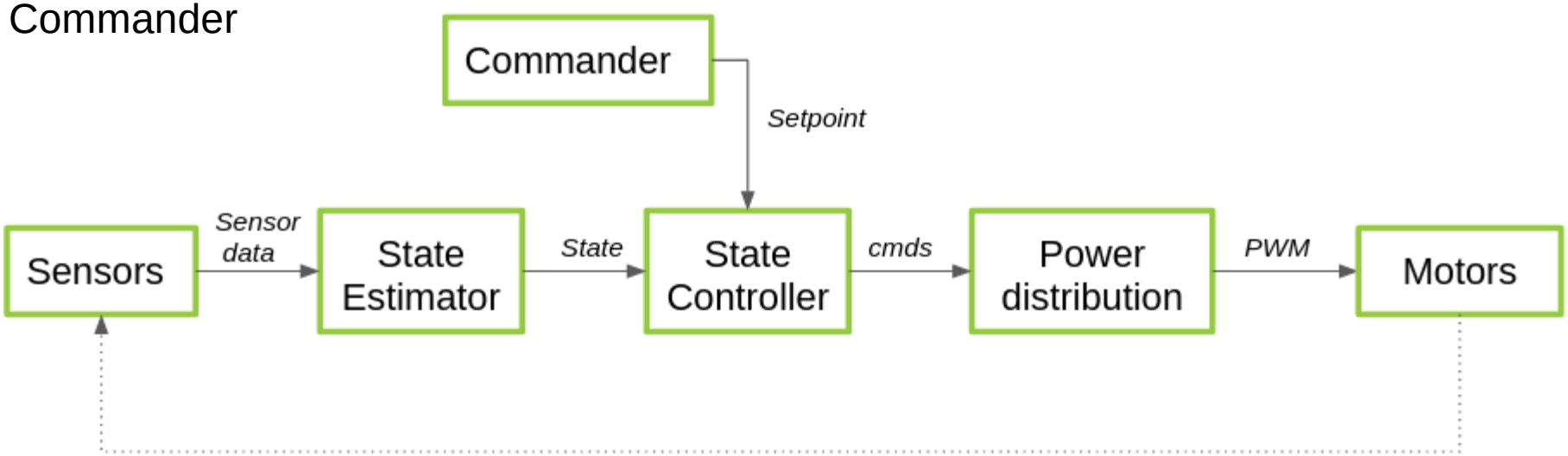
Crazyflie: The Firmware and the Client

***V. Niculescu*, H. Müller, D. Palossi**
Integrated Systems Laboratory – ETH
2nd of October 2020, Zürich

Stabilizer Block Diagram

The stabilizer represents the path from sensor acquisition to motor control.
It is the entity which ensures that the drone is properly flying. Elements:

- The State Estimator
- The State Controller
- The Commander



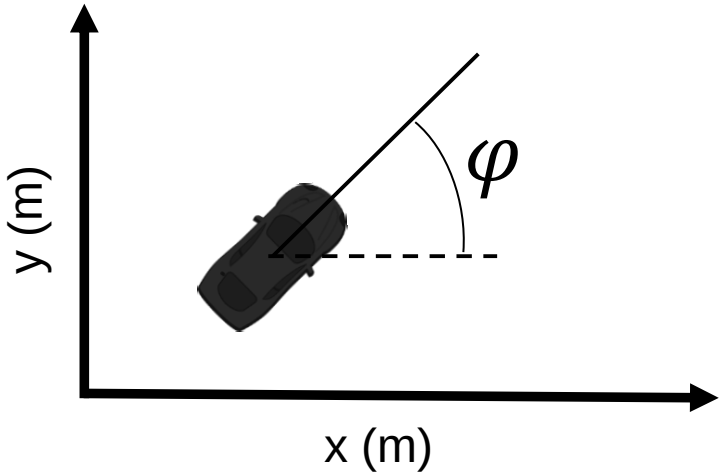
The State Space and the System Dynamics

- **STATE**: A set of variables used to describe the behavior of the system at a particular time.
- **DYNAMICS**: A set of linear/non-linear equations which describe the evolution of the system state.

1D moving object

$$\boldsymbol{x} = \begin{pmatrix} x \\ \dot{x} \end{pmatrix}$$

2D moving object



$$\boldsymbol{x} = \begin{pmatrix} x \\ \dot{x} \\ y \\ \dot{y} \\ \varphi \end{pmatrix}$$

The State Space and the System Dynamics

- **STATE**: A set of variables used to describe the behavior of the system at a particular time.
- **DYNAMICS**: A set of linear/non-linear equations which describe the evolution of the system state.

$$\boldsymbol{x} = \begin{pmatrix} x \\ \dot{x} \end{pmatrix}$$

Dynamics:

Ideal: $x(k+1) = f\big(x(k), u(k)\big)$

Real: $x(k+1) = f\big(x(k), u(k)\big) + \rho, \quad \rho \sim N(\mu, \sigma)$

The State Space and the System Dynamics

- **STATE**: A set of variables used to describe the behavior of the system at a particular time.
- **DYNAMICS**: A set of linear/non-linear equations which describe the evolution of the system state.

$$\boldsymbol{x} = \begin{pmatrix} x \\ \dot{x} \end{pmatrix}$$

Dynamics (ideal case):

$$x(k+1) = x(k) + \dot{x}(k) \cdot \Delta t + \frac{\ddot{x}(k) \cdot \Delta t^2}{2}$$

$$\dot{x}(k+1) = \dot{x}(k) + \ddot{x}(k) \cdot \Delta t$$

The State Space and the System Dynamics

Dynamics:

$$x(k+1) = x(k) + \dot{x}(k) \cdot \Delta t + \frac{\ddot{x}(k) \cdot \Delta t^2}{2}$$

$$\dot{x}(k+1) = \dot{x}(k) + \ddot{x}(k) \cdot \Delta t$$

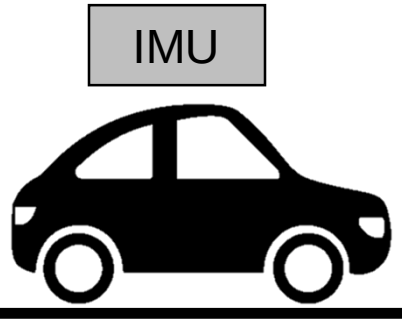
$$k = 0$$

The State Space and the System Dynamics

Dynamics:

$$x(k+1) = x(k) + \dot{x}(k) \cdot \Delta t + \frac{\ddot{x}(k) \cdot \Delta t^2}{2}$$

$$\dot{x}(k+1) = \dot{x}(k) + \ddot{x}(k) \cdot \Delta t$$



$$k = 0$$

The State Space and the System Dynamics

Dynamics:

$$x(k+1) = x(k) + \dot{x}(k) \cdot \Delta t + \frac{(\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t^2}{2}$$

$$\dot{x}(k+1) = \dot{x}(k) + (\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t$$

$$\boldsymbol{\rho} \sim N(\mu, \sigma)$$

$$k = 1$$

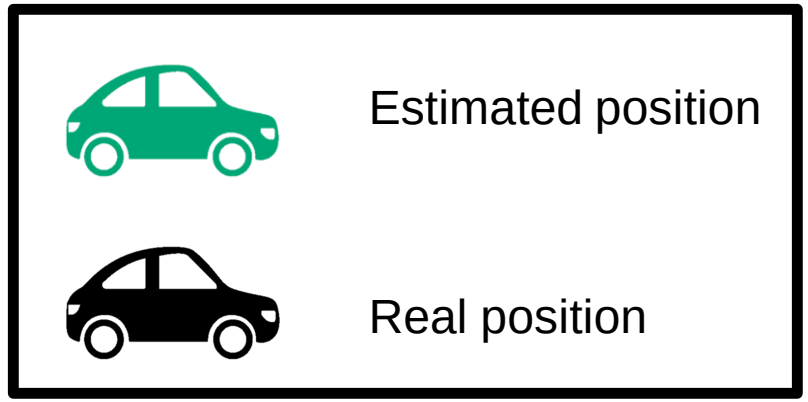
The State Space and the System Dynamics

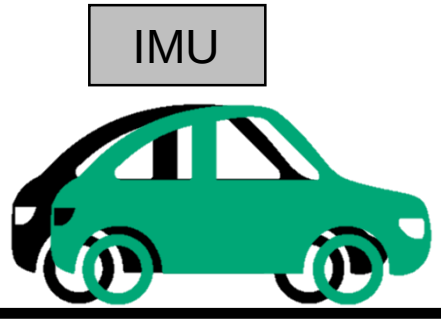
Dynamics:

$$x(k+1) = x(k) + \dot{x}(k) \cdot \Delta t + \frac{(\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t^2}{2}$$

$$\dot{x}(k+1) = \dot{x}(k) + (\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t$$

$$\boldsymbol{\rho} \sim N(\mu, \sigma)$$





$$k = 1$$

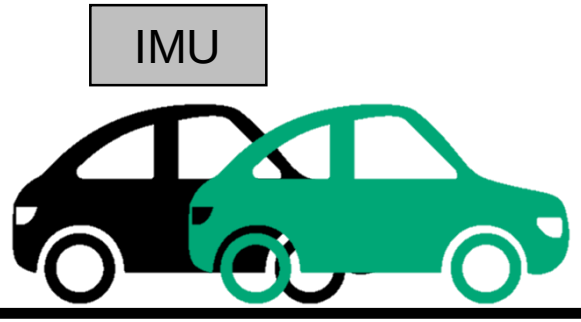
The State Space and the System Dynamics

Dynamics:

$$x(k+1) = x(k) + \dot{x}(k) \cdot \Delta t + \frac{(\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t^2}{2}$$

$$\dot{x}(k+1) = \dot{x}(k) + (\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t$$

$$\boldsymbol{\rho} \sim N(\mu, \sigma)$$



$$k = 2$$

The State Space and the System Dynamics

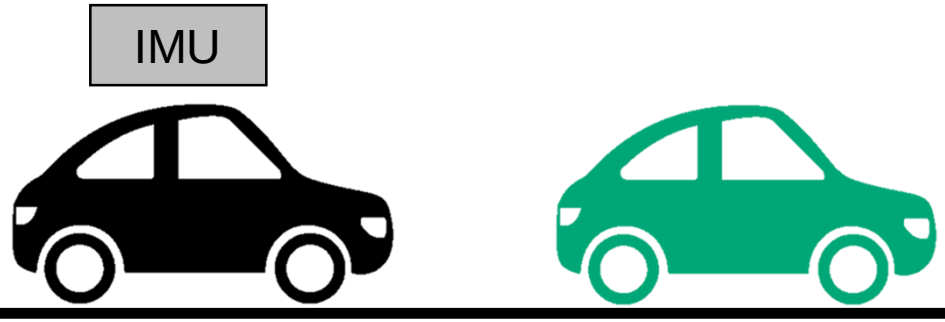
Dynamics:

$$x(k+1) = x(k) + \dot{x}(k) \cdot \Delta t + \frac{(\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t^2}{2}$$

$$\dot{x}(k+1) = \dot{x}(k) + (\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t$$

$$\boldsymbol{\rho} \sim N(\mu, \sigma)$$

ERROR GROWS UNBOUNDED!



$$k = 3$$

The State Space and the System Dynamics

Dynamics:

$$x(k+1) = x(k) + \dot{x}(k) \cdot \Delta t + \frac{(\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t^2}{2}$$

$$\dot{x}(k+1) = \dot{x}(k) + (\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t$$

$$\boldsymbol{\rho} \sim N(\mu, \sigma)$$

Measurement model (UWB):

$$z(k) = x(k) + \boldsymbol{\rho}_M, \qquad \boldsymbol{\rho}_M \sim N(\mu_M, \sigma_M)$$

$$k = 0$$

The State Space and the System Dynamics

$$x(k+1) = x(k) + \dot{x}(k) \cdot \Delta t + \frac{(\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t^2}{2}$$

$$\dot{x}(k+1) = \dot{x}(k) + (\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t$$

$$\boldsymbol{\rho} \sim N(\mu, \sigma)$$

Measurement model (UWB):

$$z(k) = x(k) + \boldsymbol{\rho}_M, \qquad \boldsymbol{\rho}_M \sim N(\mu_M, \sigma_M)$$

$$k = 1$$

The State Space and the System Dynamics

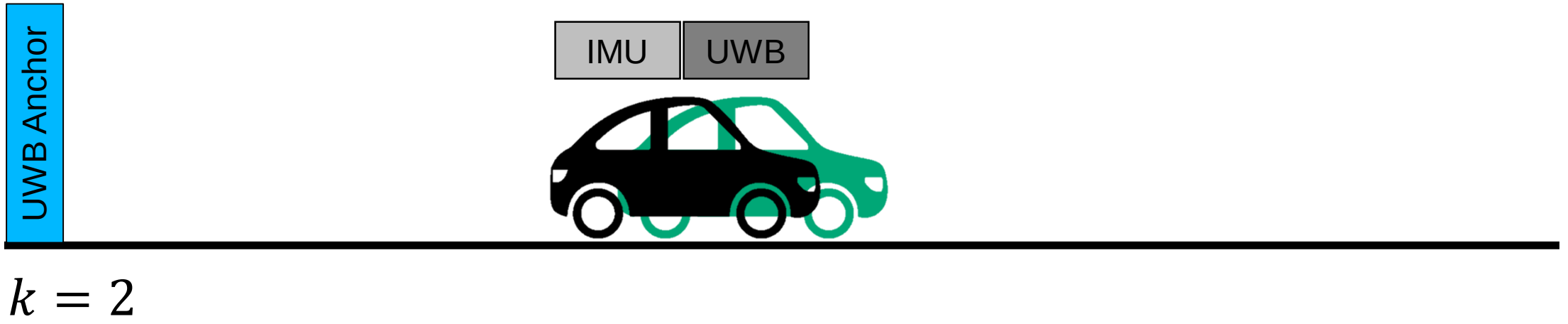
$$x(k+1) = x(k) + \dot{x}(k) \cdot \Delta t + \frac{(\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t^2}{2}$$

$$\dot{x}(k+1) = \dot{x}(k) + (\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t$$

$$\boldsymbol{\rho} \sim N(\mu, \sigma)$$

Measurement model (UWB):

$$z(k) = x(k) + \boldsymbol{\rho}_M, \qquad \boldsymbol{\rho}_M \sim N(\mu_M, \sigma_M)$$



$$k = 2$$

The State Space and the System Dynamics

$$x(k+1) = x(k) + \dot{x}(k) \cdot \Delta t + \frac{(\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t^2}{2}$$

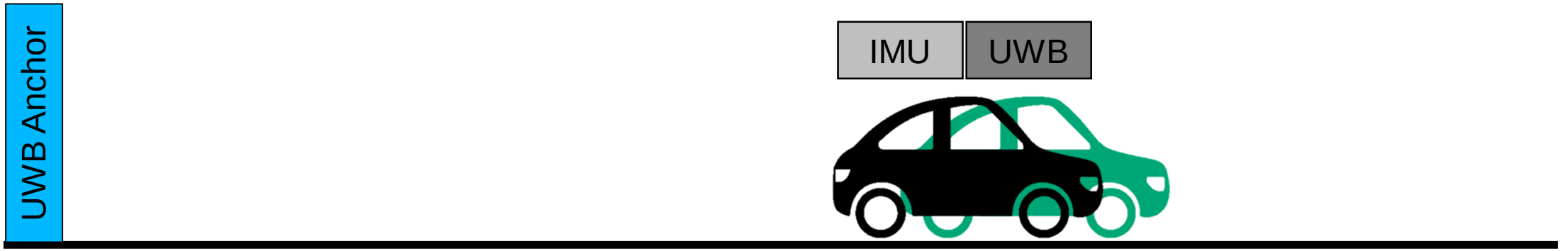
$$\dot{x}(k+1) = \dot{x}(k) + (\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t$$

$$\boldsymbol{\rho} \sim N(\mu, \sigma)$$

Measurement model (UWB):

$$z(k) = x(k) + \boldsymbol{\rho}_M, \qquad \boldsymbol{\rho}_M \sim N(\mu_M, \sigma_M)$$

ERROR IS BOUNDED!



$k = 3$

The State Estimator

- An algorithm that provides an estimate of the internal state of a given real system.
- The state estimator fuses the information from multiple sensors in order to provide a state estimate.
- Assuming that the sensor noise is always Gaussian, the Kalman Filter provides the best estimate in the mean-squared-error sense.
- A state estimator weights the “importance” of every sensor according to its standard deviation.

The Kalman Filter

1. Prediction step:

$$\hat{x}_{\mathrm{p}}(k) = A(k-1)\hat{x}_{\mathrm{m}}(k-1) + u(k-1)$$

$$P_{\mathrm{p}}(k) = A(k-1)P_{\mathrm{m}}(k-1)A^{T}(k-1) + Q(k-1)$$

2. Measurement update step:

$$P_{\mathrm{m}}(k) = \left(P_{\mathrm{p}}^{-1}(k) + H^{T}(k)R^{-1}(k)H(k)\right)^{-1}$$

$$\hat{x}_{\mathrm{m}}(k) = \hat{x}_{\mathrm{p}}(k) + P_{\mathrm{m}}(k)H^{T}(k)R^{-1}(k)\left(\bar{z}(k) - H(k)\hat{x}_{\mathrm{p}}(k)\right)$$

- Q and R are generally fixed.
- Every prediction step, we compute A and apply the formula.
- Every update step, we compute H and apply the formula.
- Every sensor has a different associated H.
- The whole estimation process is an iteration between **1.** and **2.**

The Kalman Filter

1. Prediction step:

$$\hat{x}_{\mathrm{p}}(k) = A(k-1)\hat{x}_{\mathrm{m}}(k-1) + u(k-1)$$

$$P_{\mathrm{p}}(k) = A(k-1)P_{\mathrm{m}}(k-1)A^{T}(k-1) + Q(k-1)$$

2. Measurement update step:

$$P_{\mathrm{m}}(k) = \left(P_{\mathrm{p}}^{-1}(k) + H^{T}(k)R^{-1}(k)H(k)\right)^{-1}$$

$$\hat{x}_{\mathrm{m}}(k) = \hat{x}_{\mathrm{p}}(k) + P_{\mathrm{m}}(k)H^{T}(k)R^{-1}(k)\left(\bar{z}(k) - H(k)\hat{x}_{\mathrm{p}}(k)\right)$$

- Q and R are generally fixed.
- Every prediction step, we compute A and apply the formula.
- Every update step, we compute H and apply the formula.
- Every sensor has a different associated H.
- The whole estimation process is an iteration between **1.** and **2.**

The State Space and the System Dynamics

$$x(k+1) = x(k) + \dot{x}(k) \cdot \Delta t + \frac{(\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t^2}{2}$$

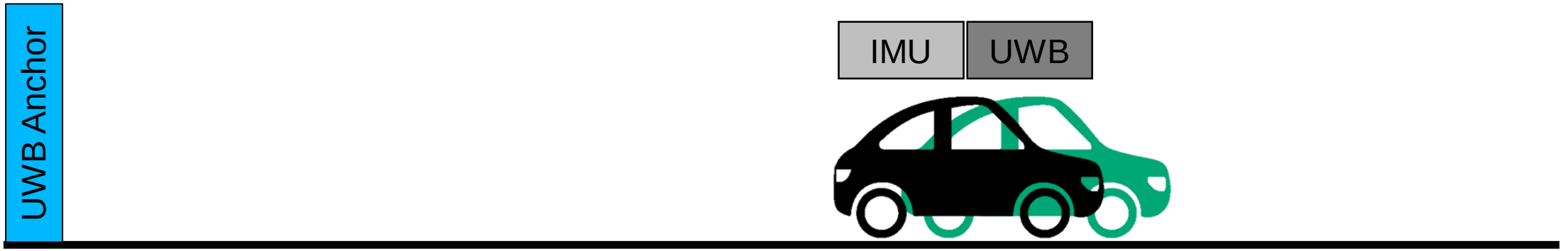
$$\dot{x}(k+1) = \dot{x}(k) + (\ddot{x}(k) + \boldsymbol{\rho}) \cdot \Delta t$$

$$\boldsymbol{\rho} \sim N(\mu, \sigma)$$

Measurement model (UWB):

$$z(k) = x(k) + \boldsymbol{\rho}_M, \qquad \boldsymbol{\rho}_M \sim N(\mu_M, \sigma_M)$$

ERROR IS BOUNDED!



$k = 3$

The Crazyflie Sensors

Crazyflie base version:

- Accelerometer
- Gyroscope
- Barometer

Limited functionality!

The Crazyflie Sensors

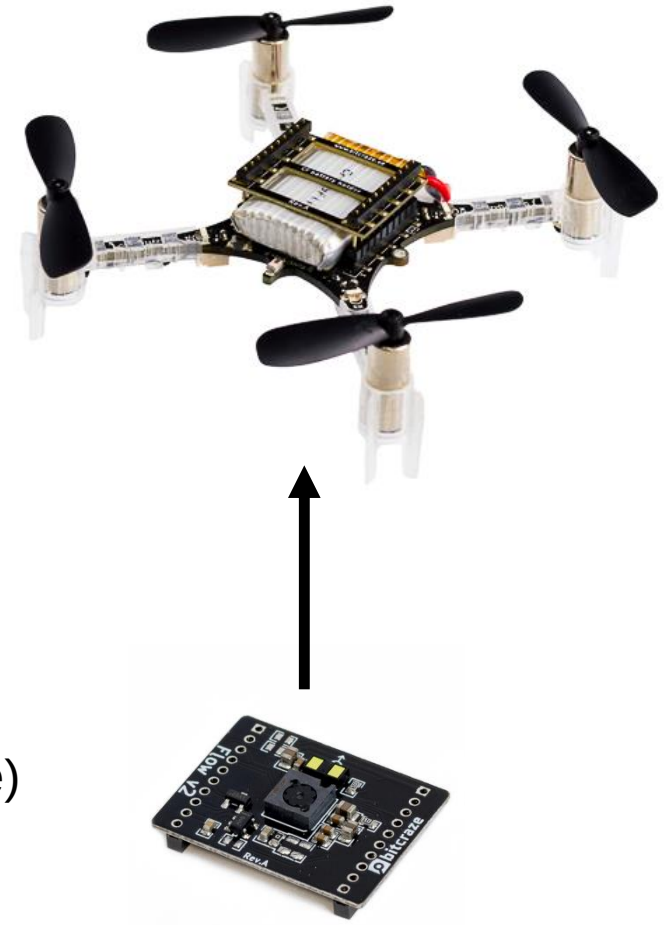
Crazyflie base version:

- Accelerometer
- Gyroscope
- Barometer

Limited functionality!

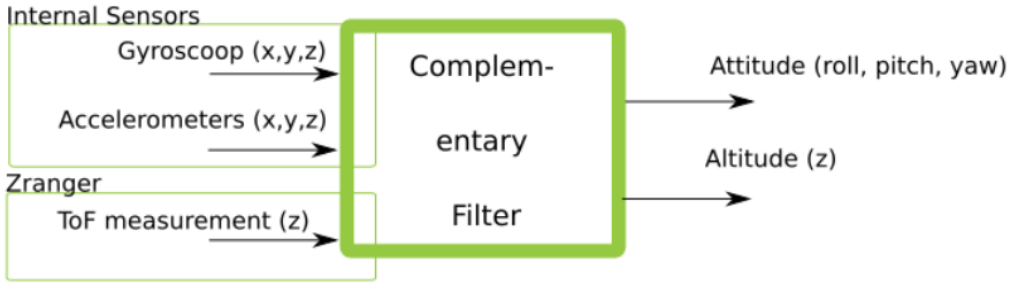
Flow Deck:

- ToF Sensor: Height measurement
- OpticalFlow sensor: Velocity measurement (body frame)



The State Estimator of the Crazyflie

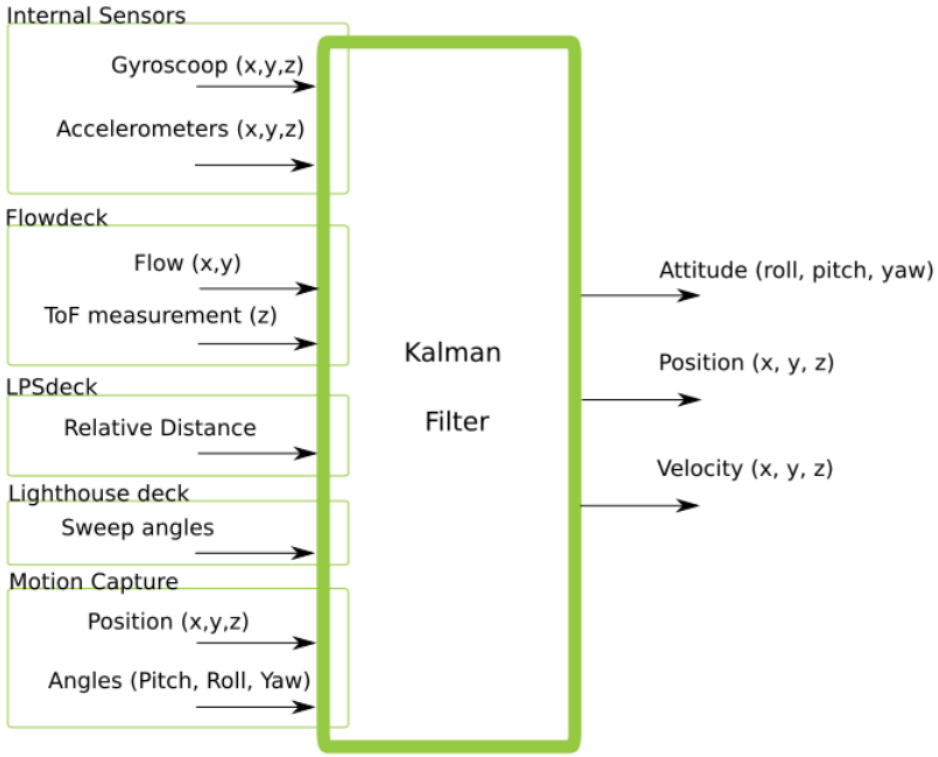
Complementary Filter



- Very lightweight and efficient filter.
- Only uses the IMU input (gyroscope, accelerometer) and the ToF measurement.
- Suitable for attitude estimation only (manual control).

Source: estimator_complementary.c, sensfusion6.c

Extended Kalman Filter



Source: estimator_kalman.c, kalman_core.c

The Estimator Implementation

estimator_kalman.c:

- Contains the high level EKF framework
- Manages and queues he incoming measurements from the sensors.
- Contains the functions for the prediction step.
- Logs the variables associated with the states

Path: crazyflie-firmware/src/modules/src/

kalman_core.c:

- Contains the implementation of the functions which update the state estimate with sensor measurements
- Implement all the “H” matrices presented in the “Kalman Filter” slide.

The Estimator Implementation

kalman_core.c:

- Contains the implementation of the functions which update the state estimate with sensor measurements
- Implement all the “H” matrices presented in the “Kalman Filter” slide.

Measurement update step:

$$P_{\mathrm{m}}(k) = \left(P_{\mathrm{p}}^{-1}(k) + H^T(k)R^{-1}(k)H(k)\right)^{-1}$$

$$\hat{x}_{\mathrm{m}}(k) = \hat{x}_{\mathrm{p}}(k) + P_{\mathrm{m}}(k)H^T(k)R^{-1}(k)\left(\bar{z}(k) - H(k)\hat{x}_{\mathrm{p}}(k)\right)$$

```
void kalmanCoreUpdateWithDistance(kalmanCoreData_t* this, distanceMeasurement_t *d)
{
  // a measurement of distance to point (x, y, z)
  float h[KC_STATE_DIM] = {0};
  arm_matrix_instance_f32 H = {1, KC_STATE_DIM, h};

  float dx = this->S[KC_STATE_X] - d->x;
  float dy = this->S[KC_STATE_Y] - d->y;
  float dz = this->S[KC_STATE_Z] - d->z;

  float measuredDistance = d->distance;

  float predictedDistance = arm_sqrt(powf(dx, 2) + powf(dy, 2) + powf(dz, 2));
  if (predictedDistance != 0.0f)
  {
    // The measurement is: z = sqrt(dx^2 + dy^2 + dz^2). The derivative dz/dX gives h.
    h[KC_STATE_X] = dx/predictedDistance;
    h[KC_STATE_Y] = dy/predictedDistance;
    h[KC_STATE_Z] = dz/predictedDistance;
  }
  else
  {
    // Avoid divide by zero
    h[KC_STATE_X] = 1.0f;
    h[KC_STATE_Y] = 0.0f;
    h[KC_STATE_Z] = 0.0f;
  }

  scalarUpdate(this, &H, measuredDistance-predictedDistance, d->stdDev);
}
```

The Estimator Implementation

kalman_core.c:

- Contains the implementation of the functions which update the state estimate with sensor measurements
- Implement all the “H” matrices presented in the “Kalman Filter” slide.

Measurement update step:

$$P_{\mathrm{m}}(k) = \left(P_{\mathrm{p}}^{-1}(k) + H^T(k)R^{-1}(k)H(k)\right)^{-1}$$

$$\hat{x}_{\mathrm{m}}(k) = \hat{x}_{\mathrm{p}}(k) + P_{\mathrm{m}}(k)H^T(k)R^{-1}(k)\left(\bar{z}(k) - H(k)\hat{x}_{\mathrm{p}}(k)\right)$$

Computing H_{UWB}

UWB measurement update:

```
void kalmanCoreUpdateWithDistance(kalmanCoreData_t* this, distanceMeasurement_t *d)
{
  // a measurement of distance to point (x, y, z)
  float h[KC_STATE_DIM] = {0};
  arm_matrix_instance_f32 H = {1, KC_STATE_DIM, h};

  float dx = this->S[KC_STATE_X] - d->x;
  float dy = this->S[KC_STATE_Y] - d->y;
  float dz = this->S[KC_STATE_Z] - d->z;

  float measuredDistance = d->distance;

  float predictedDistance = arm_sqrt(powf(dx, 2) + powf(dy, 2) + powf(dz, 2));
  if (predictedDistance != 0.0f)
  {
    // The measurement is: z = sqrt(dx^2 + dy^2 + dz^2). The derivative dz/dX gives h.
    h[KC_STATE_X] = dx/predictedDistance;
    h[KC_STATE_Y] = dy/predictedDistance;
    h[KC_STATE_Z] = dz/predictedDistance;
  }
  else
  {
    // Avoid divide by zero
    h[KC_STATE_X] = 1.0f;
    h[KC_STATE_Y] = 0.0f;
    h[KC_STATE_Z] = 0.0f;
  }

  scalarUpdate(this, &H, measuredDistance-predictedDistance, d->stdDev);
}
```

The Estimator Implementation

kalman_core.c:

- Contains the implementation of the functions which update the state estimate with sensor measurements
- Implement all the “H” matrices presented in the “Kalman Filter” slide.

Measurement update step:

$$P_{\mathrm{m}}(k) = \left(P_{\mathrm{p}}^{-1}(k) + H^T(k)R^{-1}(k)H(k)\right)^{-1}$$

$$\hat{x}_{\mathrm{m}}(k) = \hat{x}_{\mathrm{p}}(k) + P_{\mathrm{m}}(k)H^T(k)R^{-1}(k)\left(\bar{z}(k) - H(k)\hat{x}_{\mathrm{p}}(k)\right)$$

UWB measurement update:

```
void kalmanCoreUpdateWithDistance(kalmanCoreData_t* this, distanceMeasurement_t *d)
{
  // a measurement of distance to point (x, y, z)
  float h[KC_STATE_DIM] = {0};
  arm_matrix_instance_f32 H = {1, KC_STATE_DIM, h};

  float dx = this->S[KC_STATE_X] - d->x;
  float dy = this->S[KC_STATE_Y] - d->y;
  float dz = this->S[KC_STATE_Z] - d->z;

  float measuredDistance = d->distance;

  float predictedDistance = arm_sqrt(powf(dx, 2) + powf(dy, 2) + powf(dz, 2));
  if (predictedDistance != 0.0f)
  {
    // The measurement is: z = sqrt(dx^2 + dy^2 + dz^2). The derivative dz/dX gives h.
    h[KC_STATE_X] = dx/predictedDistance;
    h[KC_STATE_Y] = dy/predictedDistance;
    h[KC_STATE_Z] = dz/predictedDistance;
  }
  else
  {
    // Avoid divide by zero
    h[KC_STATE_X] = 1.0f;
    h[KC_STATE_Y] = 0.0f;
    h[KC_STATE_Z] = 0.0f;
  }

  scalarUpdate(this, &H, measuredDistance-predictedDistance, d->stdDev);
}
```

The Estimator Implementation

kalman_core.c:

- Contains the implementation of the functions which update the state estimate with sensor measurements
- Implement all the “H” matrices presented in the “Kalman Filter” slide.

Measurement update step:

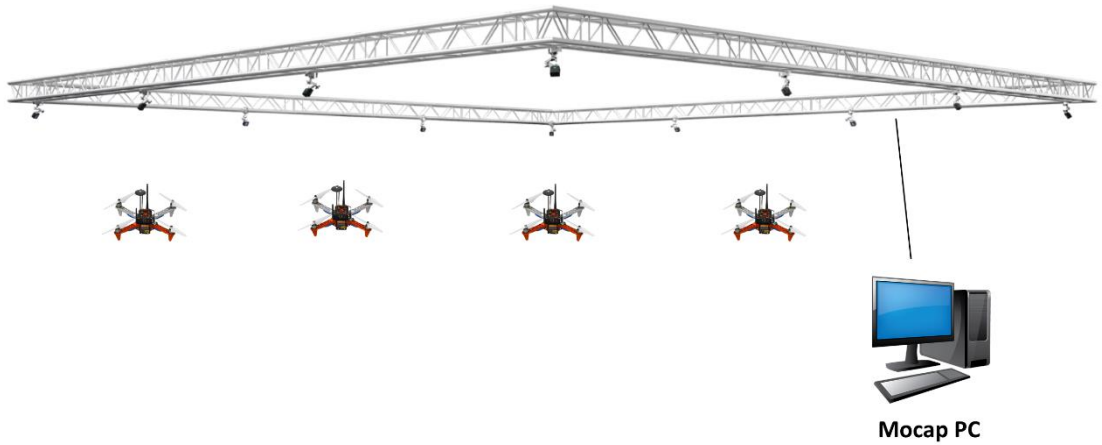
$$P_{\mathrm{m}}(k) = \left(P_{\mathrm{p}}^{-1}(k) + H^T(k)R^{-1}(k)H(k)\right)^{-1}$$

$$\hat{x}_{\mathrm{m}}(k) = \hat{x}_{\mathrm{p}}(k) + P_{\mathrm{m}}(k)H^T(k)R^{-1}(k)\left(\bar{z}(k) - H(k)\hat{x}_{\mathrm{p}}(k)\right)$$

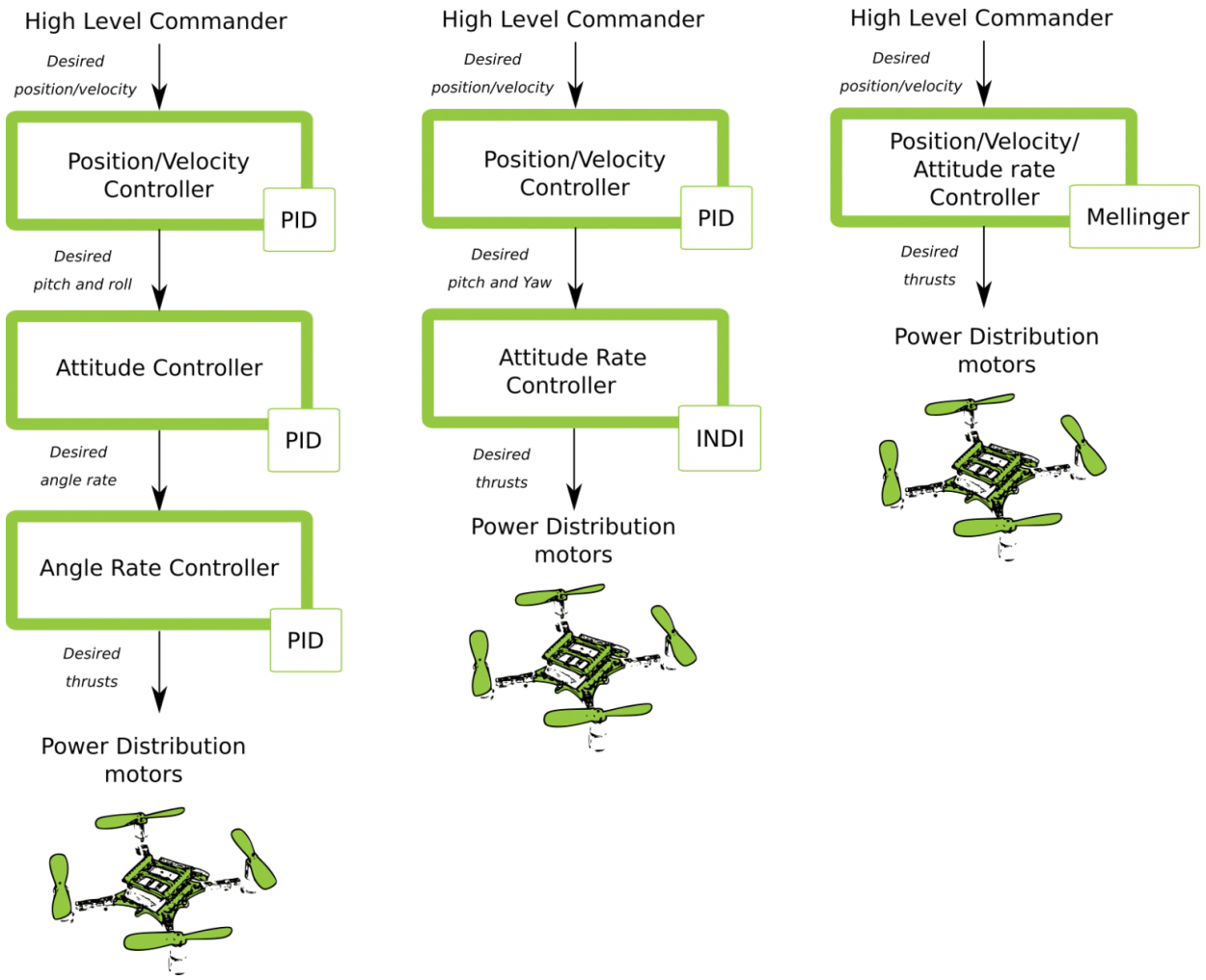
Position (moCap) measurement update:

```
void kalmanCoreUpdateWithPosition(kalmanCoreData_t* this, positionMeasurement_t *xyz)
{
  // a direct measurement of states x, y, and z
  // do a scalar update for each state, since this should be faster than updating all together
  for (int i=0; i<3; i++) {
    float h[KC_STATE_DIM] = {0};
    arm_matrix_instance_f32 H = {1, KC_STATE_DIM, h};
    h[KC_STATE_X+i] = 1;
    scalarUpdate(this, &H, xyz->pos[i] - this->S[KC_STATE_X+i], xyz->stdDev);
  }
}
```

Computing H_{moCap}

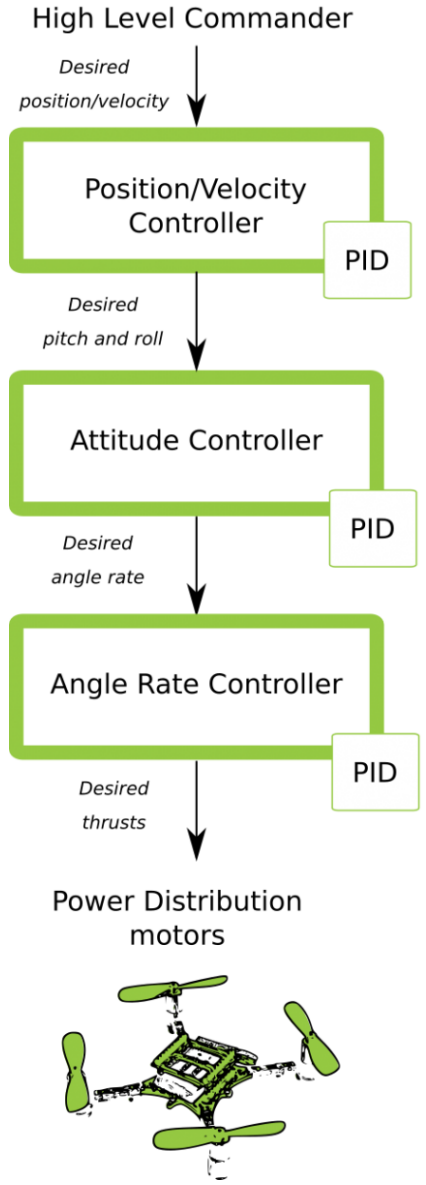


The Available State Controllers

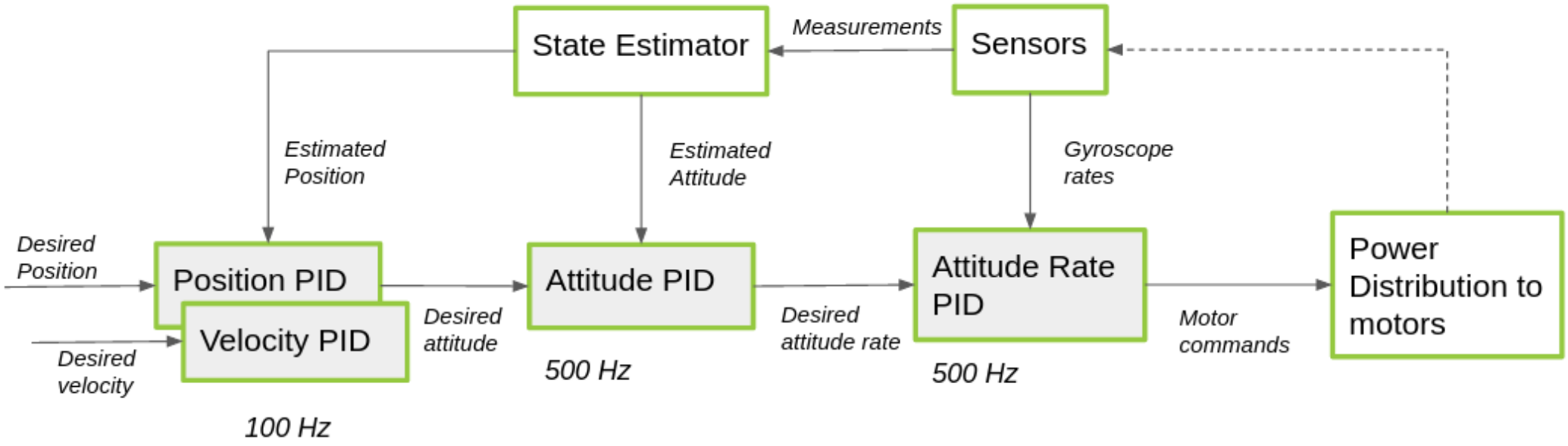


The State Controller

- 3 cascaded controllers.
- Angle Rate Controller (ART): *attitude_pid_controller.c*
  - ➢ directly controls the attitude rate.
  - ➢ relies almost directly on the gyroscope rates (slightly LPF).
  - ➢ takes the error between the desired attitude rate as input.
  - ➢ loop runs at 500 Hz.
- Attitude Controller (AC): *attitude_pid_controller.c*
  - ➢ takes in the estimated attitude of the state estimator
  - ➢ takes the error of the desired attitude setpoint
  - ➢ the output is the desired attitude rate (sent to the ART).
  - ➢ loop runs at 500 Hz.
- Position/Velocity Controller: *position_controller_pid.c*
  - ➢ receives position or velocity input from a commander which are handled on the same level.
  - ➢ loop runs at 100Hz.



The State Controller



The Setpoint

- A Setpoint is a structure that dictates the desired value of the state.
- There are two levels to control:
  - *Position (X,Y,Z)*
  - *Attitude (roll, pitch, yaw, or in quaternions)*
- *Position and attitude can be controlled in three different modes:*
  - Absolute mode (*modeAbs*)
  - Velocity mode (*modeVelocity*)
  - Disabled (*modeDisable*)

```
typedef struct setpoint_s {
  uint32_t timestamp;

  attitude_t attitude;      // deg
  attitude_t attitudeRate;  // deg/s
  quaternion_t attitudeQuaternion;
  float thrust;
  point_t position;         // m
  velocity_t velocity;      // m/s
  acc_t acceleration;       // m/s^2
  bool velocity_body;

  struct {
    stab_mode_t x;
    stab_mode_t y;
    stab_mode_t z;
    stab_mode_t roll;
    stab_mode_t pitch;
    stab_mode_t yaw;
    stab_mode_t quat;
  } mode;
} setpoint_t;
```

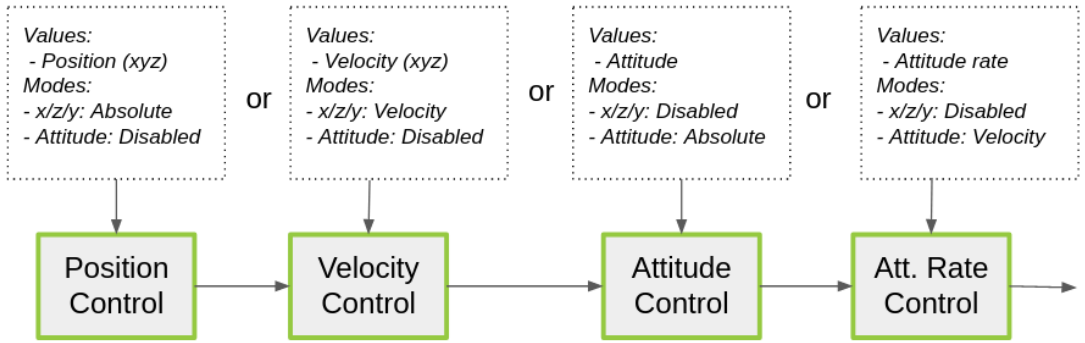
The Setpoint

- A Setpoint is a structure that dictates the desired value of the state.
- There are two levels to control:
  - *Position (X,Y,Z)*
  - *Attitude (roll, pitch, yaw, or in quaternions)*
- *Position and attitude can be controlled in three different modes:*
  - Absolute mode (*modeAbs*)
  - Velocity mode (*modeVelocity*)
  - Disabled (*modeDisable*)

```
typedef struct setpoint_s {
  uint32_t timestamp;

  attitude_t attitude;      // deg
  attitude_t attitudeRate;  // deg/s
  quaternion_t attitudeQuaternion;
  float thrust;
  point_t position;         // m
  velocity_t velocity;      // m/s
  acc_t acceleration;       // m/s^2
  bool velocity_body;

  struct {
    stab_mode_t x;
    stab_mode_t y;
    stab_mode_t z;
    stab_mode_t roll;
    stab_mode_t pitch;
    stab_mode_t yaw;
    stab_mode_t quat;
  } mode;
} setpoint_t;
```

The Setpoint

- A Setpoint is a structure that dictates the desired value of the state.
- There are two levels to control:
  - *Position (X,Y,Z)*
  - *Attitude (roll, pitch, yaw, or in quaternions)*
- *Position and attitude can be controlled in three different modes:*
  - Absolute mode (*modeAbs*)
  - Velocity mode (*modeVelocity*)
  - Disabled (*modeDisable*)




```
typedef struct setpoint_s {
  uint32_t timestamp;

  attitude_t attitude;      // deg
  attitude_t attitudeRate;  // deg/s
  quaternion_t attitudeQuaternion;
  float thrust;
  point_t position;         // m
  velocity_t velocity;      // m/s
  acc_t acceleration;       // m/s^2
  bool velocity_body;

  struct {
    stab_mode_t x;
    stab_mode_t y;
    stab_mode_t z;
    stab_mode_t roll;
    stab_mode_t pitch;
    stab_mode_t yaw;
    stab_mode_t quat;
  } mode;
} setpoint_t;
```

The Setpoint – position setpoint example

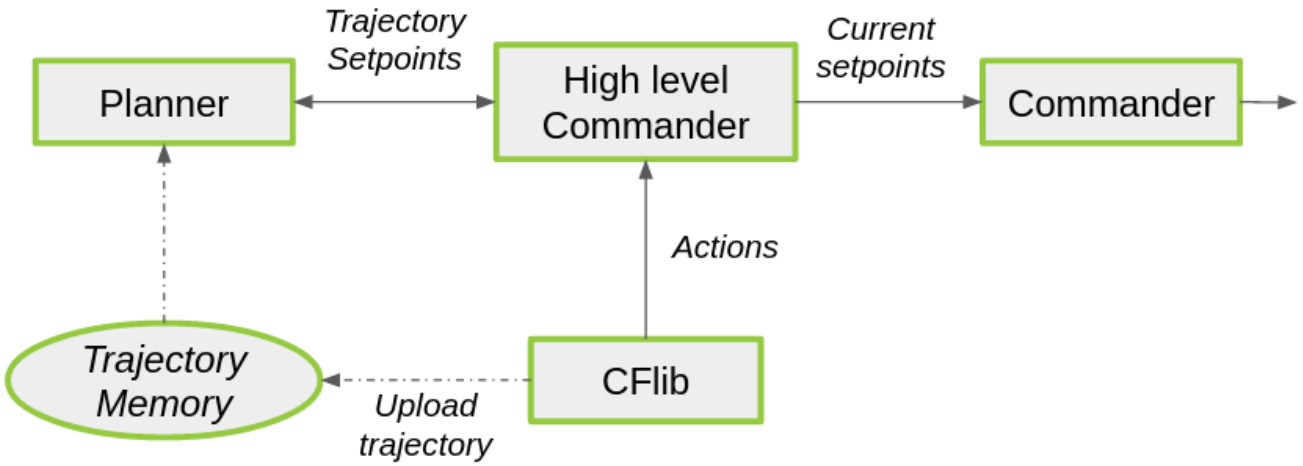
Reset the structure

Set the absolute mode for position and heading.

```
static void positionSet(setpoint_t *setpoint, float x, float y, float z, float yaw)
{
  memset(setpoint, 0, sizeof(setpoint_t));

  setpoint->mode.x = modeAbs;
  setpoint->mode.y = modeAbs;
  setpoint->mode.z = modeAbs;


  setpoint->position.x = x;
  setpoint->position.y = y;
  setpoint->position.z = z;


  setpoint->mode.yaw = modeAbs;

  setpoint->attitude.yaw = yaw;

  setpoint->mode.roll = modeDisable;
  setpoint->mode.pitch = modeDisable;
  setpoint->mode.quat = modeDisable;
}
```

The High Level Commander

- The **Commander** receives a setpoint and forwards it as reference for the position/velocity controller.
- The ***High Level Commander (HLC)*** handles setpoints from the firmware based on a predefined trajectory.
- **The Planner** generates a group of setpoints, which are handled by the HLC and sent to the commander.
- The Planner and the HLC are necessary when executing a polynomial trajectory, but the Commander can be also used independently.



The High Level Commander

- The **Commander** receives a setpoint and forwards it as reference for the position/velocity controller.
- The ***High Level Commander (HLC)*** handles setpoints from the firmware based on a predefined trajectory.
- **The Planner** generates a group of setpoints, which are handled by the HLC and sent to the commander.
- The Planner and the HLC are necessary when executing a polynomial trajectory, but the Commander can be also used independently.

- The **Commander, HLC** and **Planner** source files are located in *crazyflie-firmware/src/modules/src/ commander.c, crtp_commander_high_level.c, planner.c*

- To command the drone a specific setpoint, the following function can be used: *commanderSetSetpoint(&setpoint_vlad, 3);*

The Logger

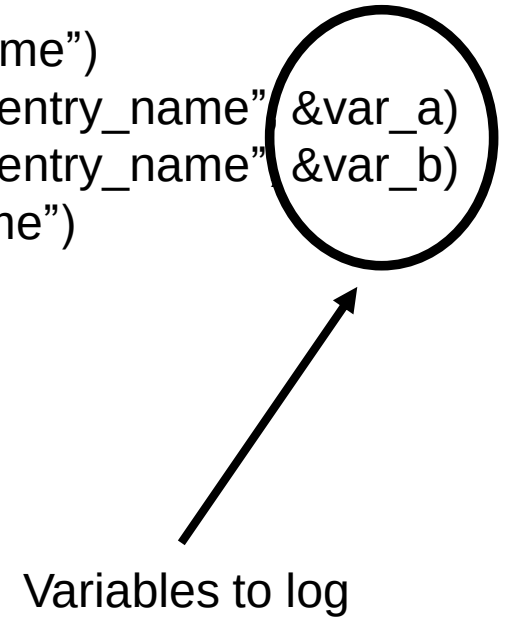
- The *Firmeware* offers logging capabilities.
- By including "log.h“, the logging macros can be used in any file.
- In order to create a new log, the following structure has to be used:

```
LOG_GROUP_START(“group_name”)
        LOG_ADD(“log_type”, “log_entry_name”, &var_a)
        LOG_ADD(“log_type”, “log_entry_name”, &var_b)
LOG_GROUP_STOP(“group_name”)
```

The Logger

- The *Firmeware* offers logging capabilities.
- By including "log.h“, the logging macros can be used in any file.
- In order to create a new log, the following structure has to be used:

```
LOG_GROUP_START(“group_name”)
        LOG_ADD(“log_type”, “log_entry_name”, &var_a)
        LOG_ADD(“log_type”, “log_entry_name”, &var_b)
LOG_GROUP_STOP(“group_name”)
```



The Logger

- The logged variables can be accessed using the Python library, the client, or from other files.

```
LOG_GROUP_START(kalman)
  LOG_ADD(LOG_FLOAT, stateX, &var_state_x)
  LOG_ADD(LOG_FLOAT, stateY, &var_state_x)
  LOG_ADD(LOG_FLOAT, stateZ, &var_state_x)
LOG_GROUP_STOP(kalman)
```

Declaring a log block in the file where the variable is visible.

```
int var_id;
var_id = logGetVarId("kalman", "stateX");
cf_pos[0] = logGetFloat(var_id);
var_id = logGetVarId("kalman", "stateY");
cf_pos[1] = logGetFloat(var_id);
var_id = logGetVarId("kalman", "stateZ");
cf_pos[2] = logGetFloat(var_id);
```

Accessing the logged variables from other files.

The Parameters

- The parameters are shared variables which can be modified from any file or from the python client.

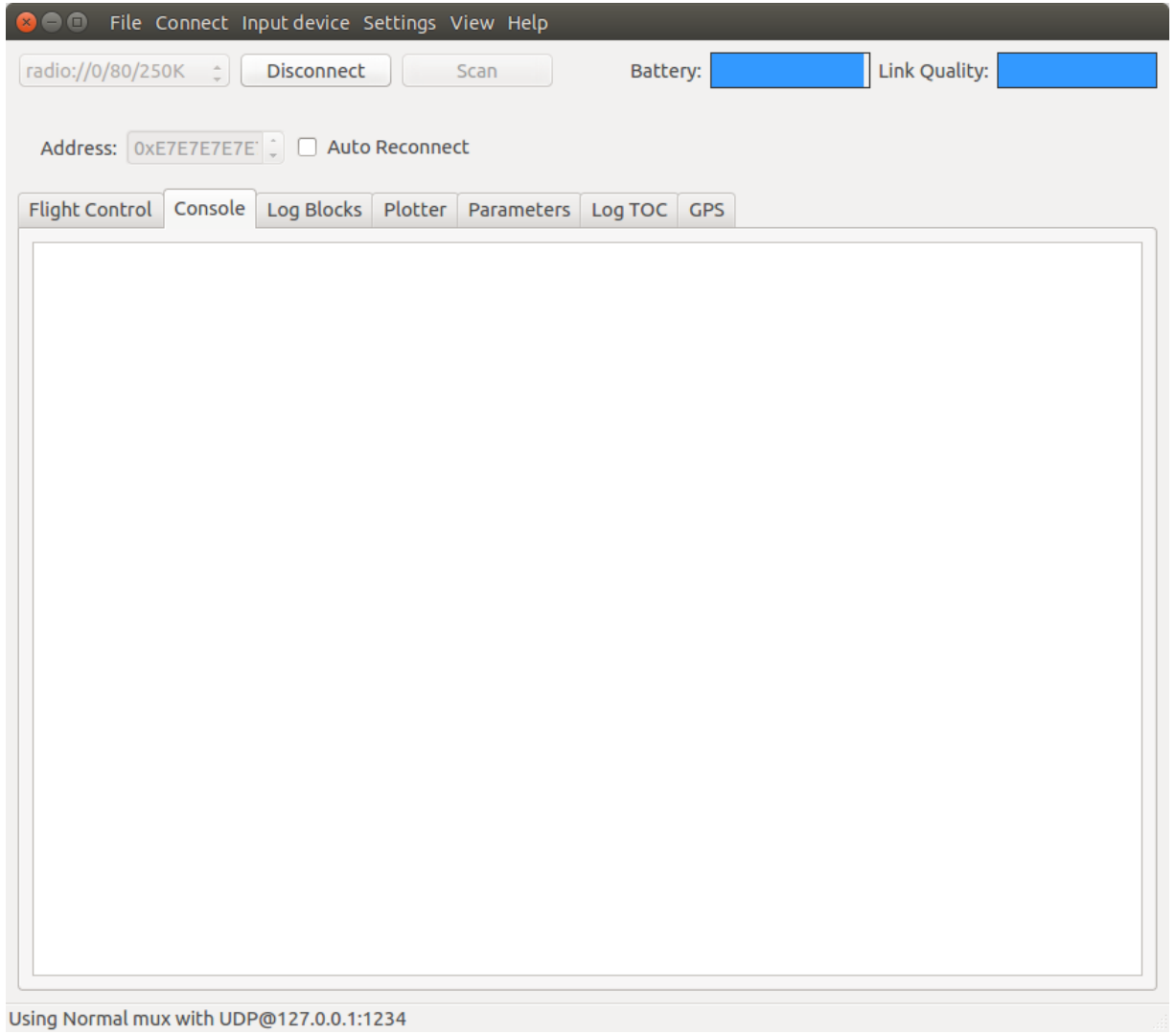
```
PARAM_GROUP_START(“group_name”)
      PARAM_ADD(“log_type”, “log_entry_name”, &var_a)
PARAM_GROUP_START(“group_name”)
```

- The parameters are bidirectional. For example, when the client commands a certain value for the parameter UWB.range, the range_uwb variable is updated.

```
PARAM_GROUP_START(UWB)
PARAM_ADD(PARAM_FLOAT, range, &range_uwb)
PARAM_GROUP_STOP(UWB)
```

The Crazyflie Client

- The client allows for the interaction with the drone.
- The user can access the logged variables and monitor their values while the system is running.
- The client allows for modifying the parameter values while the system is running.
- It can plot the state variable values.
- It provides a debug console which can display information sent from the drone with the built-in “printf” function.



The DEBUG_PRINT function

- In order to print into the debug console of the
- CF client, the DEBUG_PRINT function has to be used.
- #include "debug.h“ is necessary

```
DEBUG_PRINT(“Number of iterations: %d \n", iter_nr);
```