École Polytechnique Fédérale de Lausanne

Master Thesis

Accelerated Sensor Fusion for Drones and a Simulation Framework for Spatial

Author
Ruben Fiszel
ruben.fiszel@epfl.ch

August 29, 2017

Supervisors
Prof. Martin Odersky
LAMP | EPFL
martin.odersky@epfl.ch

Prof. Oyekunle A. Olukotun
PPL | Stanford
kunle@stanford.edu

Contents

Introduction

Moore's law end

The Moore's law[1] has ruled computation for the last 4 decades. With each generation of processor, the promise of an exponentially faster execution. Transistors are reaching the scale of 10nm, only a 100 time bigger than an atom. Unfortunately, the quantum rules of physics which govern the infinitesimally, start manifest themselves. In particular, quantum tunneling move electrons from classicly unsurmountable barrier, making computations approximate, containing a non negligible fraction of errors.

The rise of Hardware

Hardware and Software designate here respectively programs that are executed as code for a general purpose processing unit and programs that are encoded in the circuits. The dichotomy is not very well defined and we can think of it as a spectrum. General-purpose computing on graphics processing units (GPGPU) is in-between. Very efficient when appropriate and used well. They have benefited from high-investment and many generation of iterations and hence, for some tasks, can rivalize or even surpass Hardware.

Hardware has always been there but application-specific integrated circuit (ASIC) has prohibitive costs upfront (in the range of $100M for a tapeout). Reprogrammable hardware like field-programmable gate array (FPGA) have only been used marginally and for some specific industry like high-frequency trading. But now Hardware might be the only solution (until a computing revolution happen, like quantum computing, but this is not realist for the near future) to increase performance. But hardware do not enjoy the same

[1]The observation that the number of transistors in a dense integrated circuit doubles approximately every two years.

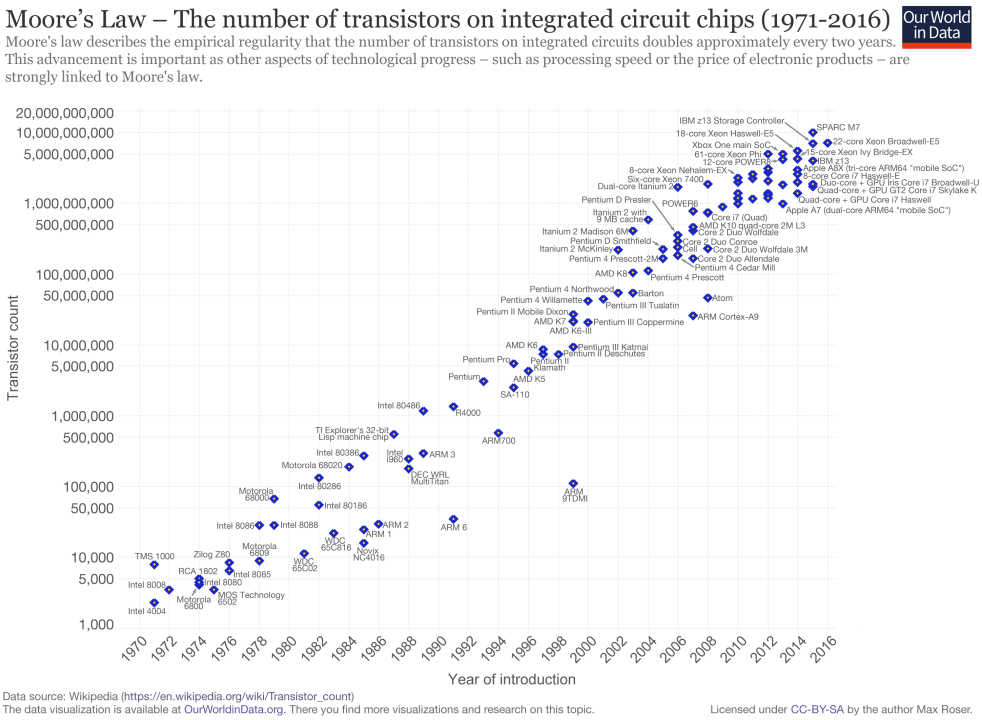


Figure 1: The number of transistors throughout the years. We can observe a recent start of a decline

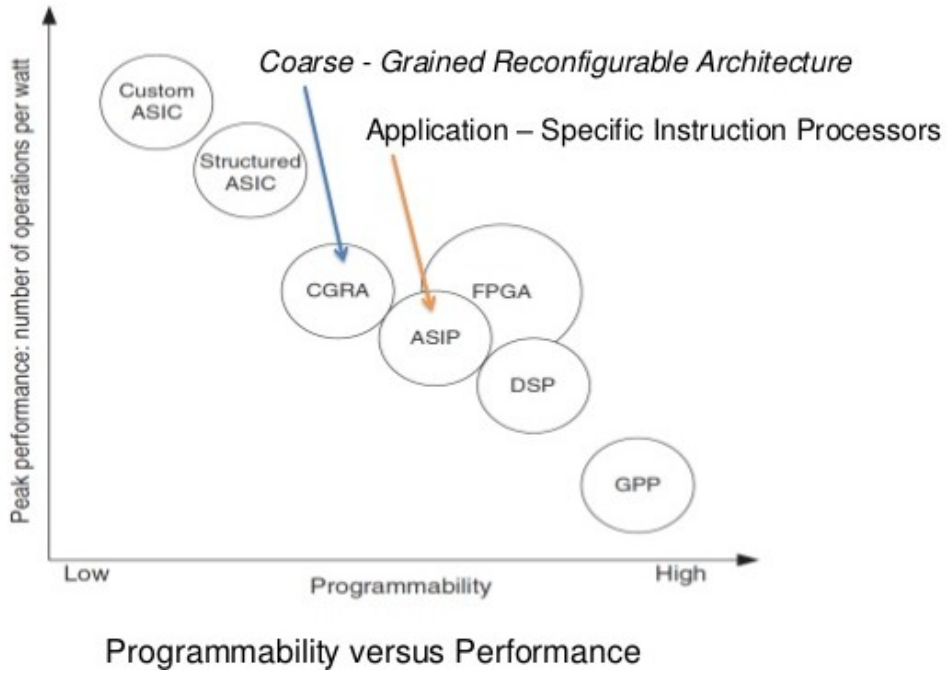


Figure 2: Hardware vs Software

quality of tool, language and integrated development environment (IDE) as software. This is the motivation behind Spatial.

Hardware as companion accelerators

In most case, hardware would be inappropriate: running an OS as hardware would be irrealist. However, as a companion to a central-processing unit (CPU also called "the host"), you are able to get the best of both world. The flexibility of software on a CPU with the speed of hardware. In this setup, hardware is considered an "accelerator" (Hence, the term "accelerating hardware"). It accelerates the most demanding subroutines of the CPU. This companionship is already present in modern computer desktops under the form of GPUs for *shader* operations and sound card for complex sound transformation/output.

The right metric: Perf/Watt

The right metric for accelerator is performance by energy, as measured in FLOPS per Watt. This is a fair metric for the comparison of different hardware because it shows the intrisic value of the architecture. If the metric was solely performance, then it would suffice to combine multiple of the same architecture. Perf per dollar is not a good metric either because you should also account for the cost of energy at runtime. Hence, Perf/Watt seems like a fair metric to compare architectures.

Spatial

At the dawn lab, under the lead of Prof. Kunle and his grad students, is developped a scala DSL spatial and its compiler to program Hardware in a higher-level, more user-friendly, more productive language than Verilog. In particular, the control flows are automatically generated when possible. This should enable software engineers to unlock the potential of Hardware. A custom CGRA, Plasticine, has been developped in parralel to Spatial. It leverages some recurrent patterns, in particular parralel patterns and aims to be the most efficient reprogrammable architecture for Spatial.

There is a large upfront cost but once at a big enough scale, Plasticine could be deployed as an accelerator for most demanding server applications and embedded systems with heavy computing requirements.

Embedded systems and drones

Embedded systems are limited by the amount of power at disposal from the battery and might also have size constraints. At the same time, especially for autonomous vehicles, there is a great need for computing power.

Thus, developping drone applications with spatial demonstrates the advantages of the platform. As a matter of fact, the filter that has been developped was only made possible because it was run on an accelerating hardware. It would be irrealist to attempt to run it on more conventional micro-transistors. This is why the family in which belong the filter developped here, particles filters, being very computationally expensive, are very seldom used for drones.

1 | Sensor fusion algorithm for POSE estimation of drones: Asynchronous Rao-Blackwellized Particle filter

POSE is the combination of the position and orientation of an object. POSE estimation is important for drones. It is a subroutine of SLAM (Simultaneous localization and mapping) and it is a central part of motion planning and motion control. More accurate and more reliable POSE estimation results in more agile, more reactive and safer drones. Drones are an intellectually stimulating subject but in the near-future they might also see their usage increase exponentially. In this context, developping and implementing new filter for POSE estimation is both important for the field of robotics but also to demonstrate the importance of hardware acceleration. Indeed, the best and last filter presented here is only made possible because it can be hardware accelerated with Spatial. However, the spatial implementation will be presented in Part III.

Before expanding on the Rao-Blackwellized particle filter, we will introduce here several other filters for POSE estimation for highly dynamic objects: Complementary filter, Kalman Filter, Extended Kalman Filter and finally Rao-Blackwellized Particle filter. The order is from the most conceptually simple, to the most complex. This order is justified because complex filters aim to alleviate some of the flaws of their simpler counterpart. It is important to understand what are those weakness and how we can alleviate them.

Drones and collision avoidance

The original motivation for the development of accelerated POSE estimation is for the task of collision avoidance by quadcopters. In particular, a collision avoidance algorithm developped at the ASL lab and demonstrated here (https://youtu.be/kdlhfMiWVV0)

Figure 1.1: Ross Allen fencing with his drone

where the drone avoids the sword attack froms its creator. At first, it was thought of accelerating the whole algorithm but it was found that one of the most demanding subroutine was pose estimation. Moreover, it was wished to increase the processing rate of the filter such that it could match the input with the fastest sampling rate: its inertial measurement unit (IMU) containing an accelerometer, a gyroscope and a magnetometer.

The flamewheel f450 is the typical drone in this category. It is surprisingly fast and agile. Given the proper command, it can generate enough thrust to avoid in a very short lapse of time any incoming object.

Sensor fusion

Sensor fusion is combining of sensory data or data derived from disparate sources such that the resulting information has less uncertainty than would be possible when these sources were used individually. In the context of drones, it is very useful because it enables to combine many unprecise sensor measurement to form a more precise measurement like having precise positionning from 2 less precise GPS (dual GPS setting). It can also permit to combine sensors with different sampling rates: typically precise sensors with

Figure 1.2: The Flamewheel f450

low sampling rate and less precise sensors with high sampling rate. Both cases are gonna be relevant here.

A fundamental explanation why this is possible comes from the central limit theorem: one sample from a distribution with a low variance is as good as n sample from a distribution with variance n times higher.

$$\mathbb{V}(X_i) = \sigma^2 \qquad \mathbb{E}(X_i) = \mu$$
$$\bar{X} = \frac{1}{n}\sum X_i$$
$$\mathbb{V}(\bar{X}) = \frac{\sigma^2}{n} \qquad \mathbb{E}(\bar{X}) = \mu$$

Notes on notation and conventions

The referential by default is the fixed world frame.

- $\mathbf{x}$ designates a vector
- x_t is the random variable of x at time t
- $x_{t1:t2}$ is the product of the random variable of x between t1 included and t2 included

- $x^{(i)}$ designates the random variable x of the arbitrary particle i
- $\hat{x}$ designates an estimated variable

POSE

POSE is the task of estimating the position and orientation of an object through time. It is a subroutine of Software Localization And Mapping (SLAM). We can formelize the problem as:

At each timestep, find the best expectation of a function of the hidden variable state (position and orientation), from their initial distribution and the history of observable random variables (such as sensor measurements).

- The state $\mathbf{x}$
- The function $g(\mathbf{x})$ such that $g(\mathbf{x}_t) = (\mathbf{p}_t, \mathbf{q}_t)$ where $\mathbf{p}$ is the position and $\mathbf{q}$ is the attitude as a quaternion.
- The observable variable $\mathbf{y}$ composed of the sensor measurements $\mathbf{z}$ and the control input $\mathbf{u}$

The algorithm inputs are:

- control inputs $\mathbf{u}_t$ (the commands sent to the flight controller)
- sensor measurements $\mathbf{z}_t$ coming from different sensors with different sampling rate
- information about the sensors (sensor measurements biases and matrix of covariance)

Data generation

The difficulties with using real flight data is that you need to get the *true* trajectory and that you need enough data to check the efficiency of the filters.

To avoid those issues, the flight data is simulated through a model of trajectory generation. This model as described in [1], The motion primitives are defined by the quadrocopter's initial state, the desired motion duration, and any combination of components of the quadrocopter's position, velocity and acceleration at the motion's end. Closed form solutions for the primitives are given, which minimize a cost function related to input aggressiveness.

The bulk of the method is that a differential equation representing the difference of position, velocity and acceleration between the starting and

ending state is solved with the Pontryagin's minimum principle using the appropriate Hamiltonian. Then, from that closed form solution, a per-axis cost can be calculated to pick the "least aggressive" trajectory out of different candidates. Finally, the feasibility of the trajectory is computed using the constraints of maximum thrust and body rate (angular velocity) limits.

For the purpose of this work, a scala implementation of the model was realized. Then, some keypoints containing gaussian components for the position, velocity acceleration, and duration were tried until a feasible set of keypoints was found. This method of data generation is both fast and a good enough approximation of the actual trajectories that a drone would perform in the real world.

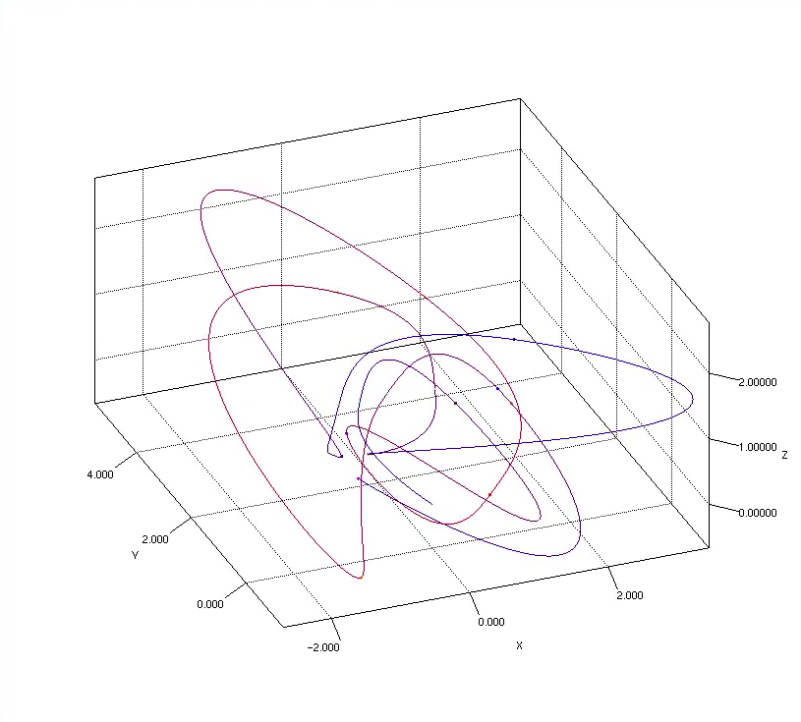


Figure 1.3: Visualisation of an example of a synthetic generated flight trajectory

Quaternion

Quaternions are extensions of complex numbers but with 3 imaginary parts. Unit quaternions can be used to represent orientation, also referred to as attitude. Quaternions algebra make rotation composition simple and quaternions avoid the issue of gimbal lock. In all filters presented, they will be used to represent the attitude.

$$\mathbf{q} = (q.r, q.i, q.j, q.k)^t = (q.r, \boldsymbol{\varrho})^T$$

Quaternion rotations composition is: $q_2 q_1$ which results in q_1 being rotated by the rotation represented by q_2. From this, we can deduce that angular velocity integrated over time is simply q^t if q is the local quaternion rotation by unit of time.

Rotation of a vector by a quaternion is done by: qvq^* where q is the quaternion representing the rotation, q^* its conjugate and v the vector to be rotated.

The distance of between two quaternions, useful as an error metric is defined by the squared Frobenius norms of attitude matrix differences [2].

$$\|A(\mathbf{q}_1) - A(\mathbf{q}_2)\|_F^2 = 6 - 2Tr[A(\mathbf{q}_1)A^t(\mathbf{q}_2)]$$

where

$$A(\mathbf{q}) = (q.r^2 - \|\boldsymbol{\varrho}\|^2)I_{3\times 3} + 2\boldsymbol{\varrho}\boldsymbol{\varrho}^T - 2q.r[\boldsymbol{\varrho}\times]$$

$$[\boldsymbol{\varrho}\times] = \begin{pmatrix} 0 & -q.k & q.j \\ q.k & 0 & -q.i \\ -q.j & q.i & 0 \end{pmatrix}$$

Helper functions and matrices

We introduce some helper matrices.

- $\mathbf{R}_{b2f}\{\mathbf{q}\}$ is the body to fixed vector rotation matrix. It transforms vector in the body frame to the fixed world frame. It takes as parameter the attitude $\mathbf{q}$.
- $\mathbf{R}_{f2b}\{\mathbf{q}\}$ is its inverse matrix (from fixed to body).
- $\mathbf{T}_{2a} = (0, 0, 1/m)^T$ is the scaling from thrust to acceleration (by dividing by the weight of the drone: $\mathbf{F} = m\mathbf{a} \Rightarrow \mathbf{a} = \mathbf{F}/m$) and then multiplying by a unit vector $(0, 0, 1)$
- $$R2Q(\boldsymbol{\theta}) = (\cos(\|\boldsymbol{\theta}\|/2), \sin(\|\boldsymbol{\theta}\|/2)\frac{\boldsymbol{\theta}}{\|\boldsymbol{\theta}\|})$$

is a function that convert from a local *rotation vector* $\boldsymbol{\theta}$ to a local quaternion rotation. The definition of this function come from converting $\boldsymbol{\theta}$ to a body-axis angle, and then to a quaternion.

- $$Q2R(\mathbf{q}) = (q.i * s, q.j * s, q.k * s)$$
  is its inverse function where $n = \arccos(q.w) * 2$ and $s = n/\sin(n/2)$
- Δt is the lapse of time between t and the next tick (t+1)

Model

The drone is assumed to have rigid-body physics. It is submitted to the gravity and its own inertia. A rigid body is a solid body in which deformation is zero or so small it can be neglected. The distance between any two given points on a rigid body remains constant in time regardless of external forces exerted on it. This enable to summarise the forces from the rotor as a thrust oriented in the direction normal to the plane formed by the 4 rotosrs, and an angular velocity.

Those variables are sufficient to describe the evolution of our drone with rigid-body physics:

- $\mathbf{a}$ the total acceleration in the fixed world frame
- $\mathbf{v}$ the velocity in the fixed world frame
- $\mathbf{p}$ the position in the fixed world frame
- $\boldsymbol{\omega}$ the angular velocity
- $\mathbf{q}$ the attitude in the fixed world frame

Sensors

The sensors at disposition are:

- **Accelerometer**: It generates $\mathbf{a_A}$ a measurement of the total acceleration in the body frame referrential the drone is submitted to at a **high** sampling rate. If the object is submitted to no acceleration then the accelerometer measure the earth's gravity field from. From that information, it could be possible to retrieve the attitude. Unfortunately, we are in a highly dynamic setting. Thus, it is possible when we can substract the drone's acceleration from the thrust to the total acceleration. This would require to know exactly the force exerced by the rotors at each instant. In this work, we assume that doing that separation, while being

theoretically possible, is too impractical. The measurements model is:

$$\mathbf{a_A}(t) = \mathbf{R}_{f2b}\{\mathbf{q}(t)\}\mathbf{a}(t) + \mathbf{a_A}^{\epsilon}$$

where the covariance matrix of the noise of the accelerometer is $\mathbf{R}_{\mathbf{a_A}\,3\times3}$ and

$$\mathbf{a_A}^{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_{\mathbf{a_A}})$$

.

- **Gyroscope**:It generates $\boldsymbol{\omega}_\mathbf{G}$ a measurement of the angular velocity in the body frame of the drone at the last timestep at a **high** sampling rate. The measurement model is:

$$\boldsymbol{\omega}_\mathbf{G}(t) = \boldsymbol{\omega} + \boldsymbol{\omega}_\mathbf{G}^{\epsilon}$$

where the covariance matrix of the noise of the accelerometer is $\mathbf{R}_{\boldsymbol{\omega}_\mathbf{G}\,3\times3}$ and

$$\boldsymbol{\omega}_{\mathbf{G}t}^{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_{\boldsymbol{\omega}_\mathbf{G}})$$

.

- **Position**: It generates $\mathbf{p_V}$ a measurement of the current positionat a **low** sampling rate. This is usually provided by a **Vicon** (for indoor), **GPS**, a **Tango** or any other position sensor. The measurement model is:

$$\mathbf{p_V}(t) = \mathbf{p}(t) + \mathbf{p_V}^{\epsilon}$$

where the covariance matrix of the noise of the position is $\mathbf{R}_{\mathbf{p_V}\,3\times3}$ and

$$\mathbf{p_V}^{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_{\mathbf{p_V}})$$

.

- **Attitude**: It generates $\mathbf{q_V}$ a measurement of the current attitute. This is usually provided in addition to the position by a **Vicon** or a **Tango** at a **low** sampling rate or the **Magnemoter** at a **high** sampling rate if the environment permit it (no high magnetic interference nearby like iron contamination). The magnetometer retrieve the attitude by assuming that the sensed magnetic field corresponds to the earth's magnetic field. The measurement model is:

$$\mathbf{q_V}(t) = \mathbf{q}(t) * R2Q(\mathbf{q_V}^{\epsilon})$$

where the 3×3 covariance matrix of the noise of the attitude in radian before being converted by $R2Q$ is $\mathbf{R}_{\mathbf{q_V}\,3\times3}$ and

$$\mathbf{q_V}^{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_{\mathbf{q_V}})$$

.

- **Optical Flow**: A camera that keeps track of the movement by comparing the difference of the position of some reference points. By using a companion distance sensor, it is able to retrieve the difference between the two perspective and thus the change in angle and position.

$$\mathbf{dq_O}(t) = (\mathbf{q}(t-k)\mathbf{q}(t)) * R2Q(\mathbf{dq_O}^{\epsilon})$$

$$\mathbf{dp_O}(t) = (\mathbf{p}(t) - \mathbf{p}(t-k)) + \mathbf{dp_O}^{\epsilon}$$

where the 3×3 covariance matrix of the noise of the attitude variation in radian before being converted by $R2Q$ is $\mathbf{R}_{\mathbf{dq_O}\,3\times3}$ and

$$\mathbf{dq_O}^{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_{\mathbf{dq_O}})$$

and the position variation covariance matrix $\mathbf{R}_{\mathbf{dp_O}\,3\times3}$ and

$$\mathbf{dp_O}^{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_{\mathbf{dp_O}})$$

.

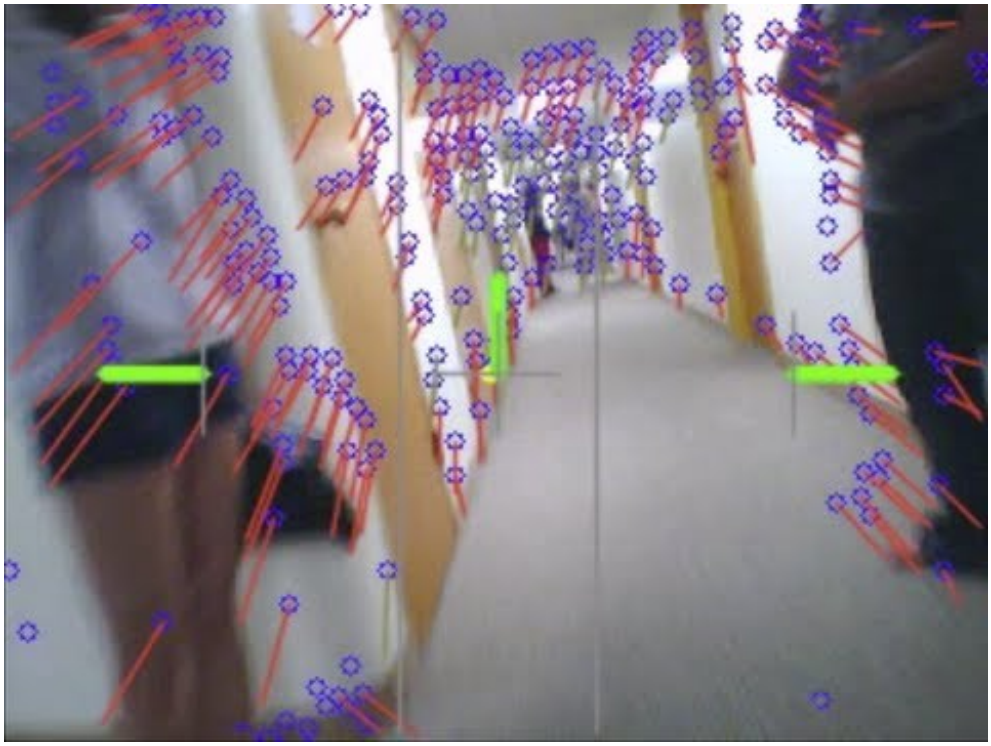


Figure 1.4: Optical flow from a moving drone

The notable difference with the position or attitude sensor is that the optical flow sensor, like the IMU, only captures time variation, not absolute values.

- **Altimeter**: An altimeter is a sensor that measure the altitude of the drone. For instance a LIDAR measure the time for the laser wave to reflect on a surface that is assumed to be the ground. A smart strategy is to only use the altimeter is oriented with a low angle to the earth, else you also have to account that angle in the estimation of the altitude.

$$z_A(t) = \sin(\text{pitch}(\mathbf{q}(\mathbf{t})))(\mathbf{p}(t).z + z_A^\epsilon)$$

$R_{z_A 3\times 3}$ and

$$z_A^\epsilon \sim \mathcal{N}(0, R_{z_A})$$

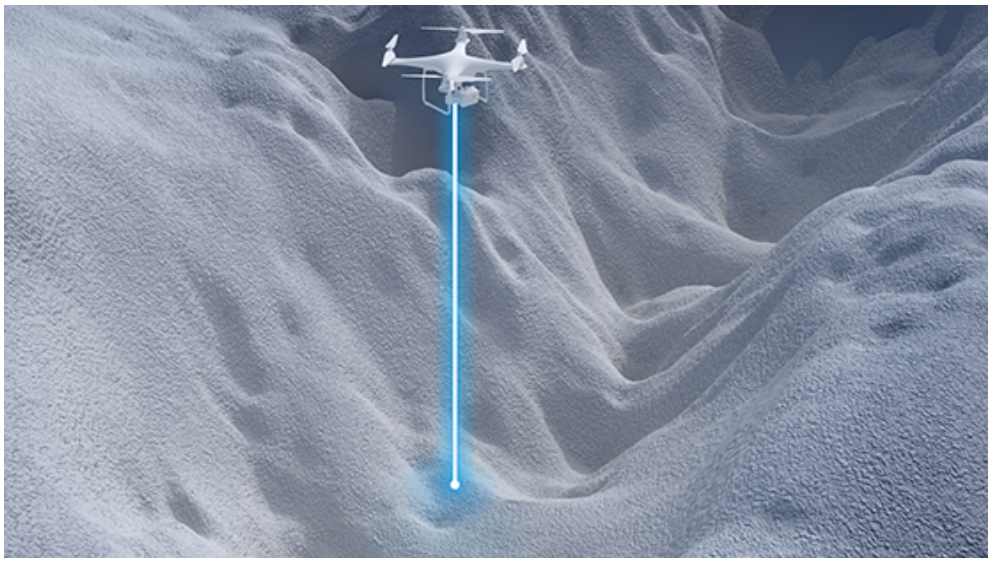


Figure 1.5: Rendering of the LIDAR laser of an altimeter

Some sensors are more relevant indoor and some others outdoor:

- **Indoor**: The sensors available indoor are the accelerometer, the gyroscope and the **Vicon**. The Vicon is a system composed of many sensors around a room that is able to track very accurately the position and orientation a mobile object. One issue with relying solely on the **Vicon** is that the sampling rate is low.

- **Outdoor**: The sensors available outdoor are the accelerometer, the gyroscope, the magnetometer, two GPS, an optical flow and an altimeter.

We assume that since the biases of the sensor could be known prior to the flight, the sensor have been calibrated and output measurements with no bias. Some filters like the ekf2 of the px4 flight stack keep track of the sensor biases but this is a state augmentation that was not deemed worthwhile.

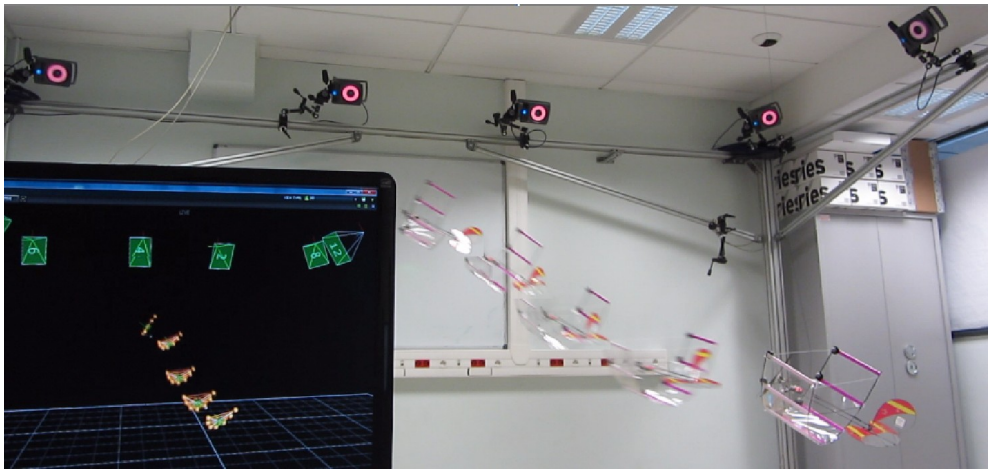


Figure 1.6: A Vicon setup

Control inputs

Observations from the control input are not strictly speaking measurements but input of the state-transition model. The IMU is a sensor, thus stricly speaking, its measurements are not control inputs. However, in the literature, it is standard to use its measurements as control inputs. One of the advantage is that the accelerometer measures acceleration and angular velocity, raw values close from the input we need in our state-transition. If we used a transformation of the thrust sent as command to the rotors, we would have to account for the rotors unprecision, the wind and other disturbances. Another advantage is that since the IMU has very high sampling rate, we can update very frequently the state with new transitions. The drawback is that the accelerometer is noisy. Fortunately, we can take into account the noise as a process model noise.

The control inputs at disposition are:

- **Acceleration**: $\mathbf{a}_{\mathbf{A}t}$ from the acceloremeter
- **Angular velocity**: $\boldsymbol{\omega}_{\mathbf{G}t}$ from the gyroscope.

Model dynamic

- $\mathbf{a}(t+1) = \mathbf{R}_{b2f}\{\mathbf{q}(t+1)\}(\mathbf{a}_{\mathbf{A}t} + \mathbf{a}_{\mathbf{A}t}^{\epsilon})$ where $\mathbf{a}_t^{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_{\mathbf{a}_t})$
- $\mathbf{v}(t+1) = \mathbf{v}(t) + \Delta t \mathbf{a}(t) + \mathbf{v}_t^{\epsilon}$ where $\mathbf{v}_t^{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_{\mathbf{v}_t})$
- $\mathbf{p}(t+1) = \mathbf{p}(t) + \Delta t \mathbf{v}(t) + \mathbf{p}_t^{\epsilon}$ where $\mathbf{p}_t^{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_{\mathbf{p}_t})$
- $\boldsymbol{\omega}(t+1) = \boldsymbol{\omega}_{\mathbf{G}t} + \boldsymbol{\omega}_{\mathbf{G}t}^{\epsilon}$ where $\mathbf{p}_t^{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}_{\boldsymbol{\omega}_{\mathbf{G}t}})$
- $\mathbf{q}(t+1) = \mathbf{q}(t) * R2Q(\Delta t \boldsymbol{\omega}(\boldsymbol{t}))$

Note that in our model, $\mathbf{q}(t+1)$ must be known. Fortunately, as we will see later, our Rao-Blackwellized Particle Filter is conditionned under the attitude so it is known.

State

The time series of the variables of our dynamic model constitute a hidden markov chain. Indeed, the model is "memoryless" and depends only on the current state and a sampled transition.

States contain variables that enable us to keep track of some of those hidden variables which is our ultimate goal (for POSE $\mathbf{p}$ and $\mathbf{q}$). States at time t are denoted by $\mathbf{x}_t$. Different filters require different state variables depending on their structure and assumptions.

Observation

Observations are revealed variables conditionned under the variables of our dynamic model. Our ultimate goal is to deduce the states from the observations.

Observations contain the control input $\mathbf{u}$ and the measurements $\mathbf{z}$.

$$\mathbf{y}_t = (\mathbf{z}_t, \mathbf{u}_t)^T = (\mathbf{p}_{\mathbf{V}t}, \mathbf{q}_{\mathbf{V}t}), (t_{Ct}, \boldsymbol{\omega}_{\mathbf{C}t}))^T$$

Filtering and smoothing

Smoothing is the statistical task of finding the expectation of the state variable from the past history of observations and multiple observation variables ahead

$$\mathbb{E}[g(\mathbf{x}_{0:t})|\mathbf{y}_{1:t+k}]$$

Which expand to,

$$\mathbb{E}[(\mathbf{p}_{0:t}, \mathbf{q}_{0:t})|(\mathbf{z}_{1:t+k}, \mathbf{u}_{1:t+k})]$$

k is a contant and the first observation is y_1

Filtering is a kind of smoothing where you only have at disposal the current observation variable ($k = 0$)

Complementary Filter

The complementary filter is the simplest of all filter and very common to retrieve the attitude because of its low computational complexity. The gyroscope and accelerometer both provide a measurement that can help us to estimate the attitude. The gyroscope indeed gives us a noisy measurement of the angular velocity from which we can retrieve the new attitude from the past one by time integration: $\mathbf{q}_t = \mathbf{q}_{t-1} * R2Q(\Delta t \omega)$.

This is commonly called "Dead reckoning"[1] and is prone to accumulation error, referred as drift. Indeed, like brownian motions, even if the process is unbiased, the variance grows with time. Reducing the noise cannot solve the issue entirely: even with extremely precise instruments, you are subject to floating point errors.

Fortunately, even though the accelerometer gives us a highly noisy (vibrations, wind, etc …) measurement of the orientation, it is not subject to drift because it does not rely on accumulation. Indeed, if not subject to other accelerations, the accelerometer measures the gravity field orientation. Since this field is oriented toward earth, it is possible to retrieve the current rotation from that field and by extension the attitude. However, in the case of a drone, it is subject to continuous and signifiant acceleration and vibration. Hence, the assumption that we retrieve the gravity field directly is wrong. Nevertheless, We could solve this by substracting the acceleration deduced from the thrust control input. It is unpractical so this approach is not pursued in this work, but understanding this filter is still useful.

The idea of the filter itself is to combine the precise "short-term" measurements of the gyroscope subject to drift with the "long-term" measurements of the accelerometer.

[1]The etymology for "Dead reckoning" comes from the mariners of the XVIIth century that used to calculate the position of the vessel with log book. The interpretation of "dead" is subject to debate. Some argue that it is a mispelling of "ded" as in "deduced". Others argue that it should be read by its old meaning: *absolute*.

State

This filter is very simple and it is only needed to store as a state the last estimated attitude along with its timestamp (to calculate Δt).

$$\mathbf{x}_t = \mathbf{q}_t$$

$$\hat{\mathbf{q}}_{t+1} = \alpha(\hat{\mathbf{q}}_t + \Delta t \omega_t) + (1 - \alpha)\mathbf{q}_{\mathbf{A}t+1}$$

$\alpha \in [0, 1]$. Usually, α is set to a high-value like 0.98. It is very intuitive to see why this should approximately "work", the data from the accelerometer continuously correct the drift from the gyroscope.

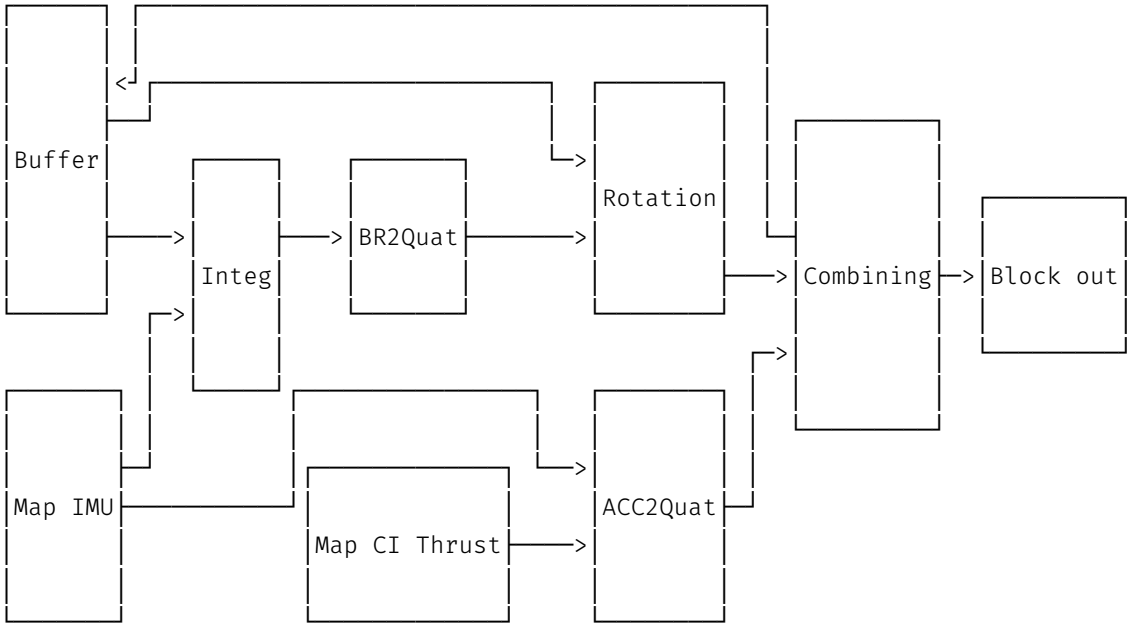


Figure 1.7: Complementary Filter graph structure

Figure 9 is the plot of the distance from the true quaternion after 15s of an arbitrary trajectory when $\alpha = 1.0$ meaning that the accelerometer does not correct the drift.

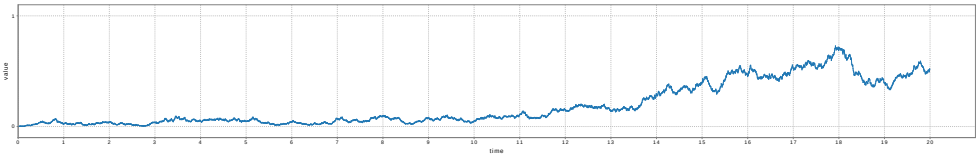


Figure 1.8: CF with alpha = 1.0

Figure 10 is that same trajectory with $\alpha = 0.98$.

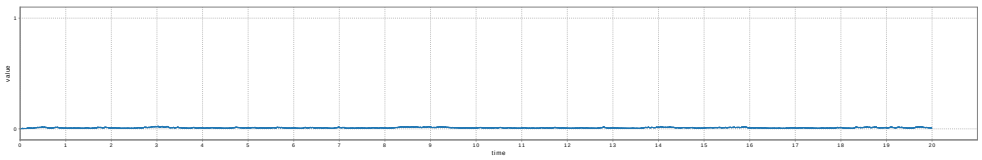


Figure 1.9: CF with alpha = 0.98

We can observe here the long-term importance of being able to correct the drift, even if ever so slightly at each timestep.

Asynchronous Augmented Complementary Filter

As explained previously, in this highly-dynamic setting, combining the gyroscope and the accelerometer to retrieve the attitude is not satisfactory. However, we can reuse the intuition from the complementary filter, which is to combine precise but drifting short-term measurements to other measurements that do not suffer from drift. This enable a simple and computionally inexpensive novel filter that we will be able to use later as a baseline. In this case, the short-term measurements are the acceleration and angular velocity from the IMU, and the non drifting measurements come from the Vicon.

We will also add the property that the data from the sensors are asynchronous. This is a consequence of the sensors having different sampling rate.

- **IMU** update

$$\mathbf{v}_t = \mathbf{v}_{t-1} + \Delta t_v \mathbf{a}_{\mathbf{A}t}$$

$$\boldsymbol{\omega}_t = \boldsymbol{\omega}_{\mathbf{G}t}$$

$$\mathbf{p}_t = \mathbf{p}_{t-1} + \Delta t \mathbf{v}_{t-1}$$

$$\mathbf{q}_t = \mathbf{q}_{t-1} R2Q(\Delta t \boldsymbol{\omega}_{t-1})$$

- **Vicon** update

$$\mathbf{p}_t = \alpha \mathbf{p}_{\mathbf{V}} + (1-\alpha)(\mathbf{p}_{t-1} + \Delta t \mathbf{v}_{t-1})$$

$$\mathbf{q}_t = \alpha \mathbf{q}_{\mathbf{V}} + (1-\alpha)(\mathbf{q}_{t-1} R2Q(\Delta t \boldsymbol{\omega}_{t-1}))$$

State

The state has to be more complex because the filter now estimates both the position and the attitude. Furthermore, because of asynchronousity, we have to store the last angular velocity, the last linear velocity, and the last

time the linear velocity has been updated (to retrieve $\Delta t_v = t - t_a$ where t_a is the last time we had an update from the accelerometer).

$$\mathbf{x}_t = (\mathbf{p}_t, \mathbf{q}_t, \boldsymbol{\omega}_t, \mathbf{a}_t, t_a)$$

The structure of this filter and all of the filters presented thereafter is as follow:

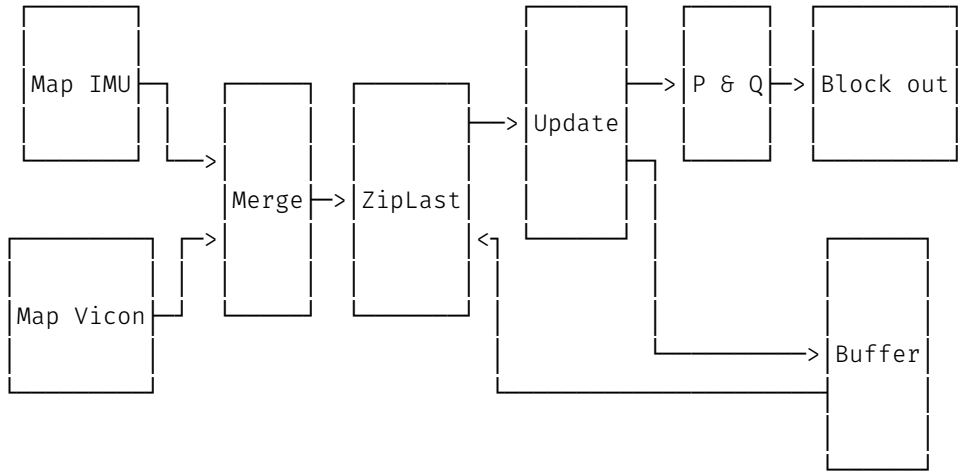


Figure 1.10: A graph of the filters structure in scala-flow

Kalman Filter

Bayesian inference

Bayesian inference is a method of statistical inference in which Bayes' theorem is used to update the probability for a hypothesis as more evidence or information becomes available. In this Bayes setting, the prior is the estimated distribution of the previous state at time $t-1$, the likelihood correspond to the likelihood of getting the new data from the sensor given the prior and finally, the posterior is the updated estimated distribution.

Model

The kalman filter requires that both the model process and the measurement process are **linear gaussian**. Linear gaussian processes are of the

form:

$$\mathbf{x}_t = f(\mathbf{x}_{t-1}) + \mathbf{w}_t$$

where f is a linear function and $\mathbf{w}_t$ a gaussian process: it is sampled from an arbitrary gaussian distribution.

The Kalman filter is a direct application of bayesian inference. It combines the prediction of the distribution given the estimated prior state and the state-transition model.

$$\mathbf{x}_t = \mathbf{F}_t\mathbf{x}_{t-1} + \mathbf{B}_t\mathbf{u}_t + \mathbf{w}_t$$

- $\mathbf{x}_t$ the state
- $\mathbf{F}_t$ the state transition model
- $\mathbf{B}_t$ the control-input model
- $\mathbf{u}_t$ the control vector
- $\mathbf{w}_t$ process noise drawn from $\mathbf{w}_t \sim N(0, \mathbf{Q}_k)$

and the estimated distribution given the data coming from the sensors.

$$\mathbf{y}_t = \mathbf{H}_t\mathbf{x}_t + \mathbf{v}_t$$

- $\mathbf{y}_t$ measurements
- $\mathbf{H}_t$ the state to measurement matrix
- $\mathbf{w}_t$ measurement noise drawn from $\mathbf{w}_t \sim N(0, \mathbf{R}_k)$

Because, both the model process and the sensor process are assumed to be linear gaussian, we can combine them into a gaussian distribution. Indeed, the product of two gaussians is gaussian.

$$P(\mathbf{x}_t) \propto P(\mathbf{x}_t^-|\mathbf{x}_{t-1}) \cdot P(\mathbf{x}_t|\mathbf{y}_t)$$
$$\mathcal{N}(\mathbf{x}_t) \propto \mathcal{N}(\mathbf{x}_t^-|\mathbf{x}_{t-1}) \cdot \mathcal{N}(\mathbf{x}_t|\mathbf{y}_t)$$

where $\mathbf{x}_t^-$ is the predicted state from the previous state and the state-transition model.

The kalman filter keep track of the parameters of that gaussian: the mean state and the covariance of the state which represent the uncertainty about our last prediction. The mean of that distribution is also the best current state estimation of the filter.

By keeping track of the uncertainty, we can optimally combine the normals by knowing what importance to give to the difference between the

expected sensor data and the actual sensor data. That factor is the Kalman gain.

- **predict**:
  - predicted **state**: $\hat{\mathbf{x}}_t^- = \mathbf{F}_t\mathbf{x}_{t-1} + \mathbf{B}_t\mathbf{u}_t$
  - predicted **covariance**: $\mathbf{\Sigma}_t^- = \mathbf{F}_{t-1}\mathbf{\Sigma}_{t-1}^-\mathbf{F}_{t-1}^T + \mathbf{Q}_t$
- **update**:
  - predicted **measurements**: $\hat{\mathbf{z}} = \mathbf{H}_t\hat{\mathbf{x}}_t^-$
  - **innovation**: $(\mathbf{z}_t - \hat{\mathbf{z}})$
  - **innovation covariance**: $\mathbf{S} = \mathbf{H}_t\mathbf{\Sigma}_t^-\mathbf{H}_t^T + \mathbf{R}_t$
  - optimal **kalman gain**: $\mathbf{K} = \mathbf{\Sigma}_t^-\mathbf{H}_t^T\mathbf{S}^{-1}$
  - updated **state**: $\mathbf{\Sigma}_t = \mathbf{\Sigma}_t^- + \mathbf{KSK}^T$
  - updated **covariance**: $\hat{\mathbf{x}}_t = \hat{\mathbf{x}}_t^- + \mathbf{K}(\mathbf{z}_t - \hat{\mathbf{z}})$

Asynchronous Kalman Filter

It is not necessary to apply the full kalman update at each measurement. Indeed, $\mathbf{H}$ can be sliced to correspond to the measurements currently available.

To be truly asynchronous, you also have to account for the different sampling rates. There is two cases :

- The required data for the update step (the control inputs) can arrive multiple time before any of the data of the update step (the measurements) occur.
- Inversely, it is possible that the measurements occur at a higher sampling rate than the control inputs.

The strategy chosen here is as follow:

1. Multiple prediction steps without any update step may happen without making the algorithm inconsistent.
2. An update is **always** immediatly preceded by a prediction step. This is a consequence of the requirement that the innovation must measure the difference between the predicted measurement from the state at the exact current time and the measurements. Thus, if the measurements are not synchronised with the control inputs, use the most likely control input for the prediction step, which might result in simply repeating them. Repeating the last control input was the method used for the accelerometer and the gyroscope data as control input.

Extended Kalman Filters

In the previous section, we have shown that the Kalman Filter is only applicable when both the process model and the measurement model are linear gaussian process. This has two aspects:

- The noise of the measurements and of the state-transition must be gaussian
- The state-transition function and the measurement to state function must be linear.

Furthermore, it is provable that kalman filters are optimal linear filters.

However, in our context, one component of the state, the attitude, is intrisically non-linear. Indeed, rotations and attitudes belong to $SO(3)$ which is not a vector space. Therefore, we cannot use *vanilla* kalman filters. The filters that we present thereafter relax those requirements.

One example of such extension is the extended kalman filter (EKF) that we will present here. The EKF relax the linearity requirement by using differentiation tocalculate an approximation of the first order of the required linear functions. Our state transition function and measurement function can now be expressed in the free forms $f(\mathbf{x}_t)$ and $h(\mathbf{x}_t)$ and we define the matrix $\mathbf{F}_t$ and $\mathbf{H}_t$ as their jacobian.

$$\mathbf{F}_{t10\times10} = \left.\frac{\partial f}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_{t-1},\mathbf{u}_{t-1}}$$

$$\mathbf{H}_{t7\times7} = \left.\frac{\partial h}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_t}$$

- **predict**:
  - predicted **state**: $\hat{\mathbf{x}}_t^- = f(\mathbf{x}_{t-1}) + \mathbf{B}_t\mathbf{u}_t$
  - predicted **covariance**: $\mathbf{\Sigma}_t^- = \mathbf{F}_{t-1}\mathbf{\Sigma}_{t-1}^-\mathbf{F}_{t-1}^T + \mathbf{Q}_t$
- **update**:
  - predicted **measurements**: $\hat{\mathbf{z}} = h(\hat{\mathbf{x}}_t^-)$
  - **innovation**: $(\mathbf{z}_t - \hat{\mathbf{z}})$
  - **innovation covariance**: $\mathbf{S} = \mathbf{H}_t\mathbf{\Sigma}_t^-\mathbf{H}_t^T + \mathbf{R}_t$
  - optimal **kalman gain**: $\mathbf{K} = \mathbf{\Sigma}_t^-\mathbf{H}_t^T\mathbf{S}^{-1}$
  - updated **state**: $\mathbf{\Sigma}_t = \mathbf{\Sigma}_t^- + \mathbf{KSK}^T$
  - updated **covariance**: $\hat{\mathbf{x}}_t = \hat{\mathbf{x}}_t^- + \mathbf{K}(\mathbf{z}_t - \hat{\mathbf{z}})$

EKF for POSE

State

For the EKF, we are gonna use the following state:

$$\mathbf{x}_t = (\mathbf{v}_t, \mathbf{p}_t, \mathbf{q}_t)^T$$

Initial state $\mathbf{x}_0$ at $(\mathbf{0}, \mathbf{0}, (1, 0, 0, 0))$

Indoor Measurements model

1. Position:
$$\mathbf{pv}(t) = \mathbf{p}(t)^{(i)} + \mathbf{pv}_t^{\epsilon}$$
where $\mathbf{pv}_t^{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_{\mathbf{pv}_t})$
2. Attitude:
$$\mathbf{qv}(t) = \mathbf{q}(t)^{(i)} * R2Q(\mathbf{qv}_t^{\epsilon})$$
where $\mathbf{qv}_t^{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_{\mathbf{qv}_t})$

Kalman prediction

The model dynamic defines the following model, state-transition function $f(\mathbf{x}, \mathbf{u})$ and process noise $\mathbf{w}$ with covariance matrix $\mathbf{Q}$

$$\mathbf{x}_t = f(\mathbf{x}_{t-1}, \mathbf{u}_t) + \mathbf{w}_t$$

$$f((\mathbf{v}, \mathbf{p}, \mathbf{q}), (\mathbf{a_A}, \boldsymbol{\omega_G})) = \begin{pmatrix} \mathbf{v} + \Delta t \mathbf{R}_{b2f}\{\mathbf{q}_{t-1}\}\mathbf{a} \\ \mathbf{p} + \Delta t \mathbf{v} \\ \mathbf{q} * R2Q(\Delta t \boldsymbol{\omega}_G) \end{pmatrix}$$

Now, we need to derive the jacobian of f. We will use sagemath to retrieve the 28 relevant different partial derivatives of q.

$$\mathbf{F}_{t10\times10} = \left.\frac{\partial f}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_{t-1}, \mathbf{u}_{t-1}}$$

$$\hat{\mathbf{x}}_t^{-(i)} = f(\mathbf{x}_{t-1}^{(i)}, \mathbf{u}_t)$$

$$\mathbf{\Sigma}_t^{-(i)} = \mathbf{F}_{t-1}\mathbf{\Sigma}_{t-1}^{-(i)}\mathbf{F}_{t-1}^T + \mathbf{Q}_t$$

Kalman measurements update

$$\mathbf{z}_t = h(\mathbf{x}_t) + \mathbf{v}_t$$

The measurement model defines $h(\mathbf{x})$

$$\begin{pmatrix} \mathbf{p_V} \\ \mathbf{q_V} \end{pmatrix} = h((\mathbf{v}, \mathbf{p}, \mathbf{q})) = \begin{pmatrix} \mathbf{p} \\ \mathbf{q} \end{pmatrix}$$

The only complex partial derivatives to calculate are the one of the acceleration, because they have to be rotated first. Once again, we use sagemath: $\mathbf{H_a}$ is defined by the script in the appendix B.

$$\mathbf{H}_{t10\times 7} = \left.\frac{\partial h}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_t} = \begin{pmatrix} \mathbf{0}_{3\times 3} & & \\ & \mathbf{I}_{3\times 3} & \\ & & \mathbf{I}_{4\times 4} \end{pmatrix}$$

$$\mathbf{R}_{t7\times 7} = \begin{pmatrix} \mathbf{R_{pv}} & \\ & \mathbf{R'_{qv}}_{4\times 4} \end{pmatrix}$$

$\mathbf{R'_{qv}}$ has to be 4×4 and has to represent the covariance of the quaternion. However, the actual covariance matrix $\mathbf{R_{qv}}$ is 3×3 and represent the noise in terms of a *rotation vector* around the x, y, z axes.

We transform this rotation vector into a quaternion using our function *R2Q*. We can compute the new covariance matrix $\mathbf{R'_{qv}}$ using Unscented Transform.

Unscented Transform

The unscented transform (UT) is a mathematical function used to estimate statistics after applying a given nonlinear transformation to a probability distribution. The idea is to use points that are representative of the original distribution, sigma points. We apply the transformation to those sigma points and calculate the new statistics using the transformed sigma points. The sigma points must have the same mean and covariance than the original distribution.

The minimal set of symmetric sigma points can be found using the covariance of the initial distribution. The $2N + 1$ minimal symmetric set of sigma points are the mean and the set of points corresponding to the mean plus and minus one of the direction corresponding to the covariance matrix. In one dimension, the square root of the variance is enough. In N-dimension, you must use the cholesky decomposition of the covariance matrix. The cholesky decomposition find the matrix L such that $\Sigma = LL^t$.

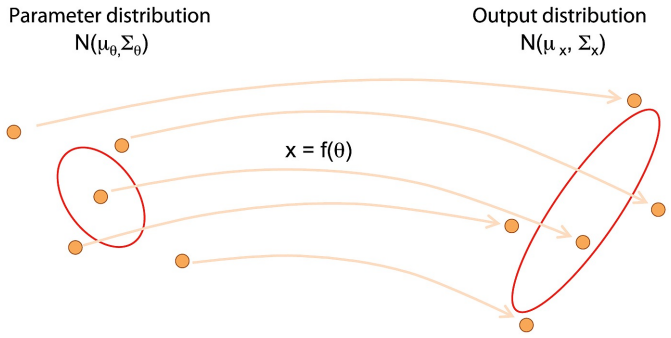


Figure 1.11: Unscented tranform

Kalman update

$$\mathbf{S} = \mathbf{H}_t \mathbf{\Sigma}_t^- \mathbf{H}_t^T + \mathbf{R}_t$$
$$\hat{\mathbf{z}} = h(\hat{\mathbf{x}}_t^-)$$
$$\mathbf{K} = \mathbf{\Sigma}_t^- \mathbf{H}_t^T \mathbf{S}^{-1}$$
$$\mathbf{\Sigma}_t = \mathbf{\Sigma}_t^- + \mathbf{K}\mathbf{S}\mathbf{K}^T$$
$$\hat{\mathbf{x}}_t = \hat{\mathbf{x}}_t^- + \mathbf{K}(\mathbf{z}_t - \hat{\mathbf{z}})$$

F partial derivatives

```
Q.<i,j,k> = QuaternionAlgebra(SR, -1, -1)

var('q0, q1, q2, q3')
var('dt')
var('wx, wy, wz')

q = q0 + q1*i + q2*j + q3*k

w = vector([wx, wy, wz])*dt
```

```
w_norm = sqrt(w[0]^2 + w[1]^2 + w[2]^2)
ang = w_norm/2
w_normalized = w/w_norm
sin2 = sin(ang)
qd = cos(ang) + w_normalized[0]*sin2*i + w_normalized[1]*sin2*j
    + w_normalized[2]*sin2*k

nq = q*qd

v = vector(nq.coefficient_tuple())

for sym in [wx, wy, wz, q0, q1, q2, q3]:
    d = diff(v, sym)
    exps = map(lambda x: x.canonicalize_radical().full_simplify(), d)
    for i, e in enumerate(exps):
        print(sym, i, e)
```

Unscented Kalman Filters

The EKF has 3 flaws in our case:

- The linearization gives an approximate form which result in approximation errors
- The prediction step of the EKF assume that the linearized form of the transformation can capture all the information needed to apply the transformation to the gaussian distribution pre-transformation. Unfortunately, this is only true near the region of the mean. The transformation of the tail of the gaussian distribution may need to be very different.
- It attempts to define a gaussian covariance matrix for the attitude quaternion. This does not make sense because it does not account for the requirement of the quaternion being in a 4 dimensional unit sphere.

The Unscented Kalman Filter (UKF) does not suffer from the two first flaws, but it is more computationally expensive as it requires a cholesky factorisation that grows exponentially in complexity with the number of dimensions.

Indeed, the UKF applies an unscented transformation to sigma points of the current approximated distribution. The statistics of the new approximated gaussian are found through this unscented transform. The EKF linearizes the transformation, the UKF approximates the resulting gaussian after the transformation. Hence, the UKF can take into account the effects of the transformation away from the mean which might be drastically different.

The implementation of an UKF still suffer greatly from quaternion

not belonging to a vector space. The approach taken by [3] is to use the error quaternion defined by $\mathbf{e}_i = \mathbf{q}_i\bar{\mathbf{q}}$. This approach has the benefit that similar quaternion differences result in similar error. But apart from that, it does not have any profound justification. We must compute a sound average weighted quaternion of all sigma points. An algorithm is described in the following section.

Average quaternion

Unfortunately, the average of quaternions components $\frac{1}{N}\sum q_i$ or *barycentric* mean is unsound: Indeed, attitude do not belong to a vector space but a homogenous Riemannian manifold (the four dimensional unit sphere). To convince yourself of the unsoundness of the *barycentric* mean, see that the addition and barycentric mean of two unit quaternion is not necessarily an unit quaternion ($(1,0,0,0)$ and $(-1,0,0,0)$ for instance. Furthermore, angle being periodic, the *barycentric* mean of a quaternion with angle $-178°$ and another with same body-axis and angle $180°$ gives $1°$ instead of the expected $-179°$.

To calculate the average quaternion, we use an algorithm which minimize a metric that correspond to the weighted attitude difference to the average, namely the weighted sum of the squared Frobenius norms of attitude matrix differences.

$$\bar{\mathbf{q}} = arg\min_{q\in\mathbb{S}^3} \sum w_i \|A(\mathbf{q}) - A(\mathbf{q}_i)\|_F^2$$

where $\mathbb{S}^3$ denotes the unit sphere.

The attitude matrix $A(\mathbf{q})$ and its corresponding Frobenius norm have been described in the quaternion section.

Intuition

The intuition of keeping track of multiple representative of the distribution is exactly the approach taken by the particle filter. The particle filter has the advantage that the distribution is never transformed back to a gaussian so there is less assumption made about the noise and the transformation. It is only required to be able to calculate the expectation from a weighted set of particles.

Particle Filter

Particle filters are sequential monte carlo methods. Like all monte carlo method, they rely on repeated sampling for estimation of a distribution.

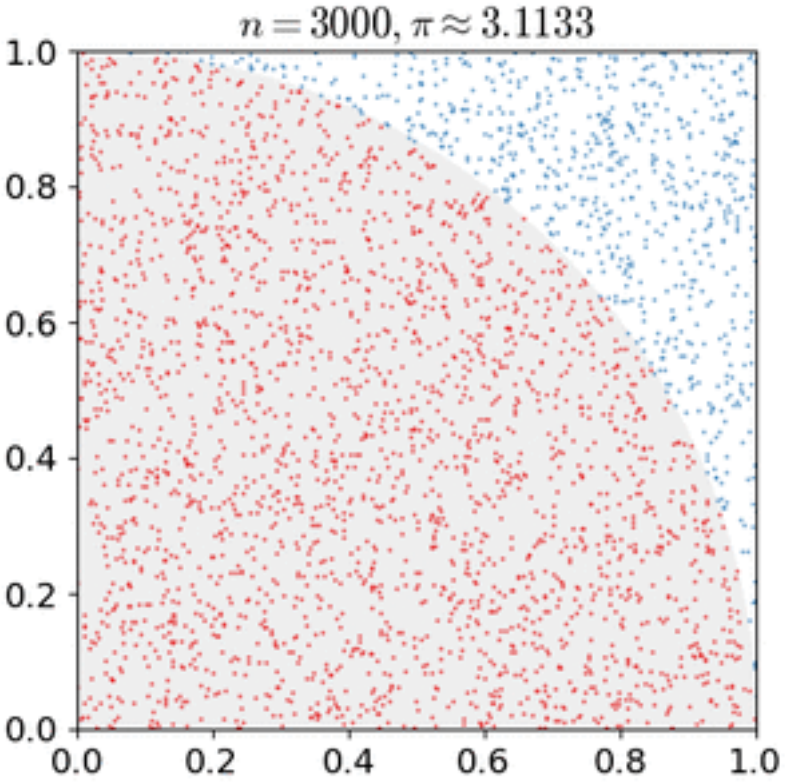


Figure 1.12: Monte carlo estimation of pi

The particle filter itself a weighted particle representation of the posterior:

$$p(\mathbf{x}) = \sum w^{(i)} \delta(\mathbf{x} - \mathbf{x}^{(i)})$$

where δ is the dirac delta function. The dirac delta function is zero everywhere except at zero, with an integral of one over the entire real line. It represents here the ideal probability density of a particle.

Importance sampling

The weights are computed through importance sampling. With importance sampling, each particle does not represent equally the distribution. Importance sampling enables to use sampling from another distribution to estimate properties from the target distribution of interest. In most cases, it can be used to focus sampling on a specific region. But in our case, it enables us to reweight particles based on their likelihood from the measurements.

Importance sampling is based on the identity:

$$\begin{aligned}\mathbb{E}[\mathbf{g}(\mathbf{x})|\mathbf{y}_{1:T}] &= \int \mathbf{g}(\mathbf{x}) p(\mathbf{x}|\mathbf{y}_{1:T}) d\mathbf{x} \\ &= \int \left[\mathbf{g}(\mathbf{x}) \frac{p(\mathbf{x}|\mathbf{y}_{1:T})}{\pi(\mathbf{x}|\mathbf{y}_{1:T})}\right] \pi(\mathbf{x}|\mathbf{y}_{1:T}) d\mathbf{x}\end{aligned}$$

Sequential Importnce Samplng

** TODO **

Particle filters are very computionally expensive and that is why their usage is not very popular currently for low-powered embedded systems like drones (But they are used in Avionics). Using accelerated hardware is one way to enable them on drones.

Resampling

When the number of effective particles is too low ($N/10$), we apply systematic resampling. The idea behind resampling is simple. The distribution is represented by a number of particles with different weights. As time goes, the repartition of weights degenerate. A large subset of particles ends up having negligible weight which make them irrelevant. In the most extreme case, one particle represents the whole distribution. To avoid that degeneration, when the weights are too unbalanced, we resample from the weights distribution: pick N times among the particle and assign them a weight of $1/N$, each pick has odd w_i to pick the particle p_i. Thus, some particles with large weights are splitted up into smaller clone particle and others with small weight are never picked. This process is similar to evolution, at each generation, the most promising branch survive and replicate while the less promising die off.

A popular method for resampling is systematic sampling as described by [4]:

Sample $U_1 \sim \mathcal{U}[0, \frac{1}{N}]$ and define $U_i = U_1 + \frac{i-1}{N}$ for $i = 2, \ldots, N$

Rao-Blackwellized Particle Filter

Introduction

Compared to a plain PF, RPBF leverage the linearity of some components of the state by assuming our model gaussian conditionned on a latent variable: Given the attitude q_t, our model is linear. This is where RPBF shines: We use particle filtering to estimate our latent variable, the attitude, and we use the optimal kalman filter to estimate the state variable.

This main inspiration from this approach is [5]. However, it differs by:

- adapt the filter to drones by taking into account that the system is too dynamic for assuming that the accelerometer simply output the gravity vector. This is solved by augmenting the state with the acceleration as shown later.
- not use measurements of the IMU as control inputs (this is usually used for wheeled vehicles because of the drift from the wheels) but have both control inputs and measurements.
- add an attitude sensor.

We introduce the latent variable $\boldsymbol{\theta}$

The latent variable $\boldsymbol{\theta}$ has for sole component the attitude:

$$\boldsymbol{\theta} = (\mathbf{q})$$

q_t is estimated from the product of the attitude of all particles $\theta^{(\mathbf{i})} = \mathbf{q}_t^{(i)}$ as the "average" quaternion $\mathbf{q}_t = avgQuat(\mathbf{q}_t^n)$. x^n designates the product of all n arbitrary particle.

The weight definition is:

$$w_t^{(i)} = \frac{p(\boldsymbol{\theta}_{0:t}^{(i)}|\mathbf{y}_{1:t})}{\pi(\boldsymbol{\theta}_{0:t}^{(i)}|\mathbf{y}_{1:t})}$$

From the definition, it is proovable that:

$$w_t^{(i)} \propto \frac{p(\mathbf{y}_t|\boldsymbol{\theta}_{0:t-1}^{(i)}, \mathbf{y}_{1:t-1})p(\boldsymbol{\theta}_t^{(i)}|\boldsymbol{\theta}_{t-1}^{(i)})}{\pi(\boldsymbol{\theta}_t^{(i)}|\boldsymbol{\theta}_{1:t-1}^{(i)}, \mathbf{y}_{1:t})} w_{t-1}^{(i)}$$

We choose the dynamic of the model as the importance distribution:

$$\pi(\boldsymbol{\theta}_t^{(i)}|\boldsymbol{\theta}_{1:t-1}^{(i)}, \mathbf{y}_{1:t}) = p(\boldsymbol{\theta}_t^{(i)}|\boldsymbol{\theta}_{t-1}^{(i)})$$

Hence,

$$w_t^{(i)} \propto p(\mathbf{y}_t|\boldsymbol{\theta}_{0:t-1}^{(i)}, \mathbf{y}_{1:t-1}) w_{t-1}^{(i)}$$

We then sum all $w_t^{(i)}$ to find the normalization constant and retrieve the actual $w_t^{(i)}$

State

$$\mathbf{x}_t = (\mathbf{v}_t, \mathbf{p}_t)^T$$

Initial state $\mathbf{x}_0 = (\mathbf{0}, \mathbf{0}, \mathbf{0})$

Initial covariance matrix $\boldsymbol{\Sigma}_{6\times6} = \epsilon \mathbf{I}_{6\times6}$

Latent variable

$$\mathbf{q}_{t+1}^{(i)} = \mathbf{q}_t^{(i)} * R2Q(\Delta t(\boldsymbol{\omega}_{\mathbf{G}t} + \boldsymbol{\omega}_{\mathbf{G}t}^{\epsilon}))$$

$\boldsymbol{\omega}_{\mathbf{G}t}^{\epsilon}$ represents the error from the control input and is sampled from $\boldsymbol{\omega}_{\mathbf{G}t}^{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_{\boldsymbol{\omega}_{\mathbf{G}t}})$

Initial attitude $\mathbf{q_0}$ is sampled such that the drone pitch and roll are none (parralel to the ground) but the yaw is unknown and uniformly distributed.

Note that $\mathbf{q}(t+1)$ is known in the model dynamic because the model is conditionned under $\boldsymbol{\theta}_{t+1}^{(i)}$.

Indoor Measurement model

1. Position:

$$\mathbf{p_V}(t) = \mathbf{p}(t)^{(i)} + \mathbf{p_V}_t^{\epsilon}$$

where $\mathbf{p}_{\mathbf{V}t}^{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_{\mathbf{p}_{\mathbf{V}t}})$

2. Attitude:

$$\mathbf{q}_{\mathbf{V}}(t) = \mathbf{q}(t)^{(i)} * R2Q(\mathbf{q}_{\mathbf{V}}^{\epsilon}_t)$$

where $\mathbf{q}_{\mathbf{V}}^{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{R}_{\mathbf{q}_{\mathbf{V}t}})$

Kalman prediction

The model dynamics define the following model, state-transition matrix $\mathbf{F}_t\{\boldsymbol{\theta}_t^{(i)}\}$, the control-input matrix $\mathbf{B}_t\{\boldsymbol{\theta}_t^{(i)}\}$, the process noise $\mathbf{w}_t\{\boldsymbol{\theta}_t^{(i)}\}$ for the Kalman filter and its covariance $\mathbf{Q}_t\{\boldsymbol{\theta}_t^{(i)}\}$

$$\mathbf{x}_t = \mathbf{F}_t\{\boldsymbol{\theta}_t^{(i)}\}\mathbf{x}_{t-1} + \mathbf{B}_t\{\boldsymbol{\theta}_t^{(i)}\}\mathbf{u}_t + \mathbf{w}_t\{\boldsymbol{\theta}_t^{(i)}\}$$

$$\mathbf{F}_t\{\boldsymbol{\theta}_t^{(i)}\}_{6\times 6} = \begin{pmatrix} \mathbf{I}_{3\times 3} & 0 \\ \Delta t\ \mathbf{I}_{3\times 3} & \mathbf{I}_{3\times 3} \end{pmatrix}$$

$$\mathbf{B}_t\{\boldsymbol{\theta}_t^{(i)}\}_{6\times 3} = \begin{pmatrix} \mathbf{R}_{b2f}\{\mathbf{q}_t^{(i)}\}\mathbf{a}_{\mathbf{A}} \\ \mathbf{0}_{3\times 3} \end{pmatrix}$$

$$\mathbf{Q}_t\{\boldsymbol{\theta}_t^{(i)}\}_{6\times 6} = \begin{pmatrix} \mathbf{R}_{b2f}\{\mathbf{q}_t^{(i)}\}(\mathbf{Q}_{\mathbf{a}_t} * dt^2)\mathbf{R}_{b2f}^t\{\mathbf{q}_t^{(i)}\} & \\ & \mathbf{Q}_{\mathbf{v}_t} \end{pmatrix}$$

$$\hat{\mathbf{x}}_t^{-(i)} = \mathbf{F}_t\{\boldsymbol{\theta}_t^{(i)}\}\mathbf{x}_{t-1}^{(i)} + \mathbf{B}_t\{\boldsymbol{\theta}_t^{(i)}\}\mathbf{u}_t$$
$$\boldsymbol{\Sigma}_t^{-(i)} = \mathbf{F}_t\{\boldsymbol{\theta}_t^{(i)}\}\boldsymbol{\Sigma}_{t-1}^{-(i)}(\mathbf{F}_t\{\boldsymbol{\theta}_t^{(i)}\})^T + \mathbf{Q}_t\{\boldsymbol{\theta}_t^{(i)}\}$$

Kalman measurement update

The measurement model defines how to compute $p(\mathbf{y}_t|\boldsymbol{\theta}_{0:t-1}^{(i)}, \mathbf{y}_{1:t-K}1)$

Indeed, The measurement model defines the observation matrix $\mathbf{H}_t\{\boldsymbol{\theta}_t^{(i)}\}$, the observation noise $\mathbf{v}_t\{\boldsymbol{\theta}_t^{(i)}\}$ and its covariance matrix $\mathbf{R}_t\{\boldsymbol{\theta}_t^{(i)}\}$ for the Kalman filter.

$$(\mathbf{a}_{\mathbf{A}t}, \mathbf{p}_{\mathbf{V}t})^T = \mathbf{H}_t\{\boldsymbol{\theta}_t^{(i)}\}(\mathbf{v}_t, \mathbf{p}_t)^T + \mathbf{v}_t\{\boldsymbol{\theta}_t^{(i)}\}$$

$$\mathbf{H}_t\{\boldsymbol{\theta}_t^{(i)}\}_{6\times 3} = \begin{pmatrix} \mathbf{0}_{3\times 3} \\ \\ & \mathbf{I}_{3\times 3} \end{pmatrix}$$

$$\mathbf{R}_t\{\boldsymbol{\theta}_t^{(i)}\}_{3\times 3} = \begin{pmatrix} \mathbf{R}_{\mathbf{pV}_t} \end{pmatrix}$$

Kalman update

$$\mathbf{S} = \mathbf{H}_t\{\boldsymbol{\theta}_t^{(i)}\}\boldsymbol{\Sigma}_t^{-(i)}(\mathbf{H}_t\{\boldsymbol{\theta}_t^{(i)}\})^T + \mathbf{R}_t\{\boldsymbol{\theta}_t^{(i)}\}$$
$$\hat{\mathbf{z}} = \mathbf{H}_t\{\boldsymbol{\theta}_t^{(i)}\}\hat{\mathbf{x}}_t^{-(i)}$$
$$\mathbf{K} = \boldsymbol{\Sigma}_t^{-(i)}\mathbf{H}_t\{\boldsymbol{\theta}_t^{(i)}\}^T\mathbf{S}^{-1}$$
$$\boldsymbol{\Sigma}_t^{(i)} = \boldsymbol{\Sigma}_t^{-(i)} + \mathbf{KSK}^T$$
$$\hat{\mathbf{x}}_t^{(i)} = \hat{\mathbf{x}}_t^{-(i)} + \mathbf{K}((\mathbf{a}_{\mathbf{A}t}, \mathbf{p}_{\mathbf{V}t})^T - \hat{\mathbf{z}})$$
$$p(\mathbf{y}_t|\boldsymbol{\theta}_{0:t-1}^{(i)}, \mathbf{y}_{1:t-1}) = \mathcal{N}((\mathbf{a}_{\mathbf{A}t}, \mathbf{p}_{\mathbf{V}t})^T; \hat{\mathbf{z}}_t, \mathbf{S})$$

Asynchronous measurements

Our measurements might have different sampling rate so instead of doing full kalman update, we only apply a partial kalman update corresponding to the current type of measurement $\mathbf{z}_t$.

For indoor, there is only one kind of sensor for the Kalman update: **pv**

Attitude reweighting

In the measurement model, the attitude defines another reweighting for importance sampling.

$$p(\mathbf{y}_t|\boldsymbol{\theta}_{0:t-1}^{(i)}, \mathbf{y}_{1:t-1}) = \mathcal{N}(Q2R({\mathbf{q}^{(i)}}^{-1}\mathbf{q}_{\mathbf{V}t});\ 0,\ \mathbf{R}_{\mathbf{q_V}})$$

Algorithm summary

1. Initiate N particles with $\mathbf{x}_0$, $\mathbf{q}_0 \sim p(\mathbf{q}_0)$, $\mathbf{\Sigma}_0$ and $w = 1/N$
2. While new sensor measurements $(\mathbf{z}_t, \mathbf{u}_t)$

- foreach N particles (i):
  1. Depending on the type of observation:
     - **IMU**:
       1. store $\boldsymbol{\omega}_{\mathbf{G}t}$ and $\mathbf{a}_{\mathbf{A}t}$ as last control inputs
       2. sample new latent variable $\boldsymbol{\theta_t}$ from $\boldsymbol{\omega}_{\mathbf{G}t}$ (which correspond to the last control inputs)
       3. apply kalman prediction from $\mathbf{a}_{\mathbf{A}t}$ (which correspond to the last control inputs)
     - **Vicon**:
       1. sample new latent variable $\boldsymbol{\theta_t}$ from $\boldsymbol{\omega}_{\mathbf{G}t}$ (which correspond to the last control inputs)
       2. apply kalman prediction from $\mathbf{a}_{\mathbf{A}t}$ (which correspond to the last control inputs)
       3. Partial kalman update with:

$$\mathbf{H}_t\{\boldsymbol{\theta}_t^{(i)}\}_{3\times 6} = (\mathbf{0}_{3\times 3} \quad \mathbf{I}_{3\times 3})$$

$$\mathbf{R}_t\{\boldsymbol{\theta}_t^{(i)}\}_{3\times 3} = \mathbf{R}_{\mathbf{p}\mathbf{V}t}$$

$$\mathbf{x}_t^{(i)} = \mathbf{H}_t\{\boldsymbol{\theta}_t^{(i)}\}\mathbf{x}_{t-1}^{(i)} + \mathbf{K}(\mathbf{p}_{\mathbf{V}t} - \hat{\mathbf{z}})$$

$$p(\mathbf{y}_t|\boldsymbol{\theta}_{0:t-1}^{(i)}, \mathbf{y}_{1:t-1}) = \mathcal{N}(\mathbf{q}_{\mathbf{V}t}; \mathbf{q}_t^{(i)},\ \mathbf{R}_{\mathbf{q}_{\mathbf{V}t}})\mathcal{N}(\mathbf{p}_{\mathbf{V}t}; \hat{\mathbf{z}}_t, \mathbf{S})$$

     - **Other sensors (Outdoor)**: As for **Vicon** but use the corresponding partial kalman update
  2. Update $w_t^{(i)}$: $w_t^{(i)} = p(\mathbf{y}_t|\boldsymbol{\theta}_{0:t-1}^{(i)}, \mathbf{y}_{1:t-1})w_{t-1}^{(i)}$

- Normalize all $w^{(i)}$ by scalaing by $1/(\sum w^{(i)})$ such that $\sum w^{(i)} = 1$
- Compute $\mathbf{p}_t$ and $\mathbf{q}_t$ as the expectation from the distribution approximated by the N particles.
- Resample if the number of effective particle is too low

Extension to outdoors

As highlighted in the Algorithm summary, the RPBF if easily extensible to other sensors. Indeed, measurements are either:

- giving information about position or velocity and their update is similar to the vicon position update as a kalman partial update

- giving information about the orientation and their update is similar to the vicon attitude update as a pure importance sampling reweighting.

As a proof-of-conceptalternative Rao-blackwellized particle filter specialized for outdoor has been developped that integrates the following sensors:

- IMU with accelerometer, gyroscope **and magnetometer**
- Altimeter
- Dual GPS (2 GPS)
- Optical Flow

The optical flow measurements are assumed to be of the form $(\Delta\mathbf{p}, \Delta\mathbf{q})$ for a Δt corresponding to its sampling rate. It is inputed to the particle filter as a likelihood:

$$p(\mathbf{y}_t|\boldsymbol{\theta}^{(i)}_{0:t-1}, \mathbf{y}_{1:t-1}) = \mathcal{N}(\mathbf{p}_{t1} + \Delta p; \mathbf{p}_{t2}; \mathbf{R}_{\mathbf{dp_{O}}_t})\mathcal{N}(\Delta\mathbf{q}; \mathbf{q}_{t1}^{-1}\mathbf{q}_{t2}; \mathbf{R}_{\mathbf{dq_{O}}_t})$$

where $t2 = t1 + \Delta t$, $\mathbf{p}_{t2}$ is the latest kalman prediction and $\mathbf{q}_{t2}$ is the latest latent variable through sampling of the attitude updates.

Results

We present a comparison of the 4 filters in 6 settings. All of them share a sampling frequency of **200Hz** for the IMU and **4Hz** for the Vicon. The RBPF is set to **1000** particles

In all scenarios, the covariance matrices of the sensors measurement are diagonal:

- $\mathbf{R}_{\mathbf{a_A}} = \sigma^2_{\mathbf{a_A}}\mathbf{I}_{3\times3}$
- $\mathbf{R}_{\boldsymbol{\omega}_\mathbf{G}} = \sigma^2_{\boldsymbol{\omega}_\mathbf{G}}\mathbf{I}_{3\times3}$
- $\mathbf{R}_{\mathbf{p_V}} = \sigma^2_{\mathbf{p_V}}\mathbf{I}_{3\times3}$
- $\mathbf{R}_{\mathbf{q_V}} = \sigma^2_{\mathbf{q_V}}\mathbf{I}_{3\times3}$

with the following settings:

- **Vicon**:
  - High-precision $\sigma^2_{\mathbf{p_V}} = \sigma^2_{\mathbf{q_V}} = 0.01$
  - Low-precision $\sigma^2_{\mathbf{p_V}} = \sigma^2_{\mathbf{q_V}} = 0.1$

- **Accelerometer**:
  - High-precision: $\sigma^2_{\mathbf{a_A}} = 0.1$
  - Low-precision: $\sigma^2_{\mathbf{a_A}} = 1.0$

- **Gyroscope**:
  - High-precision: $\sigma^2_{\omega_G} = 0.1$
  - Low-precision: $\sigma^2_{\omega_G} = 1.0$

Table 1.1: position RMSE over 5 random trajectories of 20 seconds

Vicon precision	Accel. preci.	Gyros. preci.	Augmented Complementary Filter	Extended Kalman Filter	Unscented Kalman Filter	Rao - Blackwellized Particle Filter
High	High	High	6.88e-02	3.26e-02	3.45e-02	**1.45e-02**
High	High	Low	6.10e-02	1.13e-01	9.20e-02	**2.17e-02**
High	Low	Low	4.05e-02	5.24e-02	3.29e-02	**1.61e-02**
Low	High	High	5.05e-01	5.05e-01	2.90e-01	**1.27e-01**
Low	High	Low	6.16e-01	1.09e+00	9.30e-01	**1.22e-01**
Low	Low	Low	3.57e-01	2.66e-01	3.27e-01	**1.19e-01**

Table 1.2: attitude RMSE over 5 random trajectories of 20 seconds

Vicon precision	Accel. preci.	Gyros. preci.	Augmented Complementary Filter	Extended Kalman Filter	Unscented Kalman Filter	Rao - Blackwellized Particle Filter
High	High	High	7.36e-03	5.86e-03	5.17e-03	**1.01e-04**
High	High	Low	6.37e-03	1.37e-02	9.17e-03	**6.50e-04**
High	Low	Low	6.25e-03	1.69e-02	1.02e-02	**8.34e-04**
Low	High	High	5.30e-01	3.28e-01	3.26e-01	**5.82e-03**
Low	High	Low	5.18e-01	2.99e-01	2.95e-01	**5.78e-03**
Low	Low	Low	5.90e-01	3.28e-01	3.24e-01	**3.97e-03**

Figure 1.13 is a bar plot of the first line of each table.

Figure 1.14 is the plot of the tracking of the position (x, y, z) and attitute (r, i, j, k) in the **low** vicon precision, **low** accelerometer precision and **low** gyroscope precision setting for one of random trajectory.

Conclusion

TODO

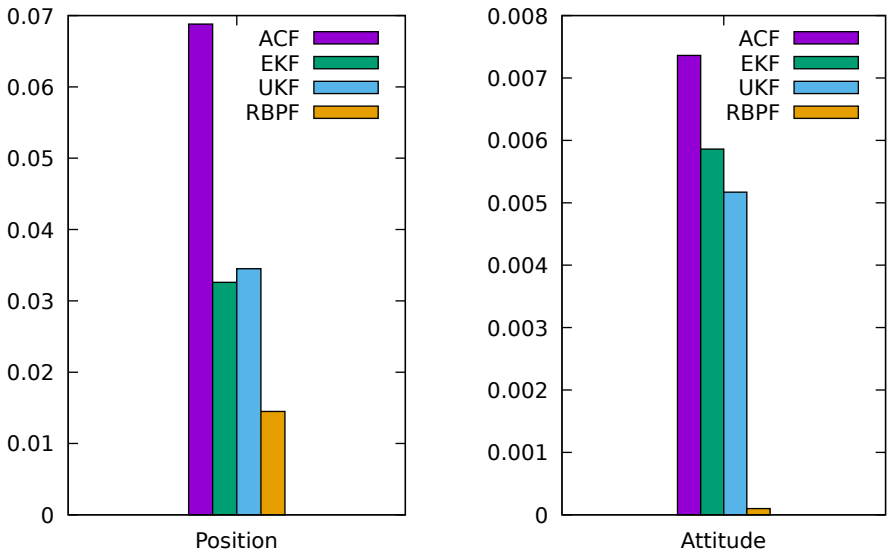


Figure 1.13: Bar plot in the High/High/High setting

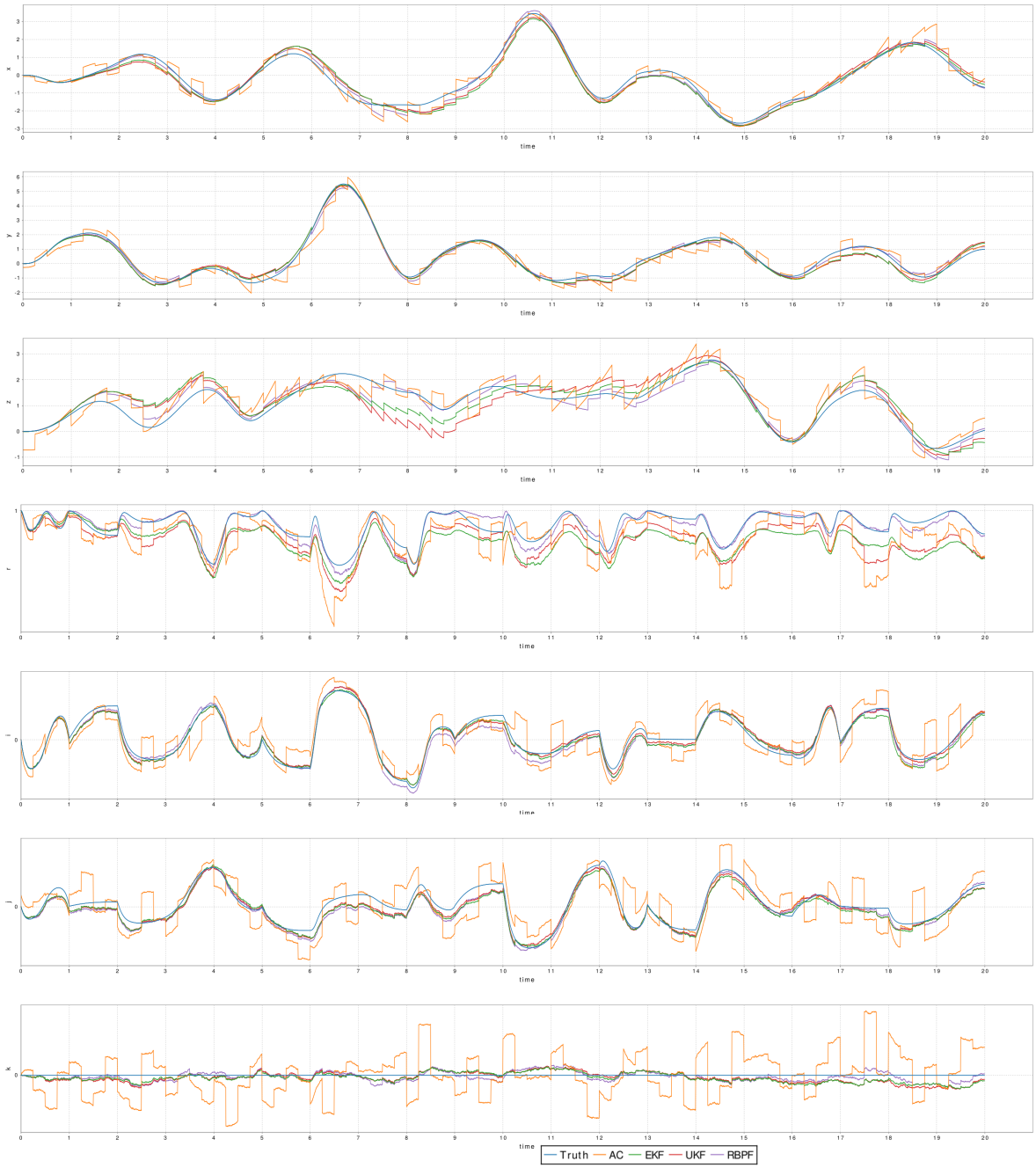


Figure 1.14: Plot of the tracking of the different filters

2 | A simulation tool for data flows with spatial integration: scala-flow

Purpose

Data flows are intuitive visual representation and abstraction of computations. As all forms of representations and abstractions, they help manage complexity, and let engineers reason on a higher level. They are common in the context of embedded systems, and most forms of data processing, in particular those related to the so called *big data*.

Spark and Simulink are popular libraries for data processing and embedded systems respectively. Spark grew popular as an alternative to Hadoop. The advantages of Spark over Hadoop was, among others, in-memory communication between nodes (as opposite of through file) and a functionnally inspired scala api that brought better abstractions and greatly reduced the number of line of code.

Simulink by MathWorks on the other hand, is a graphical programming environment for modeling, simulating and analyzing dynamic systems including potentially embedded systems. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries.

scala-flow is inspired by both of these tools. It is general purpose in the sense that it can be used to represent any dynamic systems. Nevertheless, its primary intended usage is to develop, prototype, and debug embedded systems that use hardware reprogrammed by spatial. scala-flow has a functional/composable api, displays the constructed graph, provide block constructions. It provides some strong type safety: the type of the input and output of each node is checked at compilation to ensure the soundness of the

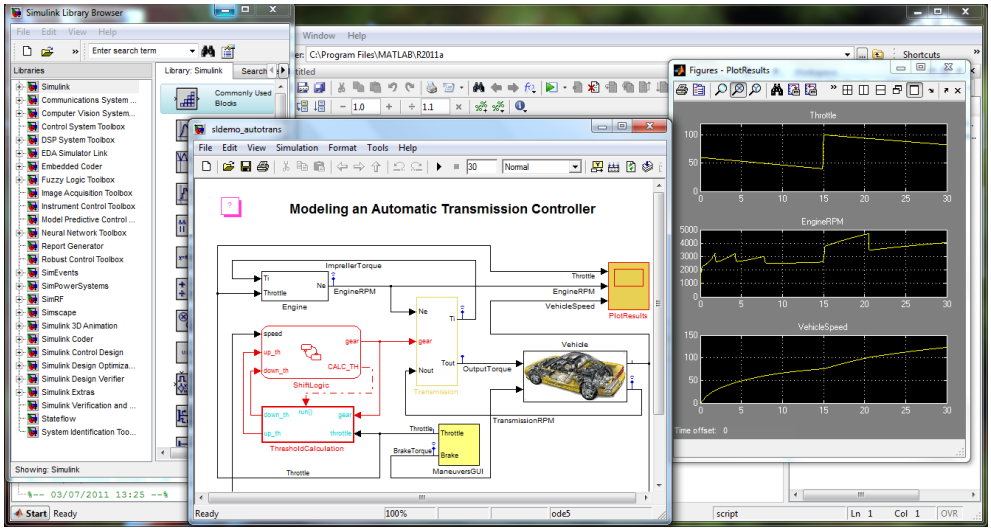


Figure 2.1: An example of the simulink interface

resulting graph.

Source, Sink and Transformations

Data is under the form of "packets" containing a value of arbitrary type, an emission timestamp and the delays the packet has encountered through the different node processing.

```
case class Timestamped[A](t: Time, v: A, dt: Time)
```

(They are called Timestamped because they represent the value with timestamp informations.)

Data get emitted from sources (nodes with no input), processed and tranformed by other nodes until they reach sinks (nodes with no output). The nodes are connected between each other in many forms.

Nodes all mix-in the common trait `Node`. Every nodes also mix-in the trait `Source[A]` whose type parameter `A` indicates the type parameters of the packets emitted by this node. Indeed, nodes can only have one output. Every nodes also minx-in the trait `SourceX[A, B, ...]` where X is the arity of the number of input for that nodes and replace by the actual arity (1, 2, 3, ...). This is similar to `FunctionX` which is the type of functions in scala.

- `Source0` indicates that the node take exactly 0 input.
- `Source1[A]` indicates that the node has 1 input whose packets are of

type A.
- `Source2[A,B]` indicates that the nodes has 2 inputs whose packets are respectively of type A and B
- etc ...

Since all nodes mix-in a `SourceX`, the compiler can check that the inputs of each node are of the right type.

An intermediary node that applies a transformation mixs-in the trait `OpX[A, B, ... , R]` where `A, B` is the type of the input, and R is the type of the output. Indeed,

```
OpX[A, B, ... , R] extends SourceX[A, B, ... ] with Source[R].
```

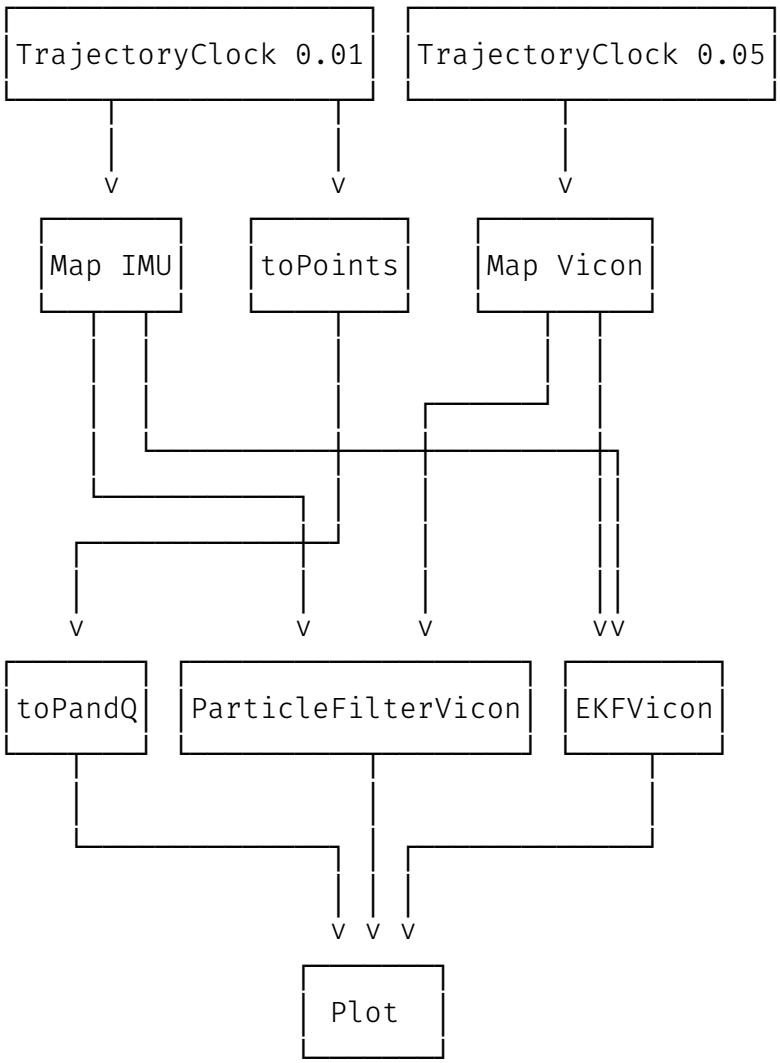
For instance `zip(sourceA, sourceB)` is an `Op[A, B, (A, B)]`

Demo

Below is the scala-flow code corresponding to a data-flow that enable to compare a particle filter, an extended kalman filter, and the true state of the underlying model. At each tick of the different clocks, a packet containing the time as value is sent to nodes representing sensors. Those sensors have access to the model that is not represented here (the trajectory of the drone) and transform the time into noisy sensor measurements and forward them to the two filters. The packets once processed by the filters are plotted by the Plot sink. The plot also take as input the true state as given by the "toPoints" transformation.

```
//****** Model ******
val dtIMU   = 0.01
val dtVicon = (dtIMU * 5)

val covAcc    = 1.0
val covGyro   = 1.0
val covViconP = 0.1
val covViconQ = 0.1

val numberParticles = 1200

val clockIMU   = new TrajectoryClock(dtIMU)
val clockVicon = new TrajectoryClock(dtVicon)

val imu   = clockIMU.map(IMU(eye(3) * covAcc, eye(3) * covGyro, dtIMU))
val vicon = clockVicon.map(Vicon(eye(3) * covViconP, eye(3) * covViconQ))

lazy val pfilter =
    ParticleFilterVicon(
      imu,
      vicon,
      numberParticles,
      covAcc,
```

```
      covGyro,
      covViconP,
      covViconQ
    )

lazy val ekfilter =
    EKFVicon(
      imu,
      vicon,
      covAcc,
      covGyro,
      covViconP,
      covViconQ
    )

val filters = List(ekfilter, pfilter)

val points = clockIMU.map(LambdaWithModel(
  (t: Time, traj: Trajectory) ⇒ traj.getPoint(t)), "toPoints")

val pqs =  points.map(x ⇒ (x.p, x.q), "toPandQ")

Plot(pqs, filters:_*)
```



The filters are block with the following signatures:

```
case class ParticleFilterVicon(rawSource1: Source[(Acceleration, Omega)],
                               rawSource2: Source[(Position, Attitude)],
                               N: Int,
                               covAcc: Real,
                               covGyro: Real,
                               covViconP: Real,
                               covViconQ: Real)
```

and similar for EKFVicon.

Blocks

Flow graphical representation

Buffer and cycles

Data

Source API

FP

Here is a simplified description of the API of each source.

When relevant, the functions have an alternative `methodNameT` function that takes themselves function whose domain is `Timestamped[A]` instead of A.

For instance, there is a

```
def foreachT(f: Timestamped[A] ⇒ Unit): Source[A]
```

that is equivalent to the `foreach` below except it can access the additional fields in `Timestamped`

```
trait Source[A] {

  /** return a new source that map every incoming packet by the function f
    * such that new packets are Timestamped[B]
    */
  def map[B](f: A ⇒ B): Source[B]

  /** return a filtered source that only broadcast
    *  the elements that satisfy the predicate b */
  def filter(b: A ⇒ Boolean): Source[A]

  /** return this source and apply the function f to each
    *  incoming packets as soon as they are received
    */
  def foreach(f: A ⇒ Unit): Source[A]

  /** return a new source that broadcast elements
    * until the first time the predicate b is not satisfied
    */
  def takeWhile(b: A ⇒ Boolean): Source[A]

  /** return a new source that accumulate As into a List[A]
    * then broadcast it when the next packet from the other
```

```
  * source clock is received
  */
def accumulate(clock: Source[Time]): Source[ListT[A]]

/** return a new source that broadcast all element inside the collection
  * returned by the application of f to all incoming packet
  */
def flatMap[C](f: A ⇒ List[C]): Source[C]

/** assumes that A is a List[Timestamped[B]].
  * returns a new source that apply the reduce function
  * over the collection contained in every incoming packet */
def reduce[B](default: B, f: (B, B) ⇒ B)
    (implicit ev: A <:< ListT[B]): Source[B]

/** return a new source that broadcast pair of the packet from this source
  * and the source provided as argument. Wait until a packet is received
  * from both source. Packets from both source are queued such
  * that independant of the order, they are never discarded
  * A2 B1 A3 B2 B3 B4 B5 A4⇒ (A1, B1), (A2, B2), (A3, B3), (A4, B4),
  * [Queue[B5]]
  */
def zip[B](s2: Source[B]): Source[Boolean]


/** return a new source that broadcast pair of the packet from this source
  * and the source provided as argument. Similar to zip except that
  * if multiple packets from the source provided as argument is received
  * before, all except the last get discarded.
  * A2 B1 A3 B2 B3 B4 B5 A4⇒ (A1, B1), (A2, B2), (A3, B3), (A4, B4),
  * [Queue[B5]]
  */
def zipLastRight[B](s2: Source[B])


/** return a new source that broadcast pair of the packet from this source
  * and the source provided as argument. Similar to zip except that all
  * packet except the last get discarded when both source are not in sync.
  * A1 A2 B1 A3 B2 B3 A4⇒ (A1, B1), (A3, B2), (B3, A4)
  */
def zipLast[B](s2: Source[B])

/** return a new source that combine this source and the provided source .
  * packets from this source are Left
  * packets from the other source are Right
  */
def merge[B](s2: Source[B]): Source[Either[A, B]]

/** return a new source that fuse this source and the provided source
  * as long they have the same type.
  * any outgoing packet is indistinguishable of origin
  */
def fusion(sources: Source[A]*): Source[A]

/** "label" every packet by the group returned by f */
def groupBy[B](f: A ⇒ B): Source[(B, A)]

/** print every incoming packet */
def debug(): Source[A]

/** return a new source that buffer 1 element and
  * broadcast the buffered element with the time of the incoming A
  */
def bufferWithTime(init: A): Source[A]
```

```
  /** return a new source that do NOT broadcast any element */
  def muted: Source[A]

  /** return a new source that broadcast one incoming packet every
    * n incoming packet.
    * The first broadcasted packet is the nth received one
    */
  def divider(n: Int): Source[A]

  /** return a pair of source from a source of pair */
  def unzip2[B, C](implicit ev: A <:< (B, C)): (Source[B], Source[C])

  /** return a new source whose every outgoing packet have an added dt
    * in their delay component
    */
  def latency(dt: Time): Source[A]

  /** return a new source whose broadcasted packets contain the time of
    * emission
    */
  def toTime: Source[Time]

  /** return a new source that do NOT broadcast the first n packets */
  def drop(n: Int): Source[A]
}


implicit class TimeSource(source: Source[Time]) {

  /** stop the broadcasting after the timeframe tf has elapsed */
  def stop(tf: Timeframe): Source[Time]

  /** add a random delay following a gaussian with corresponding
    * mean and variance */
  def latencyVariance(mean: Real, variance: Real): Source[Time]

  /** add a delay of dt */
  def latency(dt: Time): Source[Time]

  /** return a new source of the difference of time between
    * the two last emitted packets */
  def deltaTime(init: Time = 0.0): Source[Time]

}
```

The real API includes also a `name` and `silent` parameter. Both are only relevant for the graphical representation. The name of the block will be overriden by name if present and the node will be skipped in the graphical representation if silent is present.

Block

Scheduling

Batch

Replay

Spatial integration

Conclusion

TODO

3 | An interpreter for spatial

Spatial

alea jacta est

4 | Spatial implementation of an asynchronous Rao-Blackwellized Particle Filter

Spatial

alea jacta est

Conclusion

alea jacta est

References

[1] M. W. Mueller, M. Hehn, and R. D'Andrea, "A computationally efficient motion primitive for quadrocopter trajectory generation," *IEEE Transactions on Robotics*, vol. 31, no. 6, pp. 1294–1310, 2015.

[2] F. L. Markley, Y. Cheng, J. L. Crassidis, and Y. Oshman, "Averaging quaternions," *Journal of Guidance, Control, and Dynamics*, vol. 30, no. 4, pp. 1193–1197, 2007.

[3] K. Edgar, "A Quaternion-based Unscented Kalman Filter for Orientation Tracking.".

[4] A. Doucet and A. M. Johansen, "A tutorial on particle filtering and smoothing: Fifteen years later," *Handbook of nonlinear filtering*, vol. 12, nos. 656-704, p. 3, 2009.

[5] P. Vernaza and D. D. Lee, "Rao-Blackwellized particle filtering for 6-DOF estimation of attitude and position via GPS and inertial sensors," in *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, 2006, pp. 1571–1578.