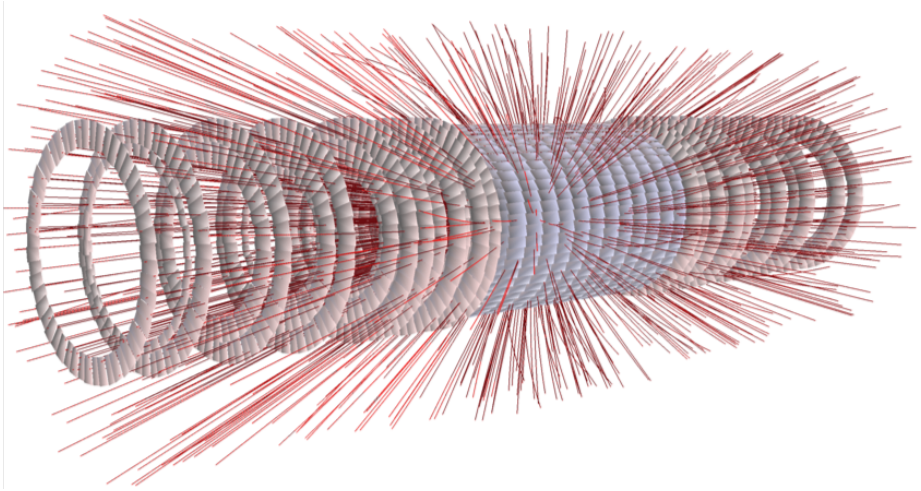
Kalman Filter in Rust - Challenge

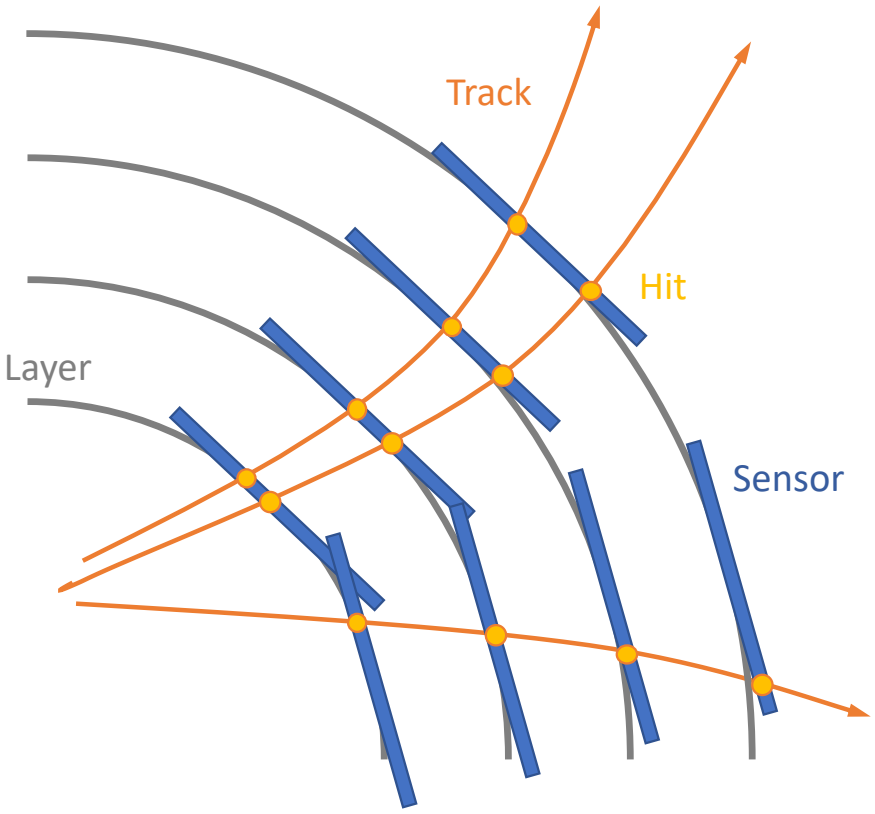
What do we use the Kalman Filter for?

In particle physics, we measure particles using sensors which register whenever they are hit by a particle. In the context of track reconstruction, we are interested in figuring out what the trajectory of a particle passing through the detector was. This is done by combining information from a number of these *hits*.

In many of the LHC experiments at CERN, the sensors that register particles are places something like this:

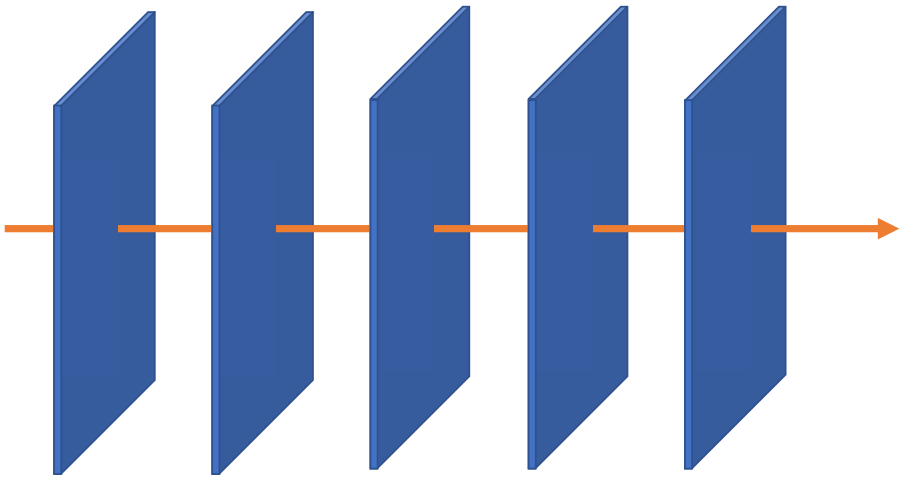


Looking at this from the side, particles passing through it would show up a bit like this:



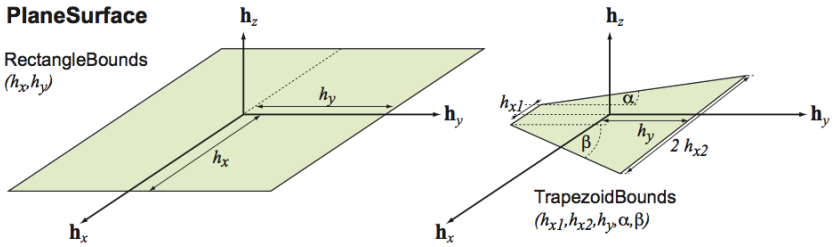
The goal of track reconstruction is to figure out where the solid black lines go, even in the presence of more than one of these *particle tracks* (as seen in the picture).

For this project, we will restrict the geometry to a simpler configuration: All the sensors will be planar surfaces, and they will all be parallel, as can be seen in the picture below.



Geometry description

The modules that make up the geometry have a local coordinate system and in this coordinate system they have certain *bounds*, which define the shape of the sensor. *Affine transformations* can be used to convert between the local coordinate system and the global one.



In the picture above, examples of the definition of the bounds in the local coordinate systems can be seen for rectangular and trapezoid shapes. This description of the geometry is the basic building block. It enables placing hits, which are always measured in the local coordinate frame of the sensor, in 3D space. This serves as the basis for further processing.

The challenge

The goal of the challenge is this:

Devise a description of planar surfaces in 3D space using Rust. This implementation should allow converting global 3D points into their corresponding

2D points on the modules. For this some way of defining vectors in 2D and 3D is required, as well as a way to store and apply transformation matrices on top those (translations and rotations). To enable conversion in both directions, there has to be a way to invert those matrices. You don't have to necessarily implement the linear algebra for this yourself, but feel free if you want to!

The implementation should also allow testing whether or not a given point is inside the active area of the module (using the concept of *bounds* mentioned above). For a basic implementation, rectangular bounds are sufficient, but keep in mind some form of extensibility or generic design.

You can have a look at some of the code in Acts' surfaces to get an idea of what an implementation in C++ might look like. Your implementation doesn't need to be identical.

The code will be judged both on correctness and on execution performance. Please also document your code, and write (unit-)tests!