

# Getting started with **MIT's** **Rolling Spider MATLAB Toolbox**



with Parrot's Rolling Spider Drone!

An MIT take-home lab for  
16.30 Feedback Control Systems



Massachusetts Institute of Technology



MIT's

# Rolling Spider MATLAB Toolbox



with Parrot's Rolling Spider Drone!

... let's you **design** and **simulate** estimation and control algorithms for a drone in MATLAB/Simulink and **autogenerates** embedded c-code that you can use to actually **fly** the drone! After your flight, recorded data can be **visualized** and analyzed.



Massachusetts Institute of Technology



**MIT's**

# **Rolling Spider MATLAB Toolbox**



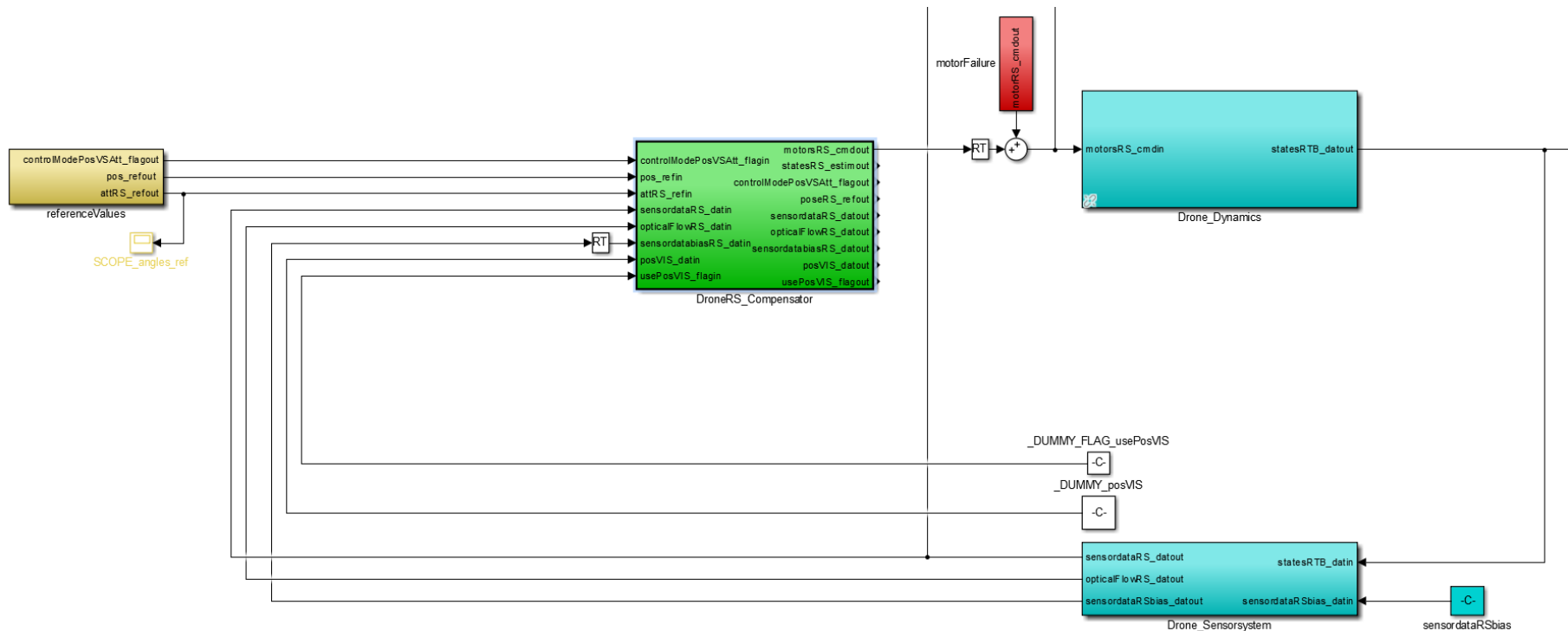
with Parrot's Rolling Spider Drone!

\* keep in mind that this toolbox is for educational purposes. It is therefore rather tuned to be easily understood than to meet software engineering standards and amazing flight performance. The first version is mainly designed for experimenting with hover flight.

--

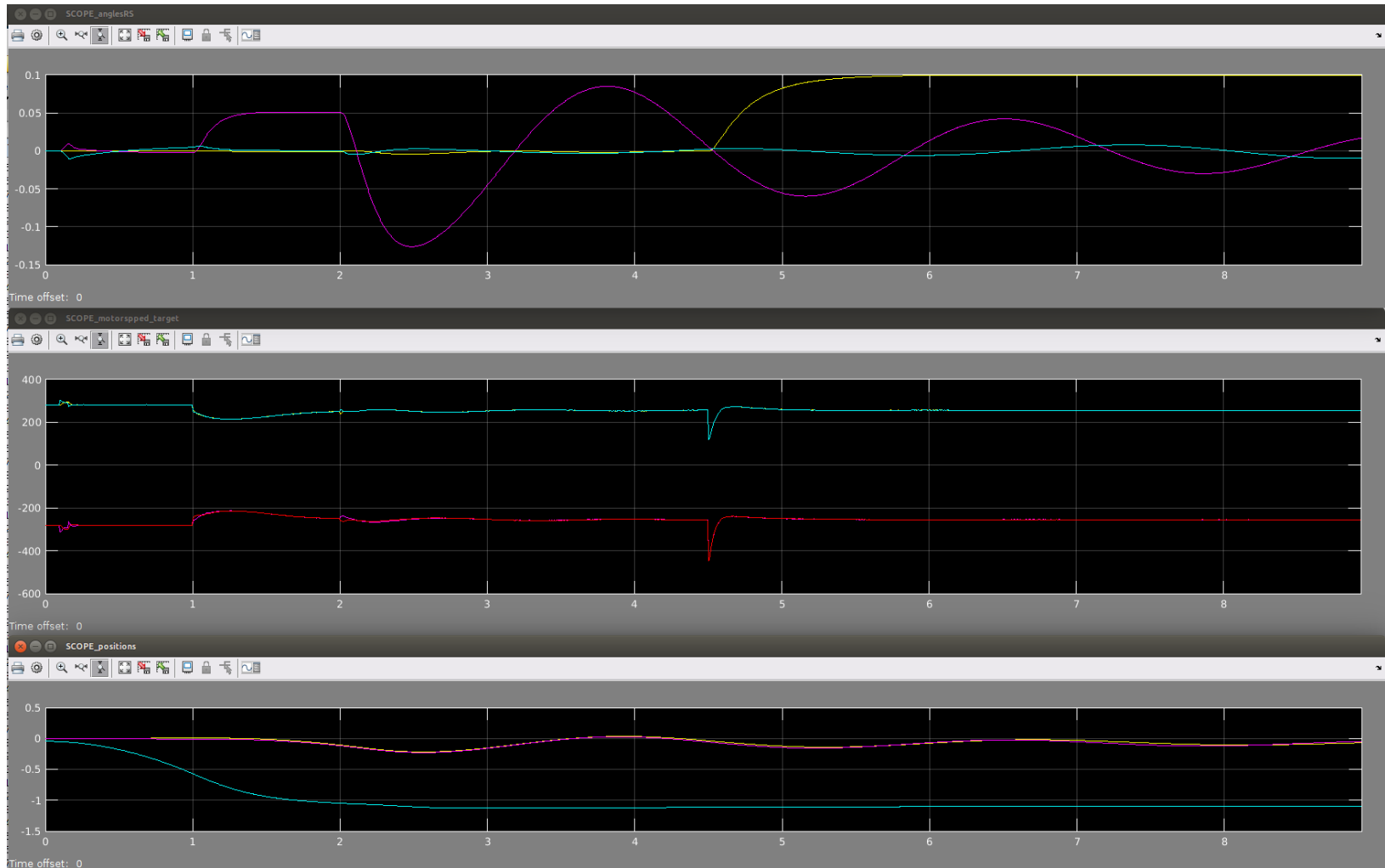
<https://github.com/Parrot-Developers/RollingSpiderEdu> for more information

# Simulate with a full Simulink model



SIMULINK model of drone's dynamics, sensor system and compensator

# Plot data from simulated flight

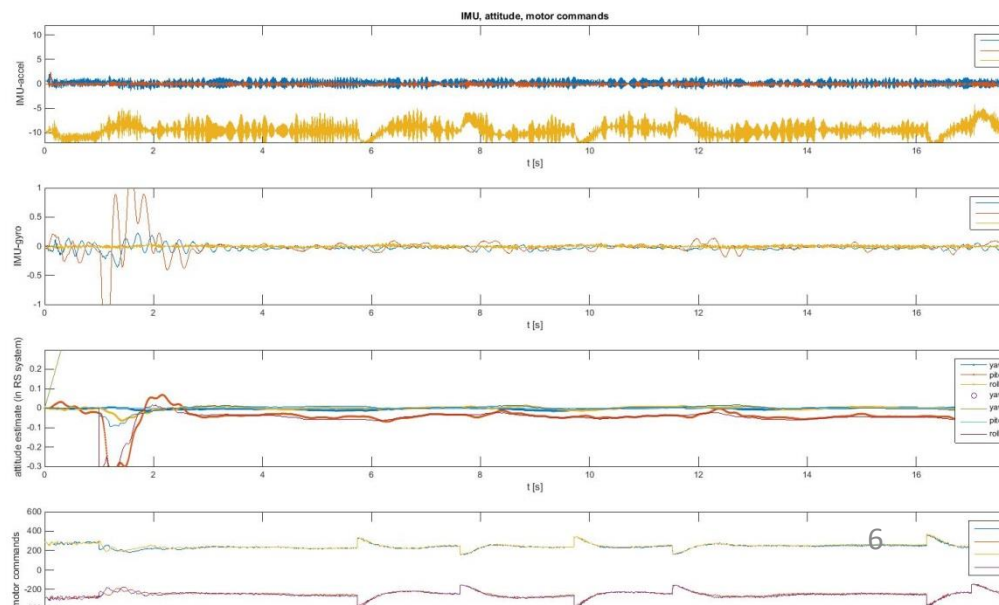
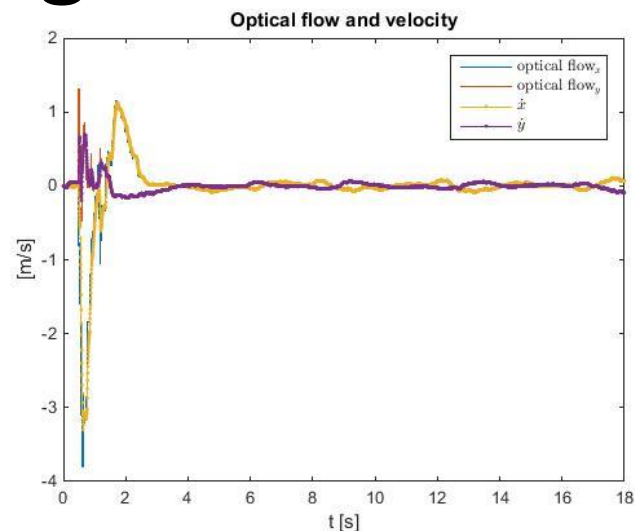
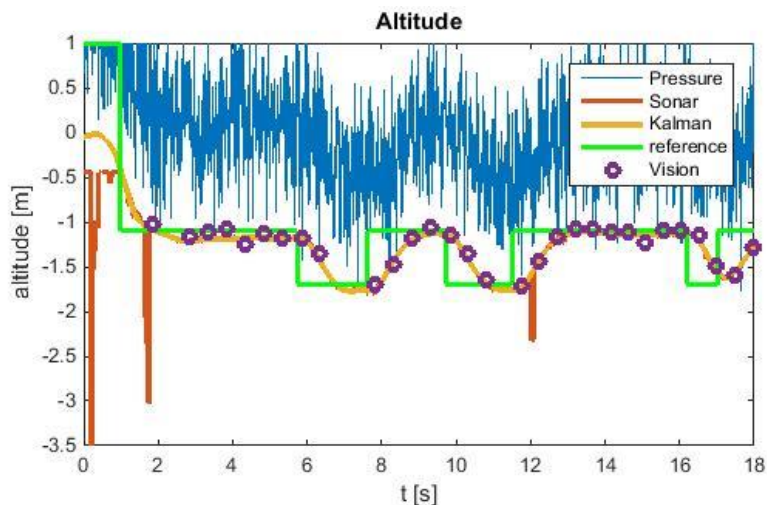


Plotting orientation, motor commands and positions from simulated flight



Massachusetts Institute of Technology

# Plot data from real flight



# Contents

- Drone Hardware
- Toolbox *„What’s the workflow?“*
  - Installation
  - Simulation and control design
  - Embedded code generation
  - Flying
  - Data Analysis
  - Dynamics Analysis
  - Beta-Feature: Vision
  - Resetting the drone firmware
- Software architecture *„How did we hack it?“*
- Troubleshooter’s FAQ

# Drone Hardware

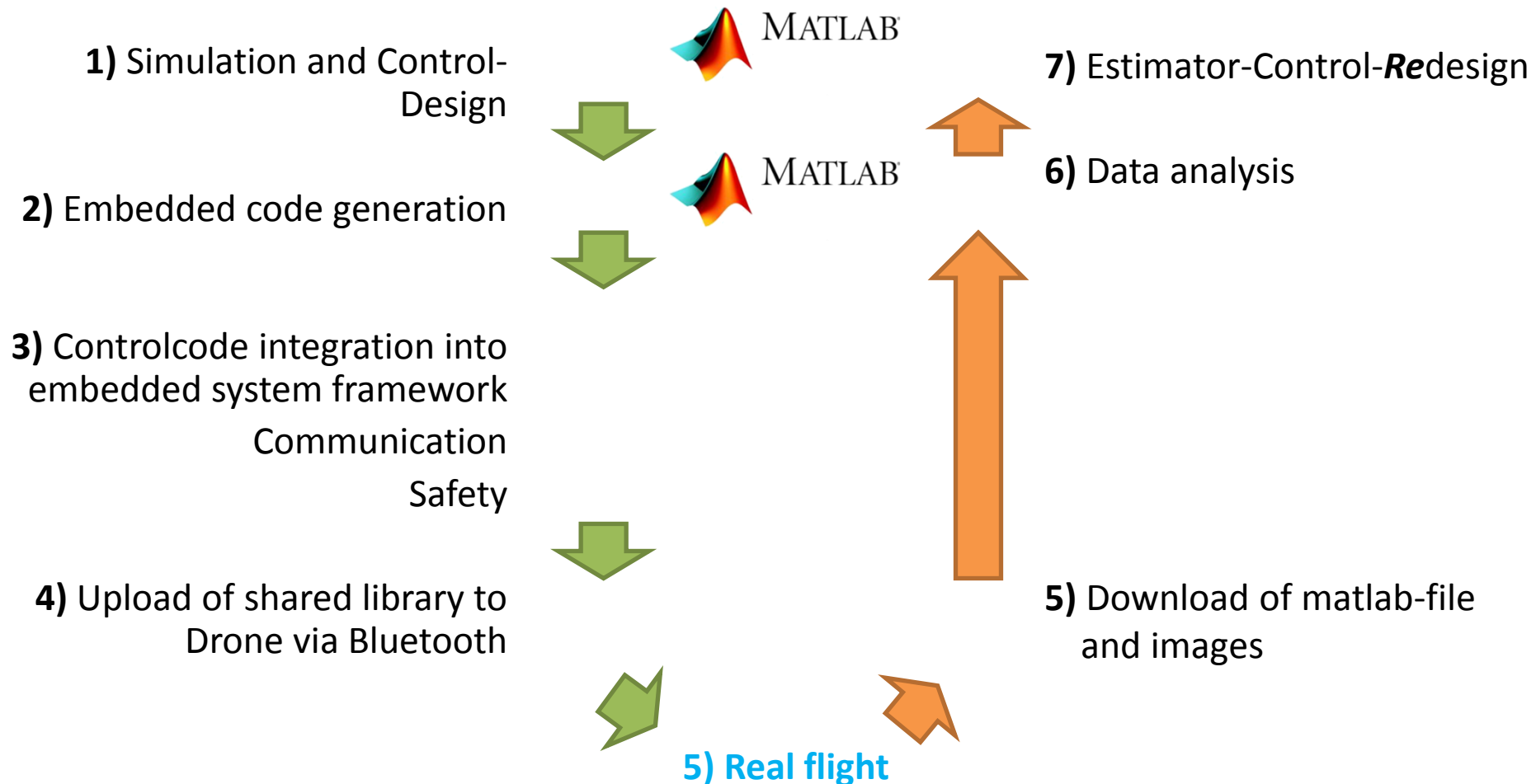
- Parrot Rolling Spider (note: not the end2015-EVO version!)
  1. Mass: 68g
  2. Motors: 33g Thrust/motor
  3. embedded linux system
  4. IMU: 6axis-accelerometer-gyroscope
  5. Altitude sensors: Sonar, Pressure sensor
  6. Vision: Downward-facing camera, 160x120
  7. Battery: 7-8min flight time
- Bluetooth BLE adapter (if your laptop does not provide it,  
e.g. IOGEAR Bluetooth 4.0 USB Micro Adapter (GBU521))
- Safety goggles
- Optional
  1. Additional battery and charger
  2. Extra set of propellers





# Toolbox

*„What's the workflow?“*





# Toolbox

A step-by-step  
tutorial to guide you from  
*installation to simulation to flight*

# Toolbox

## *Installation I: Equipping your ubuntu (once) (1/2)*

- The toolbox was designed on Ubuntu 14.04. If you don't use ubuntu natively, you can run it as a virtual machine (Windows: VMware, Mac: VMWareFusion)
- Let *[ROSMAT]* denote the path to the MIT toolbox root folder (i.e. the folder containing the README- and LICENSE file of MIT\_ROSMAT).
- Your ubuntu-system should be equipped with the following programs:
  1. MATLAB 2015
  2. Lftp  
`sudo apt-get install lftp`
  3. Bluetooth stack  
`sudo apt-get install bluez-compat`
  4. expect  
`sudo apt-get install expect`
  5. MIT's ROSMAT: Checkout <https://github.com/Parrot-Developers/RollingSpiderEdu>. Let *[ROSMAT]* be the path to the MIT-toolbox root folder containing the README- and LICENSE-file.
  6. Unpacked Gcc-arm-Toolchain to the file system root, so you have e.g. */opt/arm-2012.03/libexec* (files can be found in *[ROSMAT]/libs/gcc-arm-Toolchain*)  
- on 64bit systems, also install the following programs  
`sudo apt-get install lib32z1 lib32ncurses5 lib32bz2-1.0`
  7. To add toolbox binaries folder to PATH:  
`sudo gedit ~/.profile`  
Append this line, save the file, then lock out of ubuntu and back in again.  
`export PATH=$PATH:[ROSMAT]/bin:[ROSMAT]/bin/utils:[ROSMAT]/bin/firmware`
  8. Finally, build utils programs  
`BuildUtils.sh`

# Toolbox

## *Installation I: Equipping your ubuntu (once) (2/2)*

- Using the virtual machine provided for the MIT class *16.30 Feedback Control Systems* virtual machine, you should only need to activate your MATLAB and build utils programs (`BuildUtils.sh`). Have your Mathworks student account ready (check MIT's IST webpage for MATLAB) and open MATLAB via the desktop icon shortcut (you might have to run it from a terminal with `sudo matlab`). Sidenote, here, [ROSMAT] is `~/RollingSpiderEdu-master/MIT_MatlabToolbox`.
- The publicly available virtual machine includes all parts that the MIT virtual machine does apart from MATLAB.
- Info on MATLAB toolboxes:  
recommended: Communications System Toolbox, Computer Vision System Toolbox, Control System Toolbox, Embedded Coder, Fixed-Point Designer, MATLAB Coder, MATLAB Compiler, MATLAB Compiler SDK, Signal Processing Toolbox, Simulink, Simulink Coder, Simulink Control Design, Stateflow, Symbolic Math Toolbox  
additionally part of MIT's 16.30 VirtualMachine for MIT students: Curve Fitting Toolbox, DSP System Toolbox, Fuzzy Logic Toolbox, Global Optimization Toolbox, Image Acquisition Toolbox, Image Processing Toolbox, Instrument Control Toolbox, MATLAB Report Generator, Model Predictive Control Toolbox, Optimization Toolbox, Robotics System Toolbox, Robust Control Toolbox, Simscape, Simulink 3D Animation, Simulink Design Optimization, Simulink Design Verifier, Simulink Report Generator, Simulink Verification and Validation, System Identification Toolbox, Vision HDL Toolbox, Wavelet Toolbox

# Toolbox

## *Installation II: Flashing the drone (once) (1/3)*

The consumer drone has to be flashed with a custom firmware *once*.

1. Connect drone via USB (if using a virtual machine, make sure to connect to ubuntu)
2. Open *fvt6.txt* on drone USB, note down name and MAC address.  
(If no *fvt6.txt* can be found, skip step 3 for now and run `sudo hcitool scan` after step 9. Your drone should be listed, read the MAC address from there. Then do step 3 and continue with step 10 afterwards.)
3. Save MAC address to *DroneMACaddress.txt* by entering, in a terminal  
`DroneSetMACaddress.sh [MACADDRESS]`
4. Upload main firmware to drone by running  
`EDUfirmwareUploadSYS.sh`  
(Info: This script copies *rollingspider.edu.plf* to root folder of drone USB device)
5. Disconnect drone by ejecting USB device and removing USB cable
6. Charge battery
7. Insert battery
8. Wait until LEDs stopped blinking (firmware is now updated)  
(Note: If LEDs never blinked, redo step 1 & 3-8.)
9. Plug in bluetooth adapter (if necessary)

**...continue on next slide!**

Videotutorial: 01\_FlashingTheDrone

# Toolbox

## *Installation II: Flashing the drone (once) (2/3)*

10. Connect drone to computer by running  
`DroneConnect.sh`
11. Upload firmware files by running  
`EDUfirmwareUploadFILES.sh`  
(Info: uploads files in `[ROSMAT]/libs/EDUfirmwareFILES` to drone via ftp and IP 192.168.1.1)
12. Reboot drone  
`DroneReboot.sh`

**...continue on next slide!**

Videotutorial: 01\_FlashingTheDrone

# Toolbox

## *Installation II: Flashing the drone (once) (3/3)*

13. Connect drone again

`DroneConnect.sh`

14. Initialize drone firmware

`EDUfirmwareInitialize.sh`

(Info: This script moves firmware files to right locations and grants permissions rights:

```
mv /data/edu/dragon-prog /usr/bin/  
chmod +x /usr/bin/dragon-prog  
mv /data/edu/SpiderFlight.sh /bin/  
chmod +x /bin/SpiderFlight.sh)
```

15. Initialize drone

`DroneInitialize.sh`

(Info: This script write the computer's IP address to the drone's parameter file)

16. Done with flashing. Nice!

Videotutorial: 01\_FlashingTheDrone

# Toolbox


*(Dis-)Connecting to the drone (after restarts, ...)*

- If you want to disconnect, run in a terminal  
`DroneDisconnect.sh`
- To connect run in a terminal  
`DroneConnect.sh`



# Toolbox

## Workflow I: Simulation and Control Design

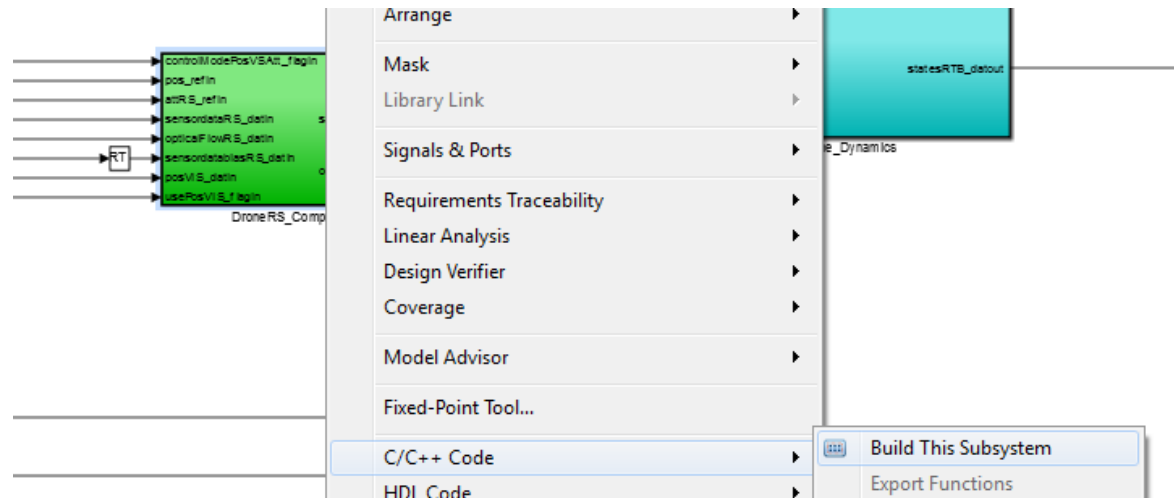
1. Open MATLAB and navigate to the `[ROSMAT]/trunk/matlab/` folder
  - `Simulation/` contains the Simulink files to design and simulate the drone with its estimators and controllers.
  - `libs/` contains parts of Peter Corke's Robotics Toolbox to simulate the dynamics of a drone, updated to (somewhat) match Parrot's Rolling Spider
  - `ExperimentAnalyzer/` contains various files to analyze sensor and dynamics data recorded while flying, or processing times from threads running on the drone.
2. Run `startup.m`, then open `sim_quadrotor.slx`
3. Design your controllers.  
 As a first approach, copy-paste a preset controller: Open `controllers/controller_PID/controller_PID.slx`, copy the `ControllerPID` block and insert it at the correct place in `sim_quadrotor.slx`.  
 For further design, Simulink can be used (mostly) freely, but keep in mind that c-code for a drone with low processing-power will be generated. (See section "Troubleshooter's FAQ" for more hints).  
*Do not change the input/output-ports of the Drone\_Compensator to avoid manual changes in the resulting c-code.*
4. Open SCOPES to have variables plotted, press  to simulate  
 Or: After simulating, go to MATLAB, type `FlightAnalyzer`

Videotutorial: 02\_DesigningControllers

# Toolbox

## Workflow II: Embedded code generation (1/2)

1. Rightclick on the *Drones\_Compensator* block, select „*Build This Subsystem*“.



In the pop-up dialogue box, click *Build*.

Videotutorial: 02\_DesigningControllers

# Toolbox

## Workflow II: Embedded code generation (2/2)

### 2. Upload your controller

(If disconnected from drone: `DroneConnect . sh` first)

`DroneUploadEmbeddedCode . sh`

(Info: this script packs the autogenerated code with the drone's c-code framework using the binary `PackEmbeddedCode`, builds the code with `make` in `[ROSMAT]/trunk/embcodes/build-arm` and uploads new shared library `[ROSMAT]/DroneExchange/librsedu.so` to drone using `ftp` to `192.168.1.1`)

- *Expert level* - With altered Simulink input/output-ports:  
Do the steps from step 2 manually. After running `PackEmbeddedCode` in its folder, replace code paragraph "Input/Outputport Declarations IO(x)"... of SIMULINK compensator block in `rsedu_control.c` with input/output-port declarations found in `ert_main.c`. Note: You also have to update function calls for initializing, stepping and packing the model in `rsedu_control.c`; found in `rsedu_control.c` with comments "IO(2)" and "IO(3)"

Videotutorial: 02\_DesigningControllers

# Toolbox

## *Workflow III: Flying (1) – Flight Phases*

The drone's flight is split into 3 phases

1. Sensor calibration  
Drone sits on the floor for 2 seconds to calibrate its sensors.
2. Take-off  
Take-off for 1 second with given power and attitude control only
3. Actual flight

# Toolbox

## Workflow III: Flying (2) – Safety Procedures

- Make yourself aware of issues described on <https://github.com/Parrot-Developers/RollingSpiderEdu/issues> and read the *Troubleshooter's FAQ* section in this document (p. 33)
- Don't charge batteries unattended
- Ensure people, animal, property, etc. safety
- Stick to Parrot's safety guidelines (see print-out)
- Wear safety glasses all the time
- Always fly with wheels installed
- Only fly indoors, open area >10'x10' over non-glossy floor
- Always test a new program with *DroneTest.sh* first, instead of *DroneRun.sh*
- Stick to software safety procedures (p.22)
- Be smart!

# Toolbox

## Workflow III: Flying (3) – Software Safety

- If the drone's main script does not crash itself, it shuts down the motors in case of a crash or a loss of optical flow
- A single flight is aborted automatically after 20 seconds (see section *Troubleshooter's FAQ*)
- For all other cases (and they will happen!)
- Always have a separate terminal open, enter `telnet 192.168.1.1` (you should already be connected to the drone), now you are logged directly onto the drone.

Type

```
killall -s SIGKILL dragon-prog; gpio 39 -d ho 1; test-SIP6_pwm -S;
```

and be ready to execute this line when the drones goes crazy!

# Toolbox

## Workflow III: Flying (4) – Settings

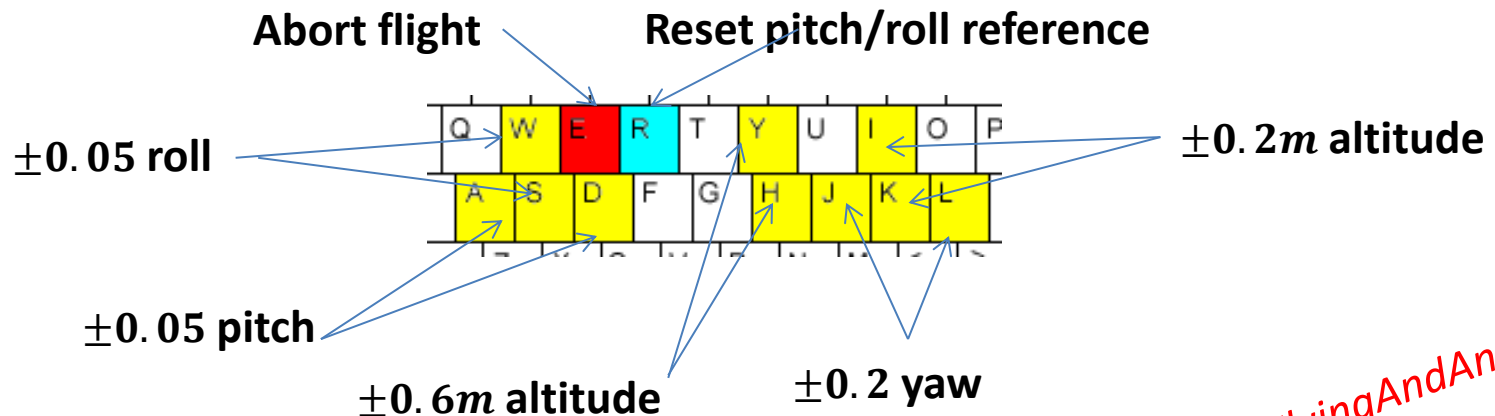
If you want to enable/disable software features

- With the drone connected, log onto drone in a new terminal `telnet 192.168.1.1`, then  
`vi /data/edu/params/paramsEDU.dat`
- Enable features with replacing '0' by '1'
  - FEAT\_TIME: records timestamps for entering and leaving the functions *rsedu\_control*,  
*rsedu\_of* (optical flow) and *rsedu\_vis* (visual position reconstruction)
  - FEAT\_OF\_ACTIVE: optical flow is used to stabilize position
  - FEAT\_POSVIS\_RUN: camera looks for landmark setup and reconstructs position if all  
landmarks found; visually reconstructed position is recorded
  - FEAT\_POSVIS\_USE: use visually reconstructed position to enhance kalman position  
estimate
  - FEAT\_NOLOOK: compute color conversion, landmark matching etc. online instead of using a  
precomputed lookup-table (don't use this, too slow)
  - FEAT\_IMSAVE:
    - 1: saves images (camera runs at 60Hz, images being recorded/saved at 10Hz)
    - 2: images are being streamed to ubuntu machine (see *rsedu\_vis.c* for  
additional instructions)
- FEAT\_NOSAFETY: 1: drone is not automatically shut down when take off-surface is not level, z-axis –  
acceleration is positive or x-y-accelerations exceed  $6\text{m/s}^2$  (dangerous setting!)
- POWERGAIN cannot be changed manually

# Toolbox

## Workflow III: Flying (5)

1. Start KeyboardPilot, i.e. the server providing the reference values, with `DroneKeyboardPilot.sh`
2. In another terminal
  - `DroneTest.sh` for a test run with 10% Power
  - `DroneRun.sh` for a full run
3. Go back to the KeyboardPilot's terminal , **hit** keyboard buttons  
(do not keep pressing them!)



Videotutorial: 03\_FlyingAndAnalyzing...



# Toolbox

## *Workflow IV: Data Analysis – FlightAnalyzer*

1. Download *RSdata.mat* from drone via ftp to *[ROSMAT]/DroneExchange/* by running `DroneDownloadFlightData.sh`  
(Alternatively (faster), connect drone via usb and run `DroneDownloadFlightDataUSB.sh`)
2. In MATLAB, load *RSdata.mat* (double-click)
3. Run MATLAB-script `FlightAnalyzer`

Videotutorial: 03\_FlyingAndAnalyzing...

# Toolbox

## *Workflow IV: Data Analysis – Software in the Loop*

Instead of a full-stack simulation, feed *recorded* sensor data through the Simulink *Drones\_Compensator* block to see what happened under the hood of estimators and controllers during the recorded flight.

1. Make sure to have loaded some flight data *RSdata.mat* and have run the *FlightAnalyzer* with recorded data once
2. In MATLAB, navigate to *[ROSMAT]/trunk/matlab/Simulation*
3. Use Simulink model *sim\_SoftwareInTheLoop\_Compensator.slx*

# Toolbox

## *Workflow IV: Data Analysis – Processing Times*

1. Download folder *ptimes/* from drone with `DroneDownloadPTimes.sh`
2. Run matlab script `PTimesAnalyzer`

# Toolbox

## *Workflow V: Dynamics Analysis*

- Check the pole placement fullstate controllers in *[ROSMAT]/trunk/matLab/Simulation/controllers/* to see examples how to utilize MATLAB and Simulink to linearize dynamics to design fullstate controllers

# Toolbox

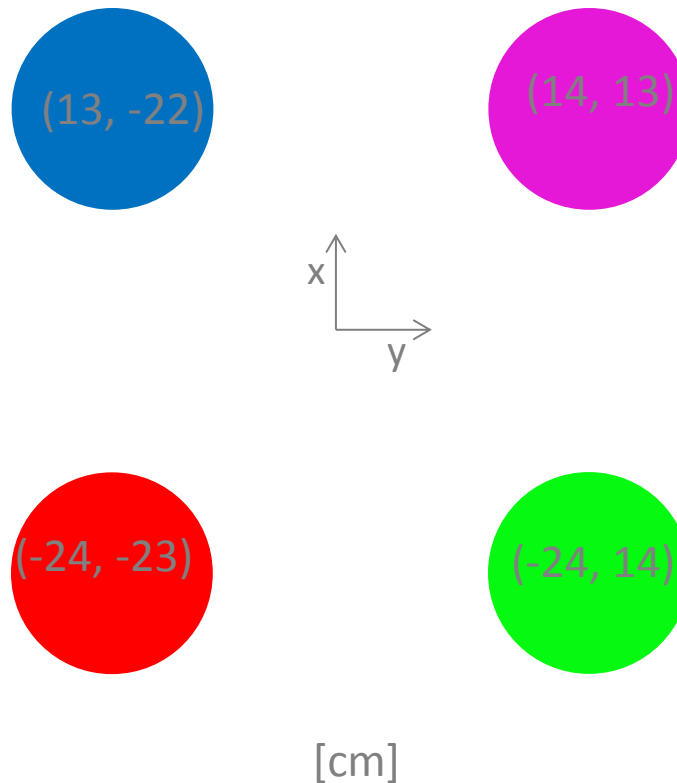
## Workflow VI: Betafeature-Vision

- Recording and postprocessing images:  
Enable the image-save feature to record images (p. 24), fly.  
Then, process recorded images: Download recorded, binary images, convert to ppm and save vision-inferred poses into pose.txt
  1. Download binary images from drone  
`DroneDownloadImages.sh`
  2. Run  
`VisionPrePostProcessor.sh`  
Follow instructions
  3. See `[ROSMAT]/DataExchange/imgs/processed` for ppm, poses, and landmark-identification images.
- Updating landmark setup:  
When using a different landmark setup, use  
`[ROSMAT]/trunk/VisionPrePostProcessor/findPostReconstructionParameters.m` to compute new vision matrices, then update them in `[ROSMAT]/trunk/embcode/rsedu_vis_helpers.c`
- The landmark's descriptors (HSV-values) might need to be updated  
(`VisionPrePostProcessor/main_offboard_image.c`)

# Toolbox

## *Workflow VI: Betafeature-Vision*

- Landmarks are colored markers on the floor
- Standard setup:



# Toolbox

## *Resetting the drone*

- To fully reset the drone to its original state,
  - Do a software update via <http://www.parrot.com/usa/support/parrot-rolling-spider/>: Click on "Software Update" and then "Download", follow the instructions on the webpage.

# Software Interface

*„How did we hack it?“*

- **Drone calls our control code @200Hz,**

input: sensor data      output: motor commands

```
void RSEDU_control(HAL_acquisition_t* hal_sensors_data,  
HAL_command_t* hal_sensors_cmd)
```

- **Drone calls our image processing code @60Hz**

input: image buffer

```
void RSEDU_image_processing(void * imagebuffer)
```

- **Drone calls our optical flow code @60Hz**

input: computed optical flow

```
void RSEDU_optical_flow(float vx, float vy, float vz, int  
defined, float qualityIndicator)
```



More : [media/SoftwareArchitecture.pdf](#)





Massachusetts Institute of Technology

# Toolbox



*Troubleshooter's FAQ*



Massachusetts Institute of Technology



# Toolbox

*Troubleshooter's FAQ:*

*Work flow*

- After pulling a new version from the official github, make sure to run `BuildUtils.sh`

# Toolbox

## *Troubleshooter's FAQ: Motors, Crashes*

- A low battery level can cause multiple problems...
  - ... the takeoff is slower - unfortunately, sonar and vision measurements work best above 0.5m. With a slower takeoff, the drone cannot reliably use those measurements for a longer time!
  - ... the motors are "weaker", the control impact therefore too.
  - ... the CPU behavior can become unpredictable.
  - Just having the drone up and running (without flying) drains the battery - it is an embedded computer after all.
  - Bottom line: Always charge your second battery while working with the first one.
- Flight time is currently limited by software to 20 seconds for a single flight. This can be extended (*rsedu\_control.c* , variable `onCycles`). However, note that recoding data takes space (especially when images are recorded). There is only a single-digit mb space available on the drone. Also note that you might have to increase the data buffer size for logging FlightAnalyzer-ready data (see *make-file* in *embcode/build\_arm*)
- If drone says "RSEDU IP not found", reupload your code (the underlying problem is that the uploaded shared library is not fully recognized (usually linker issues))
- Motors may not start because of default override from manufacturer firmware. This can occur with the drone having been upside down, too high currents while crashing or a low battery. Try starting the drone again, or reboot it with new battery.
- After a crash, check the plots of all sensor for irregularities.

# Toolbox

## *Troubleshooter's FAQ:*

## *Simulation, State Estimation*

- The underlying dynamics are modeled using Euler angles. Due to singularities flown maneuvers have to be limited to  $\pm 90^\circ$  for pitch and roll.
- Note that the position estimate is reset to zero when switching back to position control from attitude control.
- Note that the bias of the IMU is not iteratively estimated but considered to be constant during a flight. It is inferred from data gathered right before takeoff – make sure to start on level ground.
- The kalman filters have outlier handling procedures based on the deviation of the current state estimates to recent measurements. This kind of outlier handling is overly simple. Also, it can make the estimators useless for aggressive maneuvers.
- Ideas for your simple tweaks: If Kalman-estimates and measurement updates differ largely over a significant period of time, consider resetting kalman-estimates.
- Ideas for your simple tweaks : Make use of the pressure sensor to detect outliers of the sonar measurement (e.g. caused by obstacles). Be careful with how to deal with calibrating pressure offset and drift. E.g., Check if jumps of the current sonar-based altitude estimate are realistic/valid. If not, switch to pressure-based estimate and possibly back to the sonar-based one once that is close to the pressure-based one again.

# Toolbox

## *Troubleshooter's FAQ:*

## *Model Accuracy*

- The optical flow module is modeled (in SIMULINK) very far from the real implementation. Still, the optical flow is the main input to estimate the drone's velocity. Now, if we tune controllers in simulation such that they respond to velocity in a very sensitive way, we will likely run into problems running that controller on the *real* drone (because we tuned the controller to a state (velocity) that is badly estimated (on the real drone) and has low model-accuracy!).  
If the plant model was known with perfect accuracy, we could design a controller that maxes out the performance. However, with a real-world system that is modeled with inaccuracies, there is a need for robustness, achievable with trading of controller performance. You will notice this when tuning the drone's controller gains in the simulation. If you max out gains in simulation, the real drone will be unstable!
- In a simulation without drag, the sensed acceleration by the accelerometer will always point in the direction of the z-axis. Therefore, correcting the drift of the integration of angular rates from the gyro (to estimate the attitude) would only be possible in perfect hover, or with making use of the current attitude estimate. In real life, drag brings the drone into a steady-state with zero acceleration so we can infer the attitude from the direction of the gravitational vector.
- The drone does have a motor lag, i.e. the propeller rate does not change instantly. This effect is not modeled in the simulation and not addressed by the controllers. A lead compensator could help compensating for this lag.

# Toolbox

## *Troubleshooter's FAQ:*

## *From Simulation to C-Code*

- Set simulation time to *inf* (otherwise the onboard controller on the drone will be limited to the simulation time you set in the Simulink simulation)
- Avoid SIMULINK blocks that require zero-crossing detection.
- Avoid huge matrix multiplications.
- Avoid logging a lot of signals.
- Be smart, your SIMULINK model has to run on-board with limited computing power!
- Do not move files with MATLAB's file explorer, it might mess with file permissions



Massachusetts Institute of Technology



# Toolbox

*Troubleshooter's FAQ:*

*Beta-feature Vision*

- The camera pose reconstruction algorithm assumes zero pitch and roll angle when seeing the landmarks.