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 17, 2017

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

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

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Stanford University

Abstract

Drones POSE (position and orientation) estimation on drones rely on sensor fusion of its different sensors. The complexity of this task is to provide a good estimation in real-time. We have developped a novel application of an asynchronous Rao-Blackwellized Particle Filter and its implementation on hardware with the Spatial language. We have also build a new development tool: scala-flow, a data-flow simulation tool inspired by Simulink with a Spatial integration. Finally, we have build an interpreter for the Spatial language which made possible the integration of Spatial in scala-flow.

Contents

Introduction

Moore's law end

The Moore's law[1] has prevailed in the computation world 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 100 times bigger than an atom. Unfortunately, the quantum rules of physics which govern the infinitesimally, start to manifest themselves. In particular, quantum tunneling move electrons from classically insurmountable 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 synthesized as 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 in the sense that it is general purpose but relevant only for embarrassingly parallel tasks[2] and very efficient when used well. They have benefited from high-investment and many generations 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 industries like high-frequency trading. But now Hardware is the next natural step to increase performance,

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

[2]An embarrassingly parallel task is one where little or no effort is needed to separate the problem into a number of parallel tasks. This is often the case where there is little or no dependency or need for communication between those parallel tasks, or for results between them.

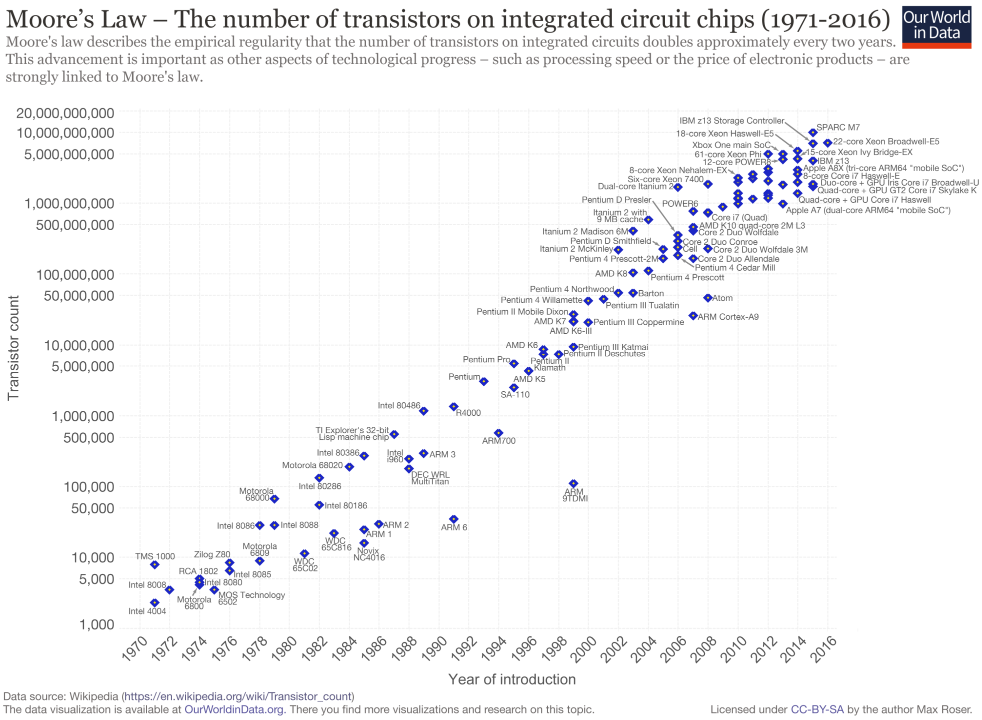


Figure 1: The number of transistors throughout the years. We can observe a recent start of a decline

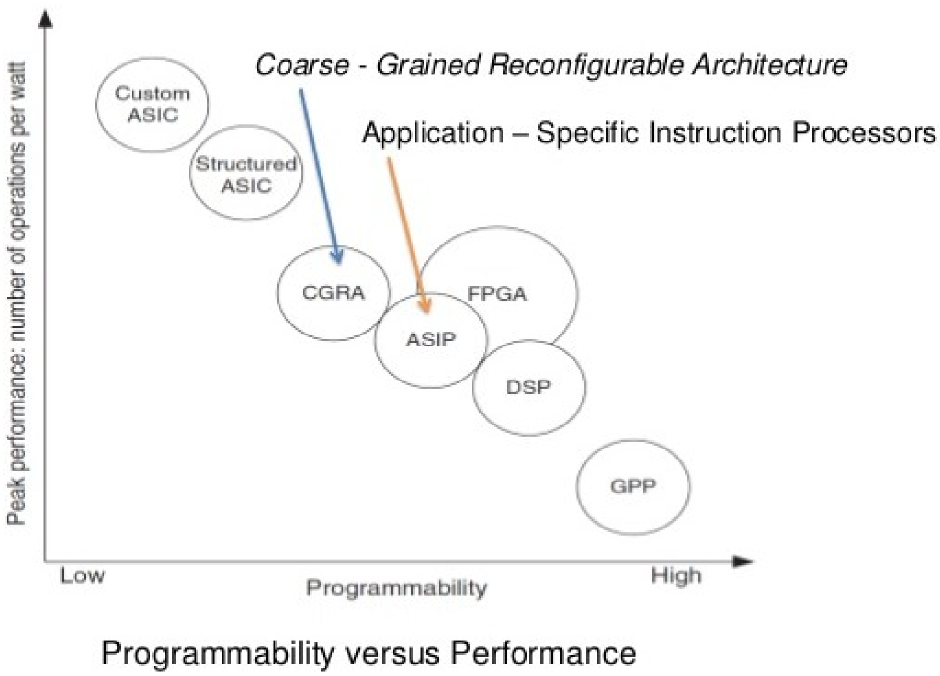


Figure 2: Hardware vs Software

at least until a computing revolution happen, like quantum computing, but this is not realist for the near future. However, hardware do not enjoy the same quality of tooling, language and integrated development environments (IDE) as software. This is one the motivation behind Spatial: bridging the gap between Software and Hardware by abstracting control flows through language constructions.

Hardware as companion accelerators

In most case, hardware would be inappropriate: running an OS as hardware would be impracticable. Nevertheless, as a companion to a central-processing unit (CPU also called “the host”), it is possible 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 and architecture because it reveals its intrinsic properties as a computing element.. If the metric was solely performance, then it would suffice to stack the same hardware and eventually a reach the scale of a super-computer. 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 developed 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 developed in parallel to Spatial. It leverages some recurrent patterns: the parallel 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 in a wide range of use-cases, from the most demanding server applications to 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, developing drone applications with spatial demonstrates the advantages of the platform. As a matter of fact, the filter implementation was only made possible because it is run on an accelerating hardware. It would be unfeasible to run it on more conventional micro-transistors. This is why the family in which belong the filter developed 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. Indeed, 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, developing 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. Furthermore, particle filters are embarrassingly parallel algorithms. Hence, they can leverage the potential of a dedicated hardware design. The spatial implementation will be presented in Part IV.

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, Particle 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 which one and how.

All the following filters are developed and tested in scala-flow. scala-flow will be expanded in part II of this thesis. For now, we will focus on the model and the results, and leave the implementation details for later.

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 developed 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 from 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 the combination of sensory data or data derived from disparate sources such that the resulting information has less uncertainty than would be possible if these sources were to be used individually. In the context of drones, it is very useful because it enables to combine many unprecise sensor measurement to form a more precisre one 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 low sampling rate and less precise sensors with high sampling rate. Both cases are gonna be relevant here.

Figure 1.2: The Flamewheel f450

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 x at time t
- $x_{t1:t2}$ is the product of the random variable 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 from [1]. Data generated this way are called synthetic data. The algorithm input are the motion primitives defined by the quadcopter's initial state, the desired motion duration, and any combination of components of the quadcopter's position, velocity and acceleration at the motion's end. The algorithm is essentially a closed form solution for the given primitives. The closed form solution minimizes a cost function related to the 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.

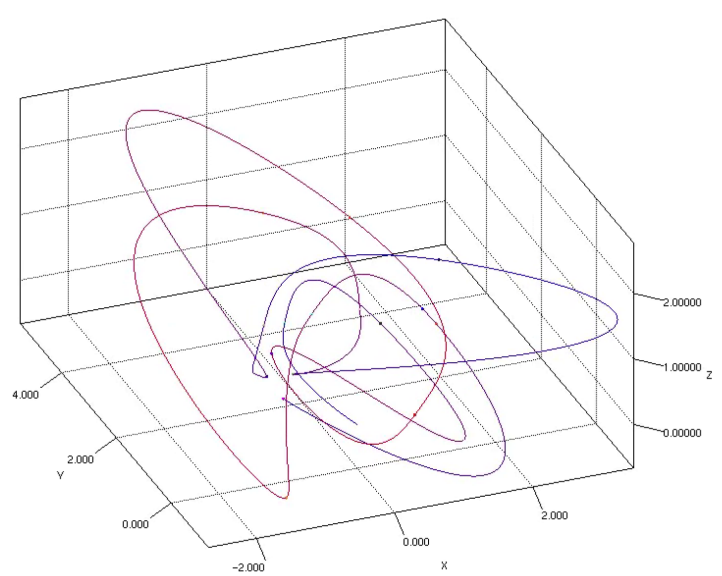


Figure 1.3: Visualization of an example of a synthetic generated flight trajectory

Quaternion

Quaternions are extensions of complex numbers 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.[1] In all filters presented, quaternions 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

[1]Gimbal lock is the loss of one degree of freedom in a three-dimensional, three-gimbal mechanism that occurs when the axes of two of the three gimbals are driven into a parallel configuration, “locking” the system into rotation in a degenerate two-dimensional space.

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. The product of two quaternions (also called Hamilton product) is computable by regrouping the same type of imaginary and real components together and accordingly to the identity:

$$i^2 = j^2 = k^2 = ijk = -1$$

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 conjugate of a quaternion is:

$$q^* = -\frac{1}{2}(q + iqi + jqj + kqk)$$

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

$$A(\mathbf{q}) = (q.r^2 - \|\boldsymbol{\varrho}\|^2)I_{3\times3} + 2\boldsymbol{\varrho}\boldsymbol{\varrho}^T - 2q.r[\boldsymbol{\varrho}\times]$$

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

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 summarize the forces from the rotor as a thrust oriented in the direction normal to the plane formed by the 4 rotors, 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 referential 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 subtract the drone's acceleration from the thrust to the total acceleration. This would require to know exactly the force exerted 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 $\mathbf{R_{a_A}}_{3\times3}$ and

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

.

- **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 position at 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 $\mathbf{R_{p_V}}_{3\times3}$ and

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

.

- **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 retrieves 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 $\mathbf{R_{q_V}}_{3\times3}$ and

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

.

- **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 $\mathbf{R_{dq_O}}_{3\times3}$ and

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

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

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

.

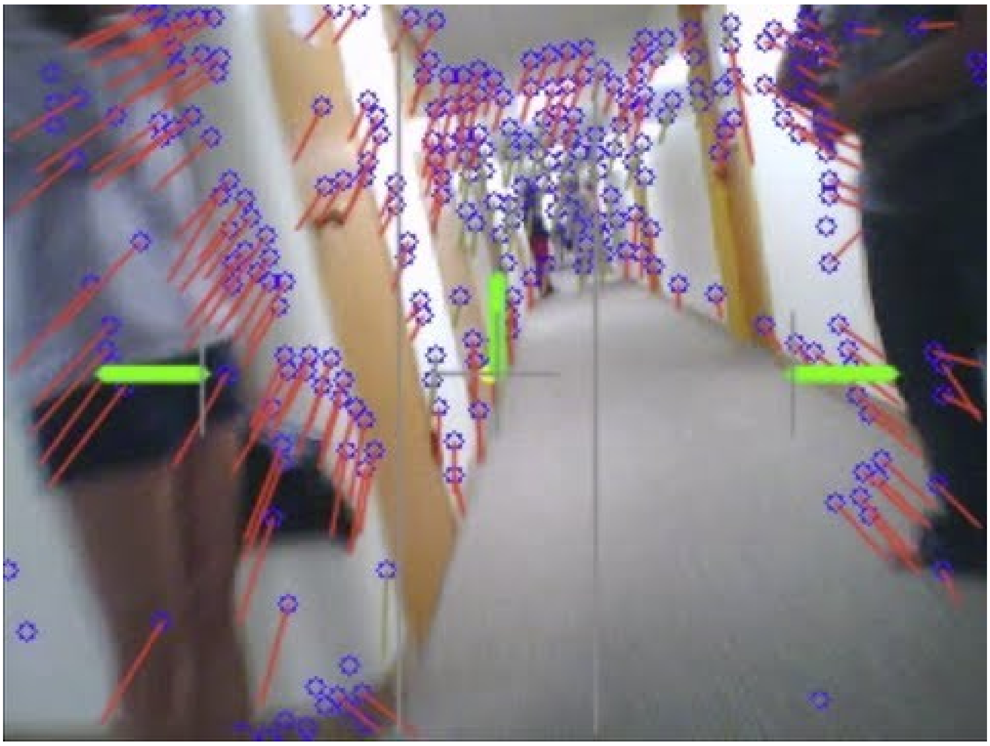


Figure 1.4: Optical flow from a moving drone

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.

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

$R_{z_A}{}_{3\times3}$ and

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

Some sensors are more relevant indoor and some others outdoor:

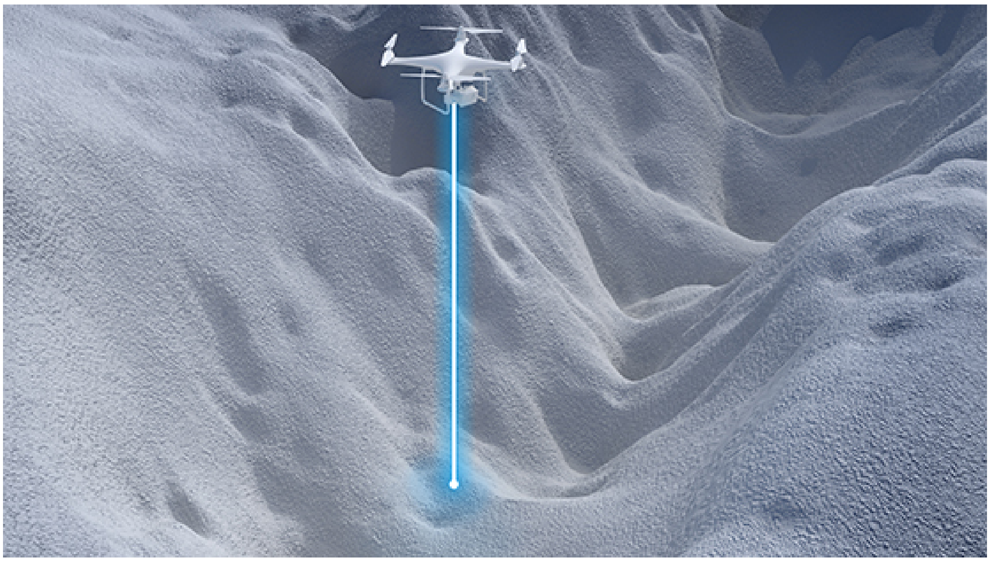


Figure 1.5: Rendering of the LIDAR laser of an altimeter

- **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.

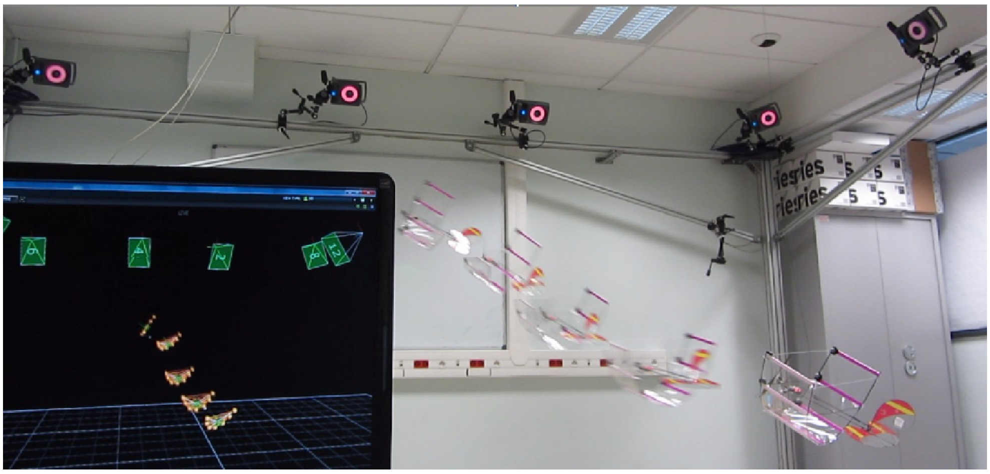


Figure 1.6: A Vicon setup

- **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.

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 strictly 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 imprecision, 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**: $\mathbf{a_A}_t$ from the acceloremeter
- **Angular velocity**: $\boldsymbol{\omega}_{\mathbf{G}t}$ from the gyroscope.

Model dynamic

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

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 conditioned 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 $\mathbf{z}$.

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

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 commonly used 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. Indeed, the gyroscope reads 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"[2] and is prone to accumula-

[2]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 misspelling of "ded" as in "deduced". Others argue that it should be read by its old meaning: *absolute*.

tion 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 significant acceleration and vibration. Hence, the assumption that we retrieve the gravity field directly is wrong. Nevertheless, We could solve this by subtracting 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.

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.

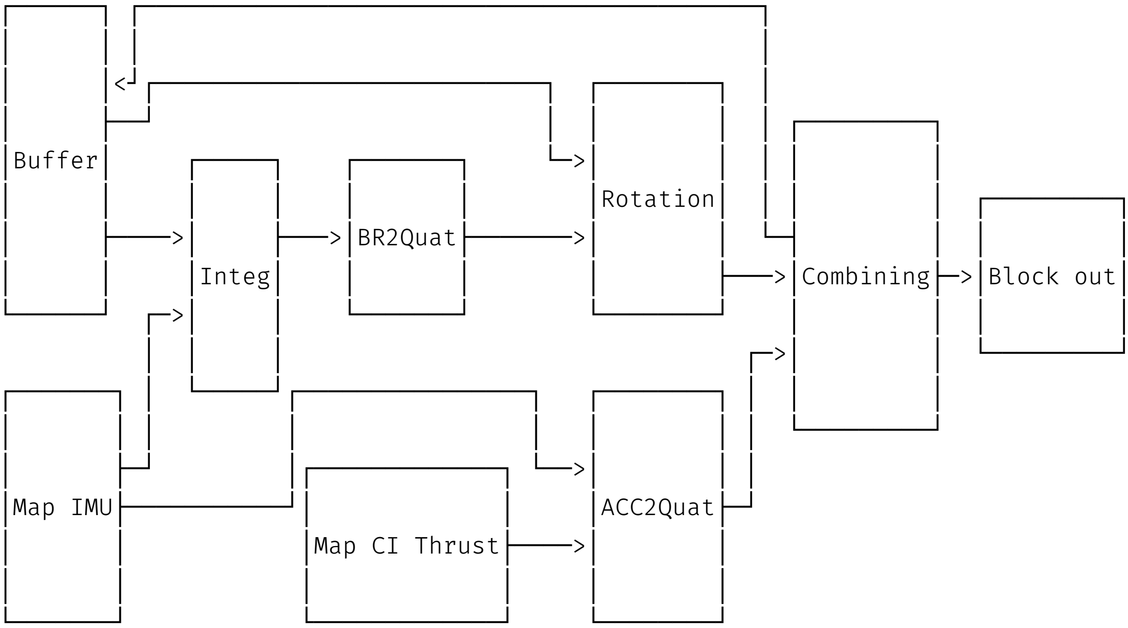


Figure 1.7: Complementary Filter graph structure

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.

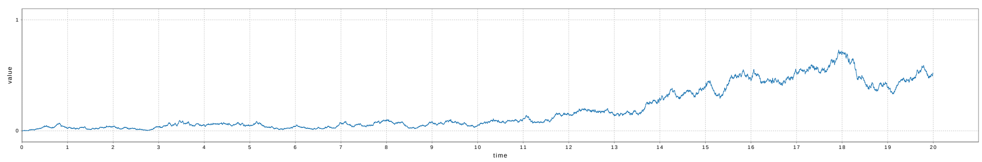


Figure 1.8: CF with alpha = 1.0

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

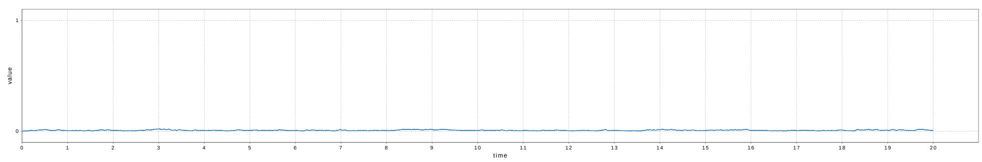


Figure 1.9: CF with alpha = 0.98

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 computationally 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. As with all following filters, we deal with asynchronicity by updating the state to the most likely state so far for any new sensor measurement incoming. 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:

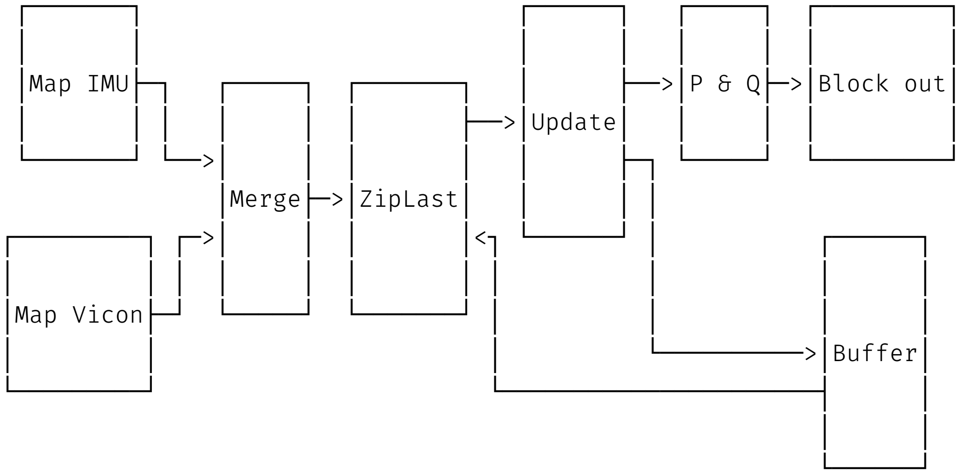


Figure 1.10: A graph of the filters structure in scala-flow

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, **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 follows:

1. Multiple prediction steps without any update step may happen without making the algorithm inconsistent.
2. An update is **always** immediately 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 synchronized with the control inputs, use the most likely control input for the prediction step. 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.

- 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 intrinsically 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 to calculate 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.

$$\mathbf{F}_{t10\times 10} = \left.\frac{\partial f}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_{t-1},\mathbf{u}_{t-1}}$$

$$\mathbf{H}_{t7\times 7} = \left.\frac{\partial h}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_{t}}$$

- **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:

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

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

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

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 $\mathbf{Q}$

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

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

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

$$\mathbf{F}_{t10\times 10} = \left.\frac{\partial f}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_{t-1}, \mathbf{u}_{t-1}}$$

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

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.

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

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

$\mathbf{R}'_{\mathbf{qv}}$ has to be 4×4 and has to represent the covariance of the quaternion. However, the actual covariance matrix $\mathbf{R}_{\mathbf{qv}}$ 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 $\mathbf{R}'_{\mathbf{qv}}$ 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 $\Sigma = LL^t$.

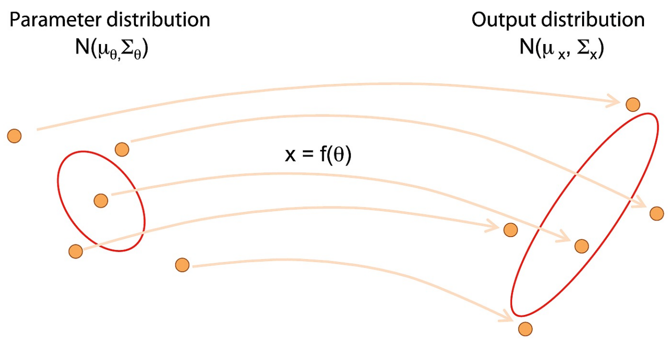


Figure 1.11: Unscented tranformation

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 computationaly expensive. This is the reason why their usage is not very popular currently for low-powered embedded systems like drones. However, they are used in Avionics for planes since the computational resources are less scarce and the precision crucial. Accelerating hardware could widen the usage of particle filters to embedded systems.

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

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 of the distribution. In our case, by choosing the right importance distribution (the dynamics of the model as

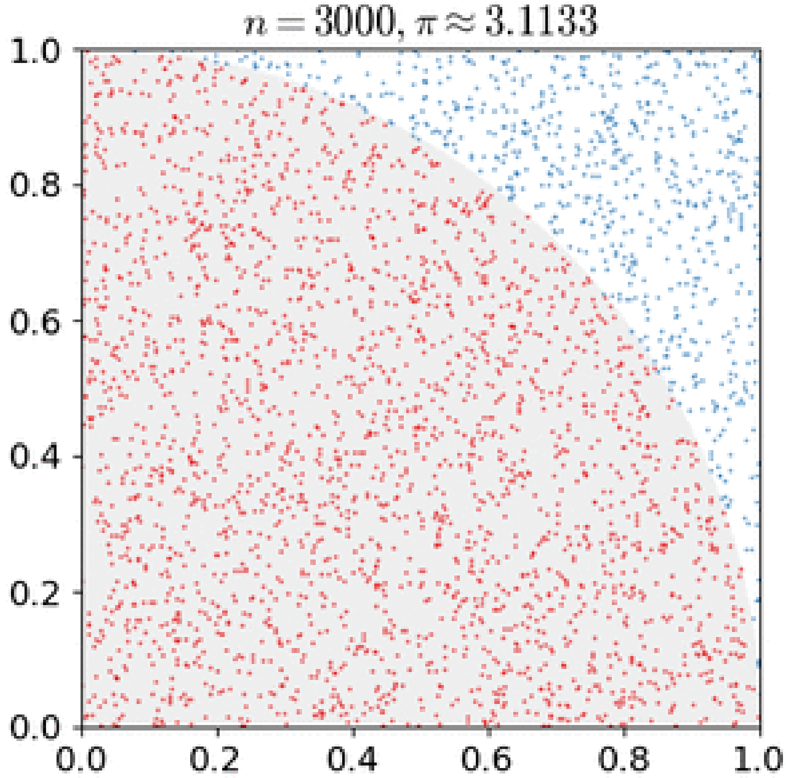


Figure 1.12: Monte Carlo estimation of pi

we will see later), we can reweight particles based on the likelihood from the measurements ($p(\mathbf{y}|\mathbf{x})$.

Importance sampling is based on the identity:

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

Thus, it can be approximated as

$$\mathbb{E}[\mathbf{g}(\mathbf{x})|\mathbf{y}_{1:T}] \approx \frac{1}{N}\sum_i^N \frac{p(\mathbf{x}^{(i)}|\mathbf{y}_{1:T})}{\pi(\mathbf{x}^{(i)}|\mathbf{y}_{1:T})}\mathbf{g}(\mathbf{x}^{(i)}) \quad \approx \sum_i^N w^{(i)}\mathbf{g}(\mathbf{x}^{(i)})$$

where N samples of $\mathbf{x}$ are drawn from the importance distribution $\pi(\mathbf{x}|\mathbf{y}_{1:T})$

And the weights are defined as:

$$w^{(i)} = \frac{1}{N}\frac{p(\mathbf{x}^{(i)}|\mathbf{y}_{1:T})}{\pi(\mathbf{x}^{(i)}|\mathbf{y}_{1:T})}$$

Computing $p(\mathbf{x}^{(i)}|\mathbf{y}_{1:T}$ is hard (if not impossible), but fortunately we can compute the unnormalized weight instead:

$$w^{(i)}* = p(\mathbf{y}_{1:T}|\mathbf{x}^{(i)})p(\mathbf{x}^{(i))}\pi(\mathbf{x}^{(i)}|\mathbf{y}_{1:T})$$

and normalizing it afterwards

$$\sum_i^N w^{(i)*} = 1 \Rightarrow w^{(i)} = \frac{w^{*(i)}}{\sum_j^N w^{*(i)}}$$

Sequential Importance Sampling

The last equation becomes more and more computationally expensive as T grows larger (the joint variable of the time series grows larger). Fortunately, Sequential Importance Sampling is an alternative recurisve algorithm that has a fixed amount of computation at each iteration:

$$
\begin{aligned}
p(\mathbf{x}_{0:k}|\mathbf{y}_{0:k}) &\propto p(\mathbf{y}_k|\mathbf{x}_{0:k},\mathbf{y}_{1:k-1})p(\mathbf{x}_k|\mathbf{y}_{1:k-1}) \\
&\propto p(\mathbf{y}_k|\mathbf{x}_k)p(\mathbf{x}_k|\mathbf{x}_{0:k-1},\mathbf{y}_{1:k-1})p(\mathbf{x}_{0:k-1}|\mathbf{y}_{1:k-1}) \\
&\propto p(\mathbf{y}_k|\mathbf{x}_k)p(\mathbf{x}_k|\mathbf{x}_{k-1})p(\mathbf{x}_{0:k-1}|\mathbf{y}_{1:k-1})
\end{aligned}
$$

The importance distribution is such that $\mathbf{x}^i_{0:k} \sim \pi(\mathbf{x}_{0:k}|\mathbf{y}_{1:k})$ with the according importance weight:

$$
w_k^{(i)} \propto \frac{p(\mathbf{y}_k|\mathbf{x}_k^{(i)})p(\mathbf{x}_k^{(i)}|\mathbf{x}_{k-1}^{(i)})p(\mathbf{x}_{0:k-1}^{(i)}|\mathbf{y}_{1:k-1})}{\pi(\mathbf{x}_{0:k}|\mathbf{y}_{1:k})}
$$

We can express the importance distribution recursively:

$$
\pi(\mathbf{x}_{0:k}|\mathbf{y}_{1:k}) = \pi(\mathbf{x}_k|\mathbf{x}_{0:k-1},\mathbf{y}_{1:k})\pi(\mathbf{x}_{0:k-1}|\mathbf{y}_{1:k-1})
$$

The recursive structure propagates to the weight itself:

$$
\begin{aligned}
w_k^{(i)} &\propto \frac{p(\mathbf{y}_k|\mathbf{x}_k^{(i)})p(\mathbf{x}_k^{(i)}|\mathbf{x}_{k-1}^{(i)})}{\pi(\mathbf{x}_k|\mathbf{x}_{0:k-1},\mathbf{y}_{1:k})}\frac{p(\mathbf{x}_{0:k-1}^{(i)}|\mathbf{y}_{1:k-1})}{\pi(\mathbf{x}_{0:k-1}|\mathbf{y}_{1:k-1})} \\
&\propto \frac{p(\mathbf{y}_k|\mathbf{x}_k^{(i)})p(\mathbf{x}_k^{(i)}|\mathbf{x}_{k-1}^{(i)})}{\pi(\mathbf{x}_k|\mathbf{x}_{0:k-1},\mathbf{y}_{1:k})}w_{k-1}^{(i)}
\end{aligned}
$$

We can further simplify the formuly by choosing the importance distribution to be the dynamics of the model:

$$
\pi(\mathbf{x}_k|\mathbf{x}_{0:k-1},\mathbf{y}_{1:k}) = p(\mathbf{x}_k^{(i)}|\mathbf{x}_{k-1}^{(i)})
$$

$$
w_k^{*(i)} = p(\mathbf{y}_k|\mathbf{x}_k^{(i)})w_{k-1}^{(i)}
$$

As previously, it is then only needed to normalize the resulting weight.

$$
\sum_i^N w^{(i)*} = 1 \Rightarrow w^{(i)} = \frac{w^{*(i)}}{\sum_j^N w^{*(i)}}
$$

Resampling

When the number of effective particles is too low (less than 1/10 of N having weight 1/10), 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 $i = 2, \dots, N$

Rao-Blackwellized Particle Filter

Introduction

Compared to a plain particle filter, 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. If a plain particle can be seen as the simple average of particle states, then the RPBF can be seen as the "average" of many Gaussians. Each particle is an optimal kalman filter conditioned on the particle's latent variable, the attitude.

Indeed, the benefit of particle filters is that they assume no particular form for the posterior distribution and transformation of the state. But as the state widens in dimensions, the number of needed particles to keep a good estimation grows exponentially. This is a consequence of ["the curse of dimensionality"}(https://en.wikipedia.org/wiki/Curse_of_dimensionality): for each dimension, we would have to consider all additional combinations of state. In our context, we have 10 dimensions ($\mathbf{v}$,$\mathbf{p}$,$\mathbf{q}$) and it would be very computationally expensive to simulate a too large number of particles.

Kalman filters on the other hand do not suffer from such exponential growth, but as explained previously, they are inadequate for non-linear transformations. RPBF is the best of both world by combining a particle filter for the non-linear components of the state (the attitude) as a latent variable, and Kalman filters for the linear components of the state (velocity and position). For ease of notation, the linear component of the state will be referred to as the state and designated by $\mathbf{x}$ even though the actual state we are concerned with should include the latent variable $\boldsymbol{\theta}$.

Related work

Related work of this approach is [5]. However, it differs by:

- adapting 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 using 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.

Latent variable

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.

As stated in the previous section, 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 and the previous section, it is provable 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 dynamics 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 $\boldsymbol{\Sigma}_{6\times 6} = \epsilon \mathbf{I}_{6\times 6}$

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 (parallel 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 conditioned under $\boldsymbol{\theta}_{t+1}^{(i)}$.

Indoor Measurement model

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

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 $\mathbf{Q}_t\{\boldsymbol{\theta}_t^{(i)}\}$

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

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

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

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

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

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.

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

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

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

Kalman update

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

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: $\mathbf{p_V}$

Attitude re-weighting

In the measurement model, the attitude defines another re-weighting for importance sampling.

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

Algorithm summary

1. Initiate N particles with $\mathbf{x}_0$, $\mathbf{q}_0 \sim p(\mathbf{q}_0)$, $\boldsymbol{\Sigma}_0$ 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 $\mathbf{a_A}_t$ 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 $\mathbf{a_A}_t$ (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:

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

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

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

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

- **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 re-weighting.

A proof-of-concept alternative Rao-blackwellized particle filter specialized for outdoor has been developed 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:

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

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. The metrics is the RMSE of the l2-norm of the position and of the Froebius norm of the attitude as described previously. All the filters 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' measurements are diagonal:

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

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: $\sigma^2_{\omega_\mathbf{G}} = 0.1$
  - Low-precision: $\sigma^2_{\omega_\mathbf{G}} = 1.0$

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 preci sion	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.

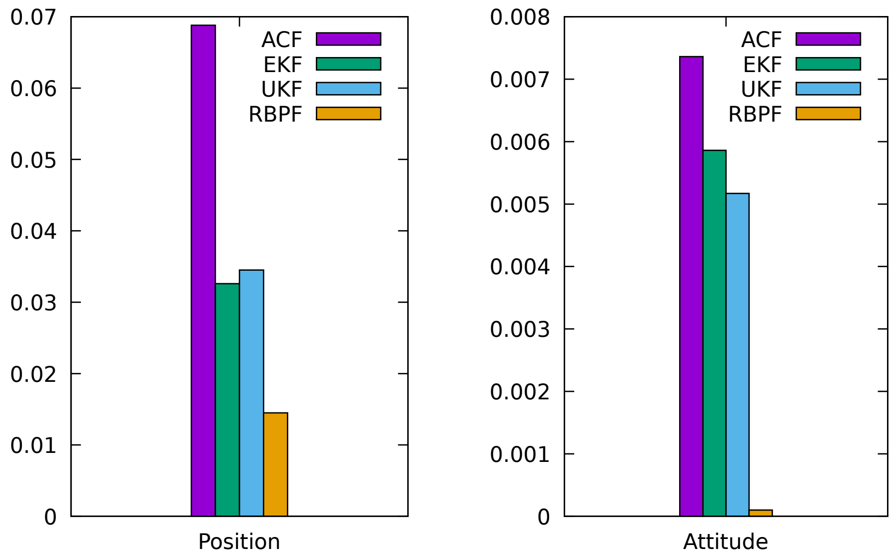


Figure 1.13: Bar plot in the High/High/High setting

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

The Rao-Blackwellized Particle Filter developed is more accurate than the alternatives, mathematically sound and computationally feasible. When implemented on hardware, this filter can be executed in real time with sensors of high and asynchronous sampling rate. It could improve POSE estimation

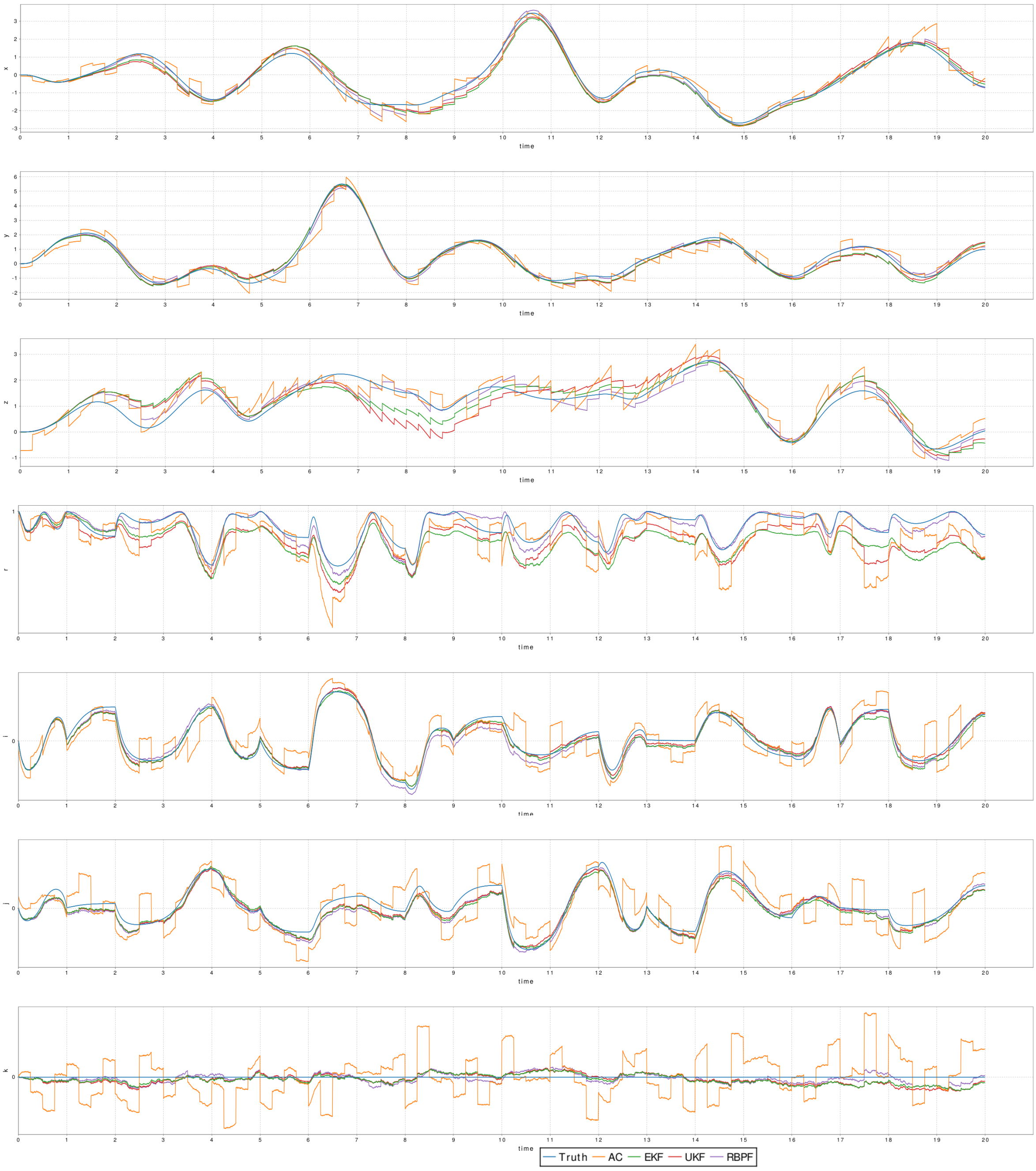


Figure 1.14: Plot of the tracking of the different filters

for all the existing drone and other robots. These improvements could unlock new abilities, potentials and increase the safeness of drone.

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

Purpose

Data flows are intuitive visual representations and abstractions of computation. As all forms of representations and abstractions, they ease complexity management, and let engineers reason on a higher level. They are common in the context of embedded systems, where sensors and electronic circuits have natural visual representations. They are also used in 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 reduced the number of line of code. Less boilerplate and duplication of code improve abstraction and ease prototyping thanks to fast iteration.

Simulink by MathWorks on the other hand, is a graphical programming environment for modeling, simulating and analyzing dynamic systems including 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 in particular those that make use of spatial programmed hardware. scala-flow has a functional/composable api, displays the constructed graph and also provides block constructions. It has strong type safety: the type of the input and output of each node is checked during compilation time to

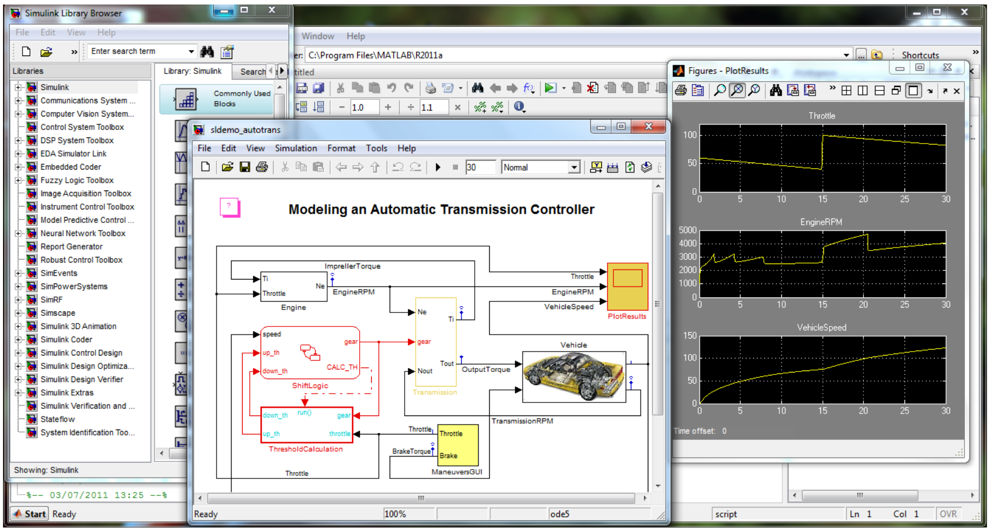


Figure 2.1: An example of the simulink interface

ensure the soundness of the resulting graph.

Source, Sink and Transformations

Data are passed from nodes to nodes under the form of typed "packets" containing a value of the given type, an emission timestamp and the delays the packet has encountered during its processing through the different nodes of the graph.

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

They are called Timestamped because they represent value and their corresponding timestamp information.

Packets get emitted from `Source0[T]` (nodes with no input), processed and transformed by other nodes until they reach sinks (nodes with no output). The nodes are connected between each other according to the structure of the data flow.

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

- `Source0` indicates that the node takes 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.

All `SourceX` must define `def listenI(x: A)` where `I` goes from 1 to X and `A` correspond to the corresponding type parameter of `SourceX`. `def listenI(x: A)` defines the action to take whenever a packet is received from the input I. Those functions are callback used to pass packets to the nodes following a listener pattern.

There is a special case, `SourceN[A, R]` which represent nodes that have an *-arity of type `A` and emit packets of type `R`. For instance, the `Plot` nodes take * number of sources and display them all on the same plot. The only constraint is that all the source nodes must emit the same kind of data of type A. Else it would not make sense to compare them. For plot specifically, `A` has also a context bound of `Data` which means that there exists a conversion from `A` to a `Seq[Float]`, to ensure that `A` is displayable in a multiplot as time series. The x-axis, the time, correspond to the timestamp of emission contained in the packet.

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.

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

For instance, `zip(sourceA, sourceB)` is an `Op[A, B, (A, B)]`. In most cases, `Op`s are a transformation of an incoming packet followed by a broadcasting (with the function `def broadcast(x: R)`) to the nodes that have for source this node.

Demo

Below is the scala-flow code corresponding to a data-flow comparing a particle filter, an extended kalman filter, and the true state of the underlying model, the trajectory of the drone. At each tick of the different clocks, a packet containing the time as value is sent to a node simulating a sensor. Those sensors have access to the underlying model and transform the time into noisy sensor measurements, then 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:_*)
```

Figure 2.2: Example of a scala-flow program

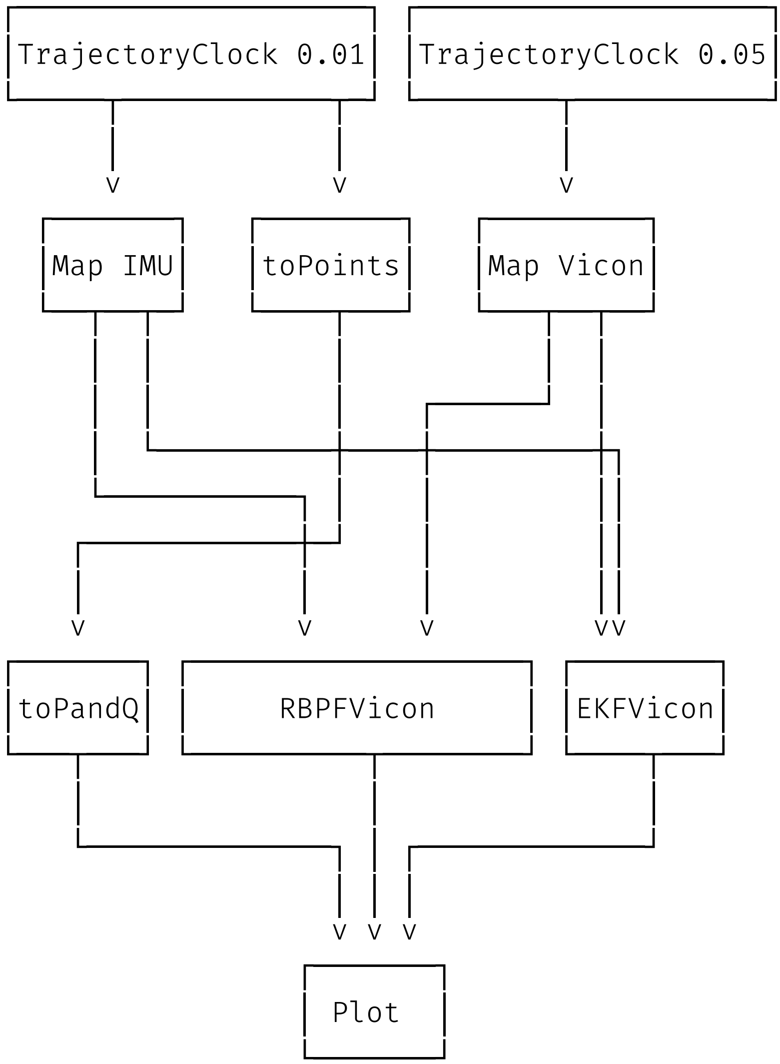


Figure 2.3: Graph representation of the data-flow

Block

A block is a node representing a group of node. That node can be summarized by its input and output such that from an external perspective, it can be considered as a simple node. Similar to the way an interface or an API hide its implementation details, a block hides its inner workings to the rest of the data-flow as long as the block receives and emits the right type of packets. This logic extends to the graphical representation. Blocks are represented as nodes in the high-level graph but expanded in an independent graph below the main one.

Similar to `OpX[A, B, ... , R]`, there exists `BlockX[A, B, ... , R]` which all extend `Block[R]` and take X sources as input. All `Block[R]` must define an `out` method of the form: `def out: Source[R]`.

For instance, the filters are blocks with the following signatures:

```
case class RBPFVicon(rawSource1: Source[(Acceleration, Omega)],
                     rawSource2: Source[(Position, Attitude)],
                     N: Int,
                     covAcc: Real,
                     covGyro: Real,
                     covViconP: Real,
                     covViconQ: Real)
    extends Block2[(Acceleration, Omega), (Position, Attitude), (Position, Attitude)] {

        def imu = source1
        def vicon = source2

        def out = ...
    }
```

Figure 2.4: Signature of the block of the particle filter

and similar for EKFVicon.

The observant reader might notice that the above block takes as arguments `rawSourceI` and not `sourceI` directly. However, the packets are processing in the body of the class as incoming from `sourceI` (`def imu = source1`). This is a consequence of intermediary `Source` potentially needing to be generated during the graph creation to synchronize multiple scheduler together. More details below.

Graph construction

A graph can be entirely re-evaluated multiple times. For instance, we might want to run our simulation more than once. A feature of scala-flow is that the nodes of a graph are immutable and can be reused between different evaluations. This enables to serialize, store or transfer a graph easily. A graph is a data structure and scala-flow follow that intuition by separating the construction of graph and its evaluations. What is specific and shortlived for the lapse of time of an evaluation of a graph are the `Channels` between the different nodes.

Channel

`Channels` are specific to a particular node and a particular "channel" of a node. A "channel" here refers to the actual I from `listenI(packet)` of a node to call. When the graph is initialized, the channels are created according to the graph structure.

If we take a look at `Channel2` for instance:

```
sealed trait Channel[A] {
  def push(x: Timestamped[A], dt: Time): Unit
}

 ...

case class Channel2[A](receiver: Source2[_, A], scheduler: Scheduler)
    extends Channel[A] {
  def push(x: Timestamped[A], dt: Time = 0) =
     ...
}
```

We see that it requires that the receiver is a `Source2`. This actually means that the receiver must have **at least** (and not exactly) 2 sources. This is a consequence of`SourceI+1` extending `SourceI`, the base case being `Source0`: `trait Source2[A, B] extends Source1[A]`.

Now, if we look at the private method `broadcast` inside `Source[T]`

```
def broadcast(x: ⇒ Timestamped[A], t: Time = 0) =
  if (!closed)
    channels.foreach(_.push(x, t))
```

We see that `broadcast` is simply pushing elements into all its private channels. The channels are set during initialization of the graph in a simple manner: The graph is traversed and for all node the corresponding channel are created for its corresponding sources.

Buffer and cycles

It is possible to create cycles inside data-flows at the express condition that the immediate node creating a cycle is exclusively a kind of node called `Buffer`. Buffers relay to the next node any incoming data but with the particularity of a buffering of one packet. `Buffer`s are created with an initial value. When the first packet arrives, the Buffer stores the incoming packet and broadcast the initial value. When another following packet arrives, the buffer stores the new packet and broadcast the previously stored one and so on.

Even using buffer nodes, declaring cycle requires additional steps:

```scala
val source: Source[A] = ...
val buffer = Buffer(merge, initA)
val zipped = source.zip(buffer)
```

This will not be valid scala because there is a circular dependency between buffer and zipped. Indeed, instantiating `buffer` require to instantiate `zipped`, which require to instantiate `buffer` ... A solution is to use `lazy val`.

```scala
val source: Source[A] = ...
lazy val buffer = Buffer(merge, initA)
lazy val zipped = source.zip(buffer)
```

`lazy val a = e` in scala implements lazy evaluation, meaning the expression `e` is not evaluated until it is needed. In our case, this makes sure that both `buffer` and `zipped` can be declared and instantiated. It suffices that their parameters are declared of the right type, they do not actually need to evaluated. At the initialization of the entire graph, there is no circular dependency either because both instance exists and will only be used during the evaluation of the graph.

Source API

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]`

which is equivalent to the `foreach` below except it can access the additional fields `t` and `dt` in `Timestamped`

```scala
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
```

```scala
  *  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)]
```

```scala
  /** 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]

}
```

Figure 2.5: API of the Sources

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.

Batteries

The following nodes are already included and pre-defined:

- `Clock`: `Source0[Time]` that takes as parameter a timeframe `dt` which corresponds to the lapse of time between each emission of packets. The packets contain as values the time of emission.

- `TestTS`: "Test Time Series" Sink that takes a source of labeled data. Labeled data are data joined with their corresponding label. This sink displays the mean error, the max error error across all datapoints and also the RMSE.

```
[info ParticleFi ] RMSE        : 1.099241e-01, 4.213478e-03
[info ParticleFi ] Mean errors: 3.026816e-01, 4.746430e-02
[info ParticleFi ] Max  errors: 7.086643e-01, 2.386466e-01
```

- `Plot`: Sink that displays the time series under the form of a plot. Can take an arbitrary number of time series, each of arbitrary dimension. In the example below, 5 time series of 2 dimensions are plotted. The plotting library is the one included in scala-breeze, used elsewhere for matrix and vector operations.

Figure 2.6: Example of a plot generated by the Plot sink

- `Jzy3dTrajectoryVisualisation`: Sink. It displays a point following a trajectory in a new window. takes a source of points as source. An example as shown in Part I.

In addition, any `scala.Stream[A]` can be transformed into a `Source0` node using `EmitterStream[A]` with `A` being the type of the `Stream`. This is how `Clock` are implemented, as an infinite scala stream of Time.

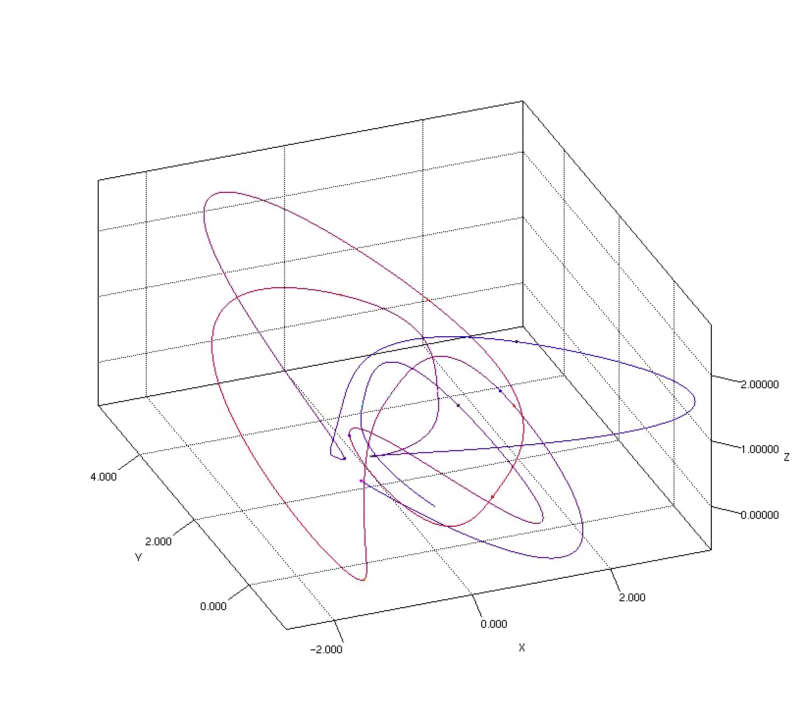


Figure 2.7: Example of a trajectory visualization

Batch

A batch is a node that process its input in “batch” mode. All the other nodes process their input in “stream” mode. By “stream” mode, it is meant that the node process the input one-by-one, as soon as it arrives. On the other hand, the “batch” mode means that the node process the incoming packets once they are all arrived, once and for all. This is the case for most sink (it makes more sense for a plot to build it once all the data is arrived). Batches are essential to the spatial integration: the nodes that simulate a spatial application can only run and treat all the data at once. Indeed, Running a spatial application involve running the spatial compiler in the background and compiling the full meta-program, including all meta-constant values.

Scheduler

Scheduling is the core mechanism of scala-flow. Scheduling ensures that packets gets emitted by the sending nodes and received by the recipient nodes at the “right time”. Since scala-flow is a simulation tool, the “scala-flow time” does not correspond at all to the real time. Scheduling emits the packets as fast as it can. Therefore, since time is an arbitrary component of the packet, the only constraint that scheduling must satisfy is emitting the packets from all nodes in the right order.

Scheduling is achieved by one or many Schedulers. Schedulers are essentially priority queue of actions. The priorities is the timestamp plus the accumulated delay of the packets. The actions are side-effect functions that emit packets to the right node by the intermediary of channels. Every node has a scheduler and enqueue action to it every time the `broadcast` method is called. The scheduler are propagated through the graph through two rules:

- Every `Source0` has for `Scheduler` the "main scheduler" available globally passed on as an implicit parameter
- Other nodes either explicitly create their own scheduler (like the batch nodes) or use the Scheduler from their `source1` input.

Only one scheduler execute actions at the same time. When a scheduler is finished, another one get started unless it was the last one. In practice, when a scheduler has no more packets to handle, there is a callback to `CloseListener` nodes and scheduler according to their `CloseListener` priority. Batches have their own scheduler and are also among `CloseListener` of the `Scheduler` of their source node, waiting for them to all finish. Batches process the accumulated packets as soon as the `CloseListener` callback is called.

All schedulers start at time 0. The current time of a scheduler is the time of the last emitted packet. `Scheduler` can en-queue new actions while the scheduler is "live" but the en-queued packet can only have a time of emission greater or equal to the current time. In the trivial case where there is no Batch, only one scheduler is needed.

Replay

`Replay` are nodes at the frontier of two schedulers. They accumulate packets from the actions of the first scheduler until they receive its `CloseListener` callback. When received, they en-queue all the accumulated actions into the second scheduler. `Replay`s are the primary mechanism of synchronization between two `Scheduler`s. A `Batch` is essentially a `Replay` with its own `Scheduler` as secondary `Scheduler`. However, a batch transforms the data before broadcasting instead of simply replaying it.

All sources of a node must share the same scheduler. Replays are automatically inserted to ensure that this rule is respected

The automatic insertion is the reason why nodes must define all `rawSourceI` but one should only externally ever use the `sourceI` methods. In most case, `rawSourceI` and `sourceI` are by definition the same. However, if a replay node has to be created, it is inserted in-between `rawSourceI` and `sourceI`.

Multi-Scheduler graph

When the graph involves multiple schedulers, depending on the graph structure, the synchronization between them might require additional replays.

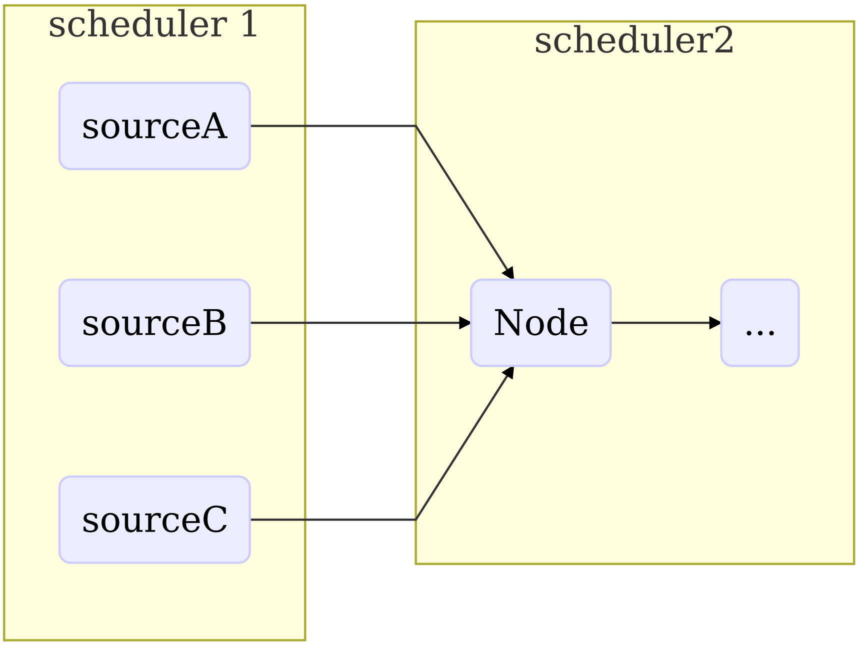


Figure 2.8: Node's sources sharing the same Scheduler

In the above structure, no replay need to be created because all sources of the node "Node" share the same scheduler. It suffices to wait for the closing callback of that scheduler.

In the above structure, intermediary replays must be created so that the node "Node" sources share the same scheduler.

InitHook

Some nodes need initialization values for each simulation evaluation. For instance, This is the case for the trajectory filters: the filters require to be given the initial position and attitude of the drone. An `InitHook[I]` is

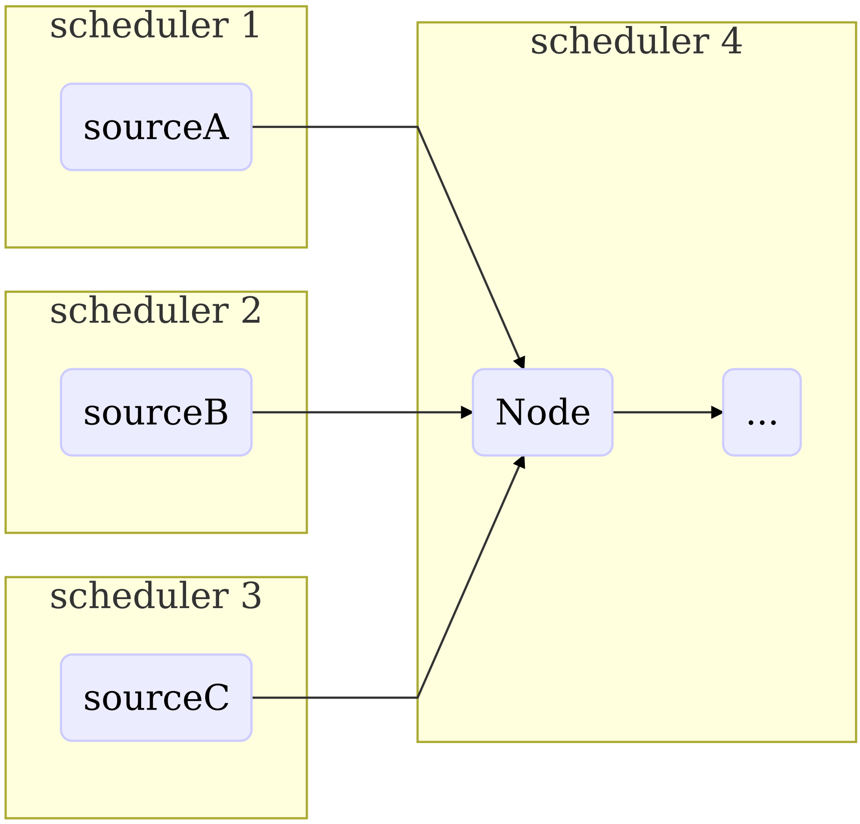


Figure 2.9: Node's sources not sharing the same Scheduler

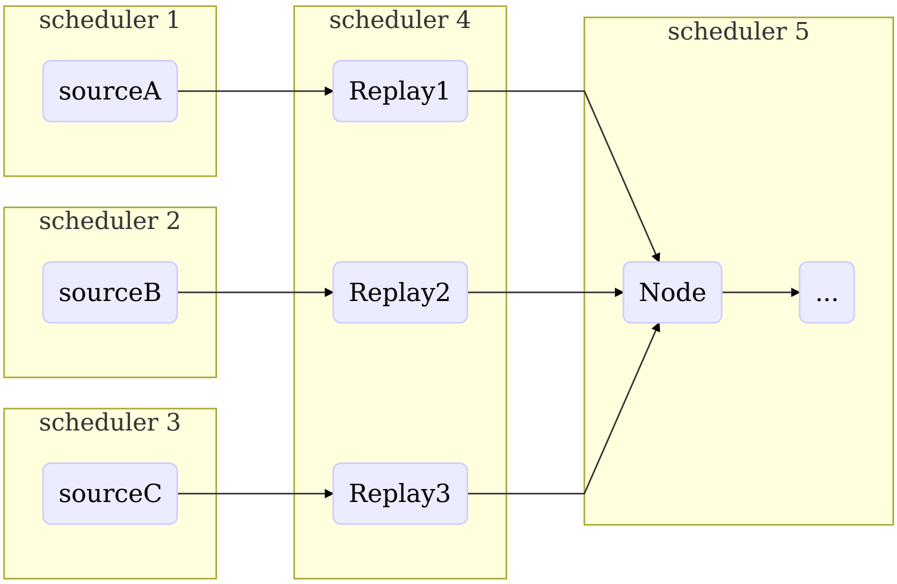


Figure 2.10: Example of Replays between inserted in-between a Node and its sources

an implicit parameter passed to the nodes during their declaration. The type parameter `I` is the type of the values that will be accessible by the nodes as initialization values.

ModelHook

Similarly, some nodes need access to a "Model". A "Model" is specific to a simulation and is an oracle that node might need to consult in order to generate data or get any other information about the external simulation environment. For instance, the sensor nodes generate noisy measurements as a function of the time based on the underlying trajectory model. Similar to `InitHook[I]`, it is passed to nodes during the graph declaration as an implicit parameter.

NodeHook

To gather the nodes and their connection between each others, a `NodeHook` is used. Every node must have access to a `NodeHook` to add itself to the registry. For the nodes that take no input, the `Source0`, the `NodeHook` is passed as an implicit parameter. For any other nodes, the `NodeHook` is propagated through the graph. All others nodes use the `NodeHook` from their `source1`. This is similar to the way `Scheduler` are propagated through the graph.

Graphical representation

The graphical representation is a graph in the `ASCII` format. The library ascii-graphs is used to generate the output in `ASCII` from sets of vertices and edges. The set of vertices and edges is retrieved from the set of nodes contained in `NodeHook` and their sources.

FlowApp

A `FlowApp[M, I]` is an extension of the scala `App`, a trait that treats the inner declaration of an object as a main program function. Its type parameters correspond respectively to the type parameter of `ModelHook[M]` and `InitHook[I]`. A `FlowApp` has the methods `drawExpandedGraph()` which display the `ASCII` representation of the graph and the method `run(model, init)` which run the evaluation of the simulation with the given model and initialization value.

Spatial integration

Scala-flow can also be used as a complementary tool for the development of applications embedding Spatial, a language to design accelerating hardware. Accelerators can be easily represented as simple transformation nodes in a data flow and hence as a regular `OpX` node in scala-flow.

`SpatialBatch` and its variants are the nodes used to embed spatial applications. `SpatialBatchRawXs` run a user-defined application. The application can use the list of incoming packet as a constant list of values. X is the number of source of the node. `SpatialBatchXs` are specialized `SpatialBatchRawXs` with additional syntactic sugar such that there is no more boilerplate and the required code is reduced to the most essential to write stream processing spatial applications. It is only to define a function `def spatial(x: TSA): SR` where `TSA` is a struct containing a value `v` of type `SA` (see below) and the packet timestamp as `t`.

If we take a look at `SpatialBatch1`'s signature,

```
abstract class SpatialBatch1[A, R, SA: Bits: Type, SR: Bits: Type]
    (val rawSource1: Source[A])
```

```
    (implicit val sa: Spatialable[A] { type Spatial = SA },
              val sr: Spatialable[R] { type Spatial = SR }
    )
```

we see that it takes type parameter A, R, SA, SR and the typeclass instances of Spatialable for SA and SR. A and R are the type members representing respectively the incoming and outgoing packet type. SA and SR are the spatial type into what they are converted to such that they can be handled by a spatial DSL. Indeed, scala.Double and spatial.Double are not the same type. The latter is a staged type part of the spatial DSL.

Spatialable[A] is a typeclass that declare a conversion from A to a spatial type (declared as the inner type member Spatial of Spatialable.

There exists a Spatialable[Time] which make the following example possible:

```
val clock1 = new Clock(0.1).stop(10)
val clock2 = new Clock(0.1).stop(10)

val spatial  = new SpatialBatch1[Time, Time, Double, Double](clock1) {
  def spatial(x: TSA) = {
    cos(x.v)
  }
}

val spatial2 = new SpatialBatch1[Time, Time, Double, Double](clock2) {
  def spatial(x: TSA) = {
    x.v + 42
  }
}


val spatial3 = new SpatialBatch2[Time, Time, Time, Double, Double, Double](spatial, spatial2) {
  def spatial(x: Either[TSA, TSB]) = {
    x match {
      case Right(t) ⇒ t.v+10
      case Left(t) ⇒ t.v-10
    }
  }
}

Plot(spatial3)
```

Figure 2.11: Usage demonstration of spatial batches

Even though it looks inconspicuous, the cos, +, - functions are actually functions from the spatial DSL. This simple scala-flow program actually compiles and runs through the interpreter 3 different spatial programs.

The development of an interpreter was required so that spatial apps could run on the same runtime than scala-flow. The interpreter development is detailed in the next part of this thesis.

Conclusion

`scala-flow` is a modern framework to simulate, develop, prototype and debug applications which have a natural representation as data-flows. Its integration with spatial makes it a good tool to include with spatial to ease the development complex applications whenever the accelerated application need to be written over multiple iterations of increasing complexity, and tested on different scenarios with modelable environment.

3 | An interpreter for spatial

Spatial: An Hardware Description Language

Building applications is only made possible thanks to the many layers of abstractions that start fundamentally at a rudimentary level. It is easy to forget how much of an exceptional feat of engineering is running an application..

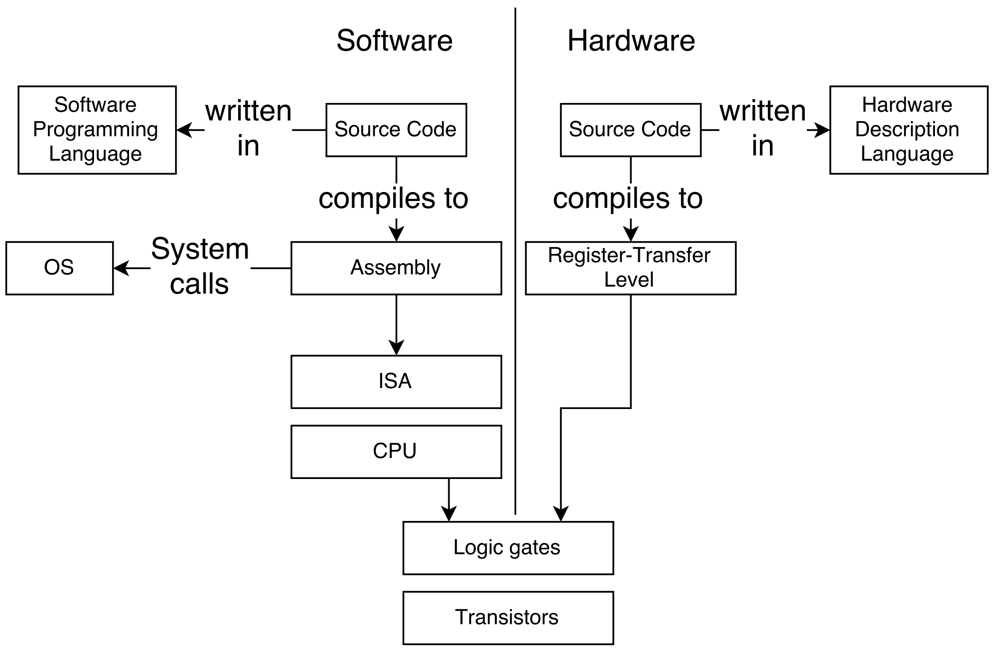


Figure 3.1: An Hardware vs Software abstraction layers overview

An Hardware Description Language (HDL) is used to describe the circuits on which applications run on. A Software programming language describe the applications themselves (imperative languages focus on the how, and functional programming languages on the what). Fundamentally, their purpose is different. But with a sufficient level of abstraction, they share many similarities.

`c := a + b` would translate in software by an instruction to store in the memory (stack or heap) the sum of `a` and `b`, stored themselves somewhere else in memory. In hardware, depending on whether `c` represents a wire, a register, a memory location in SRAM or DRAM, the circuit is changed. However, from the user perspective, the source code looks the same. One could think that it would be possible to write Hardware exactly the same way than Software, but it is delusional. Some concepts are tied to the intrinsic nature of Hardware and hopelessly inexpressible in the world of Software. A DSL that would abstract away those differences would result in a great loss of control for the user. Nevertheless, with the right level of abstraction it is possible to at least bridge the gap to a level satisfying for both the Software and Hardware engineers. This is the motivation behind Spatial.

Spatial is an (HDL) born out of the difficulties and complexity of doing Hardware. An HDL compiles to Register-Transfer Level (RTL), an equivalent to assembly in the software world. Then the RTL is synthesized as Hardware (either as Application-specific integrated circuit (ASIC) or as a bitstream reconfiguration data). The current alternatives available are Verilog, VHDL, HLS, Chisel and many others. What sets apart Spatial from the crowd is that Spatial has a higher level of abstraction by leveraging parallel patterns, abstracting control flows as language constructs and automatic timing and banking. Spatial targets "spatial architectures" constituted currently of the Field-Programmable Gate Array (FPGA) and a Coarse Grain Reconfigurable Arrays (CGRA) developed also by the lab, Plasticine. Chisel is actually the target language of Spatial for the FPGA backend. Parallel patterns and control flows are detailed in Part IV.

Spatial is at the same time a language, an Intermediate Representation (IR) and a compiler. The Spatial language is embedded in Scala as a domain specific language (DSL). The compiler is built around Argon as a set custom defined traversals, transformers and codegens. The Spatial compiler is referred to as "the staging compiler" to differentiate it from scalac, the Scala compiler.

Argon

Spatial is built on top of Argon, a fork of Lightweight Modular Staging (LMS). Argon and LMS are Scala libraries that enable staged programming (also called staged meta-programming). Thanks to Argon, language designers can specify a DSL and a custom compiler. In this DSL, users can write and run meta-programs and more specifically program generators: programs that generate other programs.

Argon is:

- **two-staged**: There is only a single meta-stage and a single object-stage. The idea behind Argon is that the meta-program is constructing an IR

programmatically in Scala through the frontend DSL, transform that IR and finally codegen the object program. All of this happening at runtime.
- **heterogenous**: The meta-program is in Scala but the generated meta-program does not have to be in Scala as well. For instance, for FPGA, the target language is both C++ and Chisel (an embedded DSL in Scala).
- **typed**: The DSL is typed which enable Scala to typecheck the construction of the IR. Furthermore, the IR is itself typed. The IR being typed ensures that language designers write sound DSL and corresponding IR.
- **automatic staging annotations**: The staging annotations are part of the frontend DSL. Implicit conversions exists from unstaged type to staged types. The staging annotations exists under the form of typeclass instances and inheritance.

Staged type

A staged type is a type that belongs to the specified DSL and has a staging annotation. Only instances of a staged type will be transformed into the initial IR.

Indeed, for a type to be considered a staged type, it must inherit from `MetaAny` and have an existing typeclass instance of `Type`. The justification behind the dual proof of membership is that the `Type` context bound is more elegant to work with in most cases. Nevertheless, it suffers that it is impossible to specialize methods such that they treat differently staged and unstaged types. Only inheritance can guarantee correct dispatching of methods according to whether the argument is staged or not. Implementing the typeclass and dual proof of membership was among the contributions of this work to Argon.

```
trait MetaAny
trait Type[A]

case class Staged() extends MetaAny
case class Unstaged()

implicit object StagedInstance extends Type[Staged]

object Attempt1 {
    //equivalent to def equal(x: Any, y: Any) =
    def equal[A, B](x: A, y: B) =
        1

    def equal[A: Type, B: Type](x: A, y: B) =
        2
}
```

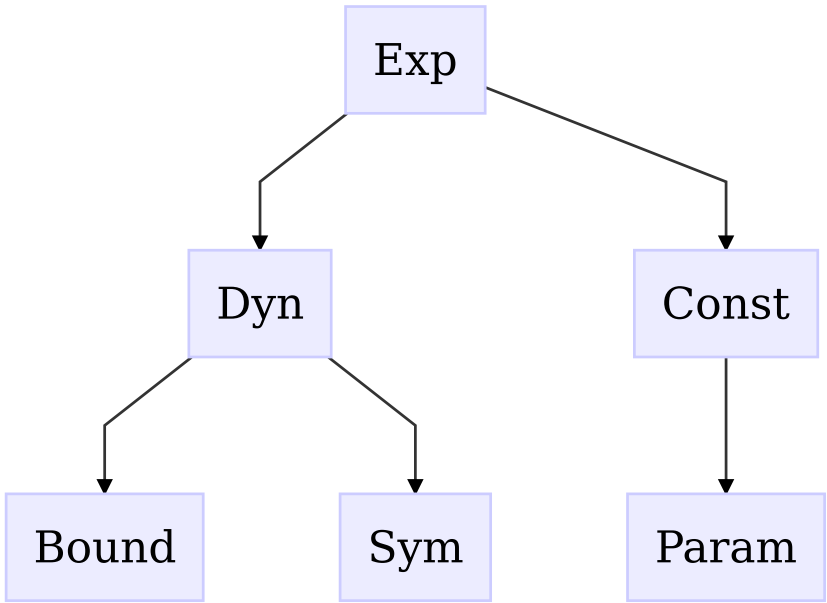
```scala
object Attempt2 {
    def equal(x: Any, y: Any) =
        1

    def equal(x: MetaAny, y: MetaAny) =
        2
}

Attempt1.equal(Unstaged(), Unstaged())
//return error: ambiguous reference to overloaded definition
Attempt1.equal(Staged(), Staged())
//return error: ambiguous reference to overloaded definition

Attempt2.equal(Unstaged(), Unstaged())
//return 1 as expected
Attempt2.equal(Staged(), Staged())
//return 2 as expected
```

Figure 3.2: Example of inheritance solving correct dispatching

IR

The IR in Argon is based on a "sea-of-nodes" representation of the object-program. The "sea-of-nodes" representation is a graph containing all kind of dependencies and anti-dependencies between the nodes of the IR. The IR is the data manipulated by the staging compiler and is transformed after each pass. The IR nodes are staged Exp values. Exp is an algebraic data type (ADT), more precisely a sum type of the following form:

- Consts are staged expressions whose value is known during staging. Since the staging compiler is aware of the value, some optimizations can be applied. For instance, constant folding can simplify the object-program considerably.
- Params are specialized Consts whose purpose is specifically for Design Space Exploration (DSE). DSE refers to the activity of exploring design alternatives prior to implementation. Users define Params as range of values and, within them, the compiler will attempt to find the best trade-off in terms of area and throughput [6] among others.
- Sym stands for "Symbol". Syms are always associated with Defs. Def is a library author defined ADT. More precisely, it is the sum type of all the product type of library author defined named staged functions. Defs take a staged type parameter and are products of other staged type only.
- Bounds are similar to Syms but are bounded by a scope. They represent staged local variables.



- Dyns are the complement of Consts and represent any staged expressions whose value is dynamic, not known during staging. It is the sum type of Bound and Sym.

Transformer and traversal

A Traversal is a pass of the compiler that traverse (iterate through) the entire IR and apply an arbitrary function. It can either be used to check if the IR is well-formed or to gather some logs and stats about the current state of the IR. Since the IR is a "sea-of-nodes", it has to be linearized first by a scheduler as a sequence of nodes. Codegen is defined a traversal.

A Transformer is a Traversal that not only traverse the IR but also transform it into a new IR.

Language virtualization

Using Scala macros, some of the primitive syntax constructions and keywords of Scala are made interoperable with spatial staged types. The following parts are currently virtualized (where cond is an `Argon.Boolean`):

- `if (cond) expr1 else expr2`
- `while (cond) expr1` (in progress)

Below, a and b are `Any`:

- `a == b`
- `a != b`
- `a.toString`
- `a + b`

Source Context

All usage of the DSL in the meta-program is accompanied with an implicit macro expansion of a `SourceContext` object. That object is passed along in the IR such that all IR node have an associated `SourceContext`. That object contains context information such as the line number, the position in the line, the method name and the content of the line on which the DSL node at the origin is located. This is how the interpreter can display the surrounding context of each interpreted instruction.

Meta-expansion

Since the DSL are embedded in Scala, it is possible to use the Scala language as a meta-programming tool directly. The construction of the IR is done in an imperative manner and only staged types are visible to the staging compiler.

```
List.tabulate(100)(_ => Reg[Int])
```

will be meta-expanded as the creation of 100 Registers.

For the same reason, when named Scala functions are defined and called inside a spatial program, the function call is not staged but inlined during meta-expansion.

```
def f(x: argon.Int) =
    //very long body

val b: argon.Boolean = ...

//thanks to language virtualization, this is syntaxic sugar for
//ifThenElse(b, f(0), f(1)) where ifThenElse is an argon defined
//function
if (b)
    f(0)
```

```
else
    f(1)
```

is expanded into

```
val b: argon.Boolean = ...

if (b)
    //very long body involving 0
else
    //very long body involving 1
```

Codegen

After the IR has been transformed through all the passes, it is sufficiently refined to be processed by the codegen. The codegen is implemented as a traversal which, after linearization by scheduling, visits each item of a sequence of pair of `Sym` and `Def`. Each pair is transformed according to the `Def` as a string in the format of the target language and written to the output file. `Def` nodes have versatile meaning since they encompass the full range of the language. Language designers add `Def` nodes to their language in a modular manner. For Spatial, each kind of data type have their associated set of `Def` which are defined in their own modules and mixed-in incrementally to the compiler. For instance, `argon.Boolean` have among others `Def` nodes that can be simplified as:

- `case class Not (a: Exp[argon.Boolean]) extends Def[argon.Boolean]`
- `case class And (a: Exp[argon.Boolean], b: Exp[argon.Boolean]) extends`
- `case class Or  (a: Exp[argon.Boolean], b: Exp[argon.Boolean]) extends`

Staging compiler flow

The full work flow of program staging through Argon is as follows: The meta-program is first compiled by scalac as an “executable meta-program”. When this executable is run, it starts meta-expansion and as a result, construct an initial IR. That initial IR goes through the different transformers which correspond to the passes of the staging compiler. Once the IR is sufficiently refined by having been through all the passes, it is codegen in the target language.

Simulation in Spatial

Synthesizing takes time, many days in some instances. It is beneficial for users to have access to an early proof of correctness of the program's logic. This justifies the existence of a simulation mode. Before the development of the interpreter, the simulation mode was a codegen whose target was a Scala program of the simulated circuit logic. The resulting Scala program is self-contained and reproduce the execution of the hardware design on the circuit but only to some extent. To mirror exactly the execution of the design, it is required to write a cycle accurate simulator. It is possible but not simple, especially writing it in a codegen form. Furthermore, a cycle-accurate simulator already exists: Synopsys Verilog Compiler Simulator (VCS). However, VCS takes Verilog as input. Hence, it cannot leverage the richer informations from the DSL and the debugging cannot be enhanced with Spatial annotations (for instance with `SourceContext`). Finally, writing a compiler is more complex than writing an equivalent interpreter.

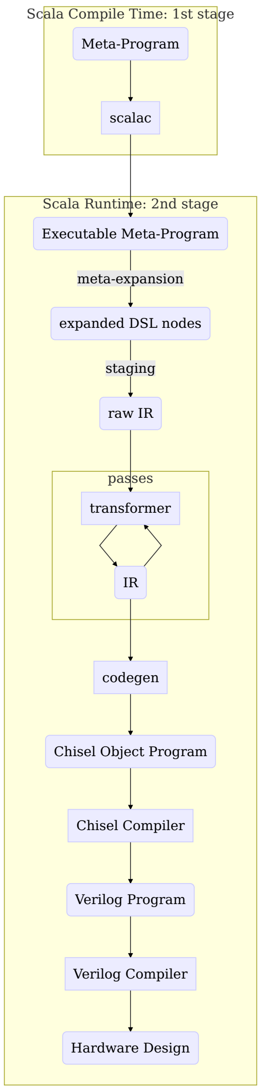


Figure 3.3: Flow diagram of the argon compiler

Benefits of the interpreter

Building an interpreter for Spatial was a requirement of having a spatial integration in scala-flow. Furthermore, it is requirement to integrate a spatial simulator into any external library. It also benefits the spatial ecosystem as a whole. Indeed, an interpreter encourages the user to have more interactions with the language and working in increasing complexity iterations thanks to fast feedback since the work flow involves less steps, is faster to launch and is more tightly integrated with spatial (The interpreter has access to `SourceContext` among others). The interpreter is not yet cycle-accurate but it is planned as future work.

Interpreter

The interpreter is implemented as an alternative to codegen. The largest benefit of this approach is that the interpreter sees an IR that has already been processed and can mirror closely the codegen and the intended evaluation of the generated code. Moreover, if one of the pass fails or throw an error, then running the interpreter will also halt at that error.

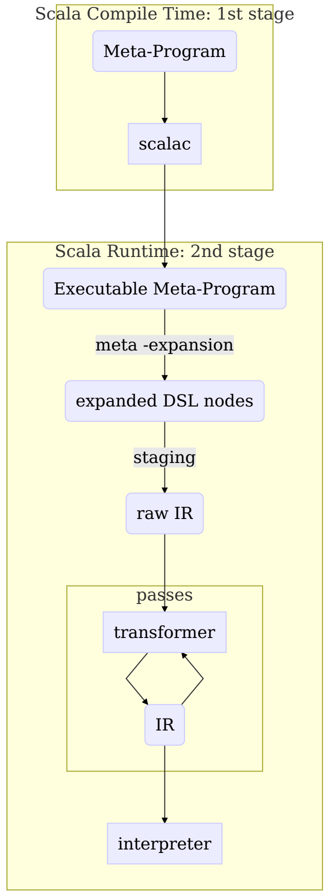


Figure 3.4: Flow diagram of the argon interpreter

Usage

Any spatial application can be run using the CLI flag `--interpreter`. If used in combination with the flag `-v` (for "verbose", setting the verbosity level to 2), each instruction interpretation will display the full state of the interpreter. If used without any verbosity flag, then the verbosity is set to 1 by default and only the name and number of the instruction is displayed at each step. Finally, if the flag `-q` (for "quiet", sets the verbosity level to 0) is used, then nothing is displayed during the interpretater execution. At all verbosity level, the state of the interpreter including the result in the output bus, if any, is displayed after the last instruction has been interpreted.

Debugging nodes

The Argon DSL has been extended with the static methods `breakpoint()` and `exit()`. The method `breakpoint()` pauses the interpreter and displays its internal state. A key must be pressed to resume the interpreter evaluation.

The method `exit()` stops the evaluation of the interpreter.

Interpreter stream

In addition to standard applications being able to run as-is, applications that rely on stream have been given some specific attention in order to

```
StreamInOutAdd.scala:11:14
      out := in + 4
[node]: StreamRead(x97,Const(true)) -> x100
[instruction #]: 16
[context]
StreamInOutAdd.scala:11:17
      out := in + 4
[node]: FixAdd(x100,Const(4)) -> x101
[instruction #]: 17
[context]
StreamInOutAdd.scala:11:11
      out := in + 4
[node]: StreamWrite(x98,x101,Const(true)) -> x102
[instruction #]: 18
[context]
StreamInOutAdd.scala:12:7
      breakpoint
[node]: BreakpointIf(Const(true)) -> x103
[instruction #]: 19
[breakpoint info]
[sram content]
[fifo content]
[reg content]
[regfiles content]
[LUT content]
[others]
x100: 6.00000e+00
x101: 1.00000e+01
x97: Queue()
x98: Queue(5.00000e+00, 6.00000e+00, 7.00000e+00, 1.00000e+01)
x99: ForeverC()
[bounds content]
Reached a breakpoint: press a key to continue or q to quit

[warn] 2 warnings found
[completed] Total time: 9.9010 seconds

[result]
Out1: Queue(5.00000e+00, 6.00000e+00, 7.00000e+00, 1.00000e+01)

[success] Total time: 16 s, completed Aug 8, 2017 1:04:02 PM
```

Figure 3.5: Screenshot of the interpreter in action

ease their usage with the interpreter. Indeed, being able to run a spatial application in the same runtime than the application compilation gives the means to do in-memory transfer between the meta-program (or the larger surrounding program), to the object program to the input streams. This is made easy by using the following pattern:

- The meta-program itself must be written in a trait extending `SpatialStream`
- The main entry point of the interpreter mix-ins the meta-program definition trait with `SpatialStreamInterpreter` and declares the input buses as well their content at start and the output buses.
- The main entry point for synthesizing mix-ins the meta-program definition trait with `SpatialStreamCompiler`

An example is provided below:

```
trait StreamInOutAdd extends SpatialStream {

  @virtualize def spatial() = {
    val in  = StreamIn[Int](In1)
    val out = StreamOut[Int](Out1)
    Accel(*) {
      out := in + 4
      breakpoint
    }
  }
}


object StreamInOutAddInterpreter extends StreamInOutAdd
    with SpatialStreamInterpreter {

  val outs = List(Out1)

  val inputs = collection.immutable.Map[Bus, List[MetaAny[_]]](
    (In1 -> List[Int](1, 2, 3, 6))
  )

}

object StreamInOutAddCompiler extends StreamInOutAdd
    with SpatialStreamCompiler
```

In-memory transfer was essential in integrating the spatial interpreter with scala-flow.

Implementation

The IR given to the interpreter follows a static single assignment (SSA) form. The interpreter is implemented as a traversal which, after linearization by scheduling, visits a sequence of pair of `Sym` and `Def`. The interpreter core is a central memory that contains the values of all symbols, and an auxiliary memory that contains the values of the temporary bounds. The pair of `Sym` and `Def` is processed by evaluating the `Def` node through modular extensions of the interpreter that mirror the modular partitioning of the IR itself. Once evaluated, the result is stored in the central memory with index the `Sym`. An `eval` function is used as an auxiliary method for the evaluation of the nodes. It takes as argument an `Exp` and can be simplified as:

- if the argument is a `Sym`, retrieve the corresponding values in the central memory
- if the argument is a `Const`, return the value of the Const

Here is a very simplified example

```
val a: argon.Int = ...
val b = 2 + a
b * 4 * b

//becomes roughly after staging and linearization
Seq(
    (x1, Add(Const(2), a)),
    (x2, Mult(x1, Const(4))),
    (x3, Mult(x2, x1))
)

//let's assume the central memory starts with:
a ← 1
//the interpreter will evaluates the seq as
x1 ← eval(2) + eval(a) = 2 + 1 = 3
x2 ← eval(x1) * eval(4) = 3 * 4 = 12
x3 ← eval(x2) * eval(x1) = 12 * 3 = 46
```

Blocks and control flows handling rely on node-defined traversals of their inner body. Loops with various parallelizing factors are handled using an interpreter specific scheduler.

The currently implemented modules of the Spatial IR for the interpreter are:

- `Controllers`
- `FileIOs`
- `Debuggings`
- `HostTransfers`
- `Regs`
- `Strings`
- `FixPts`

- `FltPts`
- `Arrays`
- `Streams`
- `Structs`
- `SRAMs`
- `DRAMs`
- `Booleans`
- `Counters`
- `Vectors`
- `FIFOs`
- `FSMs`
- `RegFiles`
- `Maths`
- `LUTs`

Conclusion

The addition of an interpreter to Argon and Spatial improves the whole eco-system and offer new possibilities. Maintenance and extension of the simulator will be easier to write in an interpreter form, especially if a cycle-accurate simulator is developed. It is hoped that the interpreter will prove itself useful in the workflow of all app developers and become a core element of Spatial.

4 Spatial implementation of an asynchronous Rao-Blackwellized Particle Filter

A Rao-Blackwellized Particle Filter turned out to be an ambitious application, the most complex that was developed so far with Spatial. It is an embarrassingly parallel algorithm and hence can leverage the parallelizable benefits of an application-specific hardware design. Developping this, We gained some insights specific about the hardware implementation of such application and some others specific to the particularities of Spatial. At the time of the writing, some spatial incomplete codegen did not allow to synthesize fully the application but it ran correctly in the simulation mode and the area usage estimation fitted on a Zynq board.

Area

The capacity of an FPGA is defined by the synthetizable area. **TODO**

Parallel patterns

Parallel patterns [7] **TO

Controls flows

Control flows (or flow of control) is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated. Sequential Parallel Pipeline: Inner Pipe: Basic form of pipelining; Pipelining of primitive instructions. Coarse-Grain: Pipelining of parralel patterns Stream Pipe: ASAP with FIFOs stack or Streams: Stream(*)

TODO

Numeric types

Numbers can be represented in two ways:

- **fixed-point**: In the fixed-point representation, an arbitrary number of bits I represent the integer value, an arbitrary number of bits D represent the decimal value. If the representation is signed, negative numbers are represented using 2's complement.

I defines the range (the maximum number that can be represented) and D defines the precision. The range is centered on 0 if the representation is signed.

In Spatial, the fixed-point type is declared by `FixPt[S: _BOOL, I: _INT, D: _INT]`. `_BOOL` is the typeclass of the types that represents a boolean. true and false types are `_TRUE` and `_FALSE`.
Likewise for`_INT`,the typeclass of types that represent a literal integer. The integers from 0 to 64 have the corresponding types `_0`, `_1`, …, `_64`.

- **floating-point**: In the floating-point representation, one bit represents the sign, an arbitrary number of bits E represent the exponent and an arbitrary number of bitsS represent the significand part.

In Spatial, the floating-point type is declared by `FltPt[S: _INT, E: _INT]`.

By comparison, in the software world, the common available numeric types for integers are fixed points: `Byte` (8-bits), `Short` (16-bits), `Int` (32-bits), `Long` (64-bits) and for real floating-point: `Float` (32-bits), `Double` (64-bits).

The floating-point representation is required for some applications because its precision increase as we get closer to 0: the space between all representable numbers around 0 diminish whereas it is uniform over the whole domain for the fixed point representation. This can be extremely important to store probabilities (since joint probability when not normalized can be infinitesimally small), or to store the result of exponentiation of negative numbers (a small difference in value might represent a big difference pre-exponentiation), or to store the values of square (we need more precision the closest we are from 0 because the line of the real squared is more "dense" the closer we are from 0). However, floating-point operations utilize more ALU than fixed-point (an increase by an order of magnitude of around 2)

Fortunately, it is easy to define a type Alias to gather all the values that should share the same representation and then switch from floating-point to the fixed-point representation and tune the allocated number of bits by editing solely the type alias.

(a) Host Interfaces

`Accel{body}`
A blocking accelerator design.

`Accel(*){body}`
A non-blocking accelerator design.

(b) Control Structures

`min until max by stride* par factor*`
A counter over the range [**min**,**max**).
  stride: optional counter stride, default is 1
  factor: optional counter parallelization, default is 1

```
if (cond){body}
[else if (cond){body} ]
[else {body} ]
```
Data-dependent execution.
Doubles as a multiplexer if all bodies return scalar values.
  cond: condition for execution of associated body
  body: arbitrary expression

`FSM(init)(continue){action}{next}`
An arbitrary finite state machine, similar to a *while* loop.
  init: the FSM's initial state
  continue: the "while" condition for the FSM
  action: arbitrary expression, executed each iteration
  next: function calculating the next state

`Foreach(counter+){body}`
A parallelizable *for* loop.
  counter: counter(s) defining the loop's iteration domain
  body: arbitrary expression, executed each loop iteration

`Reduce(accum)(counter+){func}{reduce}`
A scalar reduction loop, parallelized as a tree.
  accum: the reduction's accumulator register
  counter: counter(s) defining the loop's iteration domain
  func: arbitrary expression which produces a scalar value
  reduce: associative reduction between two scalar values

`MemReduce(accum)(counter+){func}{reduce}`
Reduction over addressable memories.
  accum: an addressable, on-chip memory for accumulation
  counter: counter(s) defining the loop's iteration domain
  func: arbitrary expression returning an on-chip memory
  reduce: associative reduction between two scalar values

`Stream(*){body}`
A streaming loop which never terminates.
  body: arbitrary expression, executed each loop iteration

`Parallel{body}`
Overrides normal compiler scheduling. All statements in the body are instead scheduled in a *fork-join* fashion.
  body: arbitrary sequence of controllers

`DummyPipe{body}`
A "loop" with exactly one iteration.
Inserted by the compiler, generally not written explicitly.
  body: arbitrary expression

(c) Optional Scheduling Directives

`Sequential.(Foreach|Reduce|MemReduce)`
Sets loop to run sequentially.

`Pipe(ii*).(Foreach|Reduce|MemReduce)`
Sets loop to be pipelined.
  ii: optional overriding initiation interval

`Stream.(Foreach|Reduce|MemReduce)`
Sets loop to be streaming.

`Parallel.(Foreach|Reduce|MemReduce)`
Informs the compiler that the loop is parallelizable.

(d) On-Chip Memories

`FIFO[T](depth)`
FIFO (queue) with a capacity of **depth** elements of type **T**

`FILO[T](depth)`
A FILO (stack) with a capacity of **depth** elements of type **T**

`LineBuffer[T](r, c)`
On-chip buffered scratchpad containing **r** buffers of **c** elements

`LUT[T](dims+)(elements+)`
Read-only Lookup Table containing supplied **elements** of type **T**

`Reg[T](reset*)`
Register holding a value of type **T**, with optional **reset** value

`RegFile[T](dims+)`
Register file of elements of type **T** with given dimensions

`SRAM[T](dims+)`
On-chip scratchpad of elements of type **T** with given dimensions

(e) Shared Host/Accelerator Memories

`ArgIn[T]`
Accelerator register initialized by the host

`ArgOut[T]`
Accelerator register visible to the host after accelerator execution

`HostIO[T]`
Accelerator register which the host may read and write at any time.

`DRAM[T](dims+)`
Burst-addressable, host-allocated off-chip memory.

(f) External Interfaces

`StreamIn[T](bus)`
Streaming input from a **bus** of external pins.

`StreamOut[T](bus)`
Streaming output to a **bus** of external pins.

(g) Design Space Parameters

```
default (min::max)
default (min::stride::max)
```
A compiler-aware design parameter with given **default** value.
Automated DSE explores the range [**min**, **max**] with optional **stride**.

Table 4.1: A subset of Spatial's syntax. Square brackets (e.g. `[T]`) represent a template's type parameter. Parameters followed by a '+' denotes an argument which can be given one or more times, while a '*' denotes that an argument is optional. `DRAM`s, `LUT`s, `RegFile`s, and `SRAM`s can be allocated with an arbitrary number of dimensions. `Foreach`, `Reduce`, and `MemReduce` support multi-dimensional iteration domains.

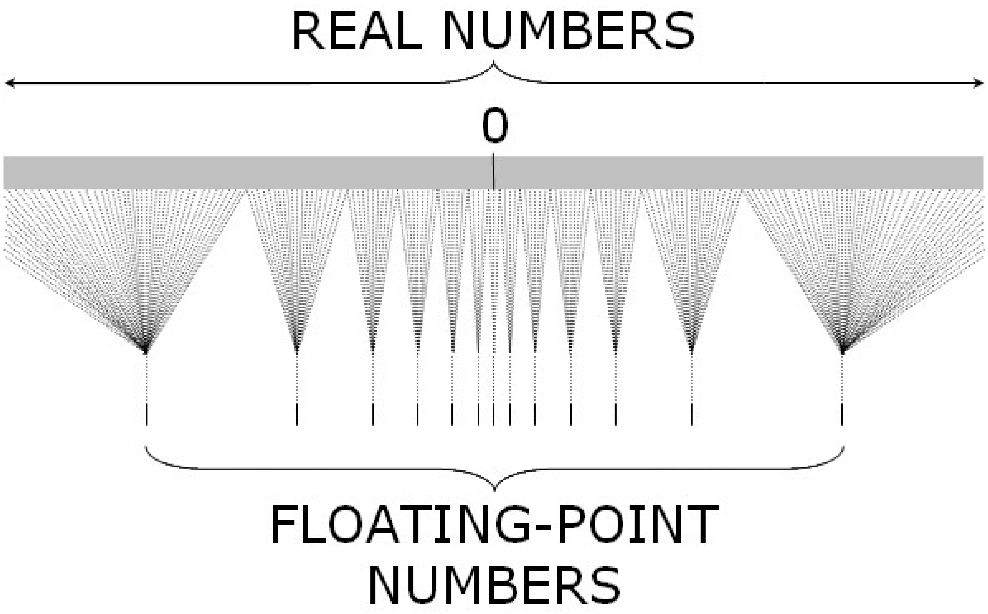


Figure 4.1: Representable Real line and its corresponding floating-point representation

```
//only this line need to be edited to change the representation
type SmallReal = FixPt[_TRUE, _4, _16]

val a: SmallReal = ...
val b: SmallReal = ...
```

vector and matrix module as stdlib

The state and uncertainty of a particle are a vector and a matrix (the matrix of covariance). All the operations involving the state and uncertainty, in particular Kalman prediction and Kalman update are matrix and vector operations. Kalman instance for instance, when written in the matrix form is reasonably compact in the matrix form but actually represent a significant amount of compute and operations. For the sake of code clarity, it is crucial to be able to keep being able to write matrix operations in a succinct syntax. Furthermore, matrix and vector operations are common need and it would be beneficial to write a reusable set of operations to Spatial. This is why a vector and matrix module was developed and added to the standard library of Spatial. It is the first module of the standard library whose purpose is to include all the common set of operations that are not part of the API of the primitives of the language.

Views

Meta-Programming

Test

Mini Particle Filter

A "mini" particle Filter has been developed at first. This version is a plain particle filter so it is not conditionned on a latent variable and there is no kalman filtering, and thus no matrix operations. **TODO** The full Mini Particle Filter source code application is contained in the Appendix and publicly available as a Spatial App on github.

Rao-Blackwellized Particle Filter

The full Rao-Blackwellized source code application is contained in the Appendix and publicly available as a Spatial App on github. **TODO** explain the FSM queue

Insights

- Changing the numeric type matter
- Doing in-place operation break pipelining
- Reducing the number of operation between first read and write is crucial

Conclusion

The Rao-Blackwellized Particle Filter is a complex application. It might have been impractical, almost to the point of unfeasibility to attempt to write it and keep an almost optimal retiming of the hardware design, in a timely manner for a single person.

Conclusion

We have presented in this work a novel approach to POSE estimation, its mathematical modeling, its implementation in Software and Hardware, and developed spatial tools, including a standalone data-flow simulation tool, to ease the Hardware implementation development and at the same time, improve the whole Spatial ecosystem.

Acknowledgments

Thank you to my parents for their continuous support, to Prof. Kunle and Prof. Odersky for supervising me and giving me the opportunity of doing this master thesis in their lab, to the entire lab of DAWN, in particular David Koeplinger, Raghu Prabhakar, Matt Feldman, Yaqi Zhang, Tian Zhao, Stefan Hadjis which accepted me as their peer for the length of my stay. I would also like to thank Nada Amin which supervised me in the semester project that led to this project and accepted to be an expert for the evaluation of the thesis. I am also grateful to the whole institution of EPFL for the education I have received those last 5 years and for which this thesis represents the culmination. Finally, to Stanford for having welcomed me for 6 months as a Visiting Researcher Student.

Appendix

Mini Particle Filter

```
type CReal = scala.Double
type SReal  = FixPt[TRUE, _16, _16]

implicit def toSReal(x: CReal) = x.to[SReal]

type STime          = SReal
type SPosition2D      = SVec2
type SAcceleration2D = SVec2
type SWeight        = SReal

@struct case class SVec2(x: SReal, y: SReal)
@struct case class TSA(t: Double, v: SAcceleration2D)
@struct case class TSB(t: Double, v: SPosition2D)
@struct case class TSR(t: Double, v: SPosition2D)

val N: scala.Int = 10
val covAcc: scala.Double = 0.01
val covGPS: scala.Double = 0.01
val dt: scala.Double = 0.1

val lutP: scala.Int = 10000

lazy val sqrtLUT =
  LUT[SReal](lutP)(List.tabulate[SReal](lutP)
      (i => math.sqrt(((i/lutP.toDouble)*5))):_*)

lazy val logLUTSmall =
  LUT[SReal](lutP)(((-9:SReal)::List.tabulate[SReal](lutP-1)
      (i => math.log(((i+1)/lutP.toDouble)*1))):_*)

lazy val logLUTBig =
  LUT[SReal](lutP)(((-9:SReal)::List.tabulate[SReal](lutP-1)
      (i => math.log(((i+1)/lutP.toDouble)*200))):_*)

lazy val expLUT =
  LUT[SReal](lutP)(List.tabulate[SReal](lutP)
      (i => math.exp(i/lutP.toDouble*20-10)):_*)

@virtualize
def log(x: SReal) = {
  if (x < 1.0)
    logLUTSmall(((x/1.0)*(lutP.toDouble)).to[Index])
  else
```

```
    logLUTBig(((x/200.0)*(lutP.toDouble)).to[Index])
}

@virtualize
def sqrt(x: SReal) = {
  if (x < 5)
    sqrtLUT(((x/5.0)*(lutP.toDouble)).to[Index])
  else
    sqrt_approx(x)
}

@virtualize
def exp(x: SReal): SReal =  {
  if (x ≤ -10)
    4.5399929762484854E-5
  else
    expLUT((((x+10)/20.0)*(lutP.toDouble)).to[Index])

}


val initV: (CReal, CReal) = (0.0, 0.0)
val initP: (CReal, CReal) = (0.0, 0.0)

val matrix = new Matrix[SReal] {
  val IN_PLACE = false
  def sqrtT(x: SReal) = sqrt(x)
  val zero           = 0
  val one            = 1
}

import matrix._

def toMatrix(v: SVec2): Matrix = {
  Matrix(2, 1, List(v.x, v.y))
}

def toVec(v: SVec2): Vec = {
  Vec(v.x, v.y)
}

def initParticles(weights: SRAM1[SWeight], states: SRAM2[SReal],
                  parFactor: Int) = {

  sqrtLUT
  expLUT
  logLUTSmall
  logLUTBig
  Foreach(0 :: N par parFactor)(x => {

    Pipe {
      Pipe { states(x, 0) = initV._1 }
      Pipe { states(x, 1) = initV._2 }
      Pipe { states(x, 2) = initP._1 }
      Pipe { states(x, 3) = initP._2 }
    }

    weights(x) = math.log(1.0 / N)

  })
}

@virtualize def spatial() = {

  val inAcc     = StreamIn[TSA](In1)
  val inGPS     = StreamIn[TSB](In2)
  val out       = StreamOut[TSR](Out1)
```

```
  val parFactor = 1 (1 -> N)

  Accel {

    val weights = SRAM[SWeight](N)
    val states  = SRAM[SReal](N, 4)

    Sequential {

      initParticles(weights, states, parFactor)

      Sequential.Foreach(1 :: 101)(i => {

        updateFromAcc(inAcc.v, dt, states, parFactor)

        if (i%5 == 0) {
          updateFromGPS(inGPS.v, weights, states, parFactor)
          normSWeights(weights, parFactor)
        }

        out := TSR(i.to[Double]*dt.to[Double],
            averagePos(weights, states, parFactor))

        if (i%5 == 0 && tooLowEffective(weights))
          resample(weights, states, parFactor)

      })
    }
  }
}

@virtualize def updateFromAcc(acc: SAcceleration2D, dt: STime,
                              states: SRAM2[SReal], parFactor: Int) = {

  Foreach(0 :: N par parFactor)(i => {
    val dv = sampleVel(acc, dt, covAcc)
    val s  = Matrix.fromSRAM1(4, states, i)
    val ds = Matrix(4, 1, List(dv(0), dv(1), s(0, 0)*dt, s(1, 0)*dt))
    val ns = s + ds
    ns.loadTo(states, i)
  })
}

@virtualize def tooLowEffective(weights: SRAM1[SReal]): Boolean = {
  val thresh = log(1.0/N)
  val c = Reduce(0)(0::N)(i => if (weights(i) > thresh) 1 else 0)(_+_)
  c < N/10
}

def updateFromGPS(pos: SPosition2D, weights: SRAM1[SReal],
                  states: SRAM2[SReal], parFactor: Int) = {
  val covPos = Matrix.eye(2, covGPS)
  Foreach(0 :: N par parFactor)(i => {
    val state = Matrix.fromSRAM1(2, states, i, false, 2)
    val lik   = unnormalizedGaussianLogPdf(toMatrix(pos), state, covPos)
    weights(i) = weights(i) + lik
  })
}

@virtualize def sampleVel(a: SAcceleration2D, dt: STime,
                          covAcc: SReal): Vec = {
  val withNoise  = gaussianVec(toVec(a), covAcc)
  val integrated = withNoise * dt
  integrated
}

def gaussianVec(mean: Vec, variance: SReal) = {
```

```
  val g1 = gaussian()
  (Vec(g1._1, g1._2) * sqrt(variance)) + mean
}

//Box-Muller
//http://www.design.caltech.edu/erik/Misc/Gaussian.html
@virtualize def gaussian() = {

  val x1 = Reg[SReal]
  val x2 = Reg[SReal]
  val w  = Reg[SReal]
  val w2 = Reg[SReal]

  FSM[Boolean, Boolean](true)(x => x)(x => {
    x1 := 2.0 * random[SReal](1.0) - 1.0
    x2 := 2.0 * random[SReal](1.0) - 1.0
    w := (x1 * x1 + x2 * x2)
  })(x => w.value >= 1.0)

  w2 := sqrt((-2.0 * log(w.value)) / w)

  val y1 = x1 * w2;
  val y2 = x2 * w2;

  (y1, y2)
}

@virtualize def normSWeights(weights: SRAM1[SWeight], parFactor: Int) = {
  val totalSWeight = Reg[SReal](0)
  val maxR         = Reg[SReal](-100)
  maxR.reset
  totalSWeight.reset
  Reduce(maxR)(0 :: N)(i => weights(i))(max(_, _))
  Reduce(totalSWeight)(0 :: N)(i => exp(weights(i) - maxR))(_ + _)
  totalSWeight := maxR + log(totalSWeight)
  Foreach(0 :: N par parFactor)(i => {
    weights(i) = weights(i) - totalSWeight
  })

}

@virtualize def resample(weights: SRAM1[SWeight], states: SRAM2[SReal],
                         parFactor: Int) = {

  val cweights  = SRAM[SReal](N)
  val outStates = SRAM[SReal](N, 4)

  val u = random[SReal](1.0)

  Foreach(0 :: N)(i => {
    if (i == 0)
      cweights(i) = exp(weights(i))
    else
      cweights(i) = cweights(i - 1) + exp(weights(i))
  })

  val k = Reg[Int](0)
  Foreach(0 :: N)(i => {
    def notDone = (cweights(k) * N < i.to[SReal] + u) && k < N
    FSM[Boolean, Boolean](notDone)(x => x)(x => k := k + 1)(x => notDone)

    Foreach(0 :: 4)(x => {
      outStates(i, x) = states(k, x)
    })

  })
```

```
  Foreach(0 :: N par parFactor)(i ⇒ {

    Foreach(0 :: 4)(x ⇒ {
      states(i, x) = outStates(i, x)
    })

    weights(i) = log(1.0 / N)
  })

}

def unnormalizedGaussianLogPdf(measurement: Matrix, state: Matrix,
                               cov: Matrix): SReal = {
  val e = (measurement - state)
  -1 / 2.0 * ((e.t * (cov.inv) * e).apply(0, 0))
}

@virtualize def averagePos(weights: SRAM1[SReal], states: SRAM2[SReal],
                           parFactor: Int): SVec2 = {
  val accumP = RegFile[SReal](2, List[SReal](0, 0))
  accumP.reset
  Foreach(0 :: N par parFactor, 0 :: 2)((i, j) ⇒ {
    accumP(j) = accumP(j) + exp(weights(i)) * states(i, j + 2)
  })

  SVec2(accumP(0), accumP(1))
}
```

Rao-Blackwellized Particle Filter

```
type SCReal = scala.Double
type SReal  = FixPt[TRUE, _16, _16]

implicit def toReal(x: SCReal) = x.to[SReal]

val N: scala.Int                                     = 10
val initV: (SCReal, SCReal, SCReal)                  = (0.0, 0.0, 0.0)
val initP: (SCReal, SCReal, SCReal)                  = (0.0, 0.0, 0.0)
val initQ: (SCReal, SCReal, SCReal, SCReal) = (1.0, 0.0, 0.0, 0.0)
val initCov = 0.00001

val initTime: SCReal  = 0.0
val covGyro: SCReal   = 1.0
val covAcc: SCReal    = 0.1
val covViconP: SCReal = 0.01
val covViconQ: SCReal = 0.01

@struct case class SVec3(x: SReal, y: SReal, z: SReal)


type STime         = SReal//Double
type SPosition     = SVec3
type SVelocity     = SVec3
type SAcceleration = SVec3
type SOmega        = SVec3
type SAttitude     = SQuat


@struct case class SQuat(r: SReal, i: SReal, j: SReal, k: SReal)
@struct case class SIMU(a: SAcceleration, g: SOmega)
```

```scala
@struct case class TSA(t: STime, v: SIMU)
@struct case class SPOSE(p: SVec3, q: SAttitude)
@struct case class TSB(t: STime, v: SPOSE)
@struct case class TSR(t: STime, pose: SPOSE)
@struct case class Particle(w: SReal, q: SQuat,
                            lastA: SAcceleration, lastQ: SQuat)


val lutP: scala.Int = 10000
val lutAcos: scala.Int = 1000

lazy val acosLUT =
  LUT[SReal](lutAcos)(List.tabulate[SReal](lutAcos)
      (i => math.acos(i/lutAcos.toDouble)):_*)

lazy val sqrtLUT =
  LUT[SReal](lutP)(List.tabulate[SReal](lutP)
      (i => math.sqrt(((i/lutP.toDouble)*5))):_*)

lazy val logLUTSmall =
  LUT[SReal](lutP)(((-9:SReal)::List.tabulate[SReal](lutP-1)
      (i => math.log(((i+1)/lutP.toDouble)*1))):_*)

lazy val logLUTBig =
  LUT[SReal](lutP)(((-9:SReal)::List.tabulate[SReal](lutP-1)
      (i => math.log(((i+1)/lutP.toDouble)*200))):_*)

lazy val expLUT =
  LUT[SReal](lutP)(List.tabulate[SReal](lutP)
      (i => math.exp(i/lutP.toDouble*20-10)):_*)


def sin(x: SReal) = sin_taylor(x)
def cos(x: SReal) = cos_taylor(x)

@virtualize
def log(x: SReal) = {
  if (x < 1.0)
    logLUTSmall(((x/1.0)*(lutP.toDouble)).to[Index])
  else
    logLUTBig(((x/200.0)*(lutP.toDouble)).to[Index])
}

@virtualize
def sqrt(x: SReal) = {
  if (x < 5)
    sqrtLUT(((x/5.0)*(lutP.toDouble)).to[Index])
  else
    sqrt_approx(x)
}

@virtualize
def exp(x: SReal): SReal =  {
  if (x <= -10)
    4.5399929762484854E-5
  else
    expLUT((((x+10)/20.0)*(lutP.toDouble)).to[Index])

}


@virtualize
def acos(x: SReal) = {
  val ind = (x*(lutP.toDouble)).to[Index]
  if (ind <= 0)
    0
  else if (ind >= lutAcos)
```

```
    PI
  else {
    val r = acosLUT(ind)
    if (x ⩾ 0)
      r
    else
      PI - r
  }
}

val matrix = new Matrix[SReal] {
  val IN_PLACE = false
  def sqrtT(x: SReal) = sqrt(x)
  val zero = 0.to[SReal]
  val one = 1.to[SReal]
}
import matrix._

def toMatrix(v: SVec3): Matrix = {
  Matrix(3, 1, List(v.x, v.y, v.z))
}

def toVec(v: SVec3): Vec= {
  Vec(v.x, v.y, v.z)
}

implicit class SQuatOps(x: SQuat) {
  def *(y: SReal)        = SQuat(x.r * y, x.i * y, x.j * y, x.k * y)
  def *(y: SQuat)        = SQuatMult(x, y)
  def dot(y: SQuat)      = x.r * y.r + x.i * y.i + x.j * y.j + x.k * y.k
  def rotateBy(q: SQuat) = q * x
  def rotate(v: SVec3): SVec3 = {
    val inv = x.inverse
    val nq  = (x * SQuat(0.0, v.x, v.y, v.z)) * inv
    SVec3(nq.i, nq.j, nq.k)
  }
  def inverse = SQuatInverse(x)
}

def SQuatMult(q1: SQuat, q2: SQuat) = {
  SQuat(
    q1.r * q2.r - q1.i * q2.i - q1.j * q2.j - q1.k * q2.k,
    q1.r * q2.i + q1.i * q2.r + q1.j * q2.k - q1.k * q2.j,
    q1.r * q2.j - q1.i * q2.k + q1.j * q2.r + q1.k * q2.i,
    q1.r * q2.k + q1.i * q2.j - q1.j * q2.i + q1.k * q2.r
  )
}

def SQuatInverse(q: SQuat) = {
  val n = q.r * q.r + q.i * q.i + q.j * q.j + q.k * q.k
  SQuat(q.r, -q.i, -q.j, q.j) * (1 / n)
}

@virtualize def initParticles(particles: SRAM1[Particle],
                              states: SRAM2[SReal],
                              covs: SRAM3[SReal],
                              parFactor: Int) = {

  acosLUT
  logLUTSmall
  logLUTBig
  sqrtLUT
  expLUT

  Sequential.Foreach(0::N par parFactor)(x ⇒ {

    Pipe {
```

```
      Pipe { states(x, 0) = initV._1 }
      Pipe { states(x, 1) = initV._2 }
      Pipe { states(x, 2) = initV._3 }
      Pipe { states(x, 3) = initP._1 }
      Pipe { states(x, 4) = initP._2 }
      Pipe { states(x, 5) = initP._3 }
    }

    val initSQuat = SQuat(initQ._1, initQ._2, initQ._3, initQ._4)

    Sequential {
      particles(x) = Particle(
        math.log(1.0 / N),
        initSQuat,
        SVec3(0.0, 0.0, 0.0),
        initSQuat
      )
      Foreach(0::6, 0::6)((i,j) ⇒
        if (i == j)
          covs(x, i, i) = initCov
        else
          covs(x, i, j) = 0
      )
    }
  })
}


@virtualize def spatial() = {

  val inSIMU     = StreamIn[TSA](In1)
  val inV       = StreamIn[TSB](In2)
  val out       = StreamOut[TSR](Out1)

  val parFactor = 1 (1 → N)

  Accel {

    val sramBUFFER = SRAM[TSR](10)

    val particles = SRAM[Particle](N)
    val states = SRAM[SReal](N, 6)
    val covs = SRAM[SReal](N, 6, 6)
    val fifoSIMU   = FIFO[TSA](100)
    val fifoV     = FIFO[TSB](100)

    val lastSTime = Reg[STime](initTime)
    val lastO    = Reg[SOmega](SVec3(0.0, 0.0, 0.0))

    Sequential {

      initParticles(particles, states, covs, parFactor)

      tsas.foreach(x ⇒ Pipe {fifoSIMU.enq(x) } )
      tsbs.foreach(x ⇒ Pipe {fifoV.enq(x) } )

      Parallel {

        Stream(*)(x ⇒ {
          fifoV.enq(inV)
        })

        Stream(*)(x ⇒ {
          fifoSIMU.enq(inSIMU)
        })
```

```
        val choice = Reg[Int]
        val dt = Reg[SReal]
        FSM[Boolean, Boolean](true)(x => x)(x => {
          Sequential {
            if ((fifoV.empty && !fifoSIMU.empty) ||
                (!fifoSIMU.empty &&
                 !fifoV.empty &&
                 fifoSIMU.peek.t < fifoV.peek.t))
                 {
              choice := 0
              val imu = fifoSIMU.peek
              val t = imu.t
              lastO := imu.v.g
              dt := (t - lastSTime).to[SReal]
              lastSTime := t
            }
            else if (!fifoV.empty) {
              choice := 1
              val t = fifoV.peek.t
              dt := (t - lastSTime).to[SReal]
              lastSTime := t
            }
            else
              choice := -1

            if (choice.value != -1) {
              updateAtt(dt, lastO, particles, parFactor)
            }
            if (choice.value == 0) {
              val imu = fifoSIMU.deq()
              imuUpdate(imu.v.a, particles, parFactor)
            }
            if (choice.value != -1) {
              kalmanPredictParticle(dt, particles, states, covs, parFactor)
            }
            if (choice.value == 1) {
              val v = fifoV.deq()
              viconUpdate(v.v, dt, particles, states, covs, parFactor)
            }
            if (choice.value != -1) {
              normWeights(particles, parFactor)

              out := TSR(lastSTime,
                  averageSPOSE(particles, states, parFactor))
              resample(particles, states, covs, parFactor)
            }
          }
        })(x => true)

      }
    }
  }

  getMem(out).foreach(x => println(x))
}

def rotationMatrix(q: SQuat) =
  Matrix(3, 3, List(
    1.0 - 2.0 * (q.j ** 2 + q.k ** 2),
        2.0 * (q.i * q.j - q.k * q.r),
            2.0 * (q.i * q.k + q.j * q.r),
    2.0 * (q.i * q.j + q.k * q.r),
        1.0 - 2.0 * (q.i ** 2 + q.k ** 2),
            2.0 * (q.j * q.k - q.i * q.r),
    2.0 * (q.i * q.k - q.j * q.r),
        2.0 * (q.j * q.k + q.i * q.r),
            1.0 - 2.0 * (q.i ** 2 + q.j ** 2)
```

```
  ))

@virtualize
def updateAtt(
  dt: SReal,
  lastO: SOmega,
  particles: SRAM1[Particle],
  parFactor: Int
) = {
  Foreach(0::N par parFactor)(i ⇒ {
    val pp = particles(i)
    val nq =
      if (dt > 0.00001)
        sampleAtt(pp.q, lastO, dt)
      else
        pp.q
    particles(i) = Particle(pp.w, nq, pp.lastA, pp.lastQ)
  })
}

@virtualize
def kalmanPredictParticle(
  dt: SReal,
  particles: SRAM1[Particle],
  states: SRAM2[SReal],
  covs: SRAM3[SReal],
  parFactor: Int
) = {
  Foreach(0::N par parFactor)(i ⇒ {


    val X: Option[SReal] = None
    val Sdt: Option[SReal] = Some(dt)
    val S1: Option[SReal] = Some(1)

    val F =
      Matrix.sparse(6, 6, IndexedSeq[Option[SReal]](
        S1, X, X, X, X, X,
        X, S1, X, X, X, X,
        X, X, S1, X, X, X,
        Sdt, X, X, S1, X, X,
        X, Sdt, X, X, S1, X,
        X, X, Sdt, X, X, S1
      ))

    val pp = particles(i)

    val U = Matrix.sparse(6, 1, IndexedSeq[Option[SReal]](
      Some(pp.lastA.x * dt),
      Some(pp.lastA.y * dt),
      Some(pp.lastA.z * dt),
      X,
      X,
      X
    ))
    val rotMatrix = rotationMatrix(pp.lastQ)
    val covFixAcc = (rotMatrix * rotMatrix.t) * (covAcc * dt * dt)
    val Q = Matrix.sparse(6, 6, IndexedSeq[Option[SReal]](
      Some(covFixAcc(0, 0)), Some(covFixAcc(0, 1)),
          Some(covFixAcc(0, 2)), X, X, X,
      Some(covFixAcc(1, 0)), Some(covFixAcc(1, 1)),
          Some(covFixAcc(1, 2)), X, X, X,
      Some(covFixAcc(2, 0)), Some(covFixAcc(2, 1)),
          Some(covFixAcc(2, 2)), X, X, X,
      X, X, X, X, X, X,
      X, X, X, X, X, X,
      X, X, X, X, X, X
```

```
    ))

    val state = Matrix.fromSRAM1(6, states, i)
    val cov = Matrix.fromSRAM2(6, 6, covs, i)

    val (nx, nsig) = kalmanPredict(state, cov, F, U, Q)
    nx.loadTo(states, i)
    nsig.loadTo(covs, i)


  })
}

@virtualize
def imuUpdate(acc: SAcceleration, particles: SRAM1[Particle],
              parFactor: Int) = {
  Foreach(0::N par parFactor)(i => {
    val pp = particles(i)
    val na = pp.q.rotate(acc)
    particles(i) = Particle(pp.w, pp.q, na, pp.q)
  })
}

@virtualize
def viconUpdate(
  vicon: SPOSE,
  dt: SReal,
  particles: SRAM1[Particle],
  states: SRAM2[SReal],
  covs: SRAM3[SReal],
  parFactor: Int) = {

  val X: Option[SReal] = None
  val S1: Option[SReal] = Some(1)

  val h = Matrix.sparse(3, 6,
    IndexedSeq[Option[SReal]](
      X, X, X, S1, X, X,
      X, X, X, X, S1, X,
      X, X, X, X, X, S1
    ))

  val r = Matrix.eye(3, covViconP)

  val viconP = toMatrix(vicon.p)

  covViconQMat
  zeroVec

  Foreach(0::N par parFactor)(i => {


    val state = Matrix.fromSRAM1(6, states, i, true)
    val cov = Matrix.fromSRAM2(6, 6, covs, i, true)

    val (nx2, nsig2, lik) = kalmanUpdate(state, cov, viconP, h, r)
    nx2.loadTo(states, i)
    nsig2.loadTo(covs, i)

    val pp = particles(i)
    val nw               = likelihoodSPOSE(vicon, lik._1, pp.q, lik._2)
    particles(i) = Particle(pp.w + nw, pp.q, pp.lastA, pp.lastQ)
  })
}


lazy val covViconQMat = Matrix.eye(3, covViconQ)
```

```scala
lazy val zeroVec = Matrix(3, 1, List[SReal](0, 0, 0))
@virtualize def likelihoodSPOSE(measurement: SPOSE,
                                expectedPosMeasure: Matrix,
                                quatState: SQuat,
                                covPos: Matrix) = {
  val wPos   = unnormalizedGaussianLogPdf(toMatrix(measurement.p),
                                          expectedPosMeasure,
                                          covPos)
  val error  = quatToLocalAngle(measurement.q.rotateBy(quatState.inverse))
  val wSQuat = unnormalizedGaussianLogPdf(error, zeroVec, covViconQMat)
  wPos + wSQuat
}


def sampleAtt(q: SQuat, om: SOmega, dt: SReal): SQuat = {
  val withNoise  = gaussianVec(toVec(om), covGyro)
  val integrated = withNoise * dt
  val lq         = localAngleToQuat(integrated)
  lq.rotateBy(q)
}

@virtualize def gaussianVec(mean: Vec, variance: SReal) = {
  val reg = RegFile[SReal](3)
  //Real sequential
  Sequential.Foreach(0::2)(i => {
    val g1 = gaussian()
      reg(i*2) = g1._1
      if (i != 1)
        reg((i*2+1)) = g1._2
  })
  (Vec(3, RegId1(reg)) :* sqrt(variance)) :+ mean
}

//Box-Muller
//http://www.design.caltech.edu/erik/Misc/Gaussian.html
@virtualize def gaussian() = {

  val x1 = Reg[SReal]
  val x2 = Reg[SReal]
  val w  = Reg[SReal]
  val w2 = Reg[SReal]

  FSM[Boolean, Boolean](true)(x => x)(x => {
    x1 := 2.0 * random[SReal](1.0) - 1.0
    x2 := 2.0 * random[SReal](1.0) - 1.0
    w := (x1 * x1 + x2 * x2)
  })(x => w.value >= 1.0)

  w2 := sqrt((-2.0 * log(w.value)) / w)

  val y1 = x1 * w2;
  val y2 = x2 * w2;

  (y1, y2)
}


@virtualize def normWeights(particles: SRAM1[Particle], parFactor: Int) = {
  val maxR = Reduce(Reg[SReal])(0::N)(i => particles(i).w)(max(_,_))
  val totalWeight = Reduce(Reg[SReal])(0::N)
      (i => exp(particles(i).w - maxR))(_+_)
  val norm = maxR + log(totalWeight)
  Foreach(0::N par parFactor)(i => {
    val p = particles(i)
    particles(i) = Particle(p.w - norm, p.q, p.lastA, p.lastQ)
  })
}
```

```scala
@virtualize def resample(particles: SRAM1[Particle], states: SRAM2[SReal],
                         covs: SRAM3[SReal], parFactor: Int) = {

  val weights = SRAM[SReal](N)
  val out = SRAM[Particle](N)
  val outStates = SRAM[SReal](N, 6)
  val outCovs = SRAM[SReal](N, 6, 6)

  val u = random[SReal](1.0)

  Foreach(0::N)(i => {
    if (i == 0)
      weights(i) = exp(particles(i).w)
    else
      weights(i) = weights(i-1) + exp(particles(i).w)
  })

  val k = Reg[Int](0)
  Sequential.Foreach(0::N)(i => {
    def notDone = (weights(k) * N < i.to[SReal] + u) && k < N
    FSM[Boolean, Boolean](notDone)(x => x)(x => k := k + 1)(x => notDone)

    Foreach(0::6)(x => {
      outStates(i, x) = states(k, x)
    })
    Foreach(0::6, 0::6)( (y, x) => {
      outCovs(i, y, x) = covs(k, y, x)
    })

    out(i) = particles(k)
  })


  Foreach(0::N)(i => {
    val p = out(i)
    Foreach(0::6)(x => {
      states(i, x) = outStates(i, x)
    })
    Foreach(0::6, 0::6)( (y, x) => {
      covs(i, y, x) = outCovs(i, y, x)
    })
    particles(i) = Particle(log(1.0/N), p.q, p.lastA, p.lastQ)
  })
}



def unnormalizedGaussianLogPdf(measurement: Matrix, state: Matrix,
                               cov: Matrix): SReal = {
  val e = (measurement :- state)
  -1/2.0*((e.t*(cov.inv)*e).apply(0, 0))
}

def localAngleToQuat(v: Vec): SQuat = {
  val n    = (v*256).norm/256
  val l    = n / 2.0
  val sl   = sin(l)
  println(v(0) + " " + v(1) + " " + v(2) + "" + n + " " + sl)
  val nrot = v :* (sl / n)
  SQuat(cos(l), nrot(0), nrot(1), nrot(2))
}

def quatToLocalAngle(q: SQuat): Matrix = {
  val r: SReal = min(q.r, 1.0)
  val n = acos(r) * 2
  val s = n / sin(n / 2)
```

```
  Matrix(3, 1, List(q.i, q.j, q.k)) :* s
}

def kalmanPredict(xp: Matrix, sigp: Matrix,
                  f: Matrix, u: Matrix,
                  q: Matrix) = {
  val xm   = f * xp :+ u
  val sigm = (f * sigp * f.t) :+ q
  (xm, sigm)
}


def kalmanUpdate(xm: Matrix, sigm: Matrix,
                 z: Matrix, h: Matrix,
                 r: Matrix) = {
  val s   = (h * sigm * h.t) :+ r
  val k = sigm * h.t * s.inv
  val sig = sigm :- (k * s * k.t)
  val za = h * xm
  val x = xm :+ (k * (z :- za))
  (x, sig, (za, s))
}

@virtualize def averageSPOSE(particles: SRAM1[Particle],
                             states: SRAM2[SReal],
                             parFactor: Int): SPOSE = {
  val firstQ = particles(0).q
  val accumP = RegFile[SReal](3, List[SReal](0, 0, 0))
  val accumQ = Reg[SQuat](SQuat(1, 0, 0, 0))
  accumP.reset
  accumQ.reset
  Parallel {
    Foreach(0::N par parFactor, 0::3)((i,j) => {
      accumP(j) = accumP(j) + exp(particles(i).w) * states(i, j+3)
    })

    Reduce(accumQ)(0::N par parFactor)(i => {
      val p = particles(i)
      if (firstQ.dot(p.q) > 0.0)
        p.q * exp(p.w)
      else
        p.q * -(exp(p.w))
    })(_ + _)
  }
  SPOSE(SVec3(accumP(0), accumP(1), accumP(2)), accumQ)
}
```

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.

[6] D. Koeplinger, C. Delimitrou, R. Prabhakar, C. Kozyrakis, Y. Zhang, and K. Olukotun, "Automatic generation of efficient accelerators for reconfigurable hardware," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016, pp. 115–127.

[7] R. Prabhakar *et al.*, "Generating configurable hardware from parallel patterns," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 651–665, 2016.