Rotation Estimation in Geometric Algebra

Mauricio Cele Lopez Belon

Abstract. We present a fast estimator of the rotation aligning two sets of corresponding vectors. Our method is faster than most conventional methods reported in literature, moreover it is robust to noise, accurate and simple to implement. It is based on minimizing a novel linear least squares error formulated in geometric algebra.

Mathematics Subject Classification (2010). Parallel algorithms 68W10; Clifford algebras, spinors 15A66.

Keywords. Geometric Algebra, Rotation Estimation, Wahba Problem.

The optimal rotation estimation from two sets of corresponding 3D vectors is a well known problem widely studied in linear algebra and quaternion algebra. On the other hand few work has been done on studying this problem using geometric algebra. Geometric algebra rotors are closely related to quaternions (quaternion algebra can be regarded as a geometric algebra defined on a set of imaginary basis vectors) we find geometric algebra to be a more natural choice for studying this problem since it is defined over a Euclidean vector space $\mathbb{R}^3$.

In this paper we introduce an efficient algorithm for estimating the best rotation aligning two sets of corresponding vectors. Our algorithm is faster than most conventional algorithms reported in literature. Our experiments demonstrate that it is robust to noise and accurate. Moreover, it is easy to implement. In fact, its implementation does not require access to any geometric algebra library, it can be implemented using any standard matrix and quaternion library. Our algorithm is based on minimizing a novel linear least squares error formulated in geometric algebra. Given the prevalence of quaternions in literature it is somewhat surprising to us that direct solution of the linear equations in quaternion form has never been exploited to derive algorithms such as the one presented, but that seems to be the case.

1. Geometric Algebra $\mathbb{G}_3$

A geometric algebra $\mathbb{G}_3$ is constructed over a real vector space $\mathbb{R}^3$, with basis vectors $\{e_1, e_2, e_3\}$. The associative geometric product is defined so that the

square of any vector is a scalar $aa = a^2 \in \mathbb{R}$. From the vector space $\mathbb{R}^3$, the geometric product generates the geometric algebra $\mathbb{G}_3$ with elements $\{X, R, A...\}$ called multivectors.

For a pair of vectors, a symmetric inner product $a \cdot b = b \cdot a$ and antisymmetric outer product $a \wedge b = -b \wedge a$ can be defined implicitly by the geometric product $ab = a \cdot b + a \wedge b$ and $ba = b \cdot a + b \wedge a$. It is easy to prove that $a \cdot b = \frac{1}{2}(ab + ba)$ is scalar, while the quantity $a \wedge b = \frac{1}{2}(ab - ba)$, called a bivector or 2-vector, is a new algebraic entity that can be visualized as the two-dimensional analogue of a direction, that is, a planar direction. Similar to vectors, bivectors can be decomposed in a bivector basis $\{e_{12}, e_{13}, e_{23}\}$ where $e_{ij} = e_i \wedge e_j$.

The outer product of three vectors $a \wedge b \wedge c$ generates a 3-vector also known as the pseudoscalar, because the trivector basis consist of single element $e_{123} = e_1 \wedge e_2 \wedge e_3$. Similarly, the scalars are regarded as 0-vectors whose basis is the number 1. It follows that the outer product of k-vectors is the completely antisymmetric part of their geometric product: $a_1 \wedge a_2 \wedge ... \wedge a_k = \langle a_1 a_2 ... a_k \rangle_k$ where the angle bracket means k-vector part, and k is its grade. The term grade is used to refer to the number of vectors in any exterior product. This product vanishes if and only if the vectors are linearly dependent. Consequently, the maximal grade for nonzero k-vectors is 3. It follows that every multivector X can be expanded into its k-vector parts and the entire algebra can be decomposed into k-vector subspaces:

$$\mathbb{G}_3 = \sum_{k=0}^{n} \mathbb{G}_3^k = \{X = \sum_{k=0}^{n} \langle X \rangle_k\}$$

This is called a *grading* of the algebra.

Reversing the order of multiplication is called reversion, as expressed by $(a_1 a_2 ... a_k)\tilde{} = a_k ... a_2 a_1$ and $(a_1 \wedge a_2 \wedge ... \wedge a_k)\tilde{} = a_k \wedge ... \wedge a_2 \wedge a_1$, and the reverse of an arbitrary multivector is defined by $\tilde{X} = \sum_{k=0}^{n} \langle \tilde{X} \rangle_k$.

Rotations are even grade multivectors known as rotors. We denote the subalgebra of rotors as $\mathbb{G}_3^+$. A rotor R can be generated as the geometric product of an even number of vectors. A reflection of any k-vector X in a plane with normal n is expressed as the sandwitch product $(-1)^k nXn$. The most basic rotor R is defined as the product of two unit vectors a and b with angle of $\frac{\theta}{2}$. The rotation plane is the bivector $B = \frac{a \wedge b}{\|a \wedge b\|}$.

$$ab = a \cdot b + a \wedge b = \cos\left(\frac{\theta}{2}\right) + B \sin\left(\frac{\theta}{2}\right).$$

Rotors act on all k-vectors using the sandwitch product $X' = RX\tilde{R}$, where $\tilde{R}$ is the reverse of R and can be obtained by reversing the order of all the products of vectors.

2. Related Works

3. Geometric Algebra Rotor Estimation

Given two sets of n corresponding vectors $P = \{p_j\}_{j=1}^n$ and $Q = \{q_j\}_{j=1}^n$, we attempt to minimize the following error function:

$$E(R) = \min_{R \in \mathbb{G}_3^+} \sum_j c_j \|q_j - R p_i \tilde{R}\|^2$$
$$s.t.\ R\tilde{R} = 1$$

where $\{c_j\}_{j=1}^n$ are scalar weights such that $\sum_j^n c_j = 1$. In that form, the minimization is a nonlinear least squares problem in R. Following Perwass, we can make it a linear least squares problem by multiplying by R on the right and using the fact that $R\tilde{R} = 1$.

$$\begin{aligned} q_j - R p_j \tilde{R} &= 0 \\ q_j R - R p_j \tilde{R} R &= 0 \\ q_j R - R p_j &= 0 \end{aligned}$$

with the normalization constraint $R\tilde{R} = 1$ now implicit in the energy $E(R)$. Although the normalization is implicit, it is not enforced, so we will need to project R back to the rotor manifold through normalization. So the equivalent linear least squares problem is:

$$E(R) = \min_{R \in \mathbb{G}_3^+} \sum_j c_j \|R p_j - q_j R\|^2$$

Let $\Psi^j : \mathbb{G}_3^+ \mapsto \mathbb{G}_3^-$ be a function that maps even grade multivectors to odd grade multivectors.

$$\Psi^j(R) = \sqrt{c_j}(R p_j - q_j R)$$

And let $F^j : \mathbb{G}_3^- \mapsto \mathbb{R}^4$ be a function that projects an odd grade multiverctor to a 4×1 column matrix.

$$F^j(X) = \begin{bmatrix} \langle X \rangle_1 \cdot e_1 \\ \langle X \rangle_1 \cdot e_2 \\ \langle X \rangle_1 \cdot e_3 \\ \langle X \rangle_3 \cdot \tilde{I} \end{bmatrix}$$

For the sake of simplicity we will refer to the composed function $F^j \circ \Psi^j : \mathbb{G}_3^+ \mapsto \mathbb{R}^4$ as simply F^j with an abuse of notation. Let F be a column vector of n functions F^j

$$F = \begin{bmatrix} F^1 \\ \vdots \\ F^n \end{bmatrix}$$

such that the energy $E(R)$ can be expressed as matrix product

$$E(R) = F^T F = \begin{bmatrix} F^{1T} & \cdots & F^{nT} \end{bmatrix} \begin{bmatrix} F^1 \\ \vdots \\ F^n \end{bmatrix}$$

The critical points of the energy $E(R)$ where it is minimized can be found solving $\nabla E(R) = 0$. Where the gradient has the following form:

$$\begin{aligned} g &= \nabla E(R) \\ g &= \nabla (F^T F) \\ g &= 2J^T F \end{aligned}$$

where J is the Jacobian matrix of F. Since the energy $E(R)$ is purely quadratic in R, the solution for $\nabla E(R) = 0$ can be found by solving a linear system of equations. An optimal rotor is in the null space of a particular matrix derived from $J^T F = 0$. An easy way to obtain a solution in the null-space is using one iteration of Newton's method. Due to linearity of the equation $J^T F = 0$ w.r.t. R one iteration suffice for obtaining a solution. The optimal increment for $E(R)$ is given by the Newton formula $\Delta R = H^{-1} \nabla E$, provided that the inverse of Hessian matrix H^{-1} exists. Noting that J does not depend on R, i.e. is constant, the Hessian matrix H takes the simple form:

$$\begin{aligned} H &= \frac{\partial (2J^T F)}{\partial R} \\ H &= 2J^T \frac{\partial F}{\partial R} \\ H &= 2J^T J \end{aligned}$$

Where H is independent of R. An optimal solution can then be found by solving a linear system.

$$R^* = R_0 - H^{-1} g(R_0)$$

Where R_0 is an initial rotor and $g(R_0)$ is the gradient of E evaluated at R_0. Of course, the choice of initial rotor R_0 has a particular effect on finding a local minimum. The energy $E(R)$ is non-convex, its shape is similar to a sinusoidal wave with infinitely many points at minimum energy value, each one at rotor $R_i = e^{-(\theta + i\pi)B}$ for $i \in \mathbb{N}$, being B the optimal unique attitude bivector and θ an optimal angle with minimal absolute value. The choice of R_0 lead us to find a local solution close to it, the choice $R_0 = [1\ 0\ 0\ 0]^T$ is optimal from the computational point of view and is also desirable because lead us to find solutions close to the *identity* rotor.

Since there are infinite solutions, H is a singular matrix and consequently the problem is ill-posed. One simple solution is to use the Tikhonov regularization on H to find an inverse $(H + \epsilon I)^{-1}$ that approaches the Moore-Penrose pseudo-inverse as ϵ approaches to zero. An optimal solution can then be found by solving a series of linear systems.

$$R_{i+1} = R_i - (H + \epsilon I)^{-1} g(R_i)$$

until convergence is reached in the sense that $|R_i - R_{i+1}| < \xi$ for a small ξ. Where we use $R_0 = [1\ 0\ 0\ 0]^T$. This algorithm has a number of advantages. Since Hessian $(H + \epsilon I)^{-1}$ is constant, it can be precomputed. Also the Jacobian matrix J^T is constant, so it can be precomputed as well. But the computation of the gradient $g(R_i) = J^T F(R_i)$ is still required at each step. Although the computation of $g(R_i)$ is relatively cheap and this approach is fast we can do better by making the gradient constant as well.

4. Fast Geometric Algebra Rotor Estimation

From now on we will consider the gradient to be constant $g = g(R_0)$. Where $R_0 = [1\ 0\ 0\ 0]^T$ is the identity rotor. However, we will add a very small regularization term to $E(R)$ to allow optimization. Our strategy is to introduce a new fixed rotor R_i within a new function $\Phi(R, R_i) = \sqrt{\epsilon}(R - R_i)$. Let us define a function $D : \mathbb{G}_3^+ \mapsto \mathbb{R}^4$ that projects a multivector of even grade to a 4×1 column matrix.

$$D(X) = \begin{bmatrix} \langle X \rangle_0 \\ X \cdot e_{12} \\ X \cdot e_{13} \\ X \cdot e_{23} \end{bmatrix}$$

For the sake of simplicity we will refer to the composed function $D \circ \Phi : \mathbb{G}_3^+ \times \mathbb{G}_3^+ \mapsto \mathbb{R}^4$ as simply D with an abuse of notation. The new term can be expressed as matrix product:

$$D^T D = \epsilon \|R - R_i\|^2$$

with Jacobian

$$J_D = \frac{\partial D}{\partial R} = \sqrt{\epsilon} I$$

where I is the 4×4 identity matrix. The energy now looks like this:

$$E_2(R, R_i) = F^T F + D^T D$$
$$E_2(R, R_i) = \sum_j c_j \|R p_j - q_j R\|^2 + \epsilon \|R - R_i\|^2$$

Where $0 < \epsilon < 1$ is constant but small (we use $\epsilon = 10^{-6}$). The regularization term $D^T D = \epsilon \|R - R_i\|^2$ helps on finding a solution *shifted* by a small amount towards the direction $\delta D = \epsilon(R - R_i)$, so one can interpret the rotor R_i as a target of displacement. The regularization is perturbing the gradient and Hessian in the following way:

$$g_2 = \nabla(F^T F + D^T D)$$
$$g_2 = \frac{\partial (F^T F)}{\partial R} + \frac{\partial (D^T D)}{\partial R}$$
$$g_2 = 2(J^T F + J_D^T D)$$
$$g_2 = 2(J^T F + \sqrt{\epsilon} D)$$

$$H_2 = 2\frac{\partial(J^T F + \sqrt{\epsilon}D)}{\partial R}$$
$$H_2 = 2J^T\frac{\partial(F)}{\partial R} + 2\sqrt{\epsilon}\frac{\partial(D)}{\partial R}$$
$$H_2 = 2(J^T J + \epsilon I)$$

It is useful to express g_2 also as $g_2 = g(R) + \sqrt{\epsilon}D(R, R_i)$ and H_2 as $H_2 = H + \epsilon I$. An optimal solution can then be found by solving a series of linear systems:

$$R_{i+1} = R_0 - (H + \epsilon I)^{-1}(g(R_0) + \sqrt{\epsilon}D(R_0, R_i))$$

This perturbation on the Hessian resembles now the Tikhonov regularization, and inverse H_2^{-1} approaches the Moore-Penrose pseudo-inverse. The perturbation on the gradient $g(R_0)$ affects its direction, forcing R_{i+1} to move a little bit towards R_i.

Notice that if we take R_i as being equal to R_0 then the gradient is not perturbed by any displacement. This is ok for finding an initial approximation R_{i+1}, moreover when the optimal rotor R^* is close to R_0 the feedback is not necessary. However, if the optimal rotor is at $\pm\pi$ from R_0, that choice might cause the optimization get stuck. For example, given two sets of corresponding vectors $P = \{p_1 = e_1, p_2 = e_2\}$ and $Q = \{q_1 = -e_1, q_2 = -e_2\}$ which are rotated by π radians to each other, the Hessian is diagonal, and its pseudo-inverse H_2^{-1} is simply the reciprocal values of its diagonal, assuming $R_i = R_0$, the gradient $g(R_0)$ coincide with the first column of H, and so the product $H_2^{-1}g(R_0) = [1\ 0\ 0\ 0]^T$ so the resulting rotor $R_{i+1} = R_0 - H_2^{-1}g(R_0) = [0\ 0\ 0\ 0]^T$. In practice, we have never experienced this problem, even with synthetic data, since the Tikhonov regularization introduces a sufficient perturbation on the pseudo-inverse to get a non-zero R_{i+1} which, after normalization, is useful as a feedback direction.

Given an estimated rotor R_{i+1} a necessary condition for it to the optimal is that the recurrence $R_{i+1} = R_0 - H_2^{-1}g_2(R_0, R_i)$ converge to a rotor R^* in the sense that $|R_i - R_{i+1}| < \xi$ for a small ξ. That is true because if R_{i+1} is already optimal, the small displacement induced by the regularization term $\sqrt{\epsilon}D(R_0, R_i)$ should increase the least squares error E_2 and the linear system should compute the last $R_{i+1} = R^*$ as it has less error. So regularization term can be seen as a feedback for reaching a stable R^*. Since the regularization term is only affecting the gradient, the computation is very cheap. The gradient $g(R_0)$ can be precomputed as well as the matrix H_2^{-1}, and the iteration is reduced to compute a matrix vector multiplication.

5. Optimal Computation of Jacobians

The differential of F^j defined as $J^j = \frac{\partial F^j}{\partial R}$ is a 4×4 matrix which four columns are the directional derivatives of $F^j = \sqrt{c_j}(Rp_j - q_j R)$ w.r.t rotor

components on the basis $\mathbb{G}_3^+$:

$$J^j = \left[\begin{array}{cccc} \frac{\partial F^j}{\partial w} & \frac{\partial F^j}{\partial e_{12}} & \frac{\partial F^j}{\partial e_{13}} & \frac{\partial F^j}{\partial e_{23}} \end{array} \right]$$

$$\frac{\partial F^j}{\partial w} = \sqrt{c_j}(p_j - q_j)$$

$$\frac{\partial F^j}{\partial w} = \sqrt{c_j} \left[\begin{array}{c} (p_j - q_j) \cdot e_1 \\ (p_j - q_j) \cdot e_2 \\ (p_j - q_j) \cdot e_3 \\ 0 \end{array} \right]$$

$$\frac{\partial F^j}{\partial e_{12}} = \sqrt{c_j}(-(p_j + q_j) \cdot e_{12} + (p_j - q_j) \wedge e_{12})$$

$$\frac{\partial F^j}{\partial e_{12}} = \sqrt{c_j} \left[\begin{array}{c} (p_j + q_j) \cdot e_2 \\ -(p_j + q_j) \cdot e_1 \\ 0 \\ (p_j - q_j) \cdot e_3 \end{array} \right]$$

$$\frac{\partial F^j}{\partial e_{13}} = \sqrt{c_j}(-(p_j + q_j) \cdot e_{13} + (p_j - q_j) \wedge e_{13})$$

$$\frac{\partial F^j}{\partial e_{13}} = \sqrt{c_j} \left[\begin{array}{c} (p_j + q_j) \cdot e_3 \\ 0 \\ -(p_j + q_j) \cdot e_1 \\ -(p_j - q_j) \cdot e_2 \end{array} \right]$$

$$\frac{\partial F^j}{\partial e_{23}} = \sqrt{c_j}(-(p_j + q_j) \cdot e_{23} + (p_j - q_j) \wedge e_{23})$$

$$\frac{\partial F^j}{\partial e_{23}} = \sqrt{c_j} \left[\begin{array}{c} 0 \\ (p_j + q_j) \cdot e_3 \\ -(p_j + q_j) \cdot e_2 \\ (p_j - q_j) \cdot e_1 \end{array} \right]$$

So the full expression of J^j is:

$$J^j = \sqrt{c_j} \left[\begin{array}{cccc} (p_j - q_j) \cdot e_1 & (p_j + q_j) \cdot e_2 & (p_j + q_j) \cdot e_3 & 0 \\ (p_j - q_j) \cdot e_2 & -(p_j + q_j) \cdot e_1 & 0 & (p_j + q_j) \cdot e_3 \\ (p_j - q_j) \cdot e_3 & 0 & -(p_j + q_j) \cdot e_1 & -(p_j + q_j) \cdot e_2 \\ 0 & (p_j - q_j) \cdot e_3 & -(p_j - q_j) \cdot e_2 & (p_j - q_j) \cdot e_1 \end{array} \right]$$

The gradient of $E(R)$ amounts to $g = \sum_j J^{jT} F^j$ and the symmetric matrix $H^j = J^{jT} J^j$ has a simple form:

$$S_1 = (q_j + p_j) \cdot e_1 \ , \ S_2 = (q_j + p_j) \cdot e_2 \ , \ S_3 = (q_j + p_j) \cdot e_3$$
$$D_1 = (p_j - q_j) \cdot e_1 \ , \ D_2 = (p_j - q_j) \cdot e_2 \ , \ D_3 = (p_j - q_j) \cdot e_3$$

$$H^j = J^{jT} J^j = c_{ij} \begin{bmatrix} D_1^2 + D_2^2 + D_3^2 & D_1 S_2 - D_2 S_1 & D_1 S_3 - D_3 S_1 & D_2 S_3 - D_3 S_2 \\ D_1 S_2 - D_2 S_1 & S_2^2 + S_1^2 + D_3^2 & S_2 S_3 - D_3 D_2 & D_3 D_1 - S_1 S_3 \\ D_1 S_3 - D_3 S_1 & S_2 S_3 - D_3 D_2 & S_3^2 + S_1^2 + D_2^2 & S_1 S_2 - D_2 D_1 \\ D_2 S_3 - D_3 S_2 & D_3 D_1 - S_1 S_3 & S_1 S_2 - D_2 D_1 & S_3^2 + S_2^2 + D_1^2 \end{bmatrix}$$

Since H^j is symmetric only 10 out of 16 elements need to be actually computed. The Hessian of $E(R)$ amounts to $H = \sum_j H^j$ and is constant.

We now can proceed to optimize our method. We incorporate the fixed initial guess $R_0 = 1$ into the gradient $g = \sum_j g^j = J^T F$.

$$F^j = \sqrt{c_j}(R_0 p_j - q_j R_0)$$
$$F^j = \sqrt{c_j}(p_j - q_j)$$

$$F^j = \sqrt{c_{ij}} \begin{bmatrix} (p_j - q_j) \cdot e_1 \\ (p_j - q_j) \cdot e_2 \\ (p_j - q_j) \cdot e_3 \\ 0 \end{bmatrix}$$

$$g^j = J^{jT} F^j = c_{ij} \begin{bmatrix} D_1^2 + D_2^2 + D_3^2 \\ D_1 S_2 - D_2 S_1 \\ D_1 S_3 - D_3 S_1 \\ D_2 S_3 - D_3 S_2 \end{bmatrix}$$

Notice that, with this replacement, the gradient $g = \sum_j g^j$ is constant. Notice also that the first column of H^j is equal to g^j, so $g = \sum_j g^j$ does not need to be explicitly calculated. Now we proceed to replace R_0 on the regularization term D, which look like this:

$$D = \sqrt{\epsilon}(R_0 - R_i)$$
$$D = \sqrt{\epsilon}(1 - R_i)$$

$$\sqrt{\epsilon} D = J_D^T D = \epsilon \begin{bmatrix} 1 - \langle R_i \rangle_0 \\ -R_i \cdot e_{12} \\ -R_i \cdot e_{13} \\ -R_i \cdot e_{23} \end{bmatrix}$$

With the above replacements, notice that all terms are constant except D which depends only on the previous iteration.:

$$R_{i+1} = R_0 - (H + \epsilon I)^{-1}(g + \sqrt{\epsilon} D(R_i))$$

the optimization loop amounts to compute a cheap matrix-vector multiplication.

6. Algorithms

There are at least two ways to obtain R_i. A *standalone* algorithm will always initialize $R_i = R_0$ and update R_i with rotor R_{i+1} calculated in the optimization loop. In a simulation, we also can take R_i from the outside and perform only one iteration of the optimization to quickly yield a non-optimal rotor for the next simulation step. We call the later the *incremental* algorithm. We have used both choices in our experiments, with excellent results in both cases. In the later case, our experiments indicates that our algorithm preserves the sense of successive rotations.

The *standalone* method is shown in Algorithm 1.

Algorithm 1 Fast Rotor Estimation

Require: $P = \{p_j\}_{j=1}^n, Q = \{q_j\}_{j=1}^n, C = \{c_j\}_{j=1}^n$

$R_0 = [1,0,0,0]^T, R_1 = [1,0,0,0]^T, H = 0_{4\times 4}$

for $j = 1$ **to** n **do**

$S_1 = (q_j + p_j) \cdot e_1\ ,\ S_2 = (q_j + p_j) \cdot e_2\ ,\ S_3 = (q_j + p_j) \cdot e_3$

$D_1 = (p_j - q_j) \cdot e_1\ ,\ D_2 = (p_j - q_j) \cdot e_2\ ,\ D_3 = (p_j - q_j) \cdot e_3$

$H = H +$

$$c_j \begin{bmatrix} D_1^2 + D_2^2 + D_3^2 & D_1S_2 - D_2S_1 & D_1S_3 - D_3S_1 & D_2S_3 - D_3S_2 \\ D_1S_2 - D_2S_1 & S_2^2 + S_1^2 + D_3^2 & S_2S_3 - D_3D_2 & D_3D_1 - S_1S_3 \\ D_1S_3 - D_3S_1 & S_2S_3 - D_3D_2 & S_3^2 + S_1^2 + D_2^2 & S_1S_2 - D_2D_1 \\ D_2S_3 - D_3S_2 & D_3D_1 - S_1S_3 & S_1S_2 - D_2D_1 & S_3^2 + S_2^2 + D_1^2 \end{bmatrix}$$

end for

$g = -\begin{bmatrix} H(0,0) & H(1,0) & H(2,0) & H(3,0) \end{bmatrix}^T$

$i = 0$

repeat

$i = i + 1$

$R_{i+1} = normalize(R_0 + (H + \epsilon I)^{-1}(g + \epsilon(R_i - R_0)))$

until $|R_i - R_{i+1}| < \xi$

return $R_{i+1}(0) + R_{i+1}(1)e_{12} + R_{i+1}(2)e_{13} + R_{i+1}(3)e_{23}$

The C++ code using the Eigen library can be found in Listing 1.

The *incremental* algorithm is best suited if we know that the two sets of vectors are almost aligned. Let us suppose that we already have suboptimal rotor estimation R_{prev}, perhaps calculated for a previous simulation step, that almost align the two sets of vectors. We can use R_{prev} to form a direction $\delta D = \epsilon(R - R_{prev})$ which we integrate in the regularization term. So the new rotor estimation is given by $R_{i+1} = R_0 - (H + \epsilon I)^{-1}(g + \epsilon(R_{prev} - R_0))$. This can be seen as moving the loop from inside the *standalone* algorithm to the outside. When a simulation provides a temporal coherence between consecutive simulation steps, the incremental estimation of rotations is a good choice, since incremental algorithm only requires to solve a linear system for computing the next best rotor. Provided that the simulation converge to stable sets of corresponding vectors, then incremental rotor estimation will converge to R^*. The Algorithm 2 shows the complete algorithm.

Algorithm 2 Incremental Fast Rotor Estimation

Require: $P = \{p_j\}_{j=1}^{n}, Q = \{q_j\}_{j=1}^{n}, C = \{c_j\}_{j=1}^{n}, R_{prev}$

$R_0 = [1,0,0,0]^T, H = 0_{4\times4}$

for $j = 1$ **to** n **do**

$S_1 = (q_j + p_j) \cdot e_1\ ,\ S_2 = (q_j + p_j) \cdot e_2\ ,\ S_3 = (q_j + p_j) \cdot e_3$

$D_1 = (p_j - q_j) \cdot e_1\ ,\ D_2 = (p_j - q_j) \cdot e_2\ ,\ D_3 = (p_j - q_j) \cdot e_3$

$H = H +$

$$c_j \begin{bmatrix} D_1^2 + D_2^2 + D_3^2 & D_1S_2 - D_2S_1 & D_1S_3 - D_3S_1 & D_2S_3 - D_3S_2 \\ D_1S_2 - D_2S_1 & S_2^2 + S_1^2 + D_3^2 & S_2S_3 - D_3D_2 & D_3D_1 - S_1S_3 \\ D_1S_3 - D_3S_1 & S_2S_3 - D_3D_2 & S_3^2 + S_1^2 + D_2^2 & S_1S_2 - D_2D_1 \\ D_2S_3 - D_3S_2 & D_3D_1 - S_1S_3 & S_1S_2 - D_2D_1 & S_3^2 + S_2^2 + D_1^2 \end{bmatrix}$$

end for

$g = -\begin{bmatrix} H(0,0) & H(1,0) & H(2,0) & H(3,0) \end{bmatrix}^T$

$\Delta R = (H + \epsilon I)^{-1}(g + \epsilon(R_{prev} - R_0))$

$R = normalize(R_0 + \Delta R)$

return $R(0) + R(1)e_{12} + R(2)e_{13} + R(3)e_{23}$

7. Experiments

Accuracy, efficiency, with/o noise

8. Comparisons

Comparisons with SVD, Horn, Flae, Valkenburg, E3GA. Accuracy, efficiency, with/o noise

9. C++ code

LISTING 1. C++ code for rotor estimation

```
Quaterniond FastRotorEstimator(
        vector<Vector3d>& P, vector<Vector3d>& Q, vector<double>& weights)
{
        Matrix4d H;
        Vector4d g, R, R_i;
        Vector3d S, D;
        const double epsilon = 1e-6;
        double wj;
        const size_t N = P.size();

        H.setZero();
        for (size_t j = 0; j < N; ++j) {
                wj = weights[j];
                S = Q[j] + P[j];
                D = P[j] - Q[j];
                H(0, 0) += wj*(D[0]*D[0] + D[1]*D[1] + D[2]*D[2]);
                H(0, 1) += wj*(D[0]*S[1] - D[1]*S[0]);
                H(0, 2) += wj*(D[0]*S[2] - D[2]*S[0]);
                H(0, 3) += wj*(D[1]*S[2] - D[2]*S[1]);
                H(1, 1) += wj*(S[1]*S[1] + S[0]*S[0] + D[2]*D[2]);
                H(1, 2) += wj*(S[1]*S[2] - D[2]*D[1]);
                H(1, 3) += wj*(D[2]*D[0] - S[0]*S[2]);
                H(2, 2) += wj*(S[2]*S[2] + S[0]*S[0] + D[1]*D[1]);
```

```
                H(2, 3) += wj*(S[0]*S[1] - D[1]*D[0]);
                H(3, 3) += wj*(S[2]*S[2] + S[1]*S[1] + D[0]*D[0]);
        }
        H(1, 0) = H(0, 1);
        H(2, 0) = H(0, 2); H(2, 1) = H(1, 2);
        H(3, 0) = H(0, 3); H(3, 1) = H(1, 3); H(3, 2) = H(2, 3);
        g = -H.col(0);
        H(0, 0) += epsilon; H(1, 1) += epsilon;
        H(2, 2) += epsilon; H(3, 3) += epsilon;
        H = H.inverse();
        R(0) = 1; R(1) = 0; R(2) = 0; R(3) = 0;
        do {
                R_i = R;
                R(0) -= 1.0;
                R.noalias() = H * (g + epsilon * R);
                R(0) += 1.0;
                R /= sqrt(R.dot(R));
        } while ((R_i - R).dot(R_i - R) > 1e-6);
        return Quaterniond(R(0), -R(3), R(2), -R(1));
}
```

LISTING 2. C++ code for rotor estimation using Newton method

```
Quaterniond RotorEstimator(
        vector<Vector3d>& P, vector<Vector3d>& Q, vector<double>& weights)
{
        const size_t MAX_VECTORS = 32;
        const size_t N = P.size() < MAX_VECTORS? P.size() : MAX_VECTORS;
        Matrix4d Jt[MAX_VECTORS], H;
        Vector4d Fi, g, Ri, R(1, 0, 0, 0);
        Vector3d D, S;
        double wi;

        H.setZero();
        for (size_t i = 0; i < N; ++i) {
                wi = weights[i];
                S = Q[i] + P[i];
                D = P[i] - Q[i];
                Matrix4d& Jti = Jt[i];
                Jti(0, 0) = wi*D[0];  Jti(0, 1) = wi*D[1];
                Jti(0, 2) = wi*D[2];  Jti(0, 3) = 0;
                Jti(1, 0) = wi*S[1];  Jti(1, 1) = -wi*S[0];
                Jti(1, 2) = 0;        Jti(1, 3) = wi*D[2];
                Jti(2, 0) = wi*S[2];  Jti(2, 1) = 0;
                Jti(2, 2) = -wi*S[0]; Jti(2, 3) = -wi*D[1];
                Jti(3, 0) = 0;        Jti(3, 1) = wi*S[2];
                Jti(3, 2) = -wi*S[1]; Jti(3, 3) = wi*D[0];
                H(0, 0) += wi*(D[0]*D[0] + D[1]*D[1] + D[2]*D[2]);
                H(0, 1) += wi*(D[0]*S[1] - D[1]*S[0]);
                H(0, 2) += wi*(D[0]*S[2] - D[2]*S[0]);
                H(0, 3) += wi*(D[1]*S[2] - D[2]*S[1]);
                H(1, 1) += wi*(S[1]*S[1] + S[0]*S[0] + D[2]*D[2]);
                H(1, 2) += wi*(S[1]*S[2] - D[2]*D[1]);
                H(1, 3) += wi*(D[2]*D[0] - S[0]*S[2]);
                H(2, 2) += wi*(S[2]*S[2] + S[0]*S[0] + D[1]*D[1]);
                H(2, 3) += wi*(S[0]*S[1] - D[1]*D[0]);
                H(3, 3) += wi*(S[2]*S[2] + S[1]*S[1] + D[0]*D[0]);
        }
        H(1, 0) = H(0, 1);
        H(2, 0) = H(0, 2); H(2, 1) = H(1, 2);
        H(3, 0) = H(0, 3); H(3, 1) = H(1, 3); H(3, 2) = H(2, 3);
        H(0, 0) += 1e-6; H(1, 1) += 1e-6;
        H(2, 2) += 1e-6; H(3, 3) += 1e-6;
        H = H.inverse();
        do {
                Ri = R;
                g.setZero();
                for (size_t i = 0; i < N; ++i) {
```

```
                    S = Q[i] + P[i];
                    D = P[i] - Q[i];
                    Fi(0) = -( S[2]*R[2] + S[1]*R[1] + D[0]*R[0]);
                    Fi(1) = -( S[2]*R[3] - S[0]*R[1] + D[1]*R[0]);
                    Fi(2) = -(-S[1]*R[3] - S[0]*R[2] + D[2]*R[0]);
                    Fi(3) = -( D[0]*R[3] - D[1]*R[2] + D[2]*R[1]);
                    g += Jt[i] * Fi;
                }
                R += H * g;
                R /= sqrt(R.dot(R));
        } while ((Ri - R).dot(Ri - R) > 1e-6);
        return Quaterniond(R[0], -R[3], R[2], -R[1]);
}
```

References

Mauricio Cele Lopez Belon
Buenos Aires, Argentina
e-mail: mclopez@outlook.com