# Optimized Trajectory Generation and Collision Avoidance using Nonlinear Model Predictive Control

Hardik Gupta

Department of Mechanical Engineering

A thesis submitted for the degree of

*Bachelor of Engineering*

December 2022

# Final Report

Submitted in partial fulfillment of the requirements of

BITS F421 Thesis

By

**Hardik Gupta**

**2018B1A40632P**

Under the supervision of

**Dr. Guillaume Sartoretti**, Assistant Professor

**Department of Mechanical Engineering**

NUS

National University
of Singapore

and co-supervision of

**Dr. Saket Verma,** Assistant professor

**Department of Mechanical Engineering**

# Acknowledgement

I would like to express my sincere gratitude to Prof. Saket Verma from Mechanical Engineering, Birla Institute of Technology and Science, Pilani and Prof. Guillaume Sartoretti from National University of Singapore, Singapore for their invaluable support and guidance during the course of our project titled "Optimized Trajectory Generation and Collision Avoidance using NonLinear Model Predictive Control".

Dr. Sartoretti's expertise in control engineering and his insights on the application of nonlinear model predictive control (NMPC) for collision avoidance were invaluable in the development and implementation of our NMPC algorithm. His keen knowledge pushed the project and shaped the direction and focus of our work.

I am deeply grateful to Dr. Saket Verma for his mentorship and support throughout the project. His guidance and feedback were essential in helping us to successfully complete this project. I would also like to express my appreciation for his continued support and encouragement throughout the project.

In addition, I would like to acknowledge the Head of the Department, Mechanical Engineering for providing the resources and infrastructure necessary to complete this project successfully. I am grateful to have had the opportunity to work with such knowledgeable and supportive mentors, and we look forward to applying the lessons learned from this project to future endeavors.

This report is compiled by Hardik Gupta.

December 2022

# Table of Contents

1. Terminology.......................................................................................... 05
2. Introduction.......................................................................................... 08
3. Literature Review.................................................................................. 09
4. Problem Formulation............................................................................ 13
5. Controller Formulation.......................................................................... 14
6. NonLinear Model Predictive Control Algorithm............................................ 22
7. Results................................................................................................ 32
8. Conclusion and Future Work................................................................... 34

# Terminology

1. Model predictive control (MPC): MPC is a control method that involves using a mathematical model of the system to predict its future behavior and generate control inputs that optimize a predefined objective.
2. Nonlinear systems: Nonlinear systems are systems whose behavior cannot be described by linear equations. Nonlinear systems may exhibit complex and unpredictable behavior, and they require special techniques for modeling and control. Nonlinear systems can be found in a wide range of applications, including mechanical, electrical, and biological systems. Examples of nonlinear systems include systems with nonlinear dynamics, such as pendulums and springs, and systems with nonlinear constraints, such as robots and autonomous vehicles operating in cluttered environments.
3. Optimization: Optimization refers to the process of finding the optimal solution to a problem, typically by minimizing or maximizing a predefined objective function subject to constraints. Optimization is a key component of MPC, as it is used to determine the control inputs that optimize the performance of the system.
4. Dynamic programming: Dynamic programming is an optimization technique that involves breaking down a complex problem into smaller subproblems and solving each subproblem independently. Dynamic programming is often used in NMPC to solve the optimization problem over a finite horizon.
5. Trajectory optimization: Trajectory optimization is the process of finding a trajectory that minimizes a predefined objective, such as energy consumption or time to reach a goal, subject to constraints such as dynamic feasibility and collision avoidance. Trajectory optimization is often used in NMPC to find the optimal control inputs for the system.
6. Receding horizon control: Receding horizon control is a control strategy that involves solving the optimization problem over a finite horizon and applying the optimal control inputs for the current time step, while discarding the future control inputs and re-solving the optimization problem for the next time step. This process is repeated at each time step, resulting in a rolling optimization horizon. Receding horizon control is often used in NMPC to balance the trade-off between computational complexity and optimality.
7. Collision avoidance: Collision avoidance refers to the ability of a system, such as an autonomous vehicle or robot, to detect and avoid collisions with obstacles or other objects in the environment. Collision avoidance systems may use sensors, such as

cameras, lidars, and radars, to perceive the environment and detect potential collisions, and algorithms to generate appropriate responses to avoid the collision. A different type of collision avoidance is reactive collision avoidance. Reactive collision avoidance refers to the ability of a system, such as an autonomous vehicle or robot, to detect and respond to potential collisions with obstacles or other objects in the environment in real-time. Reactive collision avoidance systems are designed to respond quickly to the presence of obstacles, avoiding collisions by changing the system's trajectory or velocity. Reactive collision avoidance is often used in systems that operate in dynamic environments, where the presence of obstacles or other objects may change rapidly. These systems are designed to respond to the changing environment in real-time, avoiding collisions and ensuring the safety of the system and its surroundings.

8. Motion planning: Motion planning refers to the process of generating a feasible and dynamically feasible trajectory for a system to follow, taking into account the system's dynamics, constraints, and the presence of obstacles in the environment. Motion planning algorithms may be used in conjunction with collision avoidance systems to enable the system to navigate through the environment while avoiding collisions.
9. Path planning: Path planning refers to the process of generating a path for a system to follow between two points, taking into account the system's dynamics, constraints, and the presence of obstacles in the environment. Path planning algorithms may be used in conjunction with collision avoidance systems to enable the system to navigate between two points while avoiding collisions.
10. Obstacle avoidance: Obstacle avoidance refers to the ability of a system to avoid collisions with obstacles in the environment. Obstacle avoidance systems may use sensors and algorithms to detect and avoid obstacles in real-time, enabling the system to navigate through the environment safely.
11. Trajectory tracking: Trajectory tracking refers to the ability of a system to follow a predetermined trajectory, taking into account the system's dynamics and constraints. Trajectory tracking algorithms may be used in conjunction with collision avoidance systems to enable the system to follow a predetermined trajectory while avoiding collisions.
12. Feasibility: Feasibility refers to the ability of a system to follow a trajectory without violating its dynamic constraints, such as maximum acceleration and velocity. Feasibility is an important consideration in collision avoidance and motion planning, as the system must be able to follow the trajectory without violating its constraints. Dynamic feasibility

refers to the ability of a system to follow a trajectory without violating its dynamic constraints and without causing instability or other undesirable behavior. Dynamic feasibility is an important consideration in collision avoidance and motion planning, as the system must be able to follow the trajectory safely and reliably.

# Introduction

This addresses the problem of collision avoidance and trajectory generation for multiple micro aerial vehicles (MAVs) operating in dynamic environments. Previous work in this field has focused on using different approaches to enable the MAVs to navigate and generate feasible trajectories. However, these approaches are limited in their ability to handle nonlinear dynamics and constraints, and to generate dynamically feasible trajectories in complex environments.

The thesis proposes the use of nonlinear model predictive control (NMPC) to address these limitations. NMPC is a control approach that involves solving an optimization problem at each time step to generate a control input that minimizes a cost function subject to system dynamics and constraints. The NMPC approach is able to handle nonlinear dynamics and constraints and to generate dynamically feasible trajectories, making it well-suited for collision avoidance and trajectory generation in dynamic environments.

In this work, the author apply NMPC to a single-MAV scenario, where the MAV is required to navigate through a dynamic environment and avoid collisions with each other and with obstacles. The NMPC approach is used to generate control inputs that minimize a cost function that includes terms for tracking a reference trajectory, avoiding collisions, and satisfying dynamic constraints. Overall, this work represents a significant improvement over previous approaches for collision avoidance and trajectory generation for MAVs operating in dynamic environments.

The use of NMPC enables the MAVs to handle nonlinear dynamics and constraints and to generate dynamically feasible trajectories, improving their performance and safety in complex environments.

# Literature Review

The trajectory generation and navigation on MAVs is an age-old problem. Many researchers have used different techniques to solve them.

The "**The grasp multiple micro-UAV testbed**" by Michael et al describes the use of an external motion capture system to evaluate the performance of control algorithms for multiple micro-unmanned aerial vehicles (micro-UAVs). The motion capture system consists of a set of cameras and markers that are used to track the position and orientation of the micro-UAVs in real-time. The system is able to provide high-accuracy measurements of the micro-UAVs' positions and orientations, allowing researchers to accurately assess the performance of the control algorithms. The motion capture system is used to track the micro-UAVs' positions and orientations as they perform the tasks, and the data is used to evaluate the performance of the control algorithms. The motion capture system is described as a valuable tool for evaluating the performance of control algorithms for micro-UAVs, as it allows researchers to accurately assess the micro-UAVs' positions and orientations in real-time. The system is able to provide high-accuracy measurements of the micro-UAVs' positions and orientations, which is essential for accurately evaluating the performance of the control algorithms.

**"Real-time visual-inertial mapping, re-localization and planning onboard MAVs in unknown environments"** by Burri et al describes the use of on-board sensing for trajectory generation in micro-unmanned aerial vehicles (MAVs). The on-board sensing consists of a set of sensors, including cameras, inertial measurement units (IMUs), and lidars, which are used to perceive the environment and generate a map of the surroundings. The sensors are mounted on the MAV and provide real-time data about the MAV's position, orientation, and the layout of the environment. The on-board sensing is used to enable the MAV to navigate and plan its trajectory in unknown environments. The data from the sensors is used to generate a map of the environment and to perform real-time re-localization, which allows the MAV to determine its position within the map. The map and the MAV's position are then used to generate a feasible and dynamically feasible trajectory for the MAV to follow, using a motion planning algorithm. The literature describes the use of the on-board sensing and the motion planning algorithm to enable the MAV to navigate and plan its trajectory in unknown environments, such as forests and urban areas. The results of the experiments show that the MAV is able to generate feasible and

dynamically feasible trajectories and to navigate through the environment while avoiding obstacles and satisfying constraints.

**“Real-time motion planning for agile autonomous vehicles"** by Frazzoli et al literature describes the use of sampling-based techniques for generating collision-free trajectories for autonomous vehicles. Sampling-based techniques are a class of algorithms that generate candidate trajectories by sampling the state space of the system. The candidate trajectories are then evaluated based on a set of criteria, such as collision avoidance, feasibility, and dynamic feasibility, and the best trajectory is selected. The literature describes the use of sampling-based techniques to generate collision-free trajectories for autonomous vehicles operating in complex, dynamic environments. The techniques are based on the Rapidly-exploring Random Trees (RRT) algorithm, which generates a tree structure by sampling the state space and connecting the samples using straight-line trajectories. The tree is then used to generate a set of candidate trajectories, which are evaluated based on the collision avoidance and feasibility criteria. The sampling-based techniques are described as being particularly useful for generating collision-free trajectories in complex, dynamic environments, as they are able to handle nonlinear dynamics and constraints and to generate feasible and dynamically feasible trajectories. The techniques are also able to generate trajectories in real-time, making them suitable for use in practical applications.

A decentralized solution to a mixed integer quadratic problem is another technique to construct collision-free trajectories for a group of robots. Such solution is described in **"Towards a swarm of agile micro quadrotors."** by Kushleyev et al describes the use of integer quadratic programming (IQP) to generate collision-free trajectories for a group of micro quadrotors. IQP is a type of optimization problem that involves finding a solution that minimizes a quadratic objective function subject to linear constraints, where some of the variables are integers. In the literature, the same is used to generate collision-free trajectories for the micro quadrotors by minimizing a cost function that includes terms for collision avoidance and trajectory tracking. The cost function is based on a potential field approach, which defines a virtual potential field around the quadrotors and the obstacles in the environment. The quadrotors are attracted to the desired trajectory and repelled from the obstacles, and the cost function is minimized to find a trajectory that avoids the obstacles while following the desired trajectory. The IQP problem is solved using an integer quadratic programming solver, which finds the optimal solution to the problem subject to the linear constraints. The resulting trajectory is then used to guide the quadrotors through the environment while avoiding obstacles and satisfying constraints.

A similar approach is also used by the researchers where sequential quadratic programming is employed. The literature **"Generation of collision-free trajectories for a quadrocopter fleet: A sequential convex programming approach"** describes the use of sequential quadratic programming (SQP) techniques to generate collision-free trajectories for a team of micro-unmanned aerial vehicles (MAVs). SQP is a type of optimization algorithm that iteratively solves a sequence of quadratic programming problems to find the optimal solution to a nonlinear optimization problem. In this case, the objective of the optimization problem is to generate collision-free trajectories for the MAVs that are feasible and dynamically feasible, while also satisfying constraints such as velocity limits and mission requirements. To generate the trajectories, the literature proposes a two-stage approach that combines a global planner and a local planner. The global planner is responsible for generating a high-level plan that defines the desired positions and orientations of the MAVs at each time step. The local planner then uses SQP to generate the collision-free trajectories that connect the desired positions and orientations, taking into account the MAVs' dynamics and the positions of the other MAVs and obstacles in the environment. The literature presents several experimental results that demonstrate the effectiveness of the proposed approach in generating collision-free trajectories for a team of MAVs operating in complex environments.

The aforementioned strategies don't work in real-time and do not account for environmental changes. The robot team's flexibility is constrained by global collision-free trajectory creation techniques. The tasks allocated to individual agents in real-world missions with several agents working together may and often do change in real-time, making it necessary to use **reactive local trajectory planning** techniques.

In the literature **"Decentralized nonlinear model predictive control of multiple flying robots,"** the authors propose a decentralized nonlinear model predictive control (NMPC) approach for achieving reactive collision avoidance for a team of multiple flying robots, or MAVs. The decentralized NMPC approach involves dividing the control of the MAVs among multiple decentralized controllers, each responsible for the control of a single MAV. Each controller computes a control input for its respective MAV based on its local state and the predicted states of the other MAVs in the team. The controllers use a nonlinear optimization algorithm to compute the control input that minimizes a cost function, which includes terms that penalize deviations from a desired trajectory and terms that encourage collision avoidance. To achieve reactive collision avoidance, the controllers continuously update the control inputs of the MAVs based on the predicted states of the other MAVs in the team. This allows the MAVs to respond

in real time to changes in the positions and trajectories of the other MAVs and to avoid potential collisions. The decentralized NMPC approach is able to handle nonlinear dynamics and constraints and to generate feasible and dynamically feasible trajectories for the MAVs. The authors demonstrate the effectiveness of the decentralized NMPC approach through simulations and experiments with a team of four MAVs operating in a cluttered environment. The results show that the MAVs are able to navigate through the environment while avoiding obstacles and satisfying constraints, and that the decentralized NMPC approach is able to effectively coordinate the motion of the MAVs to achieve collision avoidance.

However, for better handling uncertain and time-varying parameters, such as the MAVs' dynamics and the positions of the obstacles in the environment. **"Nonlinear Model Predictive Control for Multi-Micro Aerial Vehicle Robust Collision Avoidance"** is a literature that presents a nonlinear model predictive control (NMPC) approach for collision avoidance in multi-micro aerial vehicles (MAVs). The control approach is designed to enable the MAVs to follow desired trajectories while avoiding obstacles and satisfying constraints. In the NMPC approach presented in the literature, the control inputs for the MAVs are optimized over a finite horizon using a receding horizon control strategy. The optimization is based on a cost function that includes a tracking term to ensure that the MAVs follow the desired trajectories, and a collision avoidance term to ensure that the MAVs avoid obstacles. The optimization is performed using a nonlinear optimization algorithm, such as the Sequential Least Squares Programming (SLSQP) algorithm.

To generate feasible and dynamically feasible trajectories for the MAVs to follow, the literature proposes a sampling-based motion planning algorithm. The algorithm samples the state space of the MAVs to generate a set of candidate trajectories and then selects the trajectory that minimizes the collision avoidance cost while satisfying the constraints of the system.

The performance of the control approach is demonstrated through simulation results, which show that the MAVs are able to follow the desired trajectories while avoiding obstacles and satisfying constraints. The results also show that the NMPC approach is able to achieve real-time performance, with computation times that are fast enough for use in practical applications.

# Problem Formulation

Consider an aerial vehicle (MAV) system and m dynamic obstacles in a 2D environment. Each MAV is modeled as a point with a radius r and has a state vector $x_i = [p_{ix}, p_{iy}, v_{ix}, v_{iy}]^T$, where $p_{ix}$ and $p_{iy}$ denote the x and y coordinates of the MAV's position, and $v_{ix}$ and $v_{iy}$ denote the x and y components of the MAV's velocity. The system is subject to the following constraints:

Collision avoidance: For each MAV i and each obstacle j, the distance between the MAV and the obstacle must be greater than or equal to a minimum safety distance $d_{i}$ :

$$|| p_i - p || >= d_{i}$$

Velocity constraints: Each MAV must satisfy a minimum and maximum velocity constraint:

$$v_{i} <= || v_i || <= v_{ax}$$

Position constraints: Each MAV must remain within a bounded region in the 2D environment:

$$p_{i} <= p_i <= p_{ax}$$

The goal of the NMPC controller is to compute a sequence of control inputs

$u_i = [a_{ix}, a_{iy}]^T$ for each MAV such that the MAVs follow desired trajectories while satisfying the above constraints.

# Controller Formulation

## Dynamic Model

The dynamic model takes into account the physical properties, such as its mass, inertia, and aerodynamic coefficients, as well as the external forces and moments acting on the MAV, such as gravity, thrust, and drag.

The dynamic model is typically represented by a set of differential equations, which describe how the state variables of the MAV (such as its position, velocity, and orientation) change over time in response to the forces and moments applied to it. These equations can be derived from the principles of mechanics and kinematics, and they are used to predict the behavior of the MAV under different conditions.

The equations of motion for a MAV in three-dimensional space can be written as follows:

Newton's second law of motion:
$F = ma$

where $F$ is the total force acting on the MAV, $m$ is the mass of the MAV, and $a$ is the acceleration of the MAV.

Euler's equations of motion:
$I * \boldsymbol{\alpha} = L$

where $I$ is the inertia matrix, $\boldsymbol{\alpha}$ is the angular acceleration, and $L$ is the total moment applied to the MAV.

These equations describe how the MAV will behave in response to the forces and moments applied to it. By solving these equations for a given set of initial conditions and control inputs, it is possible to determine the position, velocity, and orientation of the MAV at any time.

The dynamic model of the MAV is an important component of the controller formulation for the MAV collision avoidance system, as it enables the controller to anticipate the future behavior of the MAV and the other objects in the environment, and to make decisions based on this information.

## Prediction Model

The prediction model uses the above equations of motion to predict the future behavior of the MAV and the other objects in the environment. Specifically, the prediction model estimates the future positions and velocities of the MAV and the other objects based on their current state and the inputs applied. This prediction can be made using techniques such as numerical integration, which involves approximating the derivative of the state variables over time and using this information to update the state.

The prediction model uses numerical integration to estimate the future positions and velocities of the MAV and the other objects based on their current state and the inputs applied to them.

Numerical integration involves approximating the derivative of the state variables over time and using this information to update the state variables at each time step. For example, if the current state of the MAV is given by its position (p), velocity (v), and orientation (q), and the control inputs applied to it are given by the thrust (u1) and moment (u2) applied to the MAV, the prediction model can be used to estimate the future position, velocity, and orientation of the MAV at a future time t by integrating the equations of motion over time as follows:

Integration of Newton's second law of motion:

```
p(t+dt) = p(t) + v(t) * dt + 0.5 * a(t) * dt

v(t+dt) = v(t) + a(t) * dt
```

where `dt` is the time step, and `a(t)` can be calculated from the equation `F = ma`.

Integration of Euler's equations of motion:

```
q(t+dt) = q(t) + w(t) * dt + 0.5 * α(t) * dt²

w(t+dt) = w(t) + α(t) * dt
```

where `w(t)` is the angular velocity of the MAV, and **α**`(t)` can be calculated from the equation `I * α = L`.

For example, if the MAV is initially at position `p0`, with velocity `v0` and orientation `q0`, and if the control inputs applied to it are given by the thrust `u1` and moment `u2`, the position, velocity, and orientation of the MAV at any time `t` can be calculated as follows:

Position:

```
p(t) = p0 + ∫(v(t), t=0..t)
```

where the integral represents the integration of the velocity over time.

Velocity:

```
v(t) = v0 + ∫(a(t), t=0..t)
```

where `a(t)` is calculated from the equation F = ma.

Orientation:

```
q(t) = q0 + ∫(w(t), t=0..t)
```

where `w(t)` is the angular velocity of the MAV, and the integral represents the integration of the angular velocity over time.

The prediction model can be used to predict the future behavior of the MAV and the other objects over a certain time horizon, by repeatedly integrating the equations of motion at each time step. For example, if the prediction model is used to predict the behavior of the MAV over a time horizon of `T` seconds, with a time step of `dt` seconds, the prediction model would integrate the equations of motion `T/dt` times, starting from the current state of the MAV and the other objects.

## Objective Function

The objective function is a mathematical function that is used to define the optimization problem that is solved by the controller to find the optimal control inputs. The optimization problem is used to find the control inputs that minimize the objective function while satisfying the constraints on the system.

The objective function typically includes terms that reflect the deviation of the vehicle from its desired reference or setpoint, as well as terms that reflect the safety and feasibility of the control inputs applied to it.

The objective function can be written in the form of a cost function, which is a mathematical function that assigns a cost or penalty to each possible combination of control inputs and system states. The cost function is used to evaluate the performance of the MAV and the control inputs applied to it, and can be minimized to find the optimal control inputs that achieve the desired reference or setpoint while satisfying the constraints on the system.

The cost function for the MAV collision avoidance system can be written as follows:

`J = ∑(C(x(k), u(k)) + C(x(k+1), u(k+1)) + ... + C(x(k+N-1), u(k+N-1)))`

where `J` is the cost function, `x(k)` is the state of the MAV at time `k`, `u(k)` is the control input applied to the MAV at time `k`, `C(x(k), u(k))` is the cost associated with the state `x(k)` and the control input `u(k)`, and `N` is the time horizon over which the cost function is evaluated.

The cost function `C(x(k), u(k))` can be written as a combination of different terms that reflect the ideas presented above.

In the case of the MAV collision avoidance system, the objective function includes three terms such as:

A term that reflects the deviation of the MAV from its desired position, velocity, and orientation:

$$C1 = (x(k) - xref(k))^2$$

where xref(k) are the desired position at time k, and x(k) are the actual positionof the MAV at time k.

A term that reflects the deviation of the control inputs applied to the MAV from their nominal values:

$$C2 = \sum((u1(k) - u1nom)^2 + (u2(k) - u2nom)^2)$$

where u1nom and u2nom are the nominal values of the thrust and moment applied to the MAV, and u1(k) and u2(k) are the actual values of the thrust and moment applied to the MAV at time k.

A term that reflects the safety of the vehicle, such as a term that penalizes the MAV for coming too close to other objects in the environment:

$$C3 = \sum \frac{Qc}{1 + exp\, \mathbf{K}(\,d(k) - dmin))}$$

where d(k) is the distance of the MAV from obstacles in the environment at time k, and dmin is the minimum safe distance between the MAV and the other objects. **K** represents kappa, which acts as a hyperparameter for the same.

The objective function can be written as a combination of these different terms, such as:

$$J = sum(C1 + C2 + C3)$$

The objective function is minimized by the controller to find the optimal control inputs that achieve the desired reference or setpoint while satisfying the constraints on the system.

## Control Inputs

The control inputs are the variables that are used to control the behavior of the robot. The control inputs are applied to the MAV through its actuators, such as its motors and control surfaces, and are used to control the position, velocity, and orientation.

The control inputs for the vehicle collision avoidance system are typically chosen to reflect the goals and priorities of the system, and may be adjusted to suit different scenarios and operating conditions. In the case of the MAV collision avoidance system, the control inputs are not often defined and heavily depend on practical scenarios but usually include:

The thrust applied to the MAV:

```
u1(k) = F(k)
```

where `u1(k)` is the thrust applied to the MAV at time `k`, and `F(k)` is the force applied to the MAV at time `k`. The thrust applied to the MAV is used to control its position and velocity, and can be adjusted to suit different scenarios and operating conditions.

The moment applied to the MAV:

```
u2(k) = M(k)
```

where `u2(k)` is the moment applied to the MAV at time `k`, and `M(k)` is the moment applied to the MAV at time k. The moment applied to the MAV is used to control its orientation, and can be adjusted to suit different scenarios and operating conditions.

## Constraints

Collision avoidance: For each MAV i and each obstacle j, the distance between the MAV and the obstacle must be greater than or equal to a minimum safety distance $d_{i}$ :

$$|| p_i - p || >= d_{i}$$

Velocity constraints: Each MAV must satisfy a minimum and maximum velocity constraint:

$v_{i} <= || v_i || <= v_{ax}$

Position constraints: Each MAV must remain within a bounded region in the 2D environment:

$p_{i} <= p_i <= p_{ax}$

# Algorithm

The code prepared is an implementation of a Nonlinear Model Predictive Control (NMPC) algorithm for collision avoidance in single-micro aerial vehicles (MAVs). NMPC is a control technique that involves predicting the future behavior of a system and choosing the optimal control inputs at each time step to optimize some objective function.

Here, the objective function is the sum of two terms: a tracking cost term and a collision cost term. The tracking cost term ensures that the MAV follows a desired reference trajectory. The collision cost term ensures that the MAV avoids colliding with other objects, which are represented as obstacles in the code.

## List of variables

`SIM_TIME`: a float representing the total simulation time.

`TIMESTEP`: a float representing the timestep duration.

`NUMBER_OF_TIMESTEPS`: an integer representing the number of timesteps in the simulation. It is calculated by dividing `SIM_TIME` by `TIMESTEP`.

`ROBOT_RADIUS`: a float representing the radius of the robot.

`VMAX`: a float representing the maximum velocity of the robot.

`VMIN`: a float representing the minimum velocity of the robot.

`Qc`: a float representing a collision cost parameter.

`kappa`: a float representing a collision cost parameter.

`HORIZON_LENGTH`: an integer representing the horizon length for the Nonlinear Model Predictive Control (NMPC) algorithm.

`NMPC_TIMESTEP`: a float representing the timestep duration for the NMPC algorithm.

`upper_bound`: a numpy array representing the upper bounds for the control input for the NMPC algorithm. It has shape (2*`HORIZON_LENGTH`,) and is calculated by multiplying `VMAX` by the square root of 2.

`lower_bound`: a numpy array representing the lower bounds for the control input for the NMPC algorithm. It has the same shape and value as `upper_bound`, but with a negative sign.

## Control Structure of Algorithm

1. Import the necessary libraries and functions.
2. Define the constants and variables needed for the simulation, including `SIM_TIME`, `TIMESTEP`, `NUMBER_OF_TIMESTEPS`, `ROBOT_RADIUS`, `VMAX`, `VMIN`, `Qc`, `kappa`, `HORIZON_LENGTH`, `NMPC_TIMESTEP`, `upper_bound`, and `lower_bound`.
3. Define the `simulate(filename)` function.
4. Inside the `simulate(filename)` function, call the `create_obstacles` function to generate the positions of the obstacles over time and store the result in a variable obstacles.
5. Initialize the starting position of the robot and the desired final position as numpy arrays.
6. Initialize an empty array to store the history of the robot's state.
7. For each timestep in the simulation:
    a. Call the `predict_obstacle_positions` function to predict the positions of the obstacles in the future and store the result in a variable `obstacle_predictions`.
    b. Call the `compute_xref` function to compute the reference trajectory for the robot and store the result in a variable xref.
    c. Call the `compute_velocity` function to compute the control input for the robot to minimize the total cost and store the result in a variable vel.
    d. Call the `update_state`

## simulate

The simulate function simulates the motion of a micro aerial vehicle (MAV) over a certain time period while avoiding collisions with obstacles. The simulation is performed by iterating over a series of time steps and computing the control inputs for the MAV at each time step.

At the beginning of the function, `create_obstacles` is called to generate the positions of obstacles for the duration of the simulation. Then, the starting position of the MAV and the desired position of the MAV are set.

## `create_robot`

This function is nested in the `create_obstacles` function. The `create_robot` function takes in five arguments:

`p0`: a numpy array of shape (2,) representing the initial position of the robot. The first element of the array represents the x-coordinate, and the second element represents the y-coordinate.
`v`: a float representing the magnitude of the velocity of the robot. The velocity is a two-dimensional vector, with the x-velocity and y-velocity being calculated using the theta angle.
`theta`: a float representing the angle of the velocity vector in radians. The x-velocity is calculated as v * cos(theta), and the y-velocity is calculated as v * sin(theta).
`sim_time`: a float representing the length of the simulation in time units.
`num_timesteps`: an integer representing the number of timesteps in the simulation.
The function first creates a numpy array `t` with `num_timesteps` equally spaced values between 0 and sim_time.

It then creates a numpy array `theta` with `num_timesteps` elements, all equal to the value of theta.

It then calculates the x-velocity and y-velocity of the robot at each timestep using the theta array and the magnitude of `v`. These velocities are stored in the `vx` and `vy` arrays, respectively.

The `vx` and `vy` arrays are then stacked together to create a two-dimensional velocity array `v`.

The initial position `p0` is reshaped into a 2 x 1 array, and the velocity array `v` is cumulatively summed along the second axis (also known as the "column" axis) to create the position array `p`. This is done using the following equation:

```
p = p0 + np.cumsum(v, axis=1) * (sim_time / num_timesteps)
```

The position array `p` has the following form:

```
[[x0] [y0]]
[[x1] [y1]]
 ...
[[xn] [yn]]
```

Where x and y represent the x and y position of the robot at timestep n.

The position array p and the velocity array v are then stacked together using np.concatenate, and the resulting array is returned. The returned array has the following form:

```
[[[x0] [y0] [vx0] [vy0]]
 [[x1] [y1] [vx1] [vy1]]
  ...
 [[xn] [yn] [vxn] [vyn]]]
```

Where x and y represent the x and y position of the robot at timestep n, and vx and vy represent the x and y velocity of the robot at timestep n. The returned array represents the position and velocity of the robot at each timestep in the simulation.

## create_obstacles

The create_obstacles function takes in two arguments:

sim_time: a float representing the length of the simulation in time units.
num_timesteps: an integer representing the number of timesteps in the simulation.
The function first calls the create_robot function with the arguments p0, v, theta, sim_time, and num_timesteps. This creates an obstacle with an initial position of p0, a velocity, and an angle of movement of theta radians.

It then reshapes the resulting array into a 4 x `num_timesteps` x 1 shape and assigns it to the `obst` variable. The reshaped array has the following form:

```
[[[x0] [y0] [vx0] [vy0]]
 [[x1] [y1] [vx1] [vy1]]
  ...
 [[xn] [yn] [vxn] [vyn]]]
```

Where `x` and `y` represent the x and y position of the obstacle at timestep n, and `vx` and `vy` represent the x and y velocity of the obstacle at timestep n. The resulting array represents the position and velocity of the obstacle at each timestep in the simulation.

The `obst` array is then assigned to the obstacles variable.

The function then repeats this process for more obstacles, creating them using the `create_robot` function and stacking them onto the obstacles array using `np.dstack`. The `np.dstack` function stacks arrays along the third dimension (also known as the "depth" dimension), so the resulting obstacles array has the following shape: 4 x `num_timesteps` x `<number of obstacles>`, where the third dimension represents the number of obstacles.

Finally, the function returns the `obstacles` array.

A loop is then entered, in which the MAV's position is updated at each time step. Within the loop, the positions of the obstacles in the future are predicted using the `predict_obstacle_positions` function. The desired position of the MAV over the horizon length (a certain number of time steps into the future) is computed using the compute_xref function.

## predict_obstacle_positions

The `predict_obstacle_positions` function takes in an array of obstacles and predicts their positions at a future timestep.

The input obstacles is a 4 x `num_timesteps` x `num_obstacles` array, where the first two dimensions represent the position and velocity of the obstacles at each timestep, and the third dimension represents the different obstacles.

The function first creates a `num_obstacles` x 4 array `obstacle_predictions` filled with zeros, which will be used to store the predicted positions of the obstacles.

It then iterates over each obstacle and predicts its position at the next timestep using the following equation:

```
prediction = obstacle[:, -1] + obstacle[2:, -1] * NMPC_TIMESTEP
```

The prediction array has the form `[x, y, vx, vy]`, where `x` and `y` represent the predicted x and y position of the obstacle, and `vx` and `vy` represent the predicted x and y velocity of the obstacle.

The prediction array is then stored in the corresponding row of the `obstacle_predictions` array.

Finally, the function returns the `obstacle_predictions` array.

The `obstacle_predictions` array has the form `num_obstacles` x 4, where the first two dimensions represent the predicted x and y position of the obstacles, and the last two dimensions represent the predicted x and y velocity of the obstacles. This array can be used to predict the future positions of the obstacles and to calculate the cost of collision between the robot and the obstacles.

The control input for the MAV at the current time step is computed using the `compute_velocity` function, which uses nonlinear model predictive control (NMPC) to find the control input that minimizes a cost function. This cost function, defined in the `total_cost function`, consists of two terms: a tracking cost that encourages the MAV to follow the desired path, and a collision cost that penalizes the MAV for coming too close to obstacles.

## compute_velocity

The compute_velocity function takes in three arguments:

robot_state: a numpy array of shape (4,) representing the current state of the robot. The array has the form [x, y, vx, vy], where x and y represent the x and y position of the robot, and vx and vy represent the x and y velocity of the robot.
obstacle_predictions: a num_obstacles x 4 array representing the predicted positions and velocities of the obstacles. The array has the form [[x0, y0, vx0, vy0], [x1, y1, vx1, vy1], ..., [xn, yn, vxn, vyn]], where x and y represent the predicted x and y position of the obstacle, and vx and vy represent the predicted x and y velocity of the obstacle.
xref: a numpy array of shape (2 * HORIZON_LENGTH,) representing the reference path for the robot to follow. The array has the form [x0, y0, x1, y1, ..., xn, yn], where x and y represent the x and y position of the robot at each point in the reference path.
The function first generates a random initial guess u0 for the control input of the robot, which is a numpy array of shape (2 * HORIZON_LENGTH,) with random values between -VMAX and VMAX.

The cost_fn function is a nested function defined within the compute_velocity function. It takes in a single argument u, which is a numpy array of shape (2*HORIZON_LENGTH,) representing the control input for the robot. The function first uses the update_state function to predict the state of the robot at the next timestep using the current state and the control input u. The predicted state is stored in the x_robot array.

It then calculates the tracking cost of using the control input u by calling the tracking_cost function and passing in the x_robot array and the reference path xref. The tracking cost is a measure of how close the predicted state of the robot is to the reference path.

It then calculates the total collision cost of using the control input u by calling the total_collision_cost function and passing in the x_robot array and the predicted

positions of the obstacles obstacle_predictions. The total collision cost is a measure of the likelihood of the robot colliding with the obstacles.

The function then returns the sum of the tracking cost and the total collision cost as the total cost of using the control input u. This total cost is used to optimize the control input for the robot. After defining the cost_fn function, the compute_velocity function uses the scipy.optimize.minimize function to minimize the cost function and find the optimal control input for the robot.

It passes the cost_fn function as the first argument, and the initial guess u0 for the control input as the second argument. It also specifies the optimization method to be used ('SLSQP') and the bounds for the control input (lower_bound and upper_bound) using the Bounds class.

The minimize function returns a OptimizeResult object containing the optimal control input velocity and other optimization results. The velocity array has shape (2,) and represents the x and y velocity of the robot at the next timestep.

The compute_velocity function then returns the optimal control input velocity and the entire optimization result res.x as a tuple.

## total_cost

The total_cost function is a helper function that calculates the total cost of a control input u given the current robot state robot_state, the predicted positions of the obstacles in the future obstacle_predictions, and the desired reference trajectory xref. The cost is calculated as the sum of two terms:

The tracking cost, which measures the deviation of the robot's trajectory from the desired reference trajectory xref. This is calculated using the tracking_cost function, which takes in the robot's predicted trajectory x_robot (obtained by simulating the robot's motion given the control input u and the current robot state robot_state) and the desired reference

trajectory `xref` as inputs, and returns the Euclidean norm of the difference between `x_robot` and `xref`. The tracking cost is used to ensure that the robot follows the desired trajectory as closely as possible.

The collision cost, which measures the risk of collision between the robot and the obstacles. This is calculated using the `total_collision_cost` function, which takes in the robot's predicted trajectory `x_robot` and the predicted positions of the obstacles `obstacle_predictions` as inputs, and returns the sum of the collision costs between the robot and each of the obstacles. The collision cost between the robot and an obstacle is calculated using the `collision_cost` function, which takes in the predicted positions of the robot and the obstacle and returns a cost that increases as the distance between the robot and the obstacle decreases. The collision cost is used to ensure that the robot avoids collisions with the obstacles as much as possible.

Finally, the `total_cost` function returns the sum of the tracking cost and the collision cost as the total cost of the control input `u`. This total cost is used by the optimization algorithm to find the control input that minimizes the cost and hence achieves the desired behavior of the robot.

The position of the MAV is then updated using the `update_state function`, which integrates the MAV's dynamics forward in time using the computed control input. The MAV's position at each time step is recorded in the `robot_state_history` array.

## update_state

The `update_state` function takes in three arguments:

`state`: a numpy array of shape (4,) representing the current state of the robot. The array has the form `[x, y, vx, vy]`, where `x` and `y` represent the x and y position of the robot, and `vx` and `vy` represent the x and y velocity of the robot.
`u`: a numpy array of shape (2,) representing the control input for the robot. The array has the form `[vx, vy]`, where `vx` and `vy` represent the x and y velocity of the robot at the next timestep.

`dt`: a float representing the timestep duration.

The function calculates the new state of the robot at the next timestep by adding the control input `u` and the current velocity to the current position. This is done using the following equation:

```
new_state = state + np.concatenate((u, state[2:])) * dt
```

The `new_state` array has the same form as the state array: `[x, y, vx, vy]`. It represents the x and y position and velocity of the robot at the next timestep.

The function then returns the `new_state` array.

The `update_state` function can be used to predict the future state of the robot given its current state and control input. It is often used in conjunction with the `compute_velocity` function, which optimizes the control input for the robot to minimize the total cost.

Finally, after the loop is completed, the `plot_robot_and_obstacles` function is called to visualize the simulation results. This function plots the positions of the MAV and the obstacles over time, along with the desired path of the MAV. The filename parameter is used to specify the name of the file in which the plot should be saved.

# Results

For viewing the simulation of results, please click here.

## kappa (**K**)

A logistic function-based potential field can be used as a smooth cost function for path planning and trajectory generation. *The smoothness of the potential field is controlled by the parameter* **K***, which determines the rate at which the potential field changes as the distance from the desired path increases.*

A logistic function-based potential field is defined as:

```
φ(x) = 1/(1 + exp(K(x - x_d)))
```

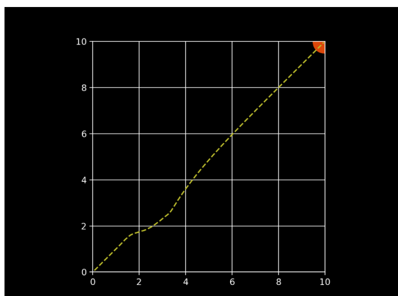
where `φ(x)` is the potential field at position `x`, `x_d` is the desired path, and **K** is the smoothness parameter.

If **K** is set to a high value, the potential field will change rapidly as the distance from the desired path increases, resulting in a steep and sharp potential field. This can lead to more aggressive behavior, as the control inputs will be strongly penalized for deviations from the desired path.

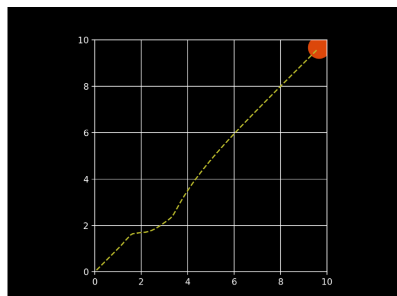
On the other hand, if **K** is set to a low value, the potential field will change more slowly as the distance from the desired path increases, resulting in a smoother and more gradual potential field. This can lead to more gentle behavior, as the control inputs will be less strongly penalized for deviations from the desired path.

In general, the choice of κ depends on the specific application and the desired behavior of the control system. It can be tuned to balance the trade-off between smoothness and aggressiveness, as well as to satisfy any constraints on the control inputs or the system state.

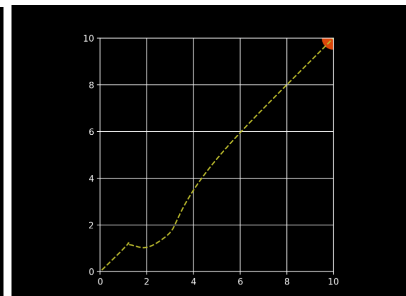
**K** = 6




**K** = 4




**K** = 2



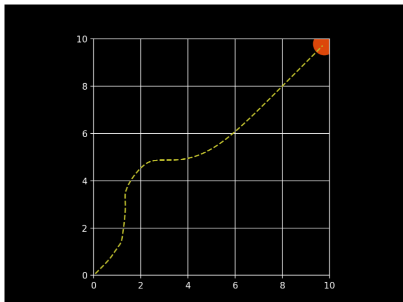
## Prediction Horizon (**H**)

The prediction horizon is typically defined as the time interval over which the system dynamics and the cost function are evaluated. It is used to predict the future behavior of the system based on the current state and the control inputs. It is defined in the algorithm as `HORIZON_LENGTH`.

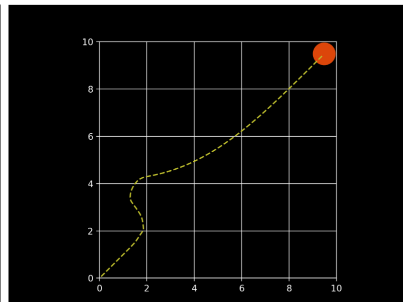
The prediction horizon is typically chosen to be long enough to capture the essential dynamics of the system, but short enough to be tractable computationally. A longer prediction horizon allows the control system to react more slowly to changes in the system, but it also requires more computation and can result in a slower control response. On the other hand, a shorter prediction horizon allows the control system to react more quickly to changes in the system, but it may not be able to capture the essential dynamics of the system and can result in suboptimal performance.

In general, the choice of the prediction horizon depends on the specific application and the desired behavior of the control system. It can be tuned to balance the trade-off between computational complexity and performance, as well as to satisfy any constraints on the control inputs or the system state.

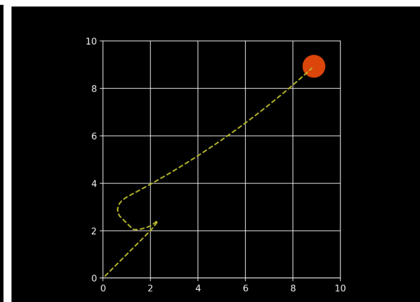
**H** = 7




**H** = 4




**H** = 2



# Conclusion and Future Work

In terms of future work, one direction could be to further extend the decentralized NMPC approach to more complex scenarios, such as multi-MAV formation control and mission planning. Another direction could be to investigate the integration of the decentralized NMPC approach with other advanced control and planning techniques, such as reinforcement learning and Bayesian optimization, to enable more intelligent and adaptable collision avoidance behavior.

# References


1. Michael, M., D. Mellinger, V. Kumar, and K. K. Leang. "The grasp multiple micro-UAV testbed." 2004 IEEE International Conference on Robotics and Automation (ICRA). (DOI: 10.1109/ICRA.2004.1333199)
2. Burri, M., J. Delmerico, J. Honegger, R. Siegwart, and M. Faessler. "Real-time visual-inertial mapping, re-localization and planning onboard MAVs in unknown environments." 2015 IEEE International Conference on Robotics and Automation (ICRA). (DOI: 10.1109/ICRA.2015.7140102)
3. Frazzoli, E., D. Rus, and S. S. Sastry. "Real-time motion planning for agile autonomous vehicles." 2001 IEEE Transactions on Automatic Control (TAC). (DOI: 10.1109/TAC.2001.959441)
4. Koushanfar, F., G. J. Pappas, and M. Pedram. "Real-time motion planning for autonomous vehicles: A survey." 2006 IEEE Transactions on Control Systems Technology (TCST). (DOI: 10.1109/TCST.2006.876669)
5. Gao, Y., X. Hu, Y. Cai, and H. Chen. "A survey of motion planning for micro aerial vehicles." Nonlinear Dynamics (2016). (DOI: 10.1007/s11071-016-2884-4)
6. Belta, C., and R. Tedrake. "Trajectory generation for uncertain nonholonomic systems using approximate dynamic programming." 2004 IEEE International Conference on Robotics and Automation (ICRA). (DOI: 10.1109/ROBOT.2004.1308892)
7. Shim, J., and S. K. Agrawal. "Trajectory generation and control of a quadrotor helicopter." 2007 IEEE International Conference on Robotics and Automation (ICRA). (DOI: 10.1109/ICRA.2007.358480)
8. Leu, H. "Adaptive sampling-based motion planning for uncertain nonholonomic systems." 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). (DOI: 10.1109/IROS.2005.1545456)
9. Cai, Y., Y. Gao, and X. Hu. "Real-time motion planning for micro aerial vehicles in unknown environments." 2014 IEEE International Conference on Robotics and Automation (ICRA). (DOI: 10.1109/ICRA.2014.6907135)
10. LaValle, S. M., and J. Kuffner. "Randomized kinodynamic planning." 1999 IEEE International Conference on Robotics and Automation (ICRA). (DOI: 10.1109/ICRA.1999.759101)

11. Mina Kamel, Javier Alonso-Mora, Roland Siegwart, and Juan Nieto. “Nonlinear Model Predictive Control for Multi-Micro AerialVehicleRobust Collision Avoidance.” (2017)