_min*tan(gam)]).T
            DubinsAirplaneSolution['q_s']   = np.array([cos(chii), sin(chii), 0]).T
            # hyperplane H_si: switch from second spiral to straight line
            DubinsAirplaneSolution['w_si']  = w1
            DubinsAirplaneSolution['q_si']  = q1
            # hyperplane H_l: switch from straight-line to end spiral
            DubinsAirplaneSolution['w_l']   = w2
            DubinsAirplaneSolution['q_l']   = q1
            # hyperplane H_e: end of Dubins path
            DubinsAirplaneSolution['w_e']   = ze
            DubinsAirplaneSolution['q_e']   = np.dot(rotz(anglend), np.array([1,0,0]).T)

        elif idx == 2: # right-left-straight-left
            theta = atan2(cle[1]-ci[1],cle[0]-ci[0])
            dist1 = R*fmod(2*pi-fmod(theta+pi/2,2*pi)+fmod(chii+pi/2,2*pi),2*pi)
            dist2 = psii*R
            dist3 = R*fmod(2*pi-fmod(anglend+pi/2,2*pi)+fmod(theta+pi/2,2*pi),2*pi)
            w1 = ci + DubinsAirplaneSolution['R']*np.dot(rotz(theta+pi/2),e1.T).T + np.array([0, 0, -
(dist1+dist2)*tan(gam)]).T
            w2 = cle + DubinsAirplaneSolution['R']*np.dot(rotz(theta+pi/2),e1.T).T - np.array([0,0,-
dist3*tan(gam)]).T
```

```python
        q1 = (w2-w1)/np.linalg.norm(w2-w1,ord=2) # direction of line

        # start spiral
        DubinsAirplaneSolution['c_s']   = crs
        DubinsAirplaneSolution['psi_s'] = anglstart-pi/2
        DubinsAirplaneSolution['lamda_s'] = 1
        DubinsAirplaneSolution['k_s']   = 0
        # intermediate-start spiral
        DubinsAirplaneSolution['c_si'] = ci + np.array([0, 0, -dist2*tan(gam)]).T
        DubinsAirplaneSolution['psi_si'] = chii+pi/2
        DubinsAirplaneSolution['lamda_si'] = -1
        DubinsAirplaneSolution['k_si']   = 0
        # end spiral
        DubinsAirplaneSolution['c_e'] = cle - np.array([0, 0, -dist3*tan(gam)]).T
        DubinsAirplaneSolution['psi_e'] = theta+pi/2
        DubinsAirplaneSolution['lamda_e'] = -1
        DubinsAirplaneSolution['k_e']   = 0
        # hyperplane H_s: switch from first to second spiral
        DubinsAirplaneSolution['w_s'] = zi - np.array([0, 0, -dist2*tan(gam)]).T
        DubinsAirplaneSolution['q_s'] = np.array([cos(chii), sin(chii), 0]).T
        # hyperplane H_si: switch from second spiral to straight line
        DubinsAirplaneSolution['w_si'] = w1
        DubinsAirplaneSolution['q_si'] = q1
        # hyperplane H_l: switch from straight-line to end spiral
        DubinsAirplaneSolution['w_l'] = w2
        DubinsAirplaneSolution['q_l'] = q1
        # hyperplane H_e: end of Dubins path
        DubinsAirplaneSolution['w_e'] = ze
        DubinsAirplaneSolution['q_e'] = np.dot(rotz(anglend), np.array([1, 0, 0]).T)

    elif idx == 3: # left-right-straight-right
        theta = atan2(cre[1]-ci[1], cre[0] - ci[0])
        dist1 = R*fmod(2*pi+fmod(theta-pi/2,2*pi)-fmod(chii-pi/2,2*pi),2*pi)
        dist2 = psii*R
```

```
            dist3 = R*fmod(2*pi+fmod(anglend-pi/2,2*pi)-fmod(theta-pi/2,2*pi),2*pi)
            w1 = ci + DubinsAirplaneSolution['R']*np.dot(rotz(theta-pi/2),e1.T).T + np.array([0, 0, -
(dist1+dist2)*tan(gam)]).T
            w2 = cre + DubinsAirplaneSolution['R']*np.dot(rotz(theta-pi/2),e1.T).T - np.array([0, 0, -
dist3*tan(gam)]).T
            q1 = (w2-w1)/np.linalg.norm(w2-w1,ord=2) # direction of line
            # start spiral
            DubinsAirplaneSolution['c_s']   = cls
            DubinsAirplaneSolution['psi_s'] = anglstart+pi/2
            DubinsAirplaneSolution['lamda_s'] = -1
            DubinsAirplaneSolution['k_s'] = 0
            # intermediate-start spiral
            DubinsAirplaneSolution['c_si'] = ci + np.array([0, 0, -dist2*tan(gam)]).T
            DubinsAirplaneSolution['psi_si'] = chii-pi/2
            DubinsAirplaneSolution['lamda_si'] = 1
            DubinsAirplaneSolution['k_si'] = 0
            # end spiral
            DubinsAirplaneSolution['c_e'] = cre - np.array([0, 0, -dist3*tan(gam)]).T
            DubinsAirplaneSolution['psi_e'] = theta-pi/2
            DubinsAirplaneSolution['lamda_e'] = 1
            DubinsAirplaneSolution['k_e']   = 0
            # hyperplane H_s: switch from first to second spiral
            DubinsAirplaneSolution['w_s'] = zi - np.array([0, 0, -dist2*tan(gam)]).T
            DubinsAirplaneSolution['q_s'] = np.array([cos(chii), sin(chii), 0]).T
            # hyperplane H_si: switch from second spiral to straight line
            DubinsAirplaneSolution['w_si'] = w1
            DubinsAirplaneSolution['q_si'] = q1
            # hyperplane H_l: switch from straight-line to end spiral
            DubinsAirplaneSolution['w_l'] = w2
            DubinsAirplaneSolution['q_l'] = q1
            # hyperplane H_e: end of Dubins path
            DubinsAirplaneSolution['w_e'] = ze
            DubinsAirplaneSolution['q_e'] = np.dot(rotz(anglend), np.array([1,0,0]).T)
```

```python
elif idx == 4: # left-right-straight-left
    ell = np.linalg.norm(cle[0:2]-ci[0:2],ord=2)
    theta = atan2(cle[1] - ci[1], cle[0] - ci[0])
    theta2 = theta - pi/2 + asin(2*R/ell)
    dist1 = R*fmod(2*pi+fmod(theta2,2*pi) - fmod(chii-pi/2,2*pi),2*pi)
    dist2 = R*psii
    dist3 = R*fmod(2*pi+fmod(theta2+pi,2*pi) - fmod(anglend+pi/2,2*pi),2*pi)
    w1 = ci + R*np.dot(rotz(theta2),e1.T).T + np.array([0, 0, -(dist1+dist2)*tan(gam)]).T
    w2 = cle + R*np.dot(rotz(theta2+pi),e1.T).T - np.array([0, 0, -dist3*tan(gam)]).T
    q1 = (w2-w1)/np.linalg.norm(w2-w1,ord=2)

    # start spiral
    DubinsAirplaneSolution['c_s'] = cls
    DubinsAirplaneSolution['psi_s'] = anglstart+pi/2
    DubinsAirplaneSolution['lamda_s'] = -1
    DubinsAirplaneSolution['k_s'] = 0
    # intermediate-start spiral
    DubinsAirplaneSolution['c_si'] = ci + np.array([0, 0, -dist2*tan(gam)]).T
    DubinsAirplaneSolution['psi_si'] = chii-pi/2
    DubinsAirplaneSolution['lamda_si'] = 1
    DubinsAirplaneSolution['k_si']   = 0
    # end spiral
    DubinsAirplaneSolution['c_e'] = cle - np.array([0, 0, -dist3*tan(gam)]).T
    DubinsAirplaneSolution['psi_e'] = theta2+pi
    DubinsAirplaneSolution['lamda_e'] = -1
    DubinsAirplaneSolution['k_e'] = 0
    # hyperplane H_s: switch from first to second spiral
    DubinsAirplaneSolution['w_s'] = zi - np.array([0, 0, -dist2*tan(gam)]).T
    DubinsAirplaneSolution['q_s'] = np.array([cos(chii), sin(chii), 0]).T
    # hyperplane H_si: switch from second spiral to straight line
    DubinsAirplaneSolution['w_si'] = w1
    DubinsAirplaneSolution['q_si'] = q1
    # hyperplane H_l: switch from straight-line to end spiral
    DubinsAirplaneSolution['w_l'] = w2
```

```
        DubinsAirplaneSolution['q_l'] = q1
        # hyperplane H_e: end of Dubins path
        DubinsAirplaneSolution['w_e'] = ze
        DubinsAirplaneSolution['q_e'] = np.dot(rotz(anglend), np.array([1, 0, 0]).T)

elif DubinsAirplaneSolution['case'] == 3:
    if idx == 1: # right-straight-left-right
        # path specific calculations
        ell = np.linalg.norm(ci[0:2] - crs[0:2],ord=2)
        theta = atan2(ci[1] - crs[1], ci[0] - crs[0])
        theta2 = theta-pi/2 + asin(2*R/ell)
        dist1 = R*fmod(2*pi+fmod(theta2,2*pi) - fmod(anglstart-pi/2,2*pi),2*pi)
        dist2 = R*fmod(2*pi+fmod(theta2+pi,2*pi)-fmod(chii+pi/2,2*pi),2*pi)
        dist3 = fabs(R_min*psii)
        w1 = crs + R*np.dot(rotz(theta2),e1.T).T + np.array([0, 0, -dist1*tan(gam)]).T
        w2 = ci + R*np.dot(rotz(theta2+pi),e1.T).T - np.array([0, 0, -(dist2+dist3)*tan(gam)]).T
        q1 = (w2-w1)/np.linalg.norm(w2-w1,ord=2)
        # start spiral
        DubinsAirplaneSolution['c_s']   = crs
        DubinsAirplaneSolution['psi_s'] = anglstart-pi/2
        DubinsAirplaneSolution['lamda_s'] = 1
        DubinsAirplaneSolution['k_s']   = 0
        # intermediate-end spiral
        DubinsAirplaneSolution['c_ei'] = ci - np.array([0, 0, -(dist2+dist3)*tan(gam)]).T
        DubinsAirplaneSolution['psi_ei'] = theta2+pi
        DubinsAirplaneSolution['lamda_ei'] = -1
        DubinsAirplaneSolution['k_ei'] = 0
        # end spiral
        DubinsAirplaneSolution['c_e'] = cre - np.array([0, 0, -dist3*tan(gam)]).T
        DubinsAirplaneSolution['psi_e'] = anglend-pi/2-psii
        DubinsAirplaneSolution['lamda_e'] = 1
        DubinsAirplaneSolution['k_e'] = 0
        # hyperplane H_s: switch from first to second spiral
        DubinsAirplaneSolution['w_s'] = w1
```

```python
    DubinsAirplaneSolution['q_s'] = q1
    # hyperplane H_l: switch from straight-line to intermediate spiral
    DubinsAirplaneSolution['w_l'] = w2
    DubinsAirplaneSolution['q_l'] = q1
    # hyperplane H_ei: switch from intermediate spiral to
    # end spiral
    DubinsAirplaneSolution['w_ei']  = zi - np.array([0, 0, -dist3*tan(gam)]).T
    DubinsAirplaneSolution['q_ei']  = np.array([cos(chii), sin(chii), 0]).T
    # hyperplane H_e: end of Dubins path
    DubinsAirplaneSolution['w_e'] = ze
    DubinsAirplaneSolution['q_e'] = np.dot(rotz(anglend),np.array([1,0,0]).T)

elif idx == 2: # right-straight-right-left
    # path specific calculations
    theta = atan2(ci[1] - crs[1], ci[0] - crs[0])
    dist1 = R*fmod(2*pi+fmod(theta-pi/2,2*pi) - fmod(anglstart-pi/2,2*pi),2*pi)
    dist2 = R*fmod(2*pi+fmod(chii-pi/2,2*pi) - fmod(theta-pi/2,2*pi),2*pi)
    dist3 = fabs(R_min*psii)
    w1 = crs + R*np.dot(rotz(theta-pi/2),e1.T).T + np.array([0, 0, -dist1*tan(gam)]).T
    w2 = ci  + R*np.dot(rotz(theta-pi/2),e1.T).T - np.array([0, 0, -(dist2+dist3)*tan(gam)]).T
    q1 = (w2-w1)/np.linalg.norm(w2-w1,ord=2)
    # start spiral
    DubinsAirplaneSolution['c_s'] = crs
    DubinsAirplaneSolution['psi_s'] = anglstart-pi/2
    DubinsAirplaneSolution['lamda_s'] = 1
    DubinsAirplaneSolution['k_s'] = 0
    # intermediate-end spiral
    DubinsAirplaneSolution['c_ei'] = ci - np.array([0, 0, -(dist2+dist3)*tan(gam)]).T
    DubinsAirplaneSolution['psi_ei'] = theta - pi/2
    DubinsAirplaneSolution['lamda_ei'] = 1
    DubinsAirplaneSolution['k_ei'] = 0
    # end spiral
    DubinsAirplaneSolution['c_e'] = cle - np.array([0, 0, -dist3*tan(gam)]).T
    DubinsAirplaneSolution['psi_e'] = anglend+pi/2+psii
```

```
            DubinsAirplaneSolution['lamda_e'] = -1
            DubinsAirplaneSolution['k_e'] = 0
            # hyperplane H_s: switch from first to second spiral
            DubinsAirplaneSolution['w_s'] = w1
            DubinsAirplaneSolution['q_s'] = q1
            # hyperplane H_l: switch from straight-line to intermediate spiral
            DubinsAirplaneSolution['w_l'] = w2
            DubinsAirplaneSolution['q_l'] = q1
            # hyperplane H_ei: switch from intermediate spiral to
            # end spiral
            DubinsAirplaneSolution['w_ei'] = zi - np.array([0, 0, -dist3*tan(gam)]).T
            DubinsAirplaneSolution['q_ei'] = np.array([cos(chii), sin(chii), 0]).T
            # hyperplane H_e: end of Dubins path
            DubinsAirplaneSolution['w_e'] = ze
            DubinsAirplaneSolution['q_e'] = np.dot(rotz(anglend), np.array([1, 0, 0]).T)
        elif idx == 3: # left-straight-left-right
            # path specific calculations
            theta = atan2(ci[1]-cls[1],ci[0]-cls[0])
            dist1 = R*fmod(2*pi-fmod(theta+pi/2,2*pi)+fmod(anglstart+pi/2,2*pi),2*pi)
            dist2 = R*fmod(2*pi-fmod(chii+pi/2,2*pi)+fmod(theta+pi/2,2*pi),2*pi)
            dist3 = fabs(R_min*psii)
            w1 = cls + DubinsAirplaneSolution['R']*np.dot(rotz(theta+pi/2),e1.T).T + np.array([0, 0, -
dist1*tan(gam)]).T
            w2 = ci + DubinsAirplaneSolution['R']*np.dot(rotz(theta+pi/2),e1.T).T - np.array([0, 0, -
(dist2+dist3)*tan(gam)]).T
            q1 = (w2-w1)/np.linalg.norm(w2-w1,ord=2) # direction of line

            # start spiral
            DubinsAirplaneSolution['c_s'] = cls
            DubinsAirplaneSolution['psi_s'] = anglstart+pi/2
            DubinsAirplaneSolution['lamda_s'] = -1
            DubinsAirplaneSolution['k_s'] = 0
            # intermediate-end spiral
            DubinsAirplaneSolution['c_ei'] = ci - np.array([0, 0, -(dist2+dist3)*tan(gam)]).T
```

```
        DubinsAirplaneSolution['psi_ei'] = theta+pi/2
        DubinsAirplaneSolution['lamda_ei'] = -1
        DubinsAirplaneSolution['k_ei'] = 0
        # end spiral
        DubinsAirplaneSolution['c_e'] = cre - np.array([0, 0, -dist3*tan(gam)]).T
        DubinsAirplaneSolution['psi_e'] = anglend-pi/2-psii
        DubinsAirplaneSolution['lamda_e'] = 1
        DubinsAirplaneSolution['k_e'] = 0
        # hyperplane H_s: switch from first to second spiral
        DubinsAirplaneSolution['w_s'] = w1
        DubinsAirplaneSolution['q_s'] = q1
        # hyperplane H_l: switch from straight-line to intermediate spiral
        DubinsAirplaneSolution['w_l'] = w2
        DubinsAirplaneSolution['q_l'] = q1
        # hyperplane H_ei: switch from intermediate spiral to
        # end spiral
        DubinsAirplaneSolution['w_ei'] = zi - np.array([0, 0, -dist3*tan(gam)]).T
        DubinsAirplaneSolution['q_ei'] = np.array([cos(chii), sin(chii), 0]).T
        # hyperplane H_e: end of Dubins path
        DubinsAirplaneSolution['w_e'] = ze
        DubinsAirplaneSolution['q_e'] = np.dot(rotz(anglend),np.array([1,0,0]).T)

    elif idx == 4: # left-straight-right-left

        # path specific calculations
        ell = np.linalg.norm(ci[0:2] - cls[0:2],ord=2)
        theta = atan2( ci[1] - cls[1], ci[0] - cls[0])
        theta2 = acos(2*R/ell)
        dist1 = R*fmod(2*pi-fmod(theta+theta2,2*pi) + fmod(anglstart+pi/2,2*pi),2*pi)
        dist2 = R*fmod(2*pi-fmod(theta+theta2-pi,2*pi)+fmod(chii-pi/2,2*pi),2*pi)
        dist3 = fabs(R_min*psii)
        w1 = cls + R*np.dot(rotz(theta+theta2),e1.T).T + np.array([0, 0, -dist1*tan(gam)]).T
        w2 = ci + R*np.dot(rotz(-pi+theta+theta2),e1.T).T - np.array([0, 0, -(dist2+dist3)*tan(gam)]).T
        q1 = (w2-w1)/np.linalg.norm(w2-w1,ord=2)
```

```
        # start spiral
        DubinsAirplaneSolution['c_s'] = cls
        DubinsAirplaneSolution['psi_s'] = anglstart+pi/2
        DubinsAirplaneSolution['lamda_s'] = -1
        DubinsAirplaneSolution['k_s'] = 0
        # intermediate-end spiral
        DubinsAirplaneSolution['c_ei'] = ci - np.array([0, 0, -(dist2+dist3)*tan(gam)]).T
        DubinsAirplaneSolution['psi_ei'] = fmod(theta+theta2-pi,2*pi)
        DubinsAirplaneSolution['lamda_ei'] = 1
        DubinsAirplaneSolution['k_ei'] = 0
        # end spiral
        DubinsAirplaneSolution['c_e'] = cle - np.array([0, 0, -dist3*tan(gam)]).T
        DubinsAirplaneSolution['psi_e'] = anglend+pi/2+psii
        DubinsAirplaneSolution['lamda_e'] = -1
        DubinsAirplaneSolution['k_e'] = 0
        # hyperplane H_s: switch from first to second spiral
        DubinsAirplaneSolution['w_s'] = w1
        DubinsAirplaneSolution['q_s'] = q1
        # hyperplane H_l: switch from straight-line to intermediate spiral
        DubinsAirplaneSolution['w_l'] = w2
        DubinsAirplaneSolution['q_l'] = q1
        # hyperplane H_ei: switch from intermediate spiral to
        # end spiral
        DubinsAirplaneSolution['w_ei'] = zi - np.array([0, 0, -dist3*tan(gam)]).T
        DubinsAirplaneSolution['q_ei'] = np.array([cos(chii), sin(chii), 0]).T
        # hyperplane H_e: end of Dubins path
        DubinsAirplaneSolution['w_e'] = ze
        DubinsAirplaneSolution['q_e'] = np.dot(rotz(anglend), np.array([1, 0, 0]).T)

if DubinsAirplaneSolution['case'] == 4:
    print '### Not Implemented Case'

return DubinsAirplaneSolution
```

```
def drawline(w1=None, q1=None, w2=None, q2=None, step=None):
    # extract line path
    r = w1
    # propagate line until cross half plane
    s = 0

    NrNc = r.shape
    if len(NrNc) == 1:
        NrNc_ind = NrNc[0]
        last_col = r[:]
    else:
        NrNc_ind = NrNc[1]
        last_col = r[:,NrNc[1]-1]

    r.shape = (3,1)
    while np.dot( (last_col - w2).T,q2 ) <= 0:
        s = s + step
        w1.shape = (3,1)
        q1.shape = (3,1)
        new_col = w1+s*q1
        new_col.shape = (3,1)
        r = np.hstack( (r,  new_col) )
        NrNc = r.shape
        if len(NrNc) == 1:
            NrNc_ind = NrNc[0]
            last_col = r[:]
        else:
            NrNc_ind = NrNc[1]
            last_col = r[:,NrNc[1]-1]

    return r

def drawspiral(R=None, gam=None, c=None, psi=None, lam=None, k=None, w=None, q=None, step=None):
```

```
# extract spiral path
r = np.zeros((1,1))
r = c.T + R*np.array( [cos(psi), sin(psi), 0] ).T
r = r.T
# determine number of required crossings of half plane
NrNc = r.shape
if len(NrNc) ==1 :
    NrNc_ind = NrNc[0]
    halfplane = np.dot( (r[0:2]-w[0:2].T),q[0:2] )
else:
    NrNc_ind = NrNc[1]
    halfplane = np.dot( (r[0:2,NrNc_ind-1]-w[0:2].T),q[0:2] )


if (halfplane > 0).all() :
    required_crossing = 2 * ( k + 1 )
else:
    required_crossing = 2 * k + 1

# propagate spiral until cross half plane the right number of times
s = 0
r.shape = (3,1)
while ( required_crossing > 0 ) or ( (halfplane <= 0).all() ):
    s = s +step
    new_col = (c + R * np.array( [ cos(lam*s+psi), sin(lam*s+psi), -s*tan(gam)] ).T )
    new_col.shape = (3,1)
    r = np.hstack( (r, new_col) )

    NrNc = r.shape
    if len(NrNc)==1 :
        NrNc_ind = NrNc[0]
        if np.sign( halfplane ) != np.sign( np.dot((r[0:2]-w[0:2].T),q[0:2]) ):
            halfplane = np.dot( ( r[0:2] - w[0:2].T ), q[0:2] )
            required_crossing = required_crossing - 1
```

```
        else:
            NrNc_ind = NrNc[1]
            if np.sign(halfplane) != np.sign( np.dot( (r[0:2,NrNc_ind-1]-w[0:2].T ), q[0:2]) ):
                halfplane = np.dot( (r[0:2,NrNc_ind-1] - w[0:2] ).T, q[0:2] )
                required_crossing = required_crossing - 1
    return r

def ExtractDubinsAirplanePath(DubinsAirplaneSolutions=None):
    # Extract the Dubins Airplane Solution in vector form

    step = 0.01

    if DubinsAirplaneSolutions['case'] == 1: # spiral - line - spiral
        r1 = drawspiral(DubinsAirplaneSolutions['R'],DubinsAirplaneSolutions['gamma'],
DubinsAirplaneSolutions['c_s'], DubinsAirplaneSolutions['psi_s'], DubinsAirplaneSolutions['lamda_s'],
DubinsAirplaneSolutions['k_s'], DubinsAirplaneSolutions['w_s'], DubinsAirplaneSolutions['q_s'],step)
        r2 = drawline(DubinsAirplaneSolutions['w_s'], DubinsAirplaneSolutions['q_s'],
DubinsAirplaneSolutions['w_l'], DubinsAirplaneSolutions['q_l'], step)
        r3 = drawspiral(DubinsAirplaneSolutions['R'], DubinsAirplaneSolutions['gamma'],
DubinsAirplaneSolutions['c_e'], DubinsAirplaneSolutions['psi_e'], DubinsAirplaneSolutions['lamda_e'],
DubinsAirplaneSolutions['k_e'], DubinsAirplaneSolutions['w_e'], DubinsAirplaneSolutions['q_e'], step)
        path = r1
        r = np.hstack((r1,r2))
        r = np.hstack((r,r3))
    elif DubinsAirplaneSolutions['case'] == 2: # spiral - spiral - line -spiral
        r1 =
drawspiral(DubinsAirplaneSolutions['R'],DubinsAirplaneSolutions['gamma'],DubinsAirplaneSolutions['c_s'],DubinsAirpla
neSolutions['psi_s'],DubinsAirplaneSolutions['lamda_s'],DubinsAirplaneSolutions['k_s'],DubinsAirplaneSolutions['w_s'
],DubinsAirplaneSolutions['q_s'],step)
        r2 =
drawspiral(DubinsAirplaneSolutions['R'],DubinsAirplaneSolutions['gamma'],DubinsAirplaneSolutions['c_si'],DubinsAirpl
aneSolutions['psi_si'],DubinsAirplaneSolutions['lamda_si'],DubinsAirplaneSolutions['k_si'],DubinsAirplaneSolutions['
w_si'],DubinsAirplaneSolutions['q_si'],step)
```

```
        r3 =
drawline(DubinsAirplaneSolutions['w_si'],DubinsAirplaneSolutions['q_si'],DubinsAirplaneSolutions['w_l'],DubinsAirpla
neSolutions['q_l'],step)
        r4 =
drawspiral(DubinsAirplaneSolutions['R'],DubinsAirplaneSolutions['gamma'],DubinsAirplaneSolutions['c_e'],DubinsAirpla
neSolutions['psi_e'],DubinsAirplaneSolutions['lamda_e'],DubinsAirplaneSolutions['k_e'],DubinsAirplaneSolutions['w_e'
],DubinsAirplaneSolutions['q_e'],step)
        path = r1
        r = np.hstack((r1,r2))
        r = np.hstack((r,r3))
        r = np.hstack((r,r4))

    elif DubinsAirplaneSolutions['case'] == 3: # spiral - line - spiral - spiral
        r1 =
drawspiral(DubinsAirplaneSolutions['R'],DubinsAirplaneSolutions['gamma'],DubinsAirplaneSolutions['c_s'],DubinsAirpla
neSolutions['psi_s'],DubinsAirplaneSolutions['lamda_s'],DubinsAirplaneSolutions['k_s'],DubinsAirplaneSolutions['w_s'
],DubinsAirplaneSolutions['q_s'],step)
        r2 =
drawline(DubinsAirplaneSolutions['w_s'],DubinsAirplaneSolutions['q_s'],DubinsAirplaneSolutions['w_l'],DubinsAirplane
Solutions['q_l'],step)
        r3 =
drawspiral(DubinsAirplaneSolutions['R'],DubinsAirplaneSolutions['gamma'],DubinsAirplaneSolutions['c_ei'],DubinsAirpl
aneSolutions['psi_ei'],DubinsAirplaneSolutions['lamda_ei'],DubinsAirplaneSolutions['k_ei'],DubinsAirplaneSolutions['
w_ei'],DubinsAirplaneSolutions['q_ei'],step)
        r4 =
drawspiral(DubinsAirplaneSolutions['R'],DubinsAirplaneSolutions['gamma'],DubinsAirplaneSolutions['c_e'],DubinsAirpla
neSolutions['psi_e'],DubinsAirplaneSolutions['lamda_e'],DubinsAirplaneSolutions['k_e'],DubinsAirplaneSolutions['w_e'
],DubinsAirplaneSolutions['q_e'],step)
        path = r1
        r = np.hstack((r1,r2))
        r = np.hstack((r,r3))
        r = np.hstack((r,r4))

    return r
```

```python
def PrintSolutionAgainstMATLAB(DubinsAirplaneSolution=None):
    print 'DubinsAirplaneSolution = '
    print '\n'
    print 'case:\t\t ' + str(DubinsAirplaneSolution['case'])
    print 'p_s:\t\t' + str(DubinsAirplaneSolution['p_s']) + '\n'
    print 'angl_s:\t\t' + str(DubinsAirplaneSolution['angl_s']) + '\n'
    print 'p_e:\t\t' + str(DubinsAirplaneSolution['p_e']) + '\n'
    print 'R:\t\t' + str(DubinsAirplaneSolution['R'])
    print 'gamma:\t\t' + str(DubinsAirplaneSolution['gamma'])
    print 'L:\t\t' + str(DubinsAirplaneSolution['L'])
    print 'c_s:\t\t' + str(DubinsAirplaneSolution['c_s']) + '\n'
    print 'psi_s:\t\t' + str(DubinsAirplaneSolution['psi_s']) + '\n'
    print 'lamda_s:\t\t' + str(DubinsAirplaneSolution['lamda_s'])
    print 'lamda_si:\t\t' + str(DubinsAirplaneSolution['lamda_si'])
    print 'k_s:\t\t' + str(DubinsAirplaneSolution['k_s'])
    print 'c_ei:\t\t' + str(DubinsAirplaneSolution['c_ei'].T) + '\n'
    print 'c_si:\t\t' + str(DubinsAirplaneSolution['c_si'].T) + '\n'
    print 'psi_ei:\t\t' + str(DubinsAirplaneSolution['psi_ei'])
    print 'lamda_e:\t\t' + str(DubinsAirplaneSolution['lamda_e'])
    print 'k_e:\t\t' + str(DubinsAirplaneSolution['k_e'])
    print 'w_s:\t\t' + str(DubinsAirplaneSolution['w_s']) + '\n'
    print 'q_s:\t\t' + str(DubinsAirplaneSolution['q_s']) + '\n'
    print 'w_si:\t\t' + str(DubinsAirplaneSolution['w_si'].T) + '\n'
    print 'q_si:\t\t' + str(DubinsAirplaneSolution['q_si'].T) + '\n'
    print 'w_l:\t\t' + str(DubinsAirplaneSolution['w_l']) + '\n'
    print 'q_l:\t\t' + str(DubinsAirplaneSolution['q_l']) + '\n'
    print 'w_ei:\t\t' + str(DubinsAirplaneSolution['w_ei'].T) + '\n'
    print 'q_ei:\t\t' + str(DubinsAirplaneSolution['q_ei'].T) + '\n'
    print 'w_e:\t\t' + str(DubinsAirplaneSolution['w_e']) + '\n'
    print 'q_e:\t\t' + str(DubinsAirplaneSolution['q_e']) + '\n'
    print 'angl_e:\t\t' + str(DubinsAirplaneSolution['angl_e']) + '\n'
```

Elementary-helping functions: **ElementaryFunctions.py**

```
#       __DUBINSAIRPLANEFUNCTIONS__
#       A small set of helping functions
#
#       Authors:
#       Kostas Alexis (konstantinos.alexis@mavt.ethz.ch)

import __builtin__

def max(a, b):
    # just a simple max function between 2 values
    if a >=b:
        return a
    else:
        return b

def min(a, b=0, nargout=0):
    # just a simple min function between 2 values
    if a <= b:
        return a
    else:
        return b
```

Plotting tools file: **PlottingTools.py**

```
#       __DUBINSAIRPLANEFUNCTIONS__
#       A small set of helping functions
#
```

```
#       Authors:
#       Kostas Alexis (konstantinos.alexis@mavt.ethz.ch)

import __builtin__

def max(a, b):
    # just a simple max function between 2 values
    if a >=b:
        return a
    else:
        return b

def min(a, b=0, nargout=0):
    # just a simple min function between 2 values
    if a <= b:
        return a
    else:
        return b
```

Proudly powered by Weebly

Collision-free Navigation

Collision-free navigation is a fundamental required capability of robots such that they can operate and function as autonomous reliable systems. With the perception systems becoming everyday more capable and especially since real-time 3D perception of the environment became

Figure 1: A hawk flying through two trees in a forest. Image from a video by Smithsonian Channel - https://youtu.be/HYGz32iv1vw

possible, motion planning for obstacle-free flight became a critical loop of aerial robotics. Eventually, aerial robotics are desired to be able to express the navigational agility in cluttered environments observed in nature. Figure 1 presents a hawk flying through two trees. It corresponds to a long-

standing paradigm of what an aerial robot should -conceptually- be able to do in the near future.

Collision-free navigation is a field on its own. Here we will briefly overview the basic problem formulation and examine one of its possible solutions. In particular, we will present the Rapidly-exploring

Random

Tree method.

Kinodynamic Motion Planning Problem

Given a dynamic system described by the Ordinary Differential Equation (ODE):

$$\frac{dx}{dt} = f(x,u), \; x(0) = x_0$$
$$x : \mathbb{R}_+ \to X \subset \mathbb{R}^d, u : \mathbb{R}_+ \to U \subset \mathbb{R}^m, f : \mathbb{R}^d \times \mathbb{R}^m \to \mathbb{R}^d$$

where x is the state trajectory, u a bounded measurable control signal and f is a Lipschitz function. Now further let:

$$X_{Obs} \in \mathbb{R}^d, \; (x_0 \notin X_{Obs})$$
$$X_{Goal} \in \mathbb{R}^d$$

corresponding to the set of obstacles (characterizing the obstacle region) and the goal set (characterizing the goal region). The goal of the kinodynamic motion planning problem is to find a control signal ***u*** such that the solution of the ODE describing the vehicle dynamics satisfies that all state trajectories ***x*** do not intersect the obstacle set X_{Obs} and there exists a time instant such that ***x*** goes through the desired goal set X_{Goal}. If no such solution exists, then the motion planning solution should return failure.

It is highlighted that even a simple version of this problem is PSPACE-hard.

Rapidly-exploring Random Tree algorithm

One particularly successful approach to the problem of finding admissible collision-free paths (even though not optimal) is based on the concept of incremental sampling-based algorithms for motion planning. Specifically, the method of **R**apidly-exploring **R**andom **T**ree (RRT) relies on the idea of exploring a tree of *feasible* trajectories of the system via *random sampling*. Below, the main steps of this algorithm are presented.

Video: Visualization of the RRT steps to find an admissible solution.

RRT Algorithm:

1. $V = \{x_{init}\}, E = 0$

2. i = 0
3. **while** i < N **do**
    1. x_{rand} = **Sample**(i)
    2. (V,E) = **Extend**((V,E), x_{rand})

Extend Procedure:

1. $x_{nearest}$ = **Nearest**(V, x_{rand})
2. xnew = **Steer**($x_{nearest}$, x_{rand})
3. If **ObstacleFree**($x_{nearest}$, x_{new}) then
    1. V = V∪{x_{new}}, E=E∪{($x_{nearest}$,x_{new})}

- The **Sample** method samples obstacle-free vehicle configurations
- The **Steer** method uses a Boundary Value Solver (BVS) that reflects the vehicle model and finds point-to-point trajectories
- The **ObstacleFree** method evaluates if a sampled trajectory is collision-free or not.

Completeness guarantees

- Probabilistic completeness: The probability of finding a solution, if one exists, approaches to one (1) as the number of samples approaches infinity

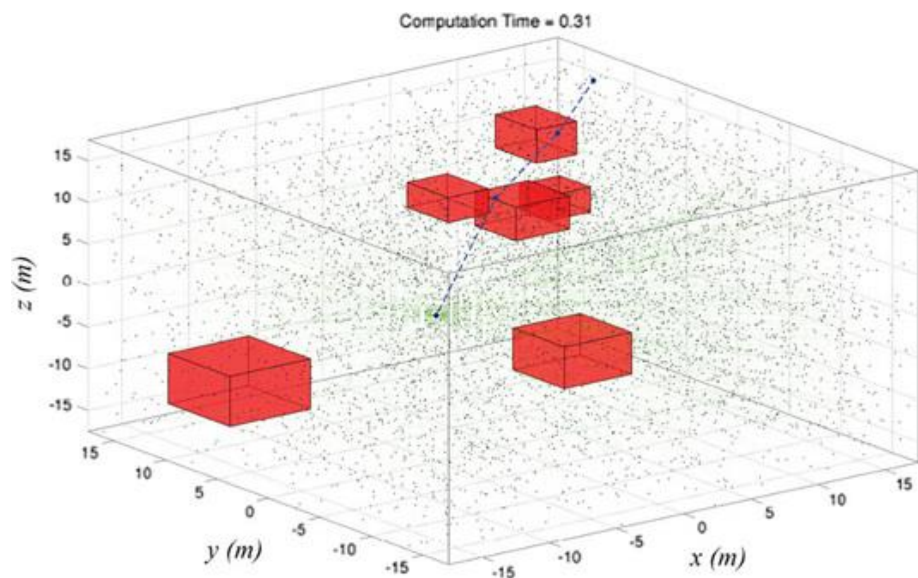
Note that the algorithm does not terminate if no solution exists. Overall the RRT method is very efficient in online problems and amenable to real-time computation. **The algorithm will find a solution if one exists, but there is no guarantee that the optimal solution will be found!**

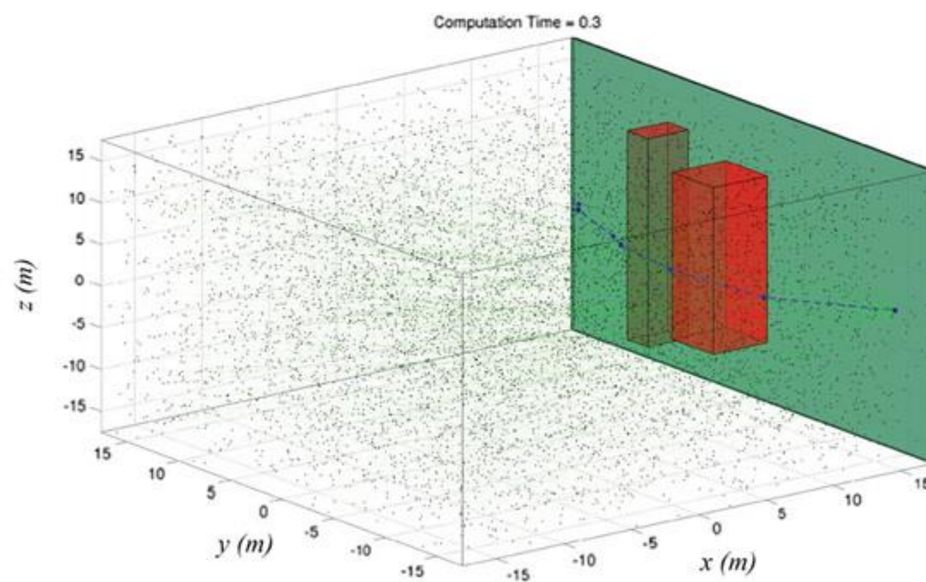
Optimal Collision-free Navigation

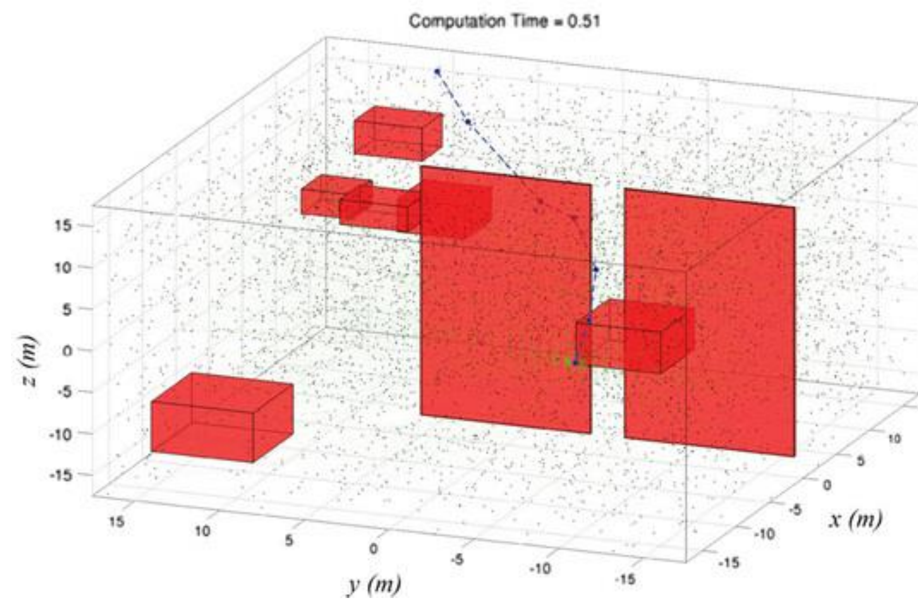
Although RRT is not optimal, one variation of it, RRT* is proven to be asymptotically optimal. Details on that algorithm can be found in the following contribution:

Karaman Sertac, and Emilio Frazzoli, ***"Sampling-based algorithms for optimal motion planning",*** The International Journal of Robotics Research 30.7 (2011): 846-894.

While indicative solutions are presented in Figure 2.


Computation Time = 0.31
z (m)
y (m)
x (m)


Computation Time = 0.3
z (m)
y (m)
x (m)


Computation Time = 0.51
z (m)
y (m)
x (m)

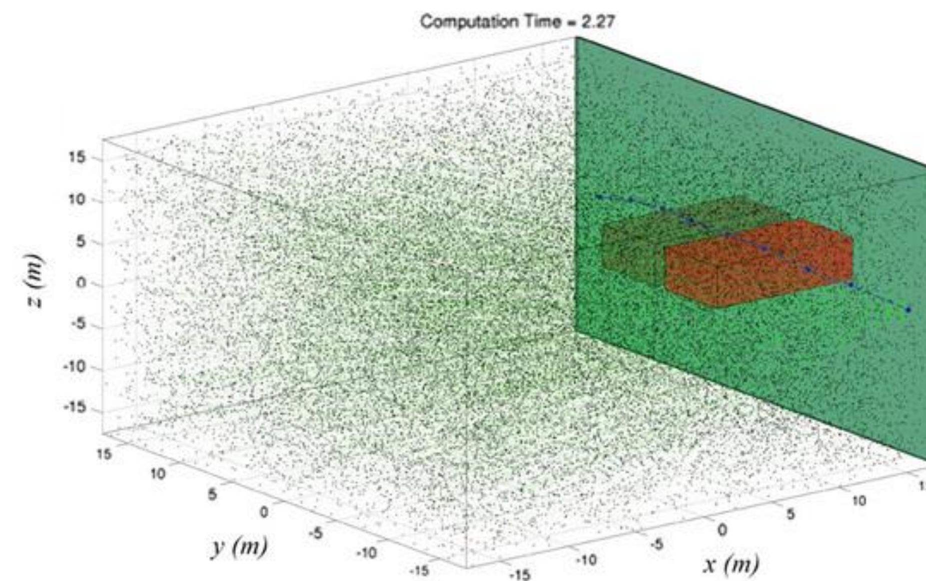

Computation Time = 2.27
z (m)
y (m)
x (m)

Figure 2: Indicative RRT* solutions. The results are taken from the following paper: K. Alexis, G. Darivianakis, M. Burri, and R. Siegwart, "**Aerial robotic contact-based inspection: planning and control**", Autonomous Robots, Springer US, DOI: 10.1007/s10514-015-9485-5, ISSN: 0929-5593, http://dx.doi.org/10.1007/s10514-015-9485-5

Simulation Tools

Simulations is a critical tool for aerial robotics research. It allows us to test an idea safely, predict the system behavior, revisit our concepts and tune the algorithms up to sufficient levels given that good models have been derived. A wide variety of simulation tools exist. Within this section we present MATLAB and Python Simpy as well as a dedicated aerial robotics simulator - RotorS. More specifically, this section contains the following subsections:

1. Simulations with SimPy
2. The RotorS Integrated Simulation Solution

Vehicle Dynamics Simulation using SimPy

A Quick Introduction to SimPy for ODE Simulations

Below, the basic concepts of SimPy - and how they apply to the problem of simulating Ordinary Differential Equations -will be overviewed.

1. Installation

Execute the following steps:

- Download SimPy
- Extract the archive, open a terminal there and type:

```
$ python setup.py install
```

You can now optionally the basic verification steps that SimPy's offers:

```
$ python -c "import simpy; simpy.test()"
```

2. Characterizing a system using Ordinary Differential Equations

A dynamic system, say a mass-spring system, can be characterized by the relation between its state variables and its inputs. This relation is expressed in the form of differential equations (and in that case "Ordinary Differential Equations"). To derive the ODE representation, we rely on laws of physics with the level of accuracy and fidelity we consider appropriate for the specific task at hand.

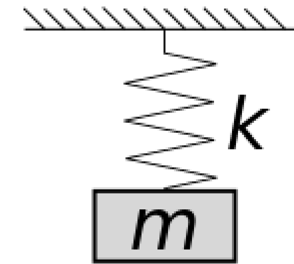


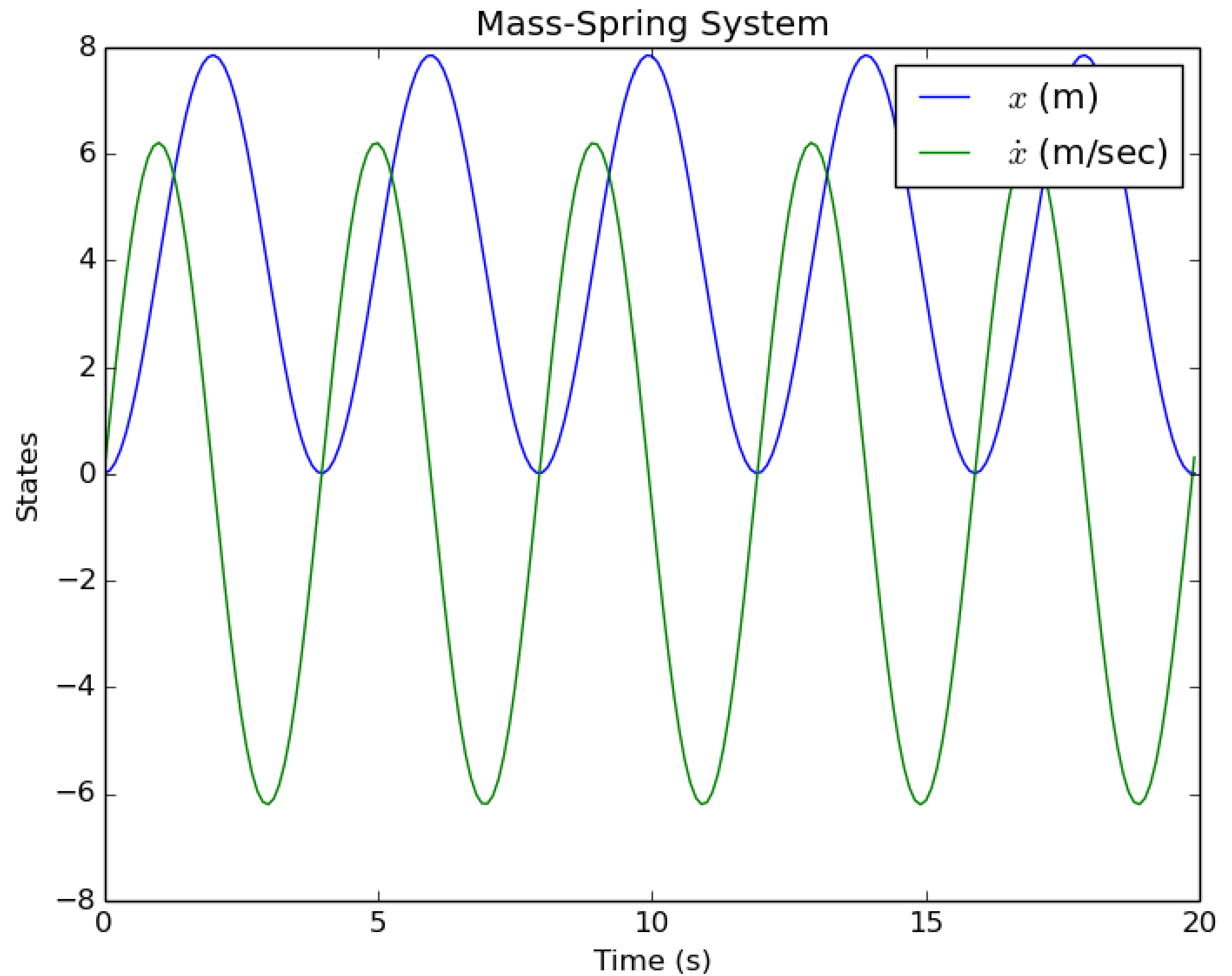
Figure 1: Sprin-mass system.

The specifc system (shown in Figure 1) can be captured by the following secord order Ordinary Differential Equation (ODE):

$$m\ddot{x} = -kx + mg \Rightarrow$$
$$\ddot{x} = -\frac{kx}{m} + g$$

Simulating such a system with the help of SimPy is particularly simple as shown in the code section below.

```
from scipy.integrate
import odeint
from numpy import *
import
matplotlib.pyplot as
plt

def
MassSpring(state,t):
  x = state[0] #
position
  xd = state[1] #
velocity

  # variables
  k = -5 # Nm
  m = 2 # kg
  g = 9.81 # m/s^2

  # mass
acceleration
  xdd = ((k*x)/m) +
g

  return [xd, xdd]

state0 = [0.0, 0.0]
t = arange(0.0,
20.0, 0.1)

state =
odeint(MassSpring,
state0, t)

plt.plot(t, state)
plt.xlabel('Time
(s)')
plt.ylabel('States')
plt.title('Mass-
Spring System')
```



The aforementioned system is particularly simple as it is expressed using linear ODEs. A slightly more difficult case arises, when one has to deal with nonlinear systems. Take for example the case of the Lorenz Attractor, a well known nonlinear system actively used to illustrate concepts of the chaos theory. Its state equations are:

$$\dot{x} = \sigma(y - x)$$
$$\dot{y} = (\rho - z)x - y$$
$$\dot{z} = xy - \beta z$$

The parameters σ,ρ,β are dominating the response of the system and if ones sets them as:

$$[\sigma, \rho, \beta] = [10, 28, 8/3]$$

Then the system exhibits chaotic behavior as shown in the simulation case study below.

```python
from scipy.integrate import odeint
from numpy import *
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def Lorenz(state,t):
  x = state[0]
  y = state[1]
  z = state[2]

  # set the constants
  sigma = 10.0
  rho = 28.0
  beta = 8.0/3.0

  xd = sigma * (y-x)
  yd = (rho-z)*x - y
  zd = x*y - beta*z

  return [xd, yd, zd]

state0 = [2.0, 3.0, 4.0]
t = arange(0.0, 30.0, 0.01)

state = odeint(Lorenz, state0, t)

fig = plt.figure()
ax = fig.gca(projection='3D')
ax.plot(state[:,0],state[:,1],state[:,2])
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
fig.show()
raw_input()
```

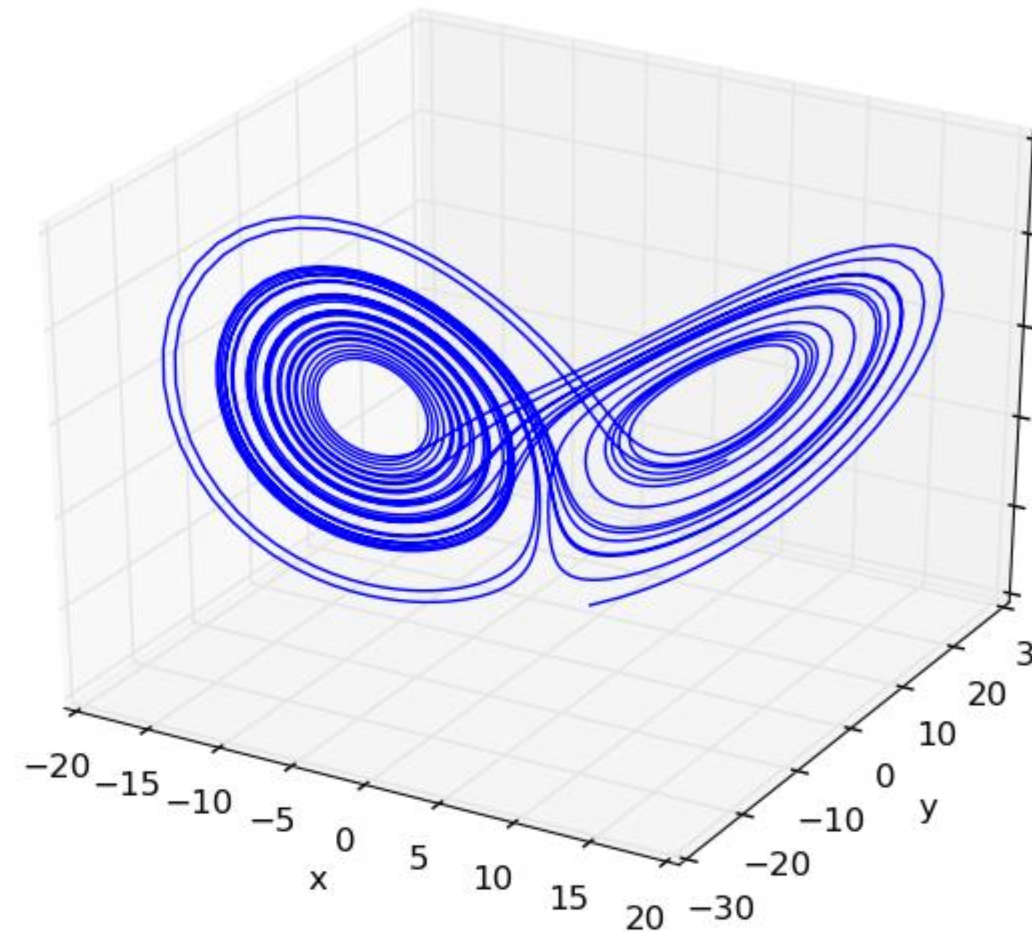


Figure 3: Lorenz Attractor expression chaotic behavior.

Simulation of a Quadrotor Aerial Robot

The classes/methods below implement the simulation of the vehicle dynamics of a quadrotor aerial robot. The system is considered to be captured using rigid body modeling and the four thrust vectors attached to it, while its self-response is only characterized by its mass and the cross-inertia terms. Within that framework, scipy is employed for the forward integration purposes of the state update, while direct calculations and linear algebra are employed for the moment/forces calculations as well as the coordinate system transformations.

File: **main.py**

```
from quadrotor_sims_params import *

if __name__ == "__main__":
    fly_quadrotor()
```

File: **quadrotor_sims_params.py**

```
#       __QUADROTORSIMSPARAMS__
#       This file implements the sim parameters
#       simple quadrotor simulation tool
#
#       Largely based on the work of https://github.com/nikhilkalige

from quadrotor_dynamics import QuadrotorDynamics
import numpy as np
import random
import time
import matplotlib.pyplot as plt

TURNS = 3

class SimulationParams(object):
    def __init__(self):
        self.mass = 1
        self.Ixx = 0.0053
```

```python
        self.length = 0.2
        self.Bup = 21.58
        self.Bdown = 3.92
        self.Cpmax = np.pi * 1800/180
        self.Cn = TURNS
        self.gravity = 9.81

    def get_acceleration(self, p0, p3):
        ap = {
            'acc': (-self.mass * self.length * (self.Bup - p0) / (4 * self.Ixx)),
            'start': (self.mass * self.length * (self.Bup - self.Bdown) / (4 * self.Ixx)),
            'coast': 0,
            'stop': (-self.mass * self.length * (self.Bup - self.Bdown) / (4 * self.Ixx)),
            'recover': (self.mass * self.length * (self.Bup - p3) / (4 * self.Ixx)),
        }
        return ap

    def get_initial_parameters(self):
        p0 = p3 = 0.9 * self.Bup
        p1 = p4 = 0.1
        acc_start = self.get_acceleration(p0, p3)['start']
        p2 = (2 * np.pi * self.Cn / self.Cpmax) - (self.Cpmax / acc_start)
        return [p0, p1, p2, p3, p4]

    def get_sections(self, parameters):
        sections = np.zeros(5, dtype='object')
        [p0, p1, p2, p3, p4] = parameters

        ap = self.get_acceleration(p0, p3)

        T2 = (self.Cpmax - p1 * ap['acc']) / ap['start']
        T4 = -(self.Cpmax + p4 * ap['recover']) / ap['stop']

        aq = 0
```

```python
        ar = 0

        sections[0] = (self.mass * p0, [ap['acc'], aq, ar], p1)

        temp = self.mass * self.Bup - 2 * abs(ap['start']) * self.Ixx / self.length
        sections[1] = (temp, [ap['start'], aq, ar], T2)

        sections[2] = (self.mass * self.Bdown, [ap['coast'], aq, ar], p2)

        temp = self.mass * self.Bup - 2 * abs(ap['stop']) * self.Ixx / self.length
        sections[3] = (temp, [ap['stop'], aq, ar], T4)

        sections[4] = (self.mass * p3, [ap['recover'], aq, ar], p4)
        return sections

def fly_quadrotor(params=None):
    gen = SimulationParams()
    quadrotor = QuadrotorDynamics()
    if not params:
        params = gen.get_initial_parameters()
    sections = gen.get_sections(params)
    state = quadrotor.update_state(sections)
    plt.plot(state)
    plt.show()
    raw_input()
```

File: **quadrotor_dynamics.py**

```python
#       __QUADROTORDYNAMICS__
#       This file implements the dynamics of a
#       simple quadrotor micro aerial vehicle
#
#       Largely based on the work of https://github.com/nikhilkalige

import numpy as np
from scipy.integrate import odeint
```

```python
class QuadrotorDynamics(object):

    def __init__(self, save_state=True, config={}):
        """
        Quadrotor Dynamics Parameters
        ----------
        save_state: Boolean
            Decides whether the state of the system should be saved
            and returned at the end
        """
        self.config = {
            # Define constants
            'gravity': 9.81,  # Earth gravity [m s^-2]
            # Define Vehicle Parameters
            'mass': 1,  # Mass [kg]
            'length': 0.2,  # Arm length [m]
            # Cross-Intertia [Ixx, Iyy, Izz]
            'inertia': np.array([0.0053, 0.0053, 0.0086]),  # [kg m^2]
            'thrustToDrag': 0.018  # thrust to drag constant [m]
        }

        self.save_state = save_state
        self._dt = 0.005  # Simulation Step
        self.config.update(config)
        self.initialize_state()

    def initialize_state(self):
        self.state = {
            # Position [x, y, z] of the quadrotor in inertial frame
            'position': np.zeros(3),
            # Velocity [dx/dt, dy/dt, dz/dt] of the quadrotor in inertial frame
            'velocity': np.zeros(3),
            # Euler angles [phi, theta, psi]
```

```python
            'orientation': np.zeros(3),
            # Angular velocity [p, q, r]
            'ang_velocity': np.zeros(3)
        }

    def motor_thrust(self, moments, total_thrust):
        """Compute Motor Thrusts

        Parameters
        ----------
        moments : numpy.array
            [Mp, Mq, Mr] moments
        total_thrust : float
            The total thrust generated by all motors

        Returns
        -------
        numpy.array
            The thrust generated by each motor [T1, T2, T3, T4]
        """
        [Mp, Mq, Mr] = moments
        thrust = np.zeros(4)
        tmp1add = total_thrust + Mr / self.config['thrustToDrag']
        tmp1sub = total_thrust - Mr / self.config['thrustToDrag']

        tmp2p = 2 * Mp / self.config['length']
        tmp2q = 2 * Mq / self.config['length']

        thrust[0] = tmp1add - tmp2q
        thrust[1] = tmp1sub + tmp2p
        thrust[2] = tmp1add + tmp2q
        thrust[3] = tmp1sub - tmp2p

        return thrust / 4.0
```

```python
def dt_eulerangles_to_angular_velocity(self, dtEuler, EulerAngles):
    """Euler angles derivatives TO angular velocities
    dtEuler = np.array([dphi/dt, dtheta/dt, dpsi/dt])
    """
    return np.dot(self.angular_rotation_matrix(EulerAngles), dtEuler)

def acceleration(self, thrusts, EulerAngles):
    """Compute the acceleration in inertial reference frame
    thrust = np.array([Motor1, .... Motor4])
    """
    force_z_body = np.sum(thrusts) / self.config['mass']
    rotation_matrix = self.rotation_matrix(EulerAngles)
    # print rotation_matrix
    force_body = np.array([0, 0, force_z_body])
    return np.dot(rotation_matrix, force_body) - np.array([0, 0, self.config['gravity']])

def angular_acceleration(self, omega, thrust):
    """Compute the angular acceleration in body frame
    omega = angular velocity :- np.array([p, q, r])
    """
    [t1, t2, t3, t4] = thrust
    thrust_matrix = np.array([self.config['length'] * (t2 - t4),
                              self.config['length'] * (t3 - t1),
                              self.config['thrustToDrag'] * (t1 - t2 + t3 - t4)])

    inverse_inertia = np.linalg.inv(self.inertia_matrix)
    p1 = np.dot(inverse_inertia, thrust_matrix)
    p2 = np.dot(inverse_inertia, omega)
    p3 = np.dot(self.inertia_matrix, omega)
    cross = np.cross(p2, p3)
    return p1 - cross

def angular_velocity_to_dt_eulerangles(self, omega, EulerAngles):
```

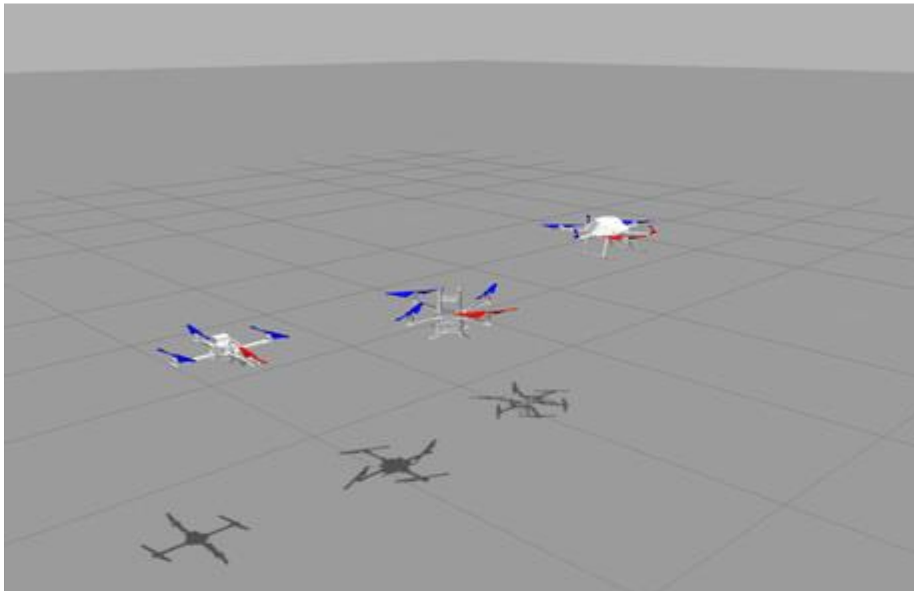
```python
    """Angular Velocity TO Euler Angles
    omega = angular velocity :- np.array([p, q, r])
    """
    rotation_matrix = np.linalg.inv(self.angular_rotation_matrix(EulerAngles))
    return np.dot(rotation_matrix, omega)

def moments(self, ref_acc, angular_vel):
    """Compute the moments

    Parameters
    ----------
    ref_acc : numpy.array
        The desired angular acceleration that the system should achieve. This
        should be of form [dp/dt, dq/dt, dr/dt]
    angular_vel : numpy.array
        The current angular velocity of the system. This
        should be of form [p, q, r]

    Returns
    -------
    numpy.array
        The desired moments of the system
    """
    inverse_inertia = np.linalg.inv(self.inertia_matrix)
    p1 = np.dot(inverse_inertia, angular_vel)
    p2 = np.dot(self.inertia_matrix, angular_vel)
    cross = np.cross(p1, p2)
    value = ref_acc + cross
    return np.dot(self.inertia_matrix, value)

def angular_rotation_matrix(self, EulerAngles):
    """Rotation matix for Angular Velocity <-> Euler Angles Conversion
    Use inverse of the matrix to convert from angular velocity to euler rates
    """
```

```
        [phi, theta, psi] = EulerAngles
        cphi = np.cos(phi)
        sphi = np.sin(phi)
        cthe = np.cos(theta)
        sthe = np.sin(theta)
        cpsi = np.cos(psi)
        spsi = np.sin(psi)
        RotMatAngV = np.array([ [1, 0,             -sthe],
                                [0, cphi,  cthe * sphi],
                                [0, -spsi, cthe * cphi]
                                ])
        return RotMatAngV

    def rotation_matrix(self, EulerAngles):
        [phi, theta, psi] = EulerAngles
        cphi = np.cos(phi)
        sphi = np.sin(phi)
        cthe = np.cos(theta)
        sthe = np.sin(theta)
        cpsi = np.cos(psi)
        spsi = np.sin(psi)

        RotMat = np.array([[cthe * cpsi, sphi * sthe * cpsi - cphi * spsi, cphi * sthe * cpsi + sphi * spsi],
                           [cthe * spsi, sphi * sthe * spsi + cphi * cpsi, cphi * sthe * spsi - sphi * cpsi],
                           [-sthe,       cthe * sphi,                      cthe * cphi]])
        return RotMat

    @property
    def inertia_matrix(self):
        return np.diag(self.config['inertia'])

    def update_state(self, piecewise_args):
        """Run the state update equations for self._dt seconds
        Update the current state of the system. It runs the model and updates
```

```
its state to self._dt seconds.

Parameters
----------
piecewise_args : array
    It contains the parameters that are needed to run each section
    of the flight. It is an array of tuples.
    [(ct1, da1, t1), (ct2, da2, t2), ..., (ctn, dan, tn)]

    ct : float
        The collective thrust generated by all motors
    da : numpy.array
        The desired angular acceleration that the system should achieve.
        This should be of form [dp/dt, dq/dt, dr/dt]
    t: float
        Time for which this section should run and should be atleast twice
        self._dt
"""
if self.save_state:
    overall_time = 0
    for section in piecewise_args:
        overall_time += section[2]

    overall_length = len(np.arange(0, overall_time, self._dt)) - (len(piecewise_args) - 1)
    # Allocate space for storing state of all sections
    final_state = np.zeros([overall_length + 100, 12])
else:
    final_state = []

# Create variable to maintain state between integration steps
self._euler_dot = np.zeros(3)
index = 0

for section in piecewise_args:
```

```
            (total_thrust, desired_angular_acc, t) = section
            if t < (2 * self._dt):
                continue

            ts = np.arange(0, t, self._dt)
            state = np.concatenate((self.state['position'], self.state['velocity'],
                                    self.state['orientation'], self.state['ang_velocity']))
            output = odeint(self._integrator, state, ts, args=(total_thrust, desired_angular_acc))
            output_length = len(output)
            # State update
            [self.state['position'], self.state['velocity'],
             self.state['orientation'], self.state['ang_velocity']] = np.split(output[output_length - 1], 4)

            if self.save_state:
                # Final state update
                final_state[index:(index + output_length)] = output
                index = index + output_length - 1

        return final_state

    def _integrator(self, state, t, total_thrust, desired_angular_acc):
        """Callback function for scipy.integrate.odeint.
            At this point scipy is used to execute the forward integration

        Parameters
        ----------
        state : numpy.array
            System State: [x, y, z, xdot, ydot, zdot, phi, theta, psi, p, q, r]
        t : float
            Time

        Returns
        -------
        numpy.array
```

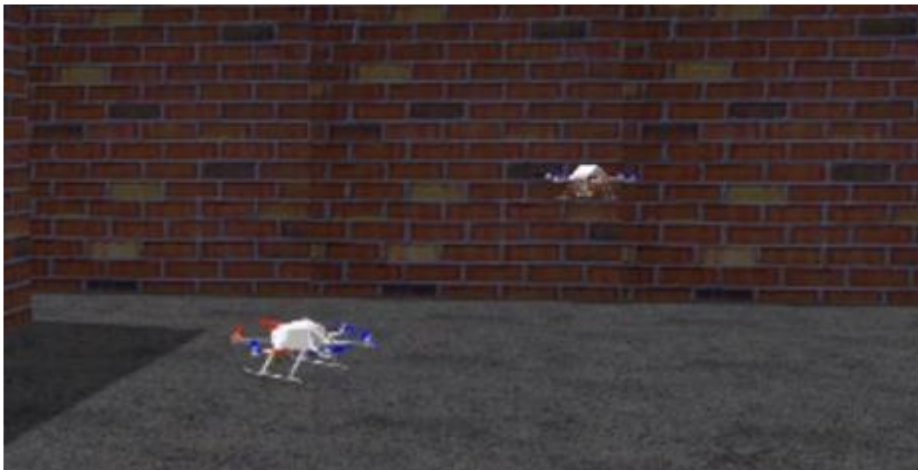
```
    Rates of the input state:
    [xdot, ydot, zdot, xddot, yddot, zddot, phidot, thetadot, psidot, pdot, qdot, rdot]
"""
# Position inertial frame [x, y, z]
pos = state[:3]
# Velocity inertial frame [x, y, z]
velocity = state[3:6]
euler = state[6:9]
# Angular velocity omega = [p, q, r]
omega = state[9:12]

# Derivative of euler angles [dphi/dt, dtheta/dt, dpsi/dt]
euler_dot = self._euler_dot

# omega = self.dt_eulerangles_to_angular_velocity(euler_dot, euler)
moments = self.moments(desired_angular_acc, omega)
thrusts = self.motor_thrust(moments, total_thrust)
# Acceleration in inertial frame
acc = self.acceleration(thrusts, euler)

omega_dot = self.angular_acceleration(omega, thrusts)
euler_dot = self.angular_velocity_to_dt_eulerangles(omega, euler)
self._euler_dot = euler_dot
# [velocity : acc : euler_dot : omega_dot]
# print 'Vel', velocity
# print 'Acc', acc
# print 'Euler dot', euler_dot
# print 'omdega dot', omega_dot
return np.concatenate((velocity, acc, euler_dot, omega_dot))
```



RotorS Simulator

RotorS is a MAV gazebo simulator developed by the Autonomous Systems Lab at ETH Zurich. It provides some multirotor models such as the AscTec Hummingbird, the AscTec Pelican, or the AscTec Firefly, but the simulator is not limited for the use with these multicopters. There are simulated sensors coming with the simulator such as an IMU, a generic odometry sensor, and the VI-Sensor, which can be mounted on the multirotor. This packages also contains some example controllers, basic worlds, a joystick interface, and example launch files. Below we provide the instructions necessary for getting started. See RotorS' wiki for more instructions and examples (https://github.com/ethz-asl/rotors_simulator/wiki).

Download the RotorS-ready Ubuntu VM



RotorS-ready Ubuntu VM

A RotorS-ready Virtual Machine has been prepared to give students easy access to the RotorS simulator. To use this Virtual Machine you will neeed:

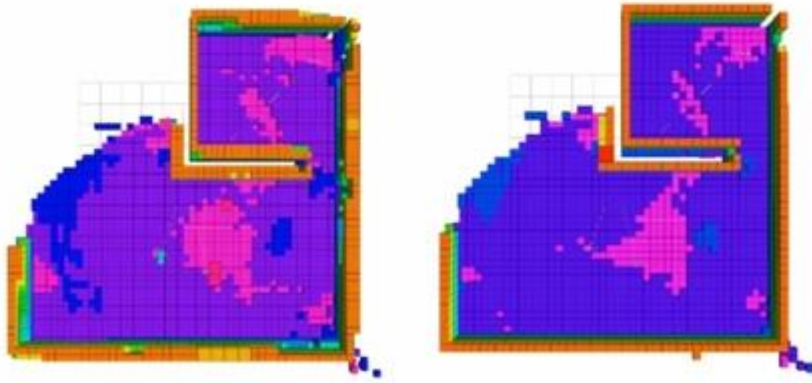
- A 64-bit Operating System with BIOS settings that support Virtualization
- VM Player software (available for free - link)
- 20GB of available HDD space (you will now download a <3GB file but it is set-up to support up to 20GB dynamic size allocation)
- 4GB RAM available to the Virtual Machine and up to 2GB of GPU memory assigned to the Virtual Machine. These settings are indicative and can be changed using the VMPlayer interface.

RotorS-ready Ubuntu VM info

- Ubuntu version: **Ubuntu 14.04.3 LTS**
- ROS version: **indigo**
- username: **rotors**
- password: **rotors**
- path for rotors: **/home/rotors/catkin_ws**

Other issues to consider:

- Ensure USB access to the VM if you want to use a gamepad
- ensure internet access to the VM



- You will need a system with GPU-based 3D Acceleration.

Why RotorS

- Open-source
- Supports models for high quality aerial robots actively used for research purposes
- Supports simulation of sensors including IMUs and Cameras
- The **Autonomous Robots Lab** at UNR employs identical physical aerial robots such as those simulated in the RotorS.

Getting Started

Installation Instructions

- To start using RotorS using the prepared Virtual Machine jump directly to "Basic Usage"
- Ideally, install ROS on a raw Ubuntu installation: to do so, follow the instructions below
- For any academic use of RotorS do not forget to **cite** its developers team originating from the Autonomous Systems Lab at ETH Zurich! Specifically cite the following BIBITEM:

```
> @incollection{RotorS_BookChapter,
> title = {RotorS - A Modular Gazebo MAV Simulator Framework},
> author = {Fadri Furrer, Michael Burri, Markus Achtelik, and Roland Siegwart},
>  year = {2015},
>  }
```

1. Install and initialize ROS indigo desktop full, additional ROS packages, catkin-tools, and wstool:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu `lsb_release -sc` main" >
/etc/apt/sources.list.d/ros-latest.list'
$ wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
$ sudo apt-get update
$ sudo apt-get install ros-indigo-desktop-full ros-indigo-joy ros-indigo-octomap-ros
python-wstool python-catkin-tools
$ sudo rosdep init
$ rosdep update
$ source /opt/ros/indigo/setup.bash
```

2. If you don't have ROS workspace yet you can do so by

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace  # initialize your catkin workspace
$ wstool init
```

Note *for setups with multiple workspaces please refer to the official documentation at http://docs.ros.org/independent/api/rosinstall/html/ by replacing `rosws` by `wstool`.*

3. Get the simulator and additional dependencies

```
$ cd ~/catkin_ws/src
$ git clone git@github.com:ethz-asl/rotors_simulator.git
$ git clone git@github.com:ethz-asl/mav_comm.git
```

Note *if you want to use `wstool` you can replace the above commands with `wstool set --git local_repo_name git@github.com:organization/repo_name.git`*

4. Build your workspace with `python_catkin_tools` (therefore you need `python_catkin_tools`)

```
$ cd ~/catkin_ws/
$ catkin init  # If you haven't done this before.
$ catkin build
```

5. Add sourcing to your `.bashrc` file

```
$ echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

Basic Usage

Basic Usage

Launch the simulator with a hex-rotor helicopter model, in our case, the AscTec Firefly.

```
$ roslaunch rotors_gazebo mav_empty_world.launch mav_name:=firefly
```

Note *The first run of gazebo might take considerably long, as it will download some models from an online database.*

The simulator starts by default in paused mode. To start it you can either

use the Gazebo GUI and press the play button

or you can send the following service call.

```
$ rosservice call gazebo/unpause_physics
```

There are some basic launch files where you can load the different multicopters with additional sensors. They can all be found in `~/catkin_ws/src/rotors_simulator/rotors_gazebo/launch`.

Getting the robot to fly

To let the multicopter fly you need to generate thrust with the rotors, this is achieved by sending commands to the multicopter, which make the rotors spin. There are currently a few ways to send commands to the multicopter, we will show one of them here. The rest is documented here in our Wiki. We will here also show how to write a stabilizing controller and how you can control the multicopter with a joystick.

Send direct motor commands

We will for now just send some constant motor velocities to the multicopter.

```
$ rostopic pub /firefly/command/motor_speed mav_msgs/Actuators '{angular_velocities: [100, 100, 100, 100, 100, 100]}'
```

Note *The size of the* `motor_speed` *array should be equal to the number of motors you have in your model of choice (e.g. 6 in the Firefly model).*

You should see (if you unpaused the simulator and you have a multicopter in it), that the rotors start spinning. The thrust generated by these motor velocities is not enough though to let the multicopter take off.

You can play with the numbers and will realize that the Firefly will take off with motor speeds of about 545 on each rotor. The multicopter is unstable though, since there is no controller running, if you just set the motor speeds.

Let the robot hover with ground truth odometry

You can let the helicopter hover with ground truth odometry (perfect state estimation), by launching:

```
$ roslaunch rotors_gazebo mav_hovering_example.launch mav_name:=firefly
```