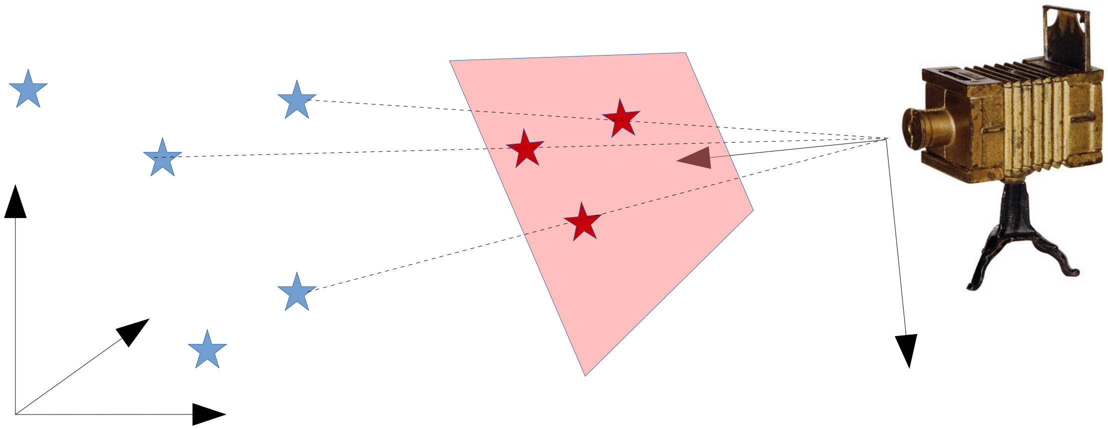
Probabilistic Robotics Course

Projective ICP [Exercise]

Lorenzo De Rebotti
derebotti@diag.uniroma1.it

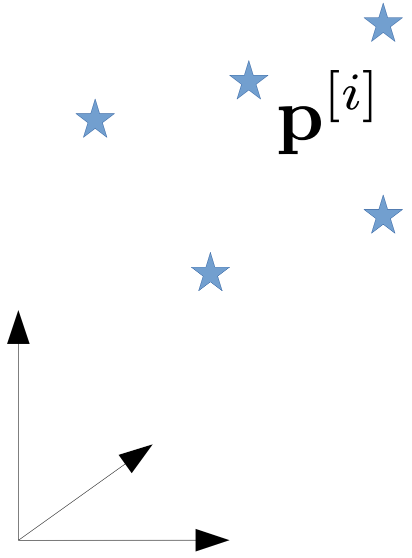
Department of Computer, Control, and Management Engineering
Sapienza University of Rome



Projective Registration

Given

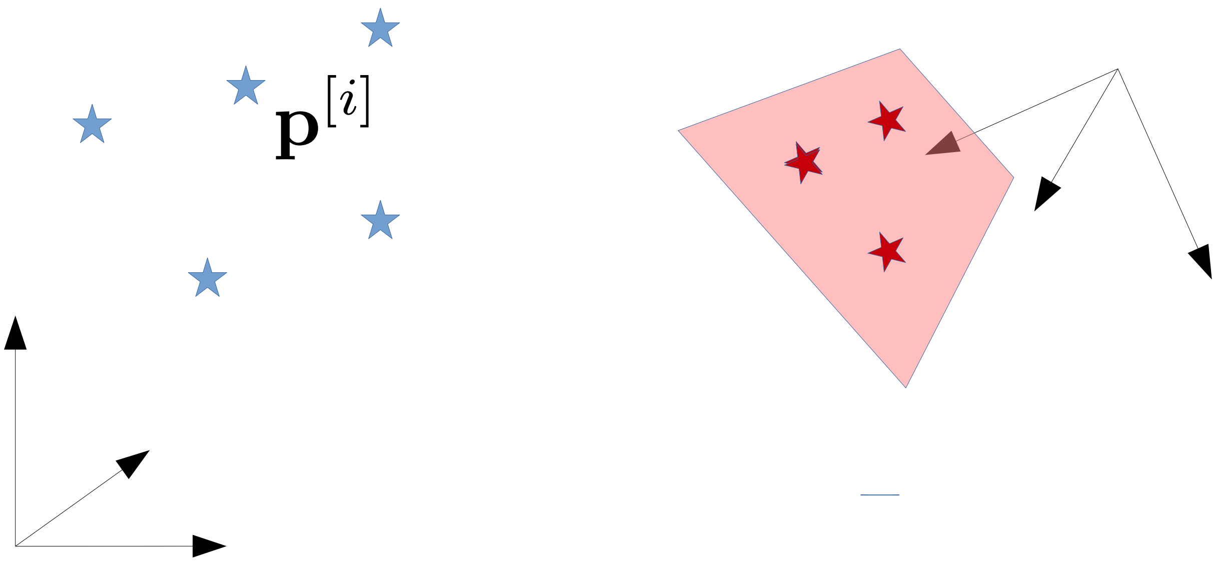
- a set of known 3D points in the world frame



Projective Registration

Given

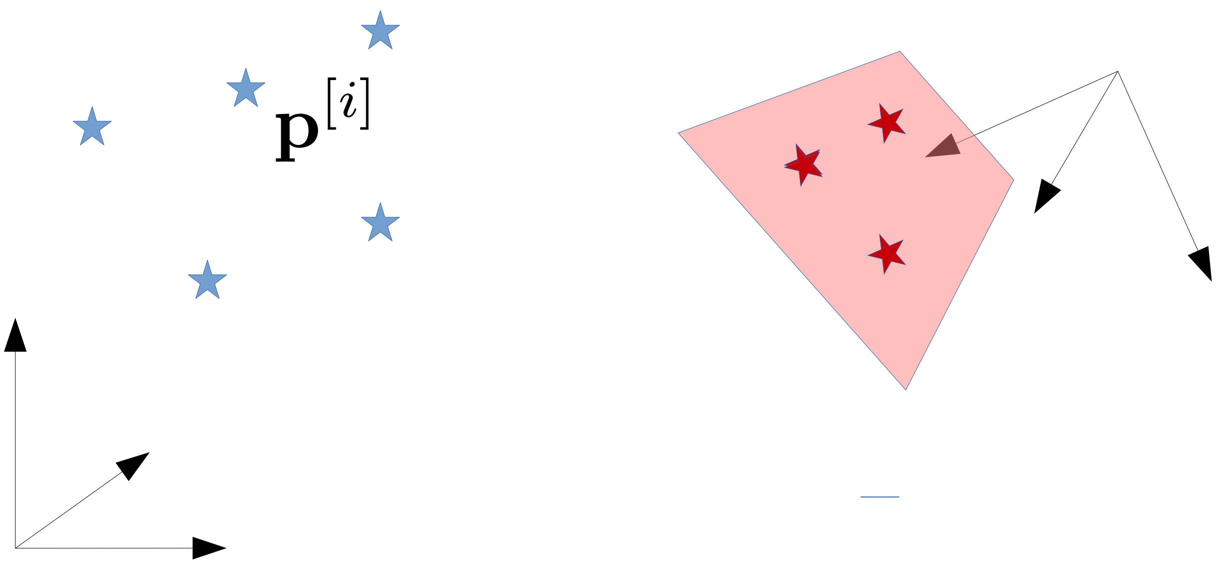
- a set of known 3D points in the world frame
- a set of 2D image projections of these points



Projective Registration

We want to find

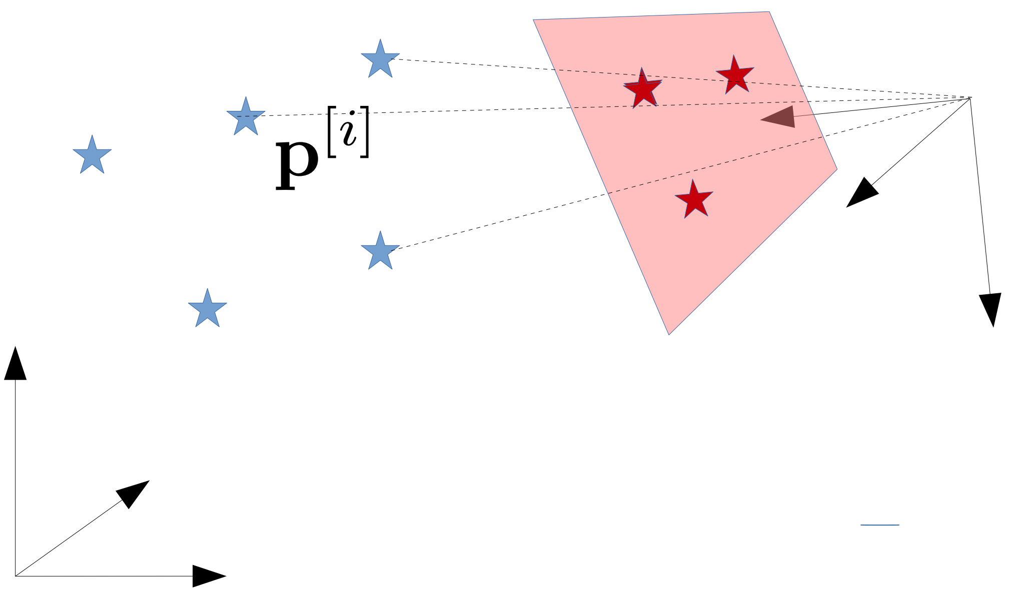
- The relative pose of the observer that minimizes the distance between prediction and measurement on the image plane



Projective Registration

We want to find

- The relative pose of the observer that minimizes the distance between prediction and measurement on the image plane



Data Association is Unknown

Outline

We will walk through a complete programming example that integrates:

- Pinhole camera model
- Data Association
- Least Squares on Manifolds

C++ implementation

How to Proceed?

We will follow these steps

- Construct a Simulation Environment
- Synthesis of the Sensor Model
- Data Association
- Least Squares
- Integrate the parts

After each step, we will do a validation

Simulation Environment

To check the components of the system, it is convenient to create a sandbox environment.

All what we need is a set of randomly drawn 3D points

To render the simulation more realistic, we will put these points along specific patterns in the space (e.g. segments).

This reflects the behavior that we could have for instance running an edge extractor on an image

Simulation Environment

Look at the file points_utils.{h,cpp}

There are some utilities to

- generate points in a world
- draw points on an opencv images

Design your program to be debugged, through visualization

Sensor Model (Pinhole Camera)

$$\mathbf{p}_{\mathrm{w}} = \begin{pmatrix} x_{\mathrm{w}} & y_{\mathrm{w}} & z_{\mathrm{w}} \end{pmatrix}^T$$

world point

$$\underbrace{\begin{pmatrix} u \\ v \end{pmatrix}}_{\mathbf{p}_{\mathrm{img}}} = \begin{pmatrix} \frac{f\,x_{\mathrm{w}}}{z_{\mathrm{w}}} + u_0 \\ \frac{f\,y_{\mathrm{w}}}{z_{\mathrm{w}}} + v_0 \end{pmatrix}$$

image point

point in camera coordinates

focal length

$$\underbrace{\begin{pmatrix} x_{\mathrm{cam}} \\ y_{\mathrm{cam}} \\ z_{\mathrm{cam}} \end{pmatrix}}_{\mathbf{p}_{\mathrm{cam}}} = \underbrace{\begin{pmatrix} f & 0 & u_0 \\ 0 & f & v_0 \\ 0 & 0 & 1 \end{pmatrix}}_{\mathbf{K}} \mathbf{p}_{\mathrm{w}}$$

principal point

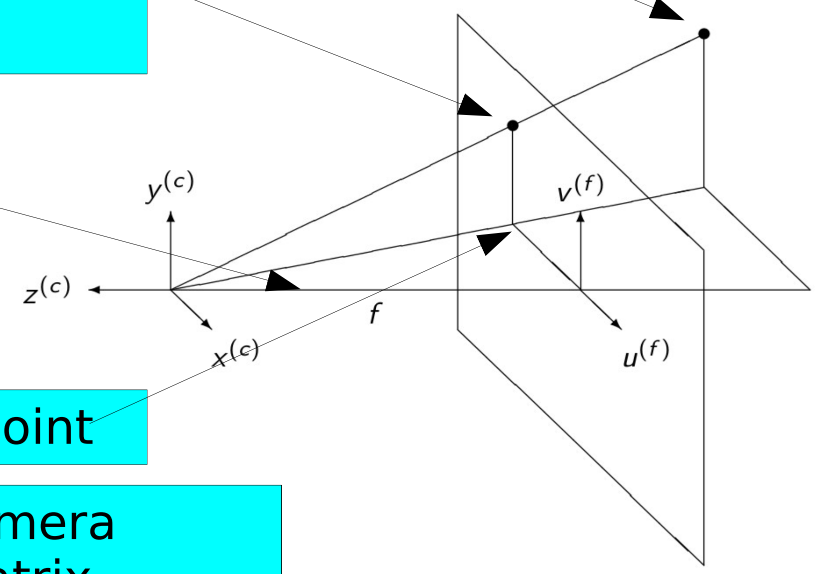
camera matrix

$$\mathbf{p}_{\mathrm{img}} = \mathrm{proj}(\mathbf{p}_{\mathrm{cam}}) = \begin{pmatrix} \frac{x_{\mathrm{cam}}}{z_{\mathrm{cam}}} \\ \frac{y_{\mathrm{cam}}}{z_{\mathrm{cam}}} \end{pmatrix}$$

$$= \mathrm{proj}(\mathbf{K}\mathbf{p}_{\mathrm{w}})$$

projection

image point expressed with camera matrix and projection



Sensor Model

If the reference system of world and camera are not the same, we can compute the projection as follows

$$\mathrm{p}_{\mathrm{img}} = \mathrm{proj}(\mathbf{K}\mathbf{X}\mathbf{p}_{\mathrm{w}})$$

world in camera transform

Implementing the Sensor Model

In a C++ implementation, it makes sense to design a class that will implement the camera functionalities.

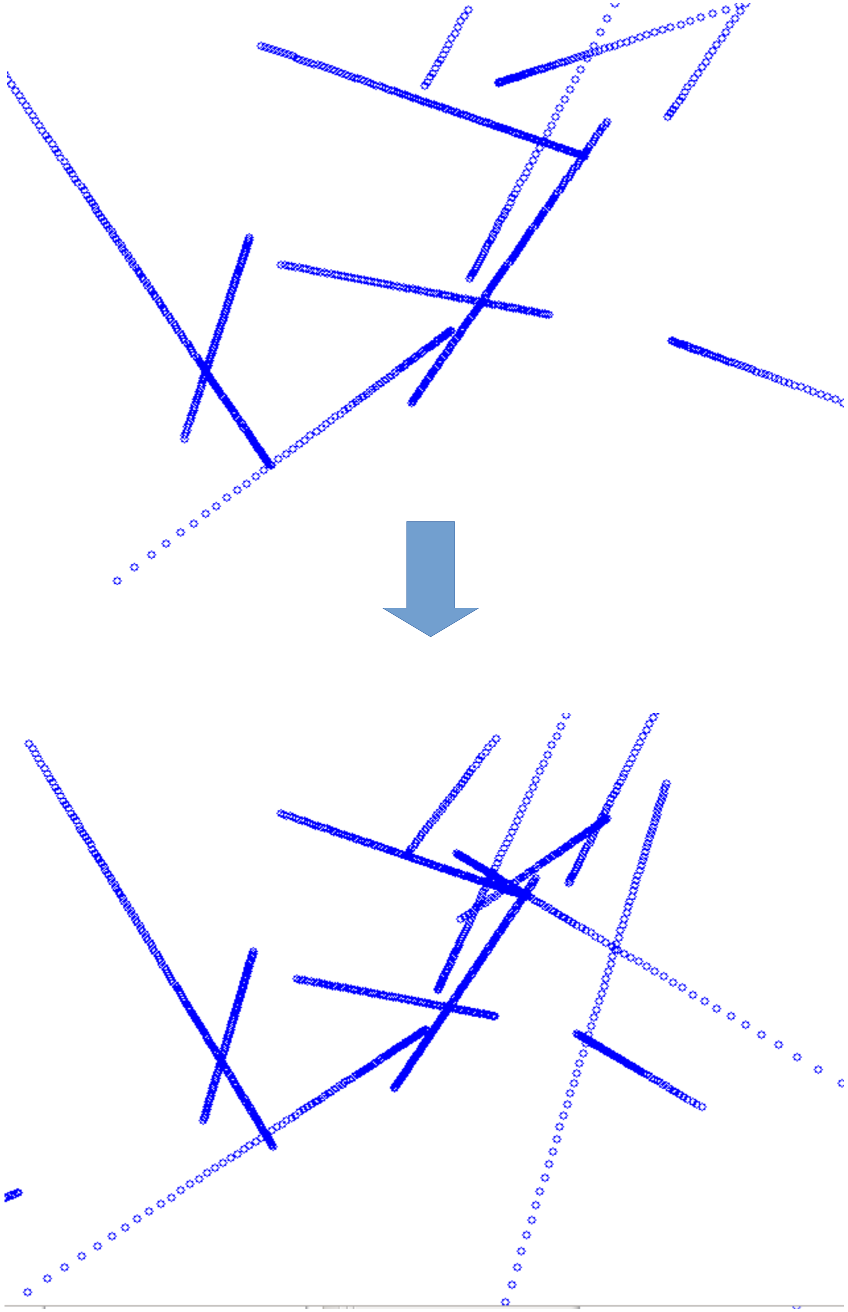
- Attributes:
  - pose of the world w.r.t. camera
  - camera matrix
- Functionalities
  - compute image coordinates of a world point, according to the attributes

see files camera.{h,cpp}

Testing Sensor Model

To check if everything is correct we want to

- construct a simulated environment with a bunch of 3d lines
- place a camera in the world
- move the camera with the arrow keys
- see the image perceived by the camera



Check the file camera_test.cpp, and run the corresponding binary

Data Association

If:

- we have a reasonable guess of the camera
- the points are sparse enough

we can approach the data association by a *nearest neighbor strategy*.

Given a transform **X** of world w.r.t. camera, we:

- project each point of the world onto the image
- we assign to each of these points the closest measured point on the image

Data Association

During one alignment, we will adjust the association based on our current guess.

A naive implementation requires O(N2) operations for each least squares iteration.

Considering that

- the measurements on the image do not change during one alignment
- the number of points is potentially large
- the image coordinates lie on a plane

a reasonable strategy is to use a distance map

Data Association

During one alignment, we will

- compute a distance map on the image, based on the measured points (once)
- at each iteration, we will assign to the measured point, the world point whose projection falls closer to it

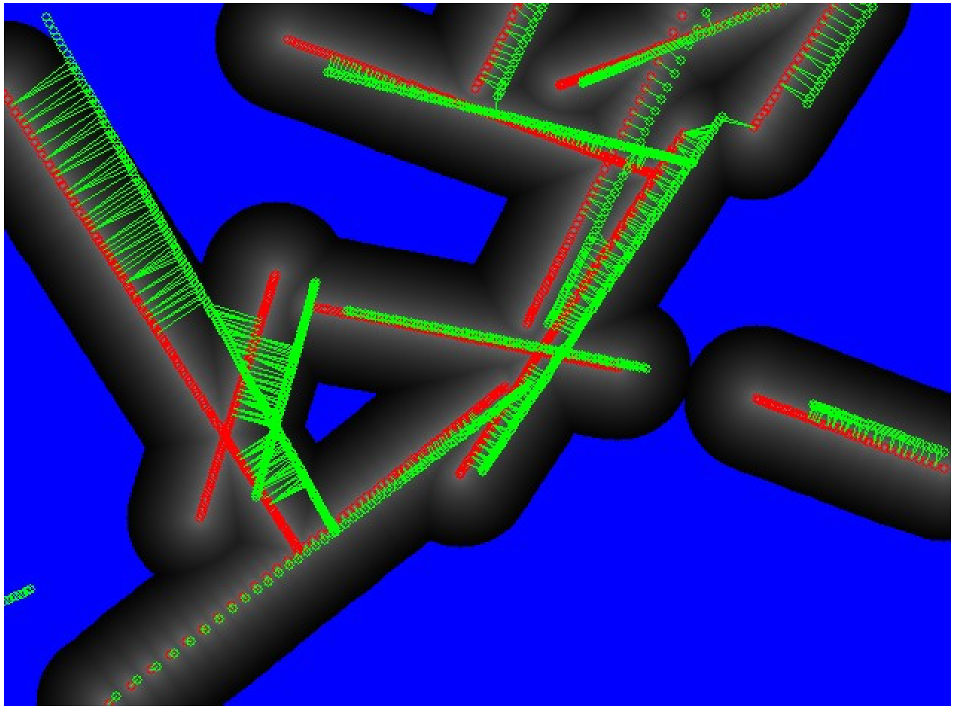
In a C++ implementation, this is encapsulated in a class DistanceMapCorrespondenceFinder

- Attributes:
  - image size
  - image points
  - max_distance
- Functionalities
  - initialization to compute the distance map
  - query for the correspondences

Data Association

To test the data association, we extend our camera_test program and show

- distance map
- computed associations



check out the file correspondence_finder_test.cpp

Least Squares

We will follow the methodology

- State, increments and boxplus

$$\begin{array}{rcc} \mathbf{X} \in SE(3) & : & \mathbf{X} = (\mathbf{R}|\mathbf{t}) \\ \mathbf{\Delta x} \in \Re^6 & : & \mathbf{\Delta x} = (\underbrace{x\ y\ z}_{\mathbf{t}}\ \underbrace{\alpha_x\ \alpha_y\ \alpha_z}_{\alpha})^T \\ \mathbf{X} \boxplus \mathbf{\Delta x} & : & v2t(\mathbf{\Delta x})\mathbf{X} \end{array}$$

- Measurements (Euclidean)

$$\mathrm{z}^{[m]} \in \Re^2 : \mathbf{z}^{[m]} = (u^{[m]}\ v^{[m]})^T$$

Least Squares

- Prediction

$$\mathbf{h}^{[n]}(\mathbf{X}) = \mathrm{proj}(\mathbf{K}\,\underbrace{\mathbf{X}\mathbf{p}^{[n]}}_{\mathbf{h}_{\mathrm{icp}}^{[n]}(\mathbf{X})})$$

$$\hat{\mathbf{p}}^{[n]} = \mathbf{h}_{\mathrm{icp}}^{[n]}(\mathbf{X}) = \mathbf{X}\mathbf{p}^{[n]}$$

$$\hat{\mathbf{p}}_{\mathrm{cam}}^{[n]} = \mathbf{K}\hat{\mathbf{p}}^{[n]}$$

This is the prediction for 3D ICP we found in the previous lesson

camera coordinates (before homogeneous division)

Least Squares

- Error

$$
\begin{aligned}
\mathbf{e}^{[n,m]}(\mathbf{X}) &= \mathbf{h}^{[n]}(\mathbf{X}) - \mathbf{z}^{[m]} \\
&= \mathrm{proj}(\mathbf{K}\,\underbrace{\mathbf{X}\mathbf{p}^{[n]}}_{\mathbf{h}_{\mathrm{icp}}^{[n]}(\mathbf{X})}) - \mathbf{z}^{[m]} \\
\mathbf{e}^{[n,m]}(\mathbf{X} \boxplus \mathbf{\Delta x}) &= \mathrm{proj}(\mathbf{K}\mathbf{h}_{\mathrm{icp}}^{[n]}(\mathbf{X} \boxplus \mathbf{\Delta x})) - \mathbf{z}^{[m]}
\end{aligned}
$$

Least Squares

- Jacobian (using chain rule)

$$
\begin{aligned}
\underbrace{\left.\frac{\partial \mathbf{e}^{[n,m]}(\mathbf{X} \boxplus \mathbf{\Delta x})}{\partial \mathbf{\Delta x}}\right|_{\mathbf{\Delta x}=\mathbf{0}}}_{\mathbf{J}^{[n]}} &= \left.\underbrace{\frac{\partial \mathrm{proj}(\mathbf{p})}{\partial \mathbf{p}}}_{\mathbf{J}_{\mathrm{proj}}(\mathbf{p})}\right|_{\mathbf{p}=\hat{\mathbf{p}}^{[n]}_{\mathrm{cam}}} \mathbf{K} \underbrace{\left.\frac{\partial \mathbf{h}^{[n]}_{\mathrm{icp}}(\mathbf{X} \boxplus \mathbf{\Delta x})}{\partial \mathbf{\Delta x}}\right|_{\mathbf{\Delta x}=\mathbf{0}}}_{\mathbf{J}^{[n]}_{\mathrm{icp}}} \\
&= \mathbf{J}_{\mathrm{proj}}\left(\hat{\mathbf{p}}^{[n]}_{\mathrm{cam}}\right) \mathbf{K} \mathbf{J}^{[n]}_{\mathrm{icp}} \\
\mathbf{J}^{[n]}_{\mathrm{icp}} &= \left(\mathbf{I}_{3\times 3} \middle| \lfloor -\hat{\mathbf{p}}^{[n]} \rfloor_{\times}\right) \\
\mathbf{J}_{\mathrm{proj}}(\mathbf{p}) &= \begin{pmatrix} \frac{1}{z} & 0 & -\frac{x}{z^2} \\ 0 & \frac{1}{z} & -\frac{y}{z^2} \end{pmatrix}
\end{aligned}
$$

Least Squares

In C++ we will write a class that implements the least squares solver

Attributes

- world_points
- image_points
- current camera pose
- kernel_threshold

Functionalities

- initialization
- one iteration(given the correspondences)

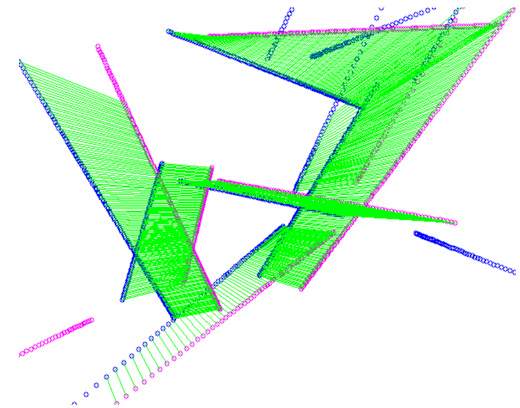
Check out the files picp_solver.* for more details

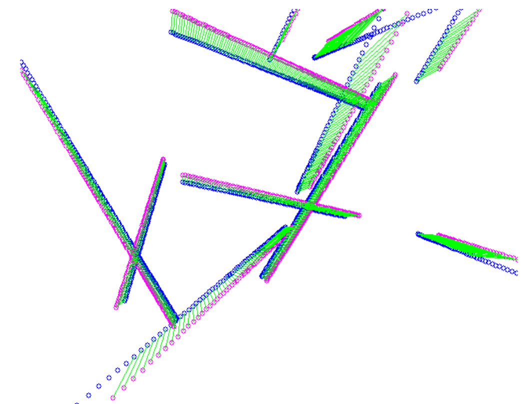
Least Squares

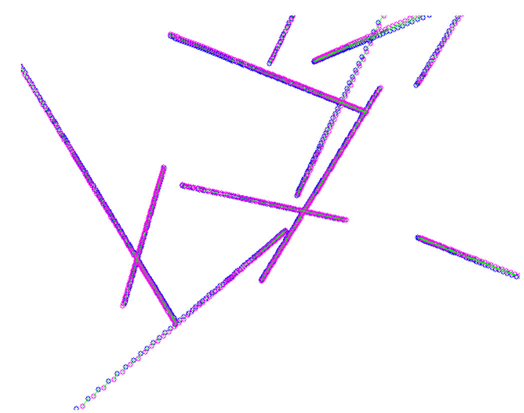
To test our least squares solution, we need to define a problem (with perfect associations) and let the algorithm run

We will modify the camera_test, to get picp_solver_test

- arrow keys; move the camera
- spacebar: performs one least squares iteration







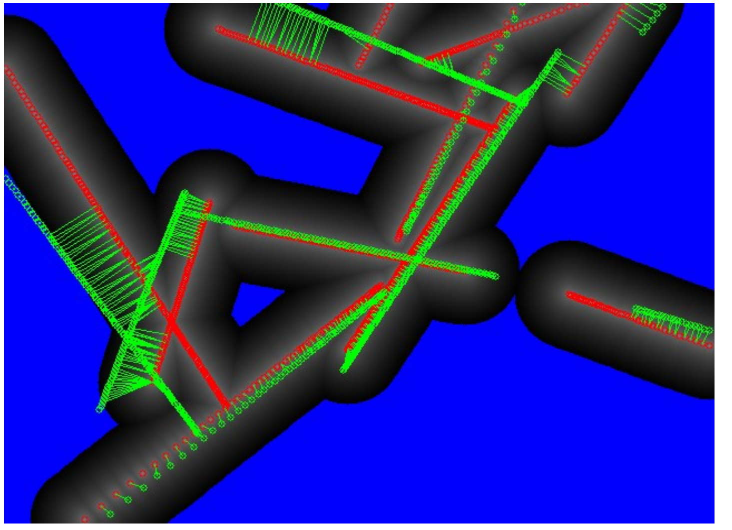
Least Squares (Hints)

When implementing the least squares part

- validate first the error function (if the system is at the optimum, the error should be 0)
- compute first the numeric jacobians (or use autodiff)
- only if needed (by speed) compute the analytic jacobians

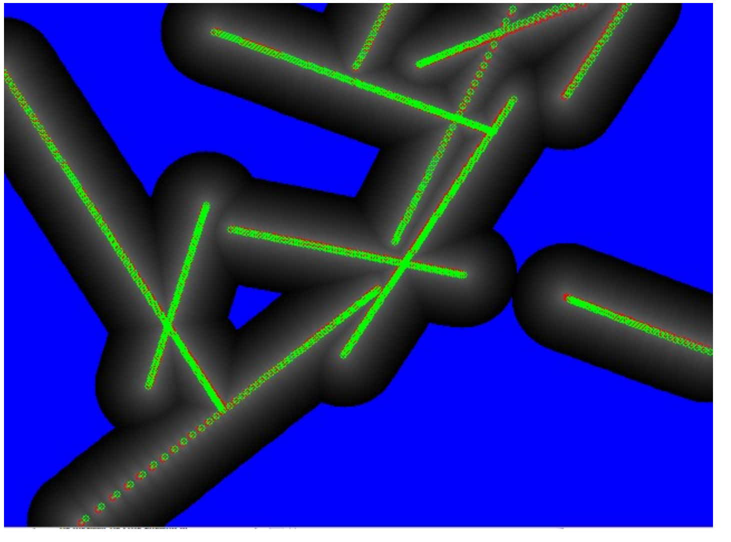
Putting together the parts

Once we have validated each single aspect we proceed to the integration



See file "picp_complete_test.cpp" and the corresponding binary

- arrow keys; move the camera
- spacebar: performs one association+least squares round



Conclusions

- Divide a non-trivial problem in smaller tasks that you can solve easily

- Sandbox the individual components of your system in order to validate the behaviors (e.g. by using a simulator)