Multiview ICP

Adrian Haarbach

August 2015

The goal of this project was to solve the multiview rigid point cloud registration problem using multiview levenberg-marquardt ICP implemented with Ceres solver.

1 ICP

The Iterative Closest Points algorithm (ICP) [1, 2, 3] is a two-step iterative algorithm to rigidly align two point clouds. The following two steps are applied in alternation until convergence:

1. compute closest point **correspondences** $\{p_i \to q_i\}_1^N$ between the two clouds
2. update the current **transformation** estimate $g \in SE(3)$ so that it minimizes a cost function defined on these correspondences.

In the following, we concentrate on step 2, meaning the update of the transformation estimate while leaving the correspondences fixed, and extend it to the multiview setting.

2 Pairwise registration

Let $p_i \leftrightarrow q_i (p_i, q_i \in \mathbb{R}^3)$ be a set of point correspondences. The goal is to find a rigid body motion $g \in SE(3)$ that minimizes the following error:

$$E(g) = \sum_{i=1}^{N} l(||d(g(p_i), q_i)||^2) \tag{1}$$

Where d is a CostFunction while l is a LossFunction. The solution $g = argmin(E)$ is the relative transformation that aligns the model cloud (the p_i) to the data cloud (the q_i) in a least squares sense.

3 Multiview registration

In the multiview setting, the roles of model and data cloud are no longer fixed, since one cloud can take on the roles of both of them. Let $C_1, \dots C_M$ be the set of point clouds that are brought to be in alignment. To generalize and formalize the notation of which cloud gets registered to which other cloud, we can encode these relations as a directed graph, with the adjacency matrix $A \in \{0,1\}^{M \times M}$, such that $A(h,k) = 1$ if cloud C_h can be registered to cloud C_k. Let $g_1, \dots g_M$ be the absolut camera poses of each view in the global reference frame. The alignment error between two clouds C_h and C_k then is:

$$E(g_h, g_k) = A(h,k) \sum_{i=1}^{N_h} l(||d(g_h(p_i^h), g_k(q_i^h))||^2) \tag{2}$$

where $\{p_i^h \rightarrow q_i^h\}$ are the N_h closest point correspondences obtained from the clouds C_h and C_k. The pairwise formulation (1) can be obtained by setting $g = g_k * g_h^{-1}$. The overall alignment error, which we want to minimize at this stage, is obtained by summing up the contribution of every pair of overlapping views:

$$E(g_1, ..., g_M) = \sum_{h=1}^{M} \sum_{k=1}^{M} A(h,k) \sum_{i=1}^{N_h} l(||d(g_h(p_i^h), g_k(q_i^h))||^2) \tag{3}$$

The solutions $g_1, ..., g_M = argmin(E)$ are the absolute camera poses that align the M clouds in a least squares sense. In contrast to the pairwise registration error (1), which has closed form solutions for the relative transformation g that aligns the two clouds, there are no closed form solutions in the multiview setting. However, rigid point cloud registration is just an instance of Non-linear least squares optimization, which can be solved efficiently using Ceres Solver [1].

3.1 Common ICP Cost Functions

3.1.1 Pairwise

The most prominent cost functions one minimizes using closed form solutions in the pairwise setting are:

$$E(g) = \sum_{i=1}^{N} ||R \cdot p_i + t - q_i||^2 \qquad \text{point to point [1, 3]} \tag{4}$$

[1] `http://ceres-solver.org/nnls_tutorial.html`

$$E(g) = \sum_{i=1}^{N} ||(R \cdot p_i + t - q_i)^T n_{q_i}||^2 \quad \text{point to plane [2]} \tag{5}$$

where $R \in SO(3), t \in \mathbb{R}^3$, so $g = (R, t) \in SE(3)$. In these cases, the loss function l is just the identity, and the cost functions d are

$$d_{point-point}(g, p, q) = R \cdot p + t - q \in \mathbb{R}^3 \tag{6}$$

$$d_{point-plane}(g, p, q, n_q) = (R \cdot p + t - q)^T n_q \in \mathbb{R}^1 \tag{7}$$

Even though there are closed form solutions in the pairwise setting for the above cost functions [4, 5], one can also use Ceres to solve the pairwise error (1) yielding a relative pose update g. Note that the evaluations of the cost functions, called the residuals, don't have to be in $\mathbb{R}^3$.

3.1.2 Multiview

The multiview setting needs only a slight modification of above cost functions. In the pairwise setting, d depends on the relative pose g. In the multiview setting, d depends on the two absolute poses $g_h = (R_h, t_h)$ and $g_k = (R_k, t_k)$.

$$d_{point-point}(g_h, g_k, p, q) = (R_h \cdot p + t_h) - (R_k \cdot q + t_k) \tag{8}$$

$$d_{point-plane}(g_h, g_k, p, q, n_q) = d_{point-point}(g_h, g_k, p, q)^T (R_k \cdot n_q) \tag{9}$$

With this slight modification of the cost functions, we can model (3) using Ceres and let it solve for $g_1, ..., g_M$. Notice that in the point-to plane case, the normals of the points in the destination cloud are only rotated, not translated, since they represent directions rather than actual points.

4 Parametrization of rigid body motions

In the above, we just wrote $g = (R, t)$ and especially assumed R to be a 3x3 rotation matrix $R \in SO(3)$, which must fulfill $RR^T = I$ and $|R| = 1$. If used in an iterative optimization algorithm, one has to optimize over 9 parameters constrained to above orthogonality conditions, while only having 3 degrees of freedom. It is much more efficient to optimize over minimal representations of rotations such as AngleAxis, Unit Quaternions or the Lie Algebra representation.

4.1 Angle Axis

According to Euler's rotation theorem, any rotation in 3D space can be expressed as a single rotation around some axis by a certain angle. The angle-axis (or axis-angle) representation of a rotation parametrizes a rotation by a unit vector $n \in \mathbb{R}^3$ as the axis of the rotation and an angle ϕ describing the magnitude of the rotation around the axis.[2]

Furthermore, the angle of rotation θ can be absorbed into the norm of the unit axis vector n, so that we only have 3 parameters for the 3 DoF: $\omega = n * \theta$. In the theory of three-dimensional rotation, Rodrigues' rotation formula, named after Olinde Rodrigues, is an efficient algorithm for rotating a vector in space, given an axis and angle of rotation.[3]

Ceres AngleAxisRotatePoint function[4] uses Rodrigues rotation formula to directly rotate a point. One first has to recover the angle and the unit axis from the minimal representation, the vector $\omega \in \mathbb{R}^3$, which represents rotational velocity, by $\phi = ||\omega||, n = \frac{\omega}{\phi}$.

$$p_{rot} = p\cos\phi + (n \times p)\sin\phi + n(n^T p)(1 - \cos\phi) \tag{10}$$

Another way is to first convert from the Angle-Axis formulation to a rotation matrix R, and then rotate the point by matrix - vector multiplication. This second form of the Rodrigues rotation formula can be derived from the first one.

$$R = I + \sin\phi[n]_x + (1 - \cos\phi)[n]_x^2$$

This is closely related to the Lie Algebra representation, where we also define the $[.]_x$ cross product matrix operator (12).

4.1.1 Cost function

With the parametrization of g as (w, t) and $w, t \in \mathbb{R}^3$, we just replace all occurrences of $R \cdot p$ (or q) in the cost functions (6), (7), (8), (9) with p_{rot}. The translational component is still represented as a 3-dimensional vector.

4.1.2 Jacobian

We rely on Ceres AutoDiffCostFunction functionality to automatically compute the necesarry derivatives of our Cost function parametrized by the angle axis representation and a translation vector.

---

2 https://en.wikipedia.org/wiki/Axis%E2%80%93angle_representation

3 https://en.wikipedia.org/wiki/Rodrigues%27_rotation_formula

4 https://github.com/ceres-solver/ceres-solver/blob/master/include/ceres/rotation.h#L566

4.2 Unit Quaternions

Yet another way to represent rotations are unit quaternions.

A quaternion q is basically the extension of a complex number $c = a + bi, i^2 = -1$ from 2 to 4 dimensions: $q = w + xi + yj + zk, i^2 = j^2 = k^2 = ijk = -1$. Just as complex numbers can be used to represent rotations in $\mathbb{R}^2$ (remember Euler's formula $\mathrm{e}^{\mathrm{i}\,\varphi} = \cos(\varphi) + \mathrm{i}\,\sin(\varphi)$), quaternions can be used to represent rotations in $\mathbb{R}^3$.

Formally, a quaternion $q \in \mathbb{H}$ may be represented by a vector $q = [q_w, q_x, q_y, q_z]^T = [q_w, q_{x:z}]^T$ together with the definitions:

$$\begin{aligned} \text{adjoint}: &\quad \bar{q} = [q_w, -q_{x:z}]^T \\ \text{norm}: &\quad ||q|| = \sqrt{q_w^2 + q_x^2 + q_y^2 + q_z^2} \\ \text{inverse}: &\quad q^{-1} = \frac{\bar{q}}{||q||} \end{aligned}$$

A unit quaternion is a quaternion with unity norm, $||q|| = 1$ and can be used to represent the orientation of a rigid body in 3D Euclidean space. Specifically, a unit quaternion can be retrieved from the axis-angle representation, with a rotation ϕ about the normalized rotation axis $n, ||n|| = 1$ via

$$q(\phi, n) = [\cos(0.5\phi), n\sin(0.5\phi)]^T$$

Moreover, there are closed form solutions for converting a unit quaternion into a rotation matrix as well as the other way around which we will use extensively. Since they don't look as neat as the previous formula, we refer to [6] for the details.

Even though Ceres comes with an implementation of unit quaternions in the same file as AngleAxisRotatePoint, we instead used Eigen's quaternions [5], which allows for much nicer syntax due to operator overloading. It is important to note that the storage order of the two differ: it is $[x, y, z, w]$ for eigen quaternions, but $[w, x, y, z]$ for the ones built into ceres.

4.2.1 Local Parametrization

Unit Quaternions are not a minimal representation, they have 4 components. Nevertheless, the 4th component, typically the scalar part w, can be recovered from the other three because the whole 4-vector must have unit norm.

This is why Ceres gives you a 3-dimensional incremental update step δ that needs to be applied to the current 4-dimensional unit quaternion. This

[5] `http://eigen.tuxfamily.org/dox/classEigen_1_1Quaternion.html`

has to be done in a class extending Local Parametrization which must implement the Plus function. For Ceres in-built quaternions, this is implemented in the QuaternionParameterization::Plus function. For our Eigen Quaternion, we copied the Plus function but adapted/permuted the indices to the different storage order:

$$Plus(q, \delta) = [sin(|\delta|)\delta/|\delta|, cos(|\delta|)] * q$$

with * being the quaternion multiplication operator. Here we assume that the last element of the quaternion vector is the real (cos theta) part ($[x, y, z, w]$ Eigen quaternion storage order).

Since we did not yet implement automatic differentiation of this local parameterization (as for our Lie Algebra using the Sophus Library), one also has to provide the correct Jacobian. For Eigen Quaternions storage order they look as follows:

$$J = \frac{\partial q}{\partial \delta} = \begin{bmatrix} w & z & -y \\ -z & w & x \\ y & -x & w \\ -x & -y & -z \end{bmatrix} \tag{11}$$

4.2.2 Cost function

With the parametrization of g as (q, t) and $q \in \mathbb{R}^4, t \in \mathbb{R}^3$, we just replace all occurrences of $R \cdot p$ (or q) in the cost functions (6), (7), (8), (9) with either a direct quaternion rotation (Eigen storage order $p = [x, y, z, w]$):

$$[p_{rot}, 0]^T = q \cdot [p, 0] \cdot q^{-1}$$

or by first converting the quaternion to a Rotation matrix R as in [6]. The translational component is still represented as a 3-dimensional vector.

4.2.3 Jacobian

In Ceres, it is possible to mix the analytic derivatives for our Local Parametrization as stated above with automatic derivatives for the complete cost function. This drastically lowered the complexity, while still providing decent efficiency, since we were able to use only well tested analytic Jacobians for the Local Parametrization, but did not have to compute analytic Jacobians for the complete cost function.

4.3 Lie Algebra of Twists

Each rigid body transformation matrix T in the Lie group $T \in \mathrm{SE}(3)$ has a minimal representation as a twist $\hat{\xi}$ in its associated Lie algebra $\hat{\xi} \in se(3)$. Each twist is uniquely defined by its twist coordinates $\xi \in \mathbb{R}^6$:

$$\xi = (\xi_1, \ldots, \xi_6)^T = (u_1, u_2, u_3, \omega_1, \omega_2, \omega_3)^T = (u^T \omega^T)^T$$

where u represents the translational velocity and ω the rotational velocity.

Let us first define the operator $[.]_x$, which is an isomorphism between $\mathbb{R}^3$ and the space $so(3)$ of all 3x3 skew symmetric matrices ($[\omega]_x = -[\omega]_x^T$).

$$[.]_x : \mathbb{R}^3 \to so(3) \subset \mathbb{R}^{3\times 3}; [\omega]_x = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \tag{12}$$

It allows to express the cross product $p \times q$ with $p, q \in \mathbb{R}^3$ as a matrix-vector multiplication: $p \times q = [p]_x * q$. Furthermore, it allows to convert from twist coordinates to a twist using the hat operator:

$$\wedge : \mathbb{R}^6 \to se(3) \subset \mathbb{R}^{4\times 4}; \hat{\xi} = \xi^\wedge = \begin{bmatrix} u \\ \omega \end{bmatrix}^\wedge = \begin{bmatrix} [\omega]_x & u \\ 0 & 0 \end{bmatrix}$$

The mapping from the twist in the Lie algebra to the transformation matrix in the Lie group is done by matrix exponentiation:

$$\begin{gathered} exp : se(3) \to SE(3) \\ exp(\hat{\xi}) = exp\left(\begin{bmatrix} [\omega]_x & u \\ 0 & 0 \end{bmatrix}\right) = \begin{bmatrix} e^{[\omega]_x} & Vu \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \end{gathered} \tag{13}$$

We can convert the vector $\omega \in \mathbb{R}^3$, which represents rotational velocity, into an axis angle representation by $\phi = ||\omega||, n = \frac{\omega}{\phi}$, to obtain a closed form solution for the Taylor series expansion $e^{[w]_x} = \sum_{i=0}^{\infty} \frac{[\omega]_x^i}{!i}$ using Rodrigues' rotation formula:

$$e^{[w]_x} = e^{[n]_x \phi} = I + sin\phi[n]_x + (1 - cos\phi)[n]_x^2$$

and similarly for V

$$V = I + \frac{1 - cos\phi}{\phi}[n]_x + \frac{\phi - sin\phi}{\phi}[n]_x^2$$

To get from twist coordinates $\xi \in \mathbb{R}^6$ to a transformation matrix $T \in SE(3) \subset \mathbb{R}^{4\times 4}$ one first has to apply the hat operator $\wedge$ and then the exponential map exp. The other direction is possible as well, by first applying the

inverse of the exponential map, called the logarithmic map log, and then the inverse of the hat operator, called the vee operator $\vee$. For our application however, we only need the forward direction, which is why just summarize the different operators here:

$$\xi \in \mathbb{R}^6 \underset{\vee(vee)}{\overset{\wedge(hat)}{\rightleftarrows}} \hat{\xi} \in se(3) \underset{log}{\overset{exp}{\rightleftarrows}} T \in SE(3)$$

Every Lie group, such as SO(3) and SE(3), is a group that is also a smooth manifold, with the property that the group operations of multiplication and inversion are smooth maps. They can locally be approximated by their corresponding Lie algebras $so(3)$ and $se(3)$, which form the tangent space of the group at the identity. This allows one to do calculus on the elements of the Lie algebra, such as calculating derivatives, which we will need for numerical minimization.

4.3.1 Cost function

With the parametrization of g by its twist coordinates $\xi \in \mathbb{R}^6$, we have a truly minimal representation of both rotation and translation in just one 6 dimensional vector.

In the Lie algebra, a point $p \in \mathbb{R}^3$ is transformed (rotated and translated) to the point y by a twist via [7][p.32] :

$$\begin{bmatrix} y \\ 0 \end{bmatrix} = \begin{bmatrix} [\omega]_x & u \\ 0 & 0 \end{bmatrix} \begin{bmatrix} p \\ 1 \end{bmatrix} = \begin{bmatrix} & -\omega_3 p_2 & +\omega_2 p_3 & +u_1 \\ \omega_3 p_1 & & -\omega_1 p_3 & +u_2 \\ -\omega_2 p_1 & +\omega_1 p_2 & & +u_3 \\ & & 0 & \end{bmatrix} \Leftrightarrow y = w \times p + u$$

4.3.2 Jacobian

The Jacobian of the twist $\hat{\xi}$ is calculated by partial derivates of $y = [y_1, y_2, y_3]^T$ and the twist coordinates $\xi = (u_1, u_2, u_3, \omega_1, \omega_2, \omega_3)^T$ [8] :

$$J = \frac{\partial y}{\partial \xi} = \begin{bmatrix} \frac{\partial y_1}{\partial \xi_1} & \cdots & \cdots & \frac{\partial y_1}{\partial \xi_6} \\ \vdots & & & \vdots \\ \frac{\partial y_3}{\partial \xi_1} & \cdots & \cdots & \frac{\partial y_3}{\partial \xi_6} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & p_3 & -p_2 \\ 0 & 1 & 0 & -p_3 & 0 & p_1 \\ 0 & 0 & 1 & p_2 & -p_1 & 0 \end{bmatrix} \tag{14}$$

References

[1] Paul J Besl and Neil D McKay. Method for registration of 3-d shapes. In *Robotics-DL tentative*, pages 586–606. International Society for Optics and Photonics, 1992.

[2] Yang Chen and Gérard Medioni. Object modeling by registration of multiple range images. In *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pages 2724–2729. IEEE, 1991.

[3] Zhengyou Zhang. Iterative point matching for registration of free-form curves and surfaces. *International journal of computer vision*, 13(2):119–152, 1994.

[4] David W Eggert, Adele Lorusso, and Robert B Fisher. Estimating 3-d rigid body transformations: a comparison of four major algorithms. *Machine Vision and Applications*, 9(5-6):272–290, 1997.

[5] Kok-Lim Low. Linear least-squares optimization for point-to-plane icp surface registration. *Chapel Hill, University of North Carolina*, 2004.

[6] James Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors. *Matrix*, 58:15–16, 2006.

[7] Y. Ma, S. Soatto, J. Kosecká, and S.S. Sastry. *An Invitation to 3-D Vision: From Images to Geometric Models*. Interdisciplinary Applied Mathematics. Springer New York, 2005.

[8] Miroslava Slavcheva. Unified pipeline for 3d reconstruction from rgb-d images using coloured truncated signed distance fields. Master's thesis, Technische Universität München.