agilicious

Release 0.0.1

Robotics and Perception Group (UZH)

Apr 17, 2023

ABOUT

Read the Paper and watch the full Video!

CHAPTER

ONE

AGILE FLIGHT DONE RIGHT!

Agilicious is a co-designed hardware and software framework tailored to autonomous, agile quadrotor flight, which has been developed and used since 2016 at the Robotics and Perception Group (RPG) of the University of Zurich. Agilicious is described in this Science Robotics 2022 paper. It is completely open-source and open-hardware and supports both model-based and neural-network-based controllers. Also, it provides high thrust-to-weight and torque-to-inertia ratios for agility, onboard vision sensors, GPU-accelerated compute hardware for real-time perception and neural-network inference, a real-time flight controller, and a versatile software stack. In contrast to existing frameworks, Agilicious offers a unique combination of flexible software and high-performance hardware. Agilicious has been used in over 30 scientific papers at our lab, including trajectory tracking for drone racing scenarios at up to 5g and 70km/h (SciRob21_Foehn, Video), vision-based acrobatic flight (RSS20 Kaufmann, Video), obstacle avoidance in both structured and unstructured environments using solely onboard perception (SciRob21_Loquercio, Video), and hardware-in-the-loop simulation in virtual-reality environments. Thanks to its versatility, we believe that Agilicious supports the next generation of scientific and industrial quadrotor research. Agilicious allows to seamlessly develop, test, reproduce and benchmark control algorithms such as Non-linear Model Predictive Control (MPC), differential-flatness-based control (DFBC) and INDI (Incremental Non-linear Dynamic Inversion) (TRO 2022, Sun, Video). Agilicious also includes our state-of-the-art, high-fidelity simulator, which uses Blade Element Momentum (BEM) theory for the propeller model and provides accurate estimates of the quadcopter's aerodynamic characteristics across the flight envelope (RSS 2021, Bauersfeld, Video).

If you use Agilicious, please cite these papers. The full list of publications using Agilicious can be found here.

1.1 License

The right to use Agilicious is strictly limited to researchers who have explicitly been granted access by UZH. To ask for access, please check the following page:

The following license applies in case you have been granted access to the Agilicious software by UZH.

(continues on next page)

(continued from previous page)

```
conditions.

LICENSE TERMS

Introduction
(i) UZH has developed the SOFTWARE under the lead of Prof. Dr. Davide Scaramuzza
    (hereinafter referred to as the PRINCIPAL INVESTIGATOR). SOFTWARE means the
    collection of computer code in all forms including, without limitation, both
    source code and binary code.
(ii) LICENSEE wishes to obtain a non-exclusive, non-transferable and royalty-free
     license of the SOFTWARE for academic, non-commercial purposes only as specified
     in this AGREEMENT.
In consideration of the above premises LICENSEE agrees as follows:
1 Grant/Scope of License
     1.1    UZH hereby grants to LICENSEE a non-exclusive, non-transferable, royalty-free
     license to use the SOFTWARE for internal academic, non- commercial purposes only.
     1.2    UZH will not provide any services or support in connection with the SOFTWARE
     or technical support within the scope of this AGREEMENT.
2 Permitted Use and Restrictions
     2.1 LICENSEE agrees that it will use the SOFTWARE, and any modifications,
     improvements, or derivatives to SOFTWARE that LICENSEE may create (collectively,
     "IMPROVEMENTS") solely for its own internal, academic, non-commercial
     purposes. The terms "academic, non-commercial", as used in this Agreement, mean
     academic or other scholarly research which (a) is not undertaken for any direct
     or indirect for-profit purposes, and (b) is not intended to produce works,
     services, or data for commercial use.
     2.2    The sale, lease or rental of any portion of the SOFTWARE or any IMPROVEMENTS
     and/or the use of any portion of the SOFTWARE or any IMPROVEMENTS in any service
     or good sold or otherwise made available to third parties is strictly prohibited.
     LICENSEE further agrees not to use any portion of the SOFTWARE or of any␣
↪IMPROVEMENTS
     in any machine-readable form outside the SOFTWARE, nor to make any copies except for
     its internal use, without prior written consent of LICENSOR.
     2.3    LICENSEE shall not sub-license, distribute, transfer, disclose or make␣
↪available
     the SOFTWARE or any IMPROVEMENTS in whole or in part to another research group of␣
↪the
     LICENSEE or to any third party without prior written permission from PRINCIPAL
     INVESTIGATOR (please submit inquiries by Email).
3. Warranty Disclaimer
     3.1    THE SOFTWARE IS PROVIDED "AS IS" AND UZH MAKES NO REPRESENTATIONS OR␣
↪WARRANTIES,
     EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT WITHOUT LIMITATION, UZH MAKES NO
     REPRESENTATIONS OR WARRANTIES OF MERCHANTIBILY OR FITNESS FOR ANY PARTICULAR PURPOSE
     OR THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET LICENSEE'S REQUIREMENTS OR
     THAT THE USE OF THE SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY'S
     PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS. FURTHERMORE, UZH DOES NOT WARRANT
     OR MAKE ANY REPRESENTATIONS REGARDING THE USE OR THE RESULTS OF THE USE OF THE
     SOFTWARE IN TERMS OF CORRECTNESS, ACCCURACY, RELIABILITY, OR OTHERWISE OR THAT␣
↪DEFECTS
     IN THE SOFTWARE WILL BE CORRECTED. UZH WILL NOT BE LIABLE FOR ANY CONSEQUENTIAL,
     INCIDENTAL, OR SPECIAL DAMAGES, OR ANY OTHER RELIEF, OR FOR ANY CLAIM BY ANY THIRD
```

(continues on next page)

(continued from previous page)

```
     PARTY, ARISING FROM THE USE OF THE SOFTWARE.
     3.2    The LICENSEE expressly acknowledges and agrees that the use of the SOFTWARE␣
↪is
     at LICENSEE's sole risk and to hold harmless and indemnify UZH, and its affiliates,
     employees or partners, from and against any third party claim arising from or in any
     way related to LICENSEE's use of SOFTWARE, violation of this AGREEMENT or any other
     actions in connection with the use of SOFTWARE.
4.  Citation
     The LICENSEE expressly acknowledges and agrees to reference the publication
↪"Agilicious:
     Open-Source and Open-Hardware Agile Quadrotor for Vision-Based Flight", AAAS Science
     Robotics, 2022 (PDF: https://rpg.ifi.uzh.ch/docs/ScienceRobotics22_Foehn.pdf) and
     to acknowledge Philipp Foehn, Elia Kaufmann, Angel Romero, Robert Penicka, Sihao␣
↪Sun,
     Leonard Bauersfeld, Thomas Laengle, Giovanni Cioffi, Yunlong Song, Antonio
     Loquercio, Davide Scaramuzza, Robotics and Perception Group, and the UNIVERSITY
     OF ZURICH as the source of the SOFTWARE in any publications reporting use of it
     or any manual or document.
5.  Title and Ownership.
     Title, ownership rights, and intellectual property rights in and to the SOFTWARE
     shall remain with UZH.
6.  Term and Termination
     6.1    This AGREEMENT shall become effective upon LICENSEE first downloading the
     SOFTWARE (“Effective Date”).
     6.2    UZH may terminate this AGREEMENT upon 30 (thirty) days advance written e-mail
     notification to LICENSEE. Upon evidence of violation of any of the terms under
     this AGREEMENT by LICENSEE, UZH may terminate this AGREEMENT without previous␣
↪notice.
     6.3    Upon termination LICENSEE is obliged to uninstall the SOFTWARE and␣
↪IMPROVEMENTS
     from all its computers and to destroy any copies of the SOFTWARE and IMPROVEMENTS
     kept according to this AGREEMENT.
     6.4 Articles  3 and 4 shall survive the termination or expiration of this AGREEMENT
     for any reason in addition to those articles surviving by operation of law.
7.  Miscellaneous
     7.1    This AGREEMENT and the license granted herein or any part thereof under this
     AGREEMENT are not assignable by LICENSEE without the prior written approval of UZH.
     7.2    Neither party shall use the names or trademarks of the other, its related
     entities and its employees, or any adaptations thereof, in any advertising,
     promotional or sales literature, or in any securities reports required by the
     respective authorities, without the prior written consent of the party so affected.
     7.3    Each party is acting as an independent contractor and not as an agent,␣
↪partner,
     or joint venture with the other party for any purpose. Neither party shall have any
     right, power or authority to act or create any obligation, express or implied, on
     behalf of the other.
     7.4    This AGREEMENT sets forth the entire AGREEMENT between the parties with␣
↪respect
     to the subject matter hereof. No supplement, modification or amendment of this
     AGREEMENT shall be binding, unless in writing signed by a duly authorized
     representative of each party to the AGREEMENT.
     7.5    Should some or several provisions of this AGREEMENT be ineffective or␣
```

(continues on next page)

(continued from previous page)

```
↪invalid,
    or should there be an omission in this AGREEMENT, the effectiveness, respectively
    the validity of the remaining provisions shall not be affected thereby. An
    ineffective, respectively, invalid provision shall be replaced by the
    interpretation of the agreement which comes nearest to the economic meaning
    and the envisaged economic purpose of the ineffective respectively, invalid␣
↪provision.
    The same applies in the case of a contractual gap.
    7.6    The terms stipulated in this AGREEMENT may not be modified in any way without
    the mutual consent of the parties in writing.
8.  Governing Law and Jurisdiction
    THIS AGREEMENT SHALL BE GOVERNED BY THE LAWS OF SWITZERLAND. Any dispute arising
    from or in connection with this AGREEMENT will be finally settled by the courts
    of Zurich, Switzerland.
```

1.2 Papers to cite if you use Agilicious

If you use the code in the academic context, please cite:

- Philipp Foehn, Elia Kaufmann, Angel Romero, Robert Penicka, Sihao Sun, Leonard Bauersfeld, Thomas Laengle, Giovanni Cioffi, Yunlong Song, Antonio Loquercio, Davide Scaramuzza,
  "Agilicious: Open-Source and Open-Hardware Agile Quadrotor for Vision-Based Flight",
  AAAS Science Robotics, 2022,
  Video, Bibtex

Additionally, please cite the following papers for the specific extensions you make use of:

- Sihao Sun, Angel Romero, Philipp Foehn, Elia Kaufmann, Davide Scaramuzza,
  "A Comparative Study of Nonlinear MPC and Differential-Flatness-based Control for Quadrotor Agile Flight",
  IEEE Transactions on Robotics, 2022,
  Video, Bibtex
- Philipp Foehn, Angel Romero, Davide Scaramuzza,
  "Time-Optimal Planning for Quadrotor Waypoint Flight",
  Science Robotics, 2021,
  Video, Bibtex
- Antonio Loquercio, Elia Kaufmann, Rene Ranftl, Mark Müller, Vladlen Koltun, Davide Scaramuzza,
  "Learning High-Speed Flight in the Wild",
  Science Robotics, 2021,
  Video, Bibtex
- Leonard Bauersfeld, Elia Kaufmann, Philipp Foehn, Sihao Sun, Davide Scaramuzza,
  "NeuroBEM: Hybrid Aerodynamic Quadrotor Model",
  RSS: Robotics, Science, and Systems, 2021,
  Video, Bibtex
- Elia Kaufmann, Antonio Loquercio, Rene Ranftl, Mark Müller, Vladlen Koltun, Davide Scaramuzza,
  "Deep Drone Acrobatics",
  RSS: Robotics, Science, and Systems, 2020,
  Video, Bibtex

- Matthias Faessler, Antonio Franchi, Davide Scaramuzza,
  "Differential Flatness of Quadrotor Dynamics Subject to Rotor Drag for Accurate Tracking of High-Speed Trajectories",
  IEEE Robotics and Automation Letters, 2018,
  Video, Bibtex

1.3 Dependencies

Strictly necessary dependencies:

- Eigen Library: http://eigen.tuxfamily.org/index.php?title=Main_Page This is absolutely necessary for building the core library.

Dependencies to use all provided functionalities (e.g., run MPC controller online, deploy using a motion capture system,...):

- Acados: https://github.com/acados/acados Framework for our MPC implementations.
- HPIPM: https://github.com/giaf/hpipm High Performance Interior Point Method solver.
- ROS VRPN Client: http://wiki.ros.org/vrpn_client_ros This client can be used to fly within motion capture systems. Install with *sudo apt install ros-${ROS_DISTRO}-vrpn-client-ros*

Optional dependencies (they allow to run unit tests in your code):

- Google Test: https://github.com/google/googletest For unit tests and continuous integration.
- Google Benchmark: https://github.com/google/benchmark For optional micro-benchmarking.

1.4 What's in it for you?

Agilicious is split in two parts, *agilib* and *agiros*.

agilib contains base classes and module implementations providing the base functionality necessary for agile flight. It contains controllers, estimators, and logic, but with minimal dependencies (mainly Eigen). It allows users to integrate Agilicious into their own infrastructure and can easily be extended with new modules. Meanwhile, *agiros* provides bindings to common ROS interfaces, which make it easy and quick to set up and get flying, both in simulation and real world.

In summary, this library offers the following modules:

1.4.1 Pipelines

The Pipeline architecture provides a modular way to combine controllers, estimators, reference trajectories into a complete control system. It is possible to create multiple pipelines and switch between them at runtime, allowing rapid prototyping.

1.4.2 Controllers

- Model Predictive Control descred in Falanga IROS'18 and Sun TRO'22
- Incremental Nonlinear Dynamic Inversion described in Sun TRO'20
- Geometric Control described in Sun TRO'22
- Cascaded PID similar to Faessler RAL'18

1.4.3 Estimators

- A standard EKF using a pose estimate and propagating with IMU measurements.
- An EKF that uses a constant acceleration model and the Pose + IMU measurements.
- A feedthrough module that provides a simple way to pipe in your own estimate sources.
- A mock estimator that can corrupt an estimate with noise and bias to simulate real-world properties.

1.4.4 References

- Hover
- Velocity Commands
- Sampled Trajectories
- Polynomial Trajectories

1.4.5 Samplers

- Time-based sampling along a given reference.
- Positional sampling along a given reference which allows to robustly handle large disturbances.

1.4.6 Bridges

- RotorS interface
- Serial bridge to interface with our own flight control software
- Serial bridge to interface through LAIRD wireless modules.
- Serial bridge to interface with BetaFlight controllers.

1.4.7 Pilot

The Pilot handles all logic and interfacing.

1.4.8 Guard

A safety guard implementation that can use a secondary estimate (e.g. motion capture) and take over if some criteria are invalidated. The guard can use a secondary pipeline configuration without relying on the user-defined modules. This allows for rapid prototyping with a virtual "safety net".

1.4.9 Simulator

A sophisticated and configurable simulation that runs magnitudes faster than real time and optionally provides aerodynamic BEM models as described in Bauersfeld RSS'21.

1.5 Getting Started

If you use ROS, simply clone agilicious into your `catkin` workspace and `catkin build`:

```
cd catkin_ws/src # Create new catkin workspace.
git clone https://github.com/catkin/catkin_simple
git clone git@github.com:uzh-rpg/agilicious.git
catkin build
```

If you want to use the library standalone, you can always:

```
cd agilib/build
cmake ..
make
```

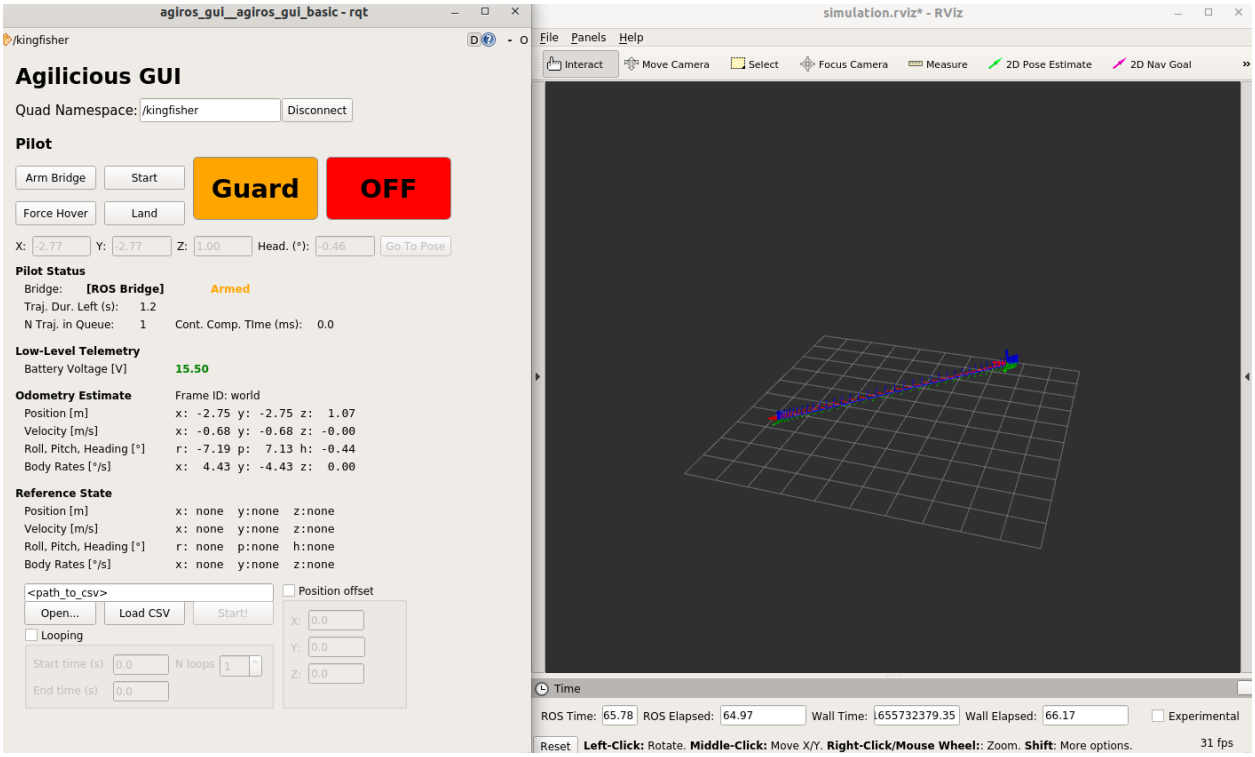
It is highly recommended to run agilicious inside a docker container, instructions are available in *Dockerized Build*.

1.5.1 Launch your first simulation

For launching simulations you need ROS. Once your compilation using `catkin build` has been successful, you can then source your catkin workspace and launch two different types of simulation:

- Agisim simulation, by running the command `roslaunch agiros agisim.launch`. This simulation environment has been written by us and supports different levels of complexity. These levels of complexity can be configured in the file `agiros/agiros/parameters/simulation.yaml`, where, for example, if we would comment the lines with "ModelRotorSimple" and uncomment the lines with "ModelRotorBEM", then a BEM simulation will be launched.
- Gazebo RotorS simulation, by running the command `roslaunch agiros simulation.launch`

These simulation environments allow the user to test their code by offering a user friendly GUI. The simplest task would be a "Go to pose", which can directly be executed by pressing buttons in the GUI. More complex maneuvers can also be performed, like trajectory tracking. For this, the trajectory can be loaded from a `.csv` file



1.5.2 How do I change the characteristics of my simulation?

Everything is structured in `.yaml` files. The most important of these `.yaml` files is the so-called "Pilot" file. For example, when launching the Agisim simulation, the `agiros/agiros/parameters/simple_sim_pilot.yaml` file is used. Here, the user can define which controller to use, which estimator to use, which quadrotor model to use, etc. If one were to change the mass of the quadrotor, for example, one would in this case need to modify the `agiros/agiros/parameters/quads/kingfisher.yaml` file.

1.6 Dockerized Build

Using our provided Dockerfile you can run the entire repository detached from the dependencies of your local system, in a docker container.

The steps to run it from scratch are:

1. Install docker: https://docs.docker.com/engine/install/ubuntu/
2. Docker post installation steps: https://docs.docker.com/engine/install/linux-postinstall/
   Summarized:
   - `sudo groupadd docker`
   - `sudo usermod -aG docker $USER`
   - `newgrp docker`
   - `docker run hello-world`
3. For computers with NVIDIA graphic cards, there are some additional steps:

- Install nvidia-docker2, following this: https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html#setting-up-nvidia-container-toolkit
- Uncomment the `NVIDIA_DOCKER_REQUIREMENT='nvidia-docker2'` part in the `launch_container.sh` file

4. Pull ros image to docker: `docker pull ros`
5. Clone agilicious repository to your host machine, and cd to the agilicious root folder.
   There, you will see a `Dockerfile`. From there, build the docker image using the Dockerfile:
   `sudo docker build --tag "ros_agilicious:latest" .` Run this command from dockerfile directory
6. Run the docker with GUI capabilities and with access to our host repository,
   such that changes in our local repository will have instant effect inside the docker container:
   `./scripts/launch_container <your_catkin_workspace>`
7. Now you have a terminal inside the container with all the dependencies needed to run the code.
   Go to `/home/agilicious/catkin_ws` and run `catkin build`.

1.7 Code Documentation

For code documentation, we now use Doxygen!

At the moment, the documentation files are not provided and have to be built locally:

1. Install doxygen `sudo apt install doxygen`
2. Run doxygen in the root folder `agilicious/` by simply issuing the `doxygen` command.
3. Open the documentation in your favourite browser by clicking `agilicious/docs/html/index.html`!

1.8 Modules Overview

To be completed soon!

1.8.1 The Pilot

The pilot is an `agilib` module implemented by the file `agilib/src/pilot/pilot.cpp`. It serves two main purposes:

- It is the entry point of the program, and therefore it is in charge of coordinating and orquestrating when and how to instantiate and run the `pipeline` module(s).
- It is the main interface point between the `agilib` library and the `agiros` ROS interface. For example, it takes care of handling the commands that come from ROS and converting them into actual actions that are then processed by the running sampler/estimator/controller.

1.8.2 The Guard

The guard is an `agilib` module implemented in `agilib/src/guard/`. The purpose of the guard is to keep the drone safe when things go wrong. It does so by running a completely separate pipeline, with separate sampler, estimator and controller running in parallel to the main pipeline. If something happens (for example, the controller crashes, the main estimator diverges, and the drone flies out of a pre-defined 3D volume), the guard triggers automatically and brings the drone back to safety. The guard is also triggered manually, if the user observes undefined/dangerous behaviour.

1.8.3 The Pipeline Concept

The Pipeline is implemented by `agilib/base/pipeline.cpp`. Its main purpose is to sequentially execute the different components that form a typical control pipeline. It runs at a fixed rate (100 Hz by default) and executes, most typically,

- First, it takes care of getting the latest valid state estimate from the `estimator`
- Then, it takes the active reference trajectory and uses the `sampler` to create a sequence of setpoints to be tracked by the controller.
- Then the `controller` (inner and possibly outer loops) are run. Different types of controllers can be selected, mainly MPC, Geometric Controller or INDI. The controller outputs a low level command.
- Finally, a `bridge` translates the control commands to actual actuator inputs.

1.8.4 Estimators

The `estimator` module group is implemented in `agilib/src/estimator`. There are different types of estimator, depending on the information they use:

- `ekf`: An EKF propagated with a constant acceleration model and updated with 6-DoF pose and IMU measurements.
- `ekf_imu`: An EKF propagated with IMU measurements and updated with 6-DoF pose measurements.
- `feedthrough`: The estimation is provided completely from an external estimator.
- `mock_vio`: A mock estimator that corrupts a ground-truth estimates with noise and bias to simulate real-world properties.

1.8.5 Samplers

The `sampler` module is implemented in `agilib/src/sampler/`. It takes care of taking a trajectory and transforming it into a sequence of setpoints that can be tracked by a controller. There are two types:

- `time_based`: It tracks a trajectory in the time domain. It uses the time allocation of the original trajectory to extract setpoints.
- `position_based`: From the current state of the drone, it searches the closest point to the trajectory and starts the sampling from there.

1.8.6 Reference Trajectories

The `reference` module is implemented in `agilib/src/references/`. It implements different ways of generating reference trajectories:

- `polynomial_trajectories`: Using the eigen library, it implements a way of generating polynomials of arbitrary high order of magnitude that can be constrained in its different derivatives. A cost function can be minimized when solving for the polynomials, allowing for generation of minimum snap, minimum jerk, or other types of polynomials.
- `sampled_trajectories`: Gets a sequence of previously sampled waypoints and converts it into a trajectory.

1.8.7 Controllers

The `controller` module is implemented in `agilib/src/controller/`. There are different kinds of controllers implemented:

- `MPC`: Model Predictive Controller. It uses *acados* with *HPIPM* solver to build an optimization problem that minimizes the distance between the designed reference and the online prediction.
- `geometric_controller`: It implements a differential flatness based geometric controller.
- `PID`: It implements a classical PID controller.
- `INDI`: Inverse Nonlinear Dynamic Inversion controller. It's primarily used for inner loop control (tracking body rates).

1.8.8 Bridges

The `bridge` is module implemented in `agilib/src/bridge/`. The bridge class offers an interface between the commands produced by the controller and the hardware. It uses a `thrust map` to convert the commands to low level signals for the designed low level controller. There are different child classes, depending on the hardware. Some of them are:

- `sbus`: Implements an interface through the SBUS interface.
- `ctrl`: Implements an interface to the `AgiNuttx` low level controller.
- `laird`: Implements an interface that uses `LAIRD` RF modules. It's usually used together with the SBUS bridge in order to get commands from an offboard controller and send them through the SBUS to the low level controller.

1.9 Simulation

The `simulator` module is implemented in `agilib/src/simulator/`. The simulation consists of two seperate parts, a low-level controller and a physics simulation.

The purpose of the low-level controller is to simulate the behavior of the controllers typically found onboard a drone: it takes a collective thrust and a bodyrate command and translates this into individual motor commands for the four motors. Every low-level controller should inherit from the `LowLevelControllerBase` class and implement a `run` function, which has access to the full drone state and the CTBR command and needs to compute desired motor RPMs for each propeller.

The second part is the physics simulation. All such models inherit from `ModelBase` and need to implement a `run` function as well. The function has read-only access to the full state of the drone. It needs to compute the derivative of the state. All models are chained together in a `ModelPipeline` which is then executed by the simulator. All parameters required by a model are stored in a corresponding `[ModelName]params` class.

Available models include:

- `ModelBodyDrag` implements quadratic body drag
- `ModelLinCubDrag` implements linear-cubic body drag (better for MPC applications that also use such a model internally)
- `ModelMotor` implements a first-order system for the motor dynamics
- `ModelPropellerBEM` implements a propeller model based on blade-element-momentum theory which models forces and propeller drag accurately
- `ModelRidgidBody` implements a ridgid body model (required to convert accelerations into velocities and velocities into positions)
- `ModelThrustTorqueSimple` implements a propeller model based on widely used square law between propeller speed and thrust

1.10 Hardware Overview

The Agilicious Platform is born as a product of years of research and countless iterations at the Robotics and Perception Group. The Agilicious Platform consits of the following main components:

- `Nvidia Jetson TX2`: Main compute unit
- `ConnectTech Quasar`: Breakout Board
- `TMotor F7 Flight Controller`: Low-Level Controller
- `F55A Pro II 3-6S 4-in-1 ESC`: Electronic Speed Controller
- `Armattan Chameleon 6"`: Main Plate

- `TMotor Veloc V2306 V2.0`: Motors
- `Azure Power SFP 5148`: Propeller
- `Tattu R-Line 4s 1800mAh 120C`: Battery

1.11 ROS Integration

Our software is split in two parts, called `agilib` and `agiros`. `agilib` is the implementation of the entire library with all its functionalities, while `agiros` serves as an interface layer between `agilib` and ROS.

The user can be referred to the file `agiros/agiros/ros_pilot.cpp` where the interface for subscription and publication is written. For example, some of the most representative (but not only) commands are:

- `enable`: arms the drone
- `start`: commands the drone to take off
- `off`: stops and disarms the drone
- `trajectory`: sends a trajectory that is followed by the drone
- `pose_estimate`: provides the drone with a pose estimate to be fused in the EKF
- `feedthrough_command`: directly sends a low level command that bypasses all controller logic and directly goes to the actuators

The function of the ROS interface is to take these commands (or any command that might be implemented by the user) and send it downstream to `agilib`.

1.12 Flightmare Integration

Flightmare is composed of two main components: a configurable rendering engine built on Unity and a flexible physics engine for dynamics simulation. Those two components are totally decoupled and can run independently from each other. Hence, we can combine **Flightmare** with **Agilicious** for **Hardware-in-the-loop (HITL) simulation**. HITL simulation allows for flying a physical quadrotor in the real-world while observing virtual photorealistic environments.

1.12.1 Download Flightmare

Clone the code:

```
git clone git@github.com:uzh-rpg/flightmare.git
```

Ignore the Flightmare ROS examples:

```
touch flightmare/flightros/CATKIN_IGNORE
```

Download Flightmare Unity Standalone:

```
curl --show-error --progress-bar --location "https://github.com/uzh-rpg/flightmare/
↪releases/download/0.0.5/RPG_Flightmare.tar.xz" | tar Jxf - -C flightmare/flightrender/␣
↪--strip 1
```

1.12.2 Hardware-in-the-loop Simulation

Now, assume you have both Agilicious and Flightmare downloaded. You need to create a new ros package that combines both. We provides some hints on how to use Unity rendering here. For details, you can check this example.

Set up Unity rendering. Example code:

```
// Flightmare Quadrotor and Unity Camera
unity_quad_ = std::make_shared<flightlib::Quadrotor>();
flightlib::Vector<3> quad_size(0.5, 0.5, 0.2);
unity_quad_->setSize(quad_size);

unity_camera_ = std::make_shared<flightlib::RGBCamera>();
flightlib::Vector<3> B_r_BC(0.0, 0.0, 0.3);
Scalar pitch_angle_deg = 0.0;
flightlib::Matrix<3, 3> R_BC =
  (Eigen::AngleAxisf(0.0 * M_PI, Eigen::Vector3f::UnitX()) *
   Eigen::AngleAxisf(-pitch_angle_deg / 180.0 * M_PI,
                     Eigen::Vector3f::UnitY()) *
   Eigen::AngleAxisf(-0.5 * M_PI, Eigen::Vector3f::UnitZ()))
    .toRotationMatrix();
double hor_fov_radians = (M_PI * (110 / 180.0));
double width = 640.0;
double height = 480.0;
// Recalculate here: https://themetalmuncher.github.io/fov-calc/;
double vertical_fov =
  2. * std::atan(std::tan(hor_fov_radians / 2) * height / width);
vertical_fov = (vertical_fov / M_PI) * 180.0;  // convert back to degrees
std::cout << "Vertical FoV is " << vertical_fov << std::endl;
unity_camera_->setFOV(vertical_fov);
unity_camera_->setWidth(width);
unity_camera_->setHeight(height);
unity_camera_->setRelPose(B_r_BC, R_BC);
unity_quad_->addRGBCamera(unity_camera_);

// Connect Unity
setUnity(unity_render_);
connectUnity();
```

Set Unity:

```
bool setUnity(const bool render) {
  unity_render_ = render;
  if (unity_render_ && unity_bridge_ == nullptr) {
    unity_bridge_ = flightlib::UnityBridge::getInstance();
    unity_bridge_->addQuadrotor(unity_quad_);
    //
    if (!loadRacetrack(pnh_)) {
      ROS_WARN("[%s] No race track is specified.", pnh_.getNamespace().c_str());
    } else {
    };

    ROS_INFO("[%s] Unity Bridge is created.", pnh_.getNamespace().c_str());
    return true;
```

(continues on next page)

(continued from previous page)

```
  } else {
    return false;
  }
}
```

Connect Unity:

```
bool connectUnity() {
  if (!unity_render_ || unity_bridge_ == nullptr) return false;
  unity_ready_ = unity_bridge_->connectUnity(unity_scene_id_);
  return unity_ready_;
}
```

The images are rendered based the current quadrotor states. Below are example codes for publishing images. Publish images:

```
void publishImages(const QuadState &state) {
  sensor_msgs::ImagePtr rgb_msg;
  frame_id_ += 1;
  // render the frame
  flightlib::QuadState unity_quad_state;
  unity_quad_state.setZero();
  unity_quad_state.p = state.p.cast<flightlib::Scalar>();
  unity_quad_state.qx = state.qx.cast<flightlib::Scalar>();
  unity_quad_->setState(unity_quad_state);
  // Warning, delay
  unity_bridge_->getRender(frame_id_);
  unity_bridge_->handleOutput(frame_id_);
  cv::Mat img;
  unity_camera_->getRGBImage(img);
  rgb_msg = cv_bridge::CvImage(std_msgs::Header(), "bgr8", img).toImageMsg();
  rgb_msg->header.stamp = ros::Time(state.t);
  image_pub_.publish(rgb_msg);
}
```