3-D Mobility Mathematical Models

Design and Analysis of a 3-D Gauss-Markov Mobility Model for Highly Dynamic Airborne Networks

-The Basic 2D Gauss-Markov Algorithm

The Gauss-Markov mobility model is a relatively simple memory-based model with a single tuning parameter, alpha α, which determines the amount of memory and variability in node movement. In the traditional 2-dimensional implementation of the Gauss-Markov model, each mobile node is assigned an initial speed and direction, as well as an average speed and direction. At set intervals of time, a new speed and direction are calculated for each node, which follow the new course until the next time step. This cycle repeats through the duration of the simulation. The new speed and direction parameters are calculated as follows:

$$s_n = \alpha s_{n-1} + (1 - \alpha)\bar{s} + \sqrt{(1-\alpha^2)}s_{x_{n-1}}$$

$$d_n = \alpha d_{n-1} + (1 - \alpha)\bar{d} + \sqrt{(1-\alpha^2)}d_{x_{n-1}}$$

Where,

- α is the tuning parameter,
- $\bar{s}$ and $\bar{d}$ are the mean speed and direction parameters, respectively,
- $s_{x_{n-1}}$ and $d_{x_{n-1}}$ are random variables from a Gaussian (normal) distribution that give some randomness to the new speed and direction parameters.

Special Case: $\alpha = 0$

When α is zero, the model becomes memoryless; the new speed and direction are based completely upon the average speed and direction variables and the Gaussian random variables.

$$s_n = \bar{s} + s_{x_{n-1}}$$

$$d_n = \bar{d} + d_{x_{n-1}}$$

Special Case: $\alpha = 1$

When α is 1, movement becomes predictable, losing all randomness. The new direction and speed values are identical to the previous direction and speed values. In short, the node continues in a straight line.

$$s_n = s_{x_{n-1}}$$

$$d_n = d_{x_{n-1}}$$

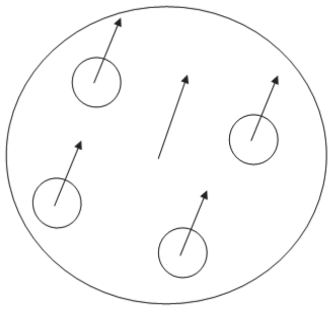
-Extending the Model to Three Dimensions

In this section, we discuss several methods to extend the basic Gauss-Markov 2-dimensional model to three dimensions. The first approach is to apply the Markov process to the x, y, and z axis of a 3-dimensional velocity vector. The velocity vector is computed as:

$$x_n = \alpha x_{n-1} + (1 - \alpha)\bar{x} + \sqrt{(1 - \alpha^2)}x_{x_{n-1}}$$
$$y_n = \alpha y_{n-1} + (1 - \alpha)\bar{y} + \sqrt{(1 - \alpha^2)}y_{x_{n-1}}$$
$$z_n = \alpha z_{n-1} + (1 - \alpha)\bar{z} + \sqrt{(1 - \alpha^2)}z_{x_{n-1}}$$

Aircraft flight can be more accurately modeled using a velocity variable combined with variables for both direction and pitch. In the second approach, we start with the speed and direction variables found in the 2-dimensional Gauss-Markov model, and add a third variable to track the vertical pitch of the mobile node with respect to the horizon as follows:

$$s_n = \alpha s_{n-1} + (1 - \alpha)\bar{s} + \sqrt{(1 - \alpha^2)}s_{x_{n-1}}$$
$$d_n = \alpha d_{n-1} + (1 - \alpha)\bar{d} + \sqrt{(1 - \alpha^2)}d_{x_{n-1}}$$
$$p_n = \alpha p_{n-1} + (1 - \alpha)\bar{p} + \sqrt{(1 - \alpha^2)}p_{x_{n-1}}$$



It is sufficient to model the aircraft movement itself using the Gauss-Markov algorithm, for which we assume the direction and pitch variables represent the actual angles at which the aircraft is moving. After calculating these variables, the algorithm must determine a new velocity vector and send that information to the ns-3 constant velocity helper, in which the new node location is calculated. Assuming the direction and pitch variables are given in radians, the velocity vector $\bar{v}$ is calculated as:

$$v_x = s_n \cos(d_n)\cos(p_n)$$
$$v_y = s_n \sin(d_n)\cos(p_n)$$
$$v_z = s_n \sin(p_n)$$

-RANDOM WALK 3D ALGORITHM

The Random Walk model implemented in ns-3 defines the movements of a node by the

- Random variables θ, that changes the direction in which the node moves
- Speed $v \epsilon [Vmin; Vmax]$ contained in a predefined speed interval.

This model has *two* modes:

1. *time mode*: Explicitly decides for how long a node will keep its current speed and direction before choosing new values
2. *distance mode*: Also decides the time before new values are chosen but based on the current speed of the node and the predefined distance value:

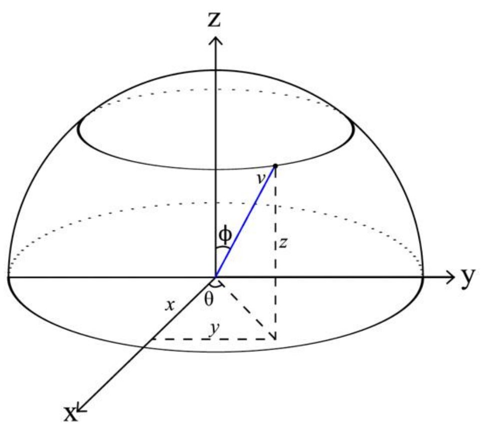


Figure 2: Visualization of coordinates in both cartesian and spherical systems.

$$time = \frac{distance}{speed}$$

If the mobile node reaches the boundary of the simulation before the resetting process is executed, the node simply bounces back by changing its direction in the corresponding axis and continues moving for the remaining time.

For example, if the x-axis is the boundary limit (equivalent to $y = 0$), then the direction in y of the velocity vector is inverted by multiplying it with -1. Similarly, if y-axis is the limit ($x = 0$) the x component of the velocity is inverted.

To enable this model to function in a three-dimensional world, a new random variable is introduced, $\phi \in [0, \pi]$. In addition to the speed and direction, now the node also selects a new pitch at every cycle. Instead of having lines as limits, now the model assumes planes. If the node reaches the limit $xy - plane$ for example, the z

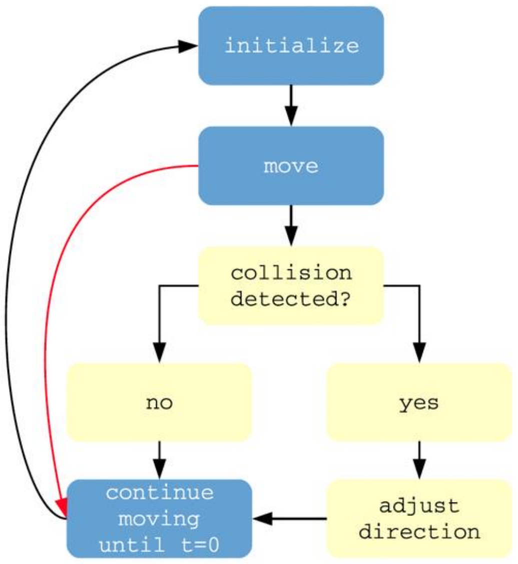


Figure 4: Simplified block diagram of collision avoidance.

component of the velocity vector is inverted to bounce the node back into the scenario S. Obstacles are assumed to be boxes and axis aligned, meaning their faces are parallel to the axis planes. The effect caused is similar to the bouncing velocity vector: if the node collides with an obstacle, it changes the corresponding component of the velocity, depending which side of the object was hit.

1. In each cycle of the Random-Walk model the node chooses new speed, direction, and pitch. These parameters basically define the ***velocity vector*** of the node.
2. Then it calculates the next position based on the $time$ it is supposed to move with those parameters.
3. If during this period any **obstacle is encountered**, the orthogonal component of the velocity vector is inverted as mentioned.
4. After the inversion it continues to move with the new values for the remaining time of the cycle, after which a new iteration takes place.

Algorithm 1: Random Walk 3D

```
Input: Boundary ($\mathbb{S}$), set of obstacles ($\mathbb{O}$), default
       distance ($default\_distance$) or $default\_time$,
       and Speed limits $[V_{min}, V_{max}]$
Output: List of Time and Position pairs
repeat
    if $mode = time$ then
        $default\_distance \leftarrow default\_time * v$;
    $\theta \leftarrow \mathcal{U}(0, 2\pi]$;
    $\phi \leftarrow \mathcal{U}[0, \pi]$;
    $v \leftarrow \mathcal{U}[V_{min}, V_{max}]$;
    $distance \leftarrow default\_distance$;
    $collides \leftarrow false$;
    for $O_i \in \mathbb{O}$ do
        if WillCollide($x, y, z, \theta, \phi, O_i$) then
            if $\Delta d >$ DistToObstacle($x, y, z, \theta, \phi, O_i$)
            then
                $\Delta d \leftarrow$ DistToObstacle($x, y, z, \theta, \phi, O_i$);
                $collides \leftarrow True$;
    if $collides$ then
        $Rebound(\Delta d)$;
    $x \leftarrow x + sin(\theta)cos(\phi)\Delta d$;
    $y \leftarrow y + sin(\theta)sin(\phi)\Delta d$;
    $z \leftarrow z + sin(\phi)\Delta d$;
until $simulation\ ends$;
```

-PSUEDOCODE: RANDOM WALK 3D

-RANDOM DIRECTION 3D ALGORITHM

First proposed with the objective of *eliminating the concentrated node density problem that occurs in the Random Waypoint Model.* In this model the nodes randomly select a point inside the limited area and keep consistent. move there with a certain speed.

Since the probability of choosing a point closer to the boundaries is lower than in the middle, the nodes in the scenario eventually end up concentrating in the center. The Random Direction model mitigates this problem by forcing the node to travel until it reaches the boundary of the delimited area. In the Random Direction model, three variables govern the trajectory of a mobile node:

1. Pause P
2. Direction θ
3. Speed v

Similar to the Random Walk model, these attributes are constrained in their limits.

1. First the node chooses a speed and direction in which it will move.
2. Then it continues moving until it reaches the boundary, standing by that position for P amount of time before resetting the parameters and beginning the cycle once again.
3. Intuitively, the node will only move again if the new direction points towards the middle of the limited area, otherwise it will remain in the paused state.

The movements in z-axis are allowed by the introduction of the pitch ϕ in the system. In the main loop of the algorithm the model selects the values for the random variables ϕ, θ, and v. Then it calculates the intersection of the new trajectory path of the node with each obstacle inside the scenario, and the scenario boundaries itself. The next position where the node will randomize the values again is the intersection with the smallest distance to travel. If an object is not in the path of the node, the intersection distance returns infinity. After the node reaches the destination, it pauses for a random amount of time, constrained within the interval $P \in [P_{min}, P_{max}]$.

```
Algorithm 2: Random Direction 3D
Input: Boundary ($\mathbb{S}$), set of obstacles ($\mathbb{O}$), Pause time
       limits ($P_{min}, P_{max}$) and Speed limits
       [$V_{min}, V_{max}$]
Output: List of Time and Position pairs
repeat
    $Pause(\mathcal{U}[P_{min}, P_{max}])$;
    $\theta \leftarrow \mathcal{U}(0, 2\pi]$;
    $\phi \leftarrow \mathcal{U}[0, \pi]$;
    $v \leftarrow \mathcal{U}[V_{min}, V_{max}]$;
    $\Delta d \leftarrow$ DistanceToBoundary$(x, y, z, \theta, \phi)$;
    for $O_i \in \mathbb{O}$ do
        if WillCollide$(x, y, z, \theta, \phi, O_i)$ then
            if $\Delta d >$ DistToObstacle$(x, y, z, \theta, \phi, O_i)$
            then
                $\Delta d \leftarrow$ DistToObstacle$(x, y, z, \theta, \phi, O_i)$;
    $\Delta t \leftarrow \Delta d / v$;
    $x \leftarrow x + sin(\theta)cos(\phi)v\Delta t$;
    $y \leftarrow y + sin(\theta)sin(\phi)v\Delta t$;
    $z \leftarrow z + sin(\phi)v\Delta t$;
until simulation ends;
```

-PSUEDOCODE: RANDOM DIRECTION 3D

Box 1:

0-180

Box 2:

181-391

Box 3:

392-581

Box 4:

581- (Mach-1)

-WEATHER CONDITIONS ON QUADCOPTER

Air temperature, wind speed, precipitation, and other atmospheric phenomena have been shown to adversely affect drone endurance, control, aerodynamics, airframe integrity, line-of-sight visibility, airspace monitoring, and sensors for navigation and collision avoidance.

The **starting sequence** is the most dangerous situation if you fly in strong winds. The ground-level turbulence of the drone is overlaid by that of the wind. The drone can be displaced or flipped over by the wind. Generally, the pilot must stand behind the drone facing the wind, so in the worst-case scenario, the drone could crash into your face.

References:

Broyles, Dan, Abdul Jabbar, and James PG Sterbenz. "Design and analysis of a 3–D gauss-markov mobility model for highly-dynamic airborne networks." In *Proceedings of the international telemetering conference (ITC), San Diego, CA*. 2010.

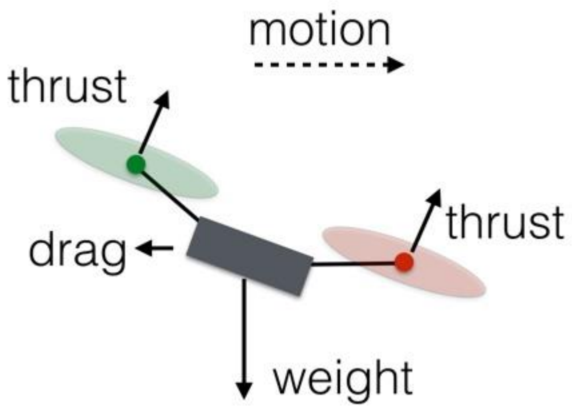
Regis, Paulo Alexandre, Suman Bhunia, and Shamik Sengupta. "Implementation of 3d obstacle compliant mobility models for uav networks in ns-3." In *Proceedings of the Workshop on ns-3*, pp. 124-131. 2016.

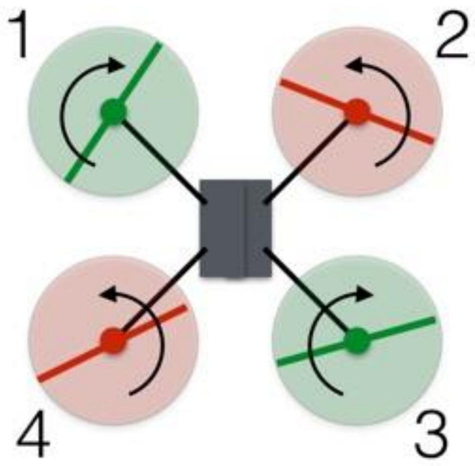
Turning Radius Factor along with Environmental Factors.

Forwards and Sideways Movement

There is no Difference between the Drone moving Forwards, Backwards, or any Direction Sideways. Every side of the Quad copter is a front side. When solving for how to move forward, we solve how to move sideways.

To fly forward, I need a forward component of thrust from the rotors. Here is a side view (with forces) of a drone moving at a constant speed.





How do we get the drone into this Position:
You could increase the rotation rate of rotors 3 and 4 (the rear ones) and decrease the rate of rotors 1 and 2. The total thrust force will remain equal to the weight, so the drone will stay at the same vertical level. Also, since one of the rear rotors is spinning counterclockwise and the other clockwise, the increased rotation of those rotors will still produce zero angular momentum. The same holds true for the front rotors, and so the drone does not rotate. However, the greater force in the back of the drone means it will tilt forward. Now a slight increase in thrust for all rotors will produce a net thrust force that has a component to balance the weight along with a forward motion component.

Controlling the Drone Using a computer:

Every movement change on the Drone requires a change to the spin rate of the Rotors. Doing that simply requires a controller that can increase or decrease the voltage to each motor.

If you have some type of computer control system, you can simply push a joystick with your thumb and let a computer handle all of that.

Turn Radius

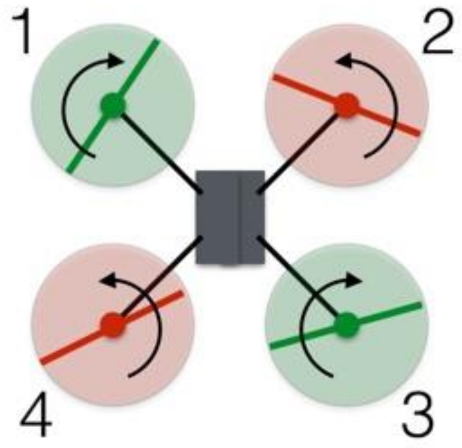
0-180 MPH

Fixed Wing Aircraft

Turning Rate of Drone:

Basic Physics behind Rotor Propulsion and Lift of Drone: Spinning blades push air down. Of course, all forces come in pairs, which means that as the rotor pushes down on the air, the air pushes up on the rotor. This is the basic idea behind lift, which comes down to controlling the upward and downward force. The faster the rotors spin, the greater the lift, and vice-versa.

Rotation (Turning):



In this Diagram, the Angular momentum of the Drone is Equal to 0 since there is an equal number of Rotors spinning in Counterclockwise as there are spinning Clockwise. To change the direction of the Drone, we must modify the angular momentum of the Drone. The equation of Angular Momentum can be found as follows:

$$L = mvr$$

Where:

- L is denoted as Angular Momentum
- m is denoted by mass
- v as Velocity
- r as Radius.

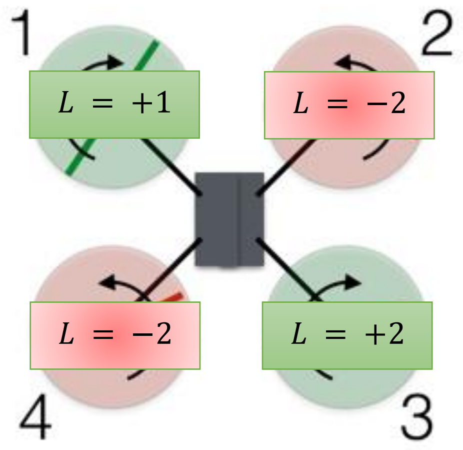
Or as:

$$L = I\,\omega$$

Where:

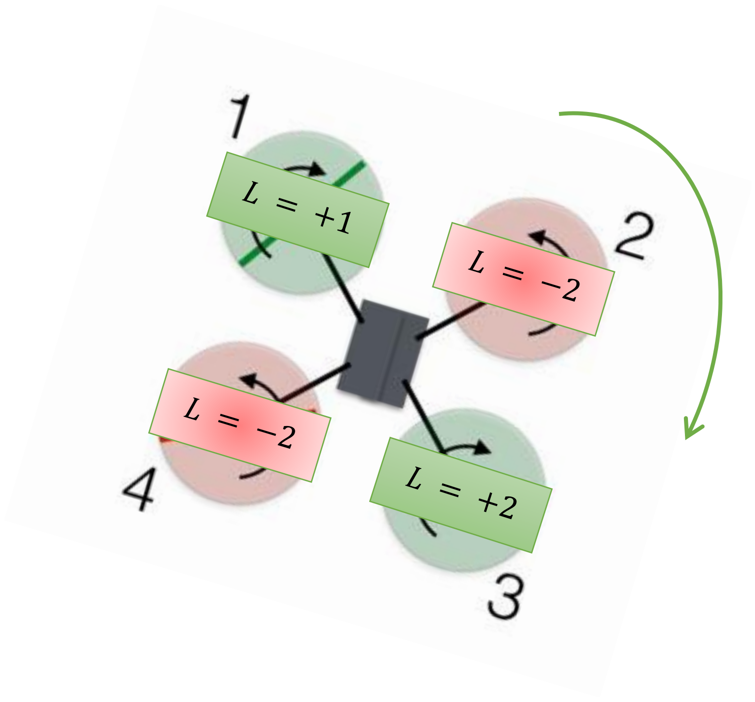
- I is denoted as the Moment of Inertia
- ω is denoted as the Angular Velocity

When it comes to our System, if there is no Torque being considered the Total Angular Momentum must remain constant which in this case is 0. If we were to want to rotate and turn, we would have to change the net Angular Momentum of the Rotors to have the Body of the Drone Compensate and restore the Net Angular Momentum of the System to 0. For example, Assume the Angular Momentums of each Rotor are as follows:

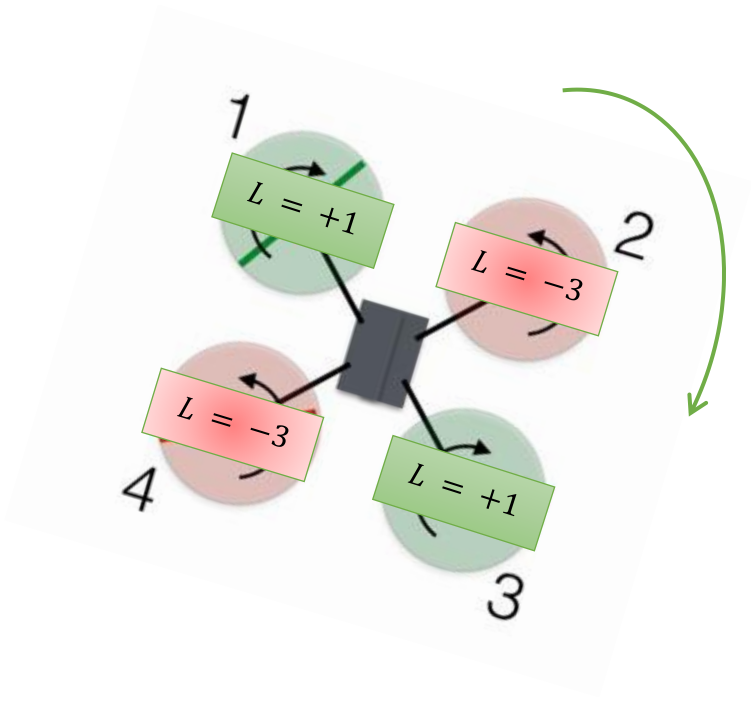


Therefore, $\Sigma_L = 0$. If we were to change the Angular Momentum of Rotor 1 to $+1$, This would cause $\Sigma_L = -1$, which cannot happen. Because of this, the body of the Drone Rotates Clockwise to have an angular momentum of $+1$ restoring the $\Sigma_L = 0$ of the System. And we have Rotation.

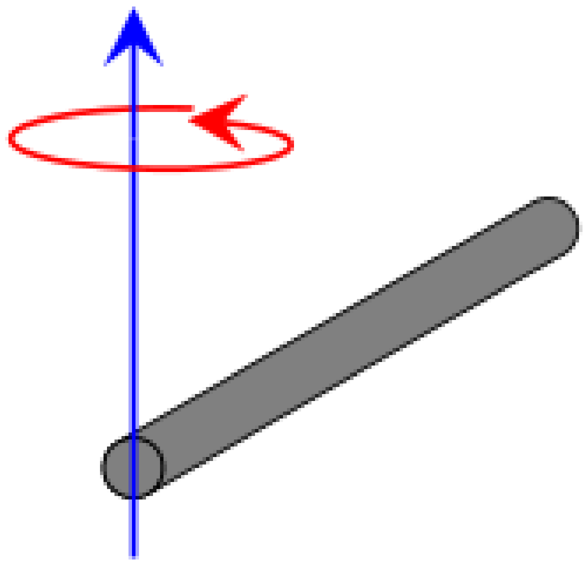
Side Wind Gusts on Drone.
New Attribute.

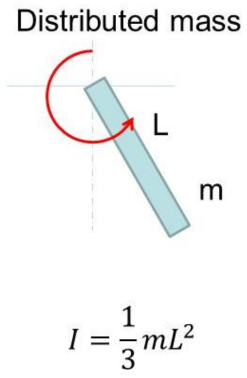


This raises another problem however, the decrease in angular momentum allowed for rotation but also decreased the Thrust from Rotor 1 and because of this the Forces pushing the Drone upward do not equal to the Downward force of Gravity. Because of this the drone descends. Along with this, the thrust of the rotors isn't balanced and therefore, the drone tips downward in the direction of rotor 1. There is a solution to this. To solve this problem, we must decrease for a pair of Rotors and increase by the same amount for the opposite directional rotors to keep the upward forces of the Drone equal to the Gravitational Force and for the Drone to remain hovering. For example, to turn clockwise as the previous example, we would do the following instead:

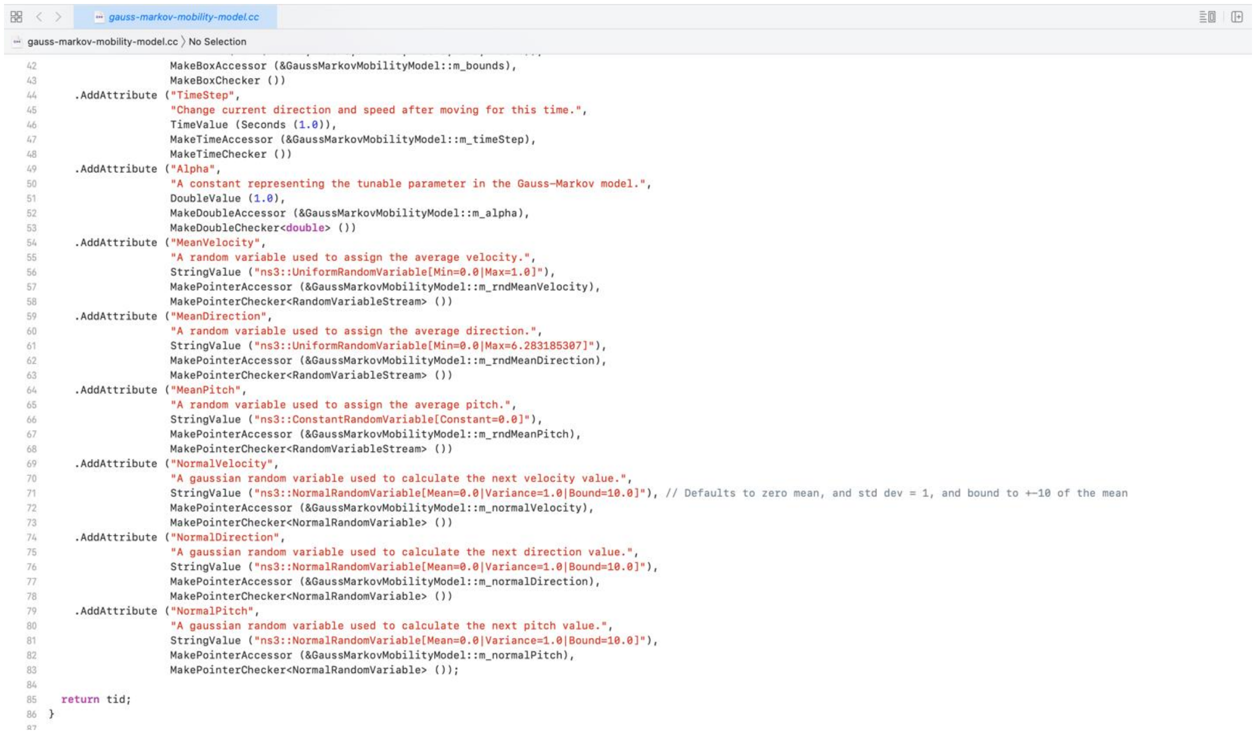


Moment of Inertia for Propeller Blades (In Progress)



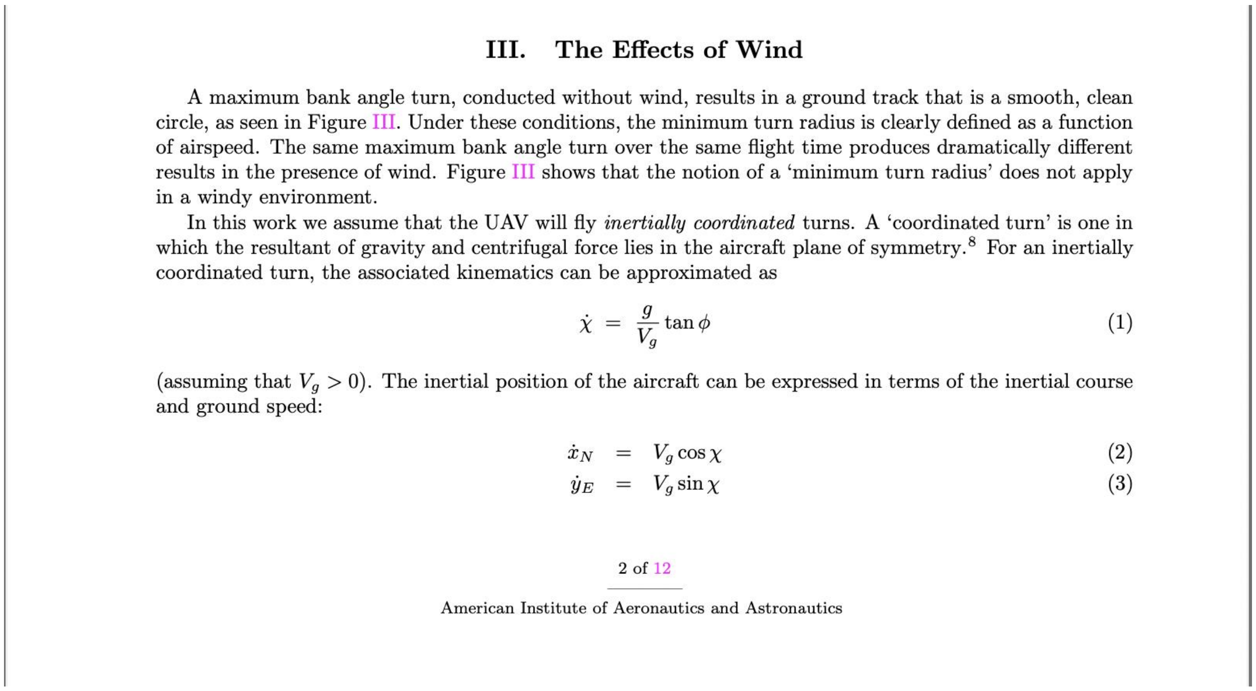


For a Propeller with many Blades, take sum of all Blades Moment of Inertia.


gauss-markov-mobility-model.cc
gauss-markov-mobility-model.cc 〉 No Selection

```
                   MakeBoxAccessor (&GaussMarkovMobilityModel::m_bounds),
                   MakeBoxChecker ())
    .AddAttribute ("TimeStep",
                   "Change current direction and speed after moving for this time.",
                   TimeValue (Seconds (1.0)),
                   MakeTimeAccessor (&GaussMarkovMobilityModel::m_timeStep),
                   MakeTimeChecker ())
    .AddAttribute ("Alpha",
                   "A constant representing the tunable parameter in the Gauss-Markov model.",
                   DoubleValue (1.0),
                   MakeDoubleAccessor (&GaussMarkovMobilityModel::m_alpha),
                   MakeDoubleChecker<double> ())
    .AddAttribute ("MeanVelocity",
                   "A random variable used to assign the average velocity.",
                   StringValue ("ns3::UniformRandomVariable[Min=0.0|Max=1.0]"),
                   MakePointerAccessor (&GaussMarkovMobilityModel::m_rndMeanVelocity),
                   MakePointerChecker<RandomVariableStream> ())
    .AddAttribute ("MeanDirection",
                   "A random variable used to assign the average direction.",
                   StringValue ("ns3::UniformRandomVariable[Min=0.0|Max=6.283185307]"),
                   MakePointerAccessor (&GaussMarkovMobilityModel::m_rndMeanDirection),
                   MakePointerChecker<RandomVariableStream> ())
    .AddAttribute ("MeanPitch",
                   "A random variable used to assign the average pitch.",
                   StringValue ("ns3::ConstantRandomVariable[Constant=0.0]"),
                   MakePointerAccessor (&GaussMarkovMobilityModel::m_rndMeanPitch),
                   MakePointerChecker<RandomVariableStream> ())
    .AddAttribute ("NormalVelocity",
                   "A gaussian random variable used to calculate the next velocity value.",
                   StringValue ("ns3::NormalRandomVariable[Mean=0.0|Variance=1.0|Bound=10.0]"), // Defaults to zero mean, and std dev = 1, and bound to +-10 of the mean
                   MakePointerAccessor (&GaussMarkovMobilityModel::m_normalVelocity),
                   MakePointerChecker<NormalRandomVariable> ())
    .AddAttribute ("NormalDirection",
                   "A gaussian random variable used to calculate the next direction value.",
                   StringValue ("ns3::NormalRandomVariable[Mean=0.0|Variance=1.0|Bound=10.0]"),
                   MakePointerAccessor (&GaussMarkovMobilityModel::m_normalDirection),
                   MakePointerChecker<NormalRandomVariable> ())
    .AddAttribute ("NormalPitch",
                   "A gaussian random variable used to calculate the next pitch value.",
                   StringValue ("ns3::NormalRandomVariable[Mean=0.0|Variance=1.0|Bound=10.0]"),
                   MakePointerAccessor (&GaussMarkovMobilityModel::m_normalPitch),
                   MakePointerChecker<NormalRandomVariable> ());

  return tid;
}
```



III. The Effects of Wind

A maximum bank angle turn, conducted without wind, results in a ground track that is a smooth, clean circle, as seen in Figure III. Under these conditions, the minimum turn radius is clearly defined as a function of airspeed. The same maximum bank angle turn over the same flight time produces dramatically different results in the presence of wind. Figure III shows that the notion of a 'minimum turn radius' does not apply in a windy environment.

In this work we assume that the UAV will fly *inertially coordinated* turns. A 'coordinated turn' is one in which the resultant of gravity and centrifugal force lies in the aircraft plane of symmetry.[8] For an inertially coordinated turn, the associated kinematics can be approximated as

$$\dot{\chi} = \frac{g}{V_g} \tan\phi \tag{1}$$

(assuming that $V_g > 0$). The inertial position of the aircraft can be expressed in terms of the inertial course and ground speed:

$$\dot{x}_N = V_g \cos\chi \tag{2}$$
$$\dot{y}_E = V_g \sin\chi \tag{3}$$

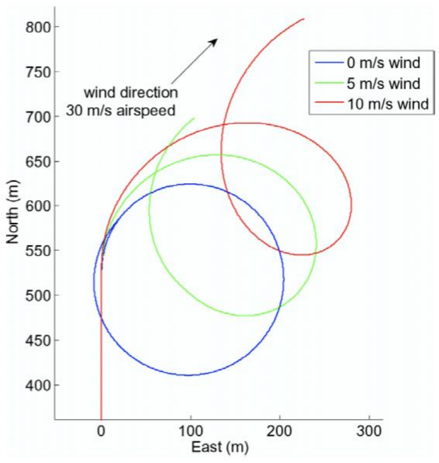


Figure 1. Open loop flight simulation. Wind is 0, 5, and 10 m/s from 225°.

1. Send Materials
2. References
3. Take details on Attribute Source Code
4. Determine a Code for our own Attributes in Detail
5. Find a volunteer to create and compile Attribute Code in C++.