Lambda Twist: An Accurate Fast Robust Perspective Three Point (P3P) Solver.

Mikael Persson[1] [0000−0002−5931−9396] and Klas Nordberg[1] *

Computer Vision Laboratory, Linköping University, Sweden

Abstract. We present Lambda Twist; a novel P3P solver which is accurate, fast and robust. Current state-of-the-art P3P solvers find all roots to a quartic and discard geometrically invalid and duplicate solutions in a post-processing step. Instead of solving a quartic, the proposed P3P solver exploits the underlying elliptic equations which can be solved by a fast and numerically accurate diagonalization. This diagonalization requires a single real root of a cubic which is then used to find the, up to four, P3P solutions. Unlike the direct quartic solvers our method never computes geometrically invalid or duplicate solutions.
Extensive evaluation on synthetic data shows that the new solver has better numerical accuracy and is faster compared to the state-of-the-art P3P implementations. Implementation and benchmark are available on github.

Keywords: P3P · PnP · Visual Odometry · Camera Geometry

1 Introduction

Pose estimation from projective observations of known model points, also known as the Perspective n-point Problem (PnP), is extensively used in geometric computer vision systems. In particular, finding the camera pose (orientation and position) from observations of n 3D points in relation to a world coordinate system is often the first step in visual odometry and augmented reality systems[12,7]. It is also an important part in structure from motion and reconstruction of unordered images [1]. The minimal PnP case with a finite number of solutions requires three ($n = 3$) observations in a nondegenerate configuration and is known as the P3P problem (Figure 1).

We are concerned with the latency and accuracy critical scenarios of odometry on low power hardware and AR/VR. Since both latency and localization errors independently not only break immersion, but also cause nausea, accurate solutions and minimal latency are crucial. As an example application, vision based localization for AR/VR places a few markers/beacons on a target, which are then found using a high speed camera. Ideally we would then solve the pose directly on chip without sending the full image stream elsewhere, mandating minimal cost. Further, because the markers are placed on a small area and the camera is of relatively low resolution, the markers are close to each other, meaning numerical issues due to near degenerate cases are common and the algorithm must be robust. The experiments will show that we have made substantial progress on both speed and accuracy compared to state-of-the-art.

* name.surname@liu.se

Surveying the literature we find that P3P solvers are either direct or triangulating. Triangulating methods first triangulate the points in the camera coordinate system using a pose invariant, leaving only distances as unknowns and then solve for the pose. In this case the rotation is solved for as either a quaternion or $\mathbf{R} \in SO(3)$ depending on end user preference. This allows the geometric feasibility constraints, i.e. each point must lie in front of the camera, to limit the solutions prior to computing the pose. In contrast, direct methods parametrize the pose in the input coefficients using a projective invariant. Therefore, they have to apply the feasibility constraint after the solutions are found, as a post processing step.

To our knowledge, all direct methods, including state-of-the-art [6,8], are based on finding the four roots of a quartic. This requires complex arithmetics and root polishing to achieve high numerical accuracy. In contrast, triangulating methods can find all P3P solutions by diagonalizing a three by three matrix. Further, the methods by Kneip and Ke output $\mathbf{R} \in SO(3)$, which if unit quaternion representation is desired requires careful conversion. Thus, triangulating methods have a potential advantage in terms of numerical complexity and accuracy. This motivates us to revisit the P3P problem. To this end we derive a novel triangulating P3P solution designed to provide high numerical accuracy and computational performance. Unlike earlier approaches based on similar pose invariants, we use a novel solution path. This allows us to discard infeasible and duplicate solutions at an earlier stage, thereby saving computation. We explicitly exploit that only a single real root of the diagonalizing cubic is required. We find this root using Newton's method with an initializing heuristic. This improves the numerical accuracy and again saves computation.

We believe that the main strength of our algorithm is the relative numerical stability of solving for one root of a cubic compared to the multiple iterative solvers used in the solution of a quartic. Note that while discarding invalid and duplicate solutions in advance saves us time, there is another more subtle problem which our solution avoids. Note that the numeric accuracy of quartic solver roots decrease severely with increasing root multiplicity. Because such roots correspond to duplicate or near duplicate solutions and because such duplicates are common see Section 4.4, this reduces the numerical accuracy of the quartic based solvers beyond what might otherwise be expected. Further this predicts the behavior seen in the experiments where a incorrect solution found by Ke's solver will be refined to a duplicate root with further iterations. Since we compute the single root we need iteratively, the multiplicity of the cubic does not significantly affect the numerical accuracy of our solver. Further, since the different roots of the cubic do not correspond to different or duplicate solutions, higher multiplicities are rare. Finally, the initialization used makes it very likely that the root we converge to is the one with the best numerical properties.

Although grounded in geometry, the proposed method is derived using methods from linear algebra. The method is parameterized simply in the signed distance to the three 3D points. Although difficult to compare, we believe that the proposed approach is comparatively simple to understand.

Through extensive experiments we show that our solver achieves superior computational and significantly better numerical performance over prior state of the art.

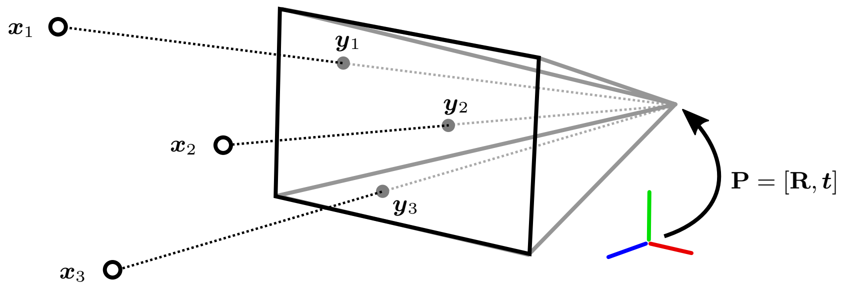

Fig. 1. The P3P setup, where each pair $\{\boldsymbol{x}_1, \boldsymbol{y}_1\}$, $\{\boldsymbol{x}_2, \boldsymbol{y}_2\}$ and $\{\boldsymbol{x}_3, \boldsymbol{y}_3\}$ are used in the calculation of the camera pose $\mathbf{P} = [\mathbf{R}, \boldsymbol{t}]$.

1.1 Related Work

Several methods for solving P3P can be found in the literature. The first solution for P3P was published 1841 by Grunert [4], which demonstrated that P3P has up to 4 feasible solutions. Since then, a large number of solvers have been published, which improve the original formulation in various ways. An overview of several of the relevant P3P algorithms until 1991 is presented in the paper by Haralick et al. [5]. More recent methods can be found in the work by Gao et al. [2], Kneip et al. [8], and by Ke et al. [6].

A common theme in a majority of P3P solvers is that each valid solution is associated with a root of a quartic. What differs between the solvers is how this quartic is formulated from the known data, and how the solution are associated with the roots. To find all valid solutions to P3P these methods thus need to find all real roots to a quartic.

A few of the proposed methods, e.g. the one by Finsterwalder, accounted for by Haralick et al. [5], and the one by Grafarend et al. [3], have observed that pairs of P3P solutions can be associated with the roots of a cubic. This observation has not been given much attention in the literature on P3P. In particular, it has not been discussed in the more recent P3P methods.

In this paper we present a novel P3P solver that, while it shares the projective invariant beginnings with Grafarend et al. [9], follows a different solution path which enables an efficient implementation

2 Problem Formulation

Given a calibrated pinhole camera, three 3D points $\boldsymbol{x}_i = (x_i, y_i, z_i)$, and corresponding homogeneous image coordinates $\boldsymbol{y}_i \sim (u_i, v_i, 1)$ such that $|\boldsymbol{y}_i| = 1$, then:

$$\lambda_i \boldsymbol{y}_i = \mathbf{R}\boldsymbol{x}_i + \boldsymbol{t},\ i \in \{1, 2, 3\}, \tag{1}$$

where the rotation $\mathbf{R} \in SO(3)$ together with the translation $\boldsymbol{t} \in \mathbb{R}^3$ define the pose of the camera. In short a P3P solver is a function $[\mathbf{R}_k, \boldsymbol{t}_k] = \text{P3P}(\boldsymbol{x}_{1:3}, \boldsymbol{y}_{1:3})$. Depending on the configuration of the points, P3P has up to four solutions.

2.1 Requirements

It is well known that a necessary condition to assure a finite set of solutions, is that neither the 3D points nor the 2D points are collinear[5]. In addition there are two specific requirements any solution should meet. First, they should be real. Second, since the parameters λ_k is the signed distance from the camera center for each 3D point, λ_1, λ_2, and λ_3 are required to be positive and real. This is a geometric feasibility condition on the solutions, which implies that all three 3D points are "in front of" the camera.

3 Lambda Twist Derivation

In this section we derive the the proposed algorithm for solving P3P. The starting point is (1), and the first step is to eliminate $\boldsymbol{t}$ and $\mathbf{R}$, leaving only the signed distance parameters λ_i as unknowns. More precisely, they have to solve a system of three inhomogeneous quadratic equations. As we will see, the solutions to the three inhomogeneous equations can be determined by first reducing the problem to a pair of homogeneous quadratic equations. Solving these homogeneous quadratic equations is isomorphic to finding the intersection of two ellipses in the plane. We diagonalize the elliptic equations by finding a rank 2 linear combination of the constraints. Finding this rank 2 combination requires a single root of a cubic polynomial. In general, this root gives us up to four sets of positive $(\lambda_1, \lambda_2, \lambda_3)$, corresponding to a vector $\boldsymbol{\Lambda} \in \mathbb{R}^3$, an element of the "lambda-space". Finally, for each geometrically feasible set of $\boldsymbol{\Lambda}_k$ parameters, we determine a corresponding camera pose $(\mathbf{R}_k, \boldsymbol{t}_k)$.

3.1 Pose Invariant Constraints

In principle, the homogeneous image coordinates $\boldsymbol{y}_i$ in (1) can be multiplied by an arbitrary non-zero scalar. By doing so, the parameters λ_i have to scale inversely. If the scaling is positive, each λ_i then represents a scaled depth, but the sign rules described in Section 2.1 still apply. In the following, we scale each $\boldsymbol{y}_i$ such that $\|\boldsymbol{y}_i\| = 1$.

We begin the elimination with the translation $\boldsymbol{t}$, by taking the pairwise differences of the three equations in (1), then taking into account that $\mathbf{R} \in SO(3)$ and $\|\boldsymbol{y}_i\| = 1$:

$$\lambda_i \boldsymbol{y}_i - \lambda_j \boldsymbol{y}_j = \mathbf{R}(\boldsymbol{x}_i - \boldsymbol{x}_j), \implies \tag{2}$$

$$\begin{aligned} &|\lambda_i \boldsymbol{y}_i - \lambda_j \boldsymbol{y}_j|^2 = |\boldsymbol{x}_i - \boldsymbol{x}_j|^2 \overset{def}{=} a_{ij}, \implies \\ &\lambda_i^2 + \lambda_j^2 - 2b_{ij} = a_{ij}, \qquad (3) \\ &b_{ij} \overset{def}{=} \boldsymbol{y}_i^T \boldsymbol{y}_j. \end{aligned}$$

This leaves us with only λ_i as free variables to be determined from the three non-trivial ij i.e. $\{12, 13, 23\}$. This is what we will do next.

3.2 Three Inhomogeneous Quadratic Polynomials

The constraints on the scaled depth parameters λ_i in (3) can be formulated in a more compact way as

$$\boldsymbol{\Lambda}^\top \mathbf{M}_{12} \boldsymbol{\Lambda} = a_{12}, \qquad \boldsymbol{\Lambda}^\top \mathbf{M}_{13} \boldsymbol{\Lambda} = a_{13}, \qquad \boldsymbol{\Lambda}^\top \mathbf{M}_{23} \boldsymbol{\Lambda} = a_{23}, \tag{4}$$

where

$$\mathbf{M}_{12} = \begin{pmatrix} 1 & -b_{12} & 0 \\ -b_{12} & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad \mathbf{M}_{13} = \begin{pmatrix} 1 & 0 & -b_{13} \\ 0 & 0 & 0 \\ -b_{13} & 0 & 1 \end{pmatrix}, \quad \mathbf{M}_{23} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & -b_{23} \\ 0 & -b_{23} & 1 \end{pmatrix}.$$

There is a geometric interpretation of the variables: $b_{ij} = \boldsymbol{y}_i^T \boldsymbol{y}_j$ is the cosine of the angle between the projection rays of image point i and j, and a_{ij} is the squared distance between 3D points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. Since we require that the 3D points are not collinear, it follows that $a_{ij} > 0, \forall\, ij$.

Some observations can be made from (4). First, we see that if $\boldsymbol{\Lambda}$ solves these three equations, then so does $-\boldsymbol{\Lambda}$. However as mentioned in Section 2.1 only solutions where all $\lambda_i > 0$ are of interest. A second observation is that the set of $\boldsymbol{\Lambda}$ which solves an equation in (4) forms an elliptic cylinder centered on each of the three axes in the lambda-space. Each solution $\boldsymbol{\Lambda}$ is a point where all three cylinders intersect each other. In general there are 8 such points, taking the sign flip of $\boldsymbol{\Lambda}$ into account, though some of these intersection points may be complex. As mentioned in Section 2.1, we want to remove complex $\boldsymbol{\Lambda}$ and also any $\boldsymbol{\Lambda}$ which does not lie in the “positive octant” of the lambda-space, where $\lambda_1, \lambda_2, \lambda_3 > 0$.

3.3 Two Homogeneous Quadratic Polynomials

A linear combination of Eqn (4) provide a set of homogeneous polynomials which are compactly formulated as

$$\mathbf{D}_1 = \mathbf{M}_{12}a_{23} - \mathbf{M}_{23}a_{12} = \left(\boldsymbol{d}_{11}\boldsymbol{d}_{12}\boldsymbol{d}_{13}\right), \tag{5}$$

$$\mathbf{D}_2 = \mathbf{M}_{13}a_{23} - \mathbf{M}_{23}a_{13} = \left(\boldsymbol{d}_{21}\boldsymbol{d}_{22}\boldsymbol{d}_{23}\right), \tag{6}$$

$$\boldsymbol{\Lambda}^\top \mathbf{D}_i \boldsymbol{\Lambda} = 0, \forall i \in \{1, 2\}, \tag{7}$$

where $\boldsymbol{d}_{ij}$ is column j of the matrix $\mathbf{D}_i$.

Since (7) are homogeneous polynomials, any requirement on the norm of $\boldsymbol{\Lambda}$ is lost. By substituting the solution into one of the equations in (4) the proper scale is found. The equations in (7) specify two ellipses in (λ_1, λ_2) if $\lambda_3 = 1$. Thus the solution represents the intersections of the ellipses.

3.4 The Cubic Polynomial

We now turn to the problem of finding $\boldsymbol{\Lambda}$ that solves (7). For now we assume that $\mathbf{D}_1$ and $\mathbf{D}_2$ are indefinite as the rare cases when this is not true is either trivial or degenerate. In Eq (7), each quadratic form has a solutions set in the form of an elliptic double cone in the lambda-space. These two cones intersect along, at most, four lines. This is true for any quadratic form specified as a linear combination of $\mathbf{D}_1$ and $\mathbf{D}_2$ as well. A special case of such a double cone is a pair of planes, where each plane intersects with two of the lines. Such a degenerate case occurs precisely when the linear combination has rank 2. A plane which includes $\boldsymbol{\Lambda}$ is then an extra linear constraint in addition to the previous quadratic relations. This observation is key for the proposed solver, and we will next consider how to determine these planes, and how they can be used to find all valid $\boldsymbol{\Lambda}$.

We form $\mathbf{D}_0 = \mathbf{D}_1 + \gamma \mathbf{D}_2$ that has a corresponding solution space which is simple to determine. We find this $\mathbf{D}_0$ by solving

$$0 = \det(\mathbf{D}_0) = \det(\mathbf{D}_1 + \gamma \mathbf{D}_2). \tag{8}$$

This corresponds to a cubic polynomial in γ:

$$c_3\gamma^3 + c_2\gamma^2 + c_1\gamma + c_0. \tag{9}$$

The four coefficients are given by

$$\boldsymbol{c} = \begin{pmatrix} c_3 \\ c_2 \\ c_1 \\ c_0 \end{pmatrix} = \begin{pmatrix} \det(\mathbf{D}_2) \\ \boldsymbol{d}_{21}^T(\boldsymbol{d}_{12} \times \boldsymbol{d}_{13}) + \boldsymbol{d}_{22}^T(\boldsymbol{d}_{13} \times \boldsymbol{d}_{11}) + \boldsymbol{d}_{23}^T(\boldsymbol{d}_{11} \times \boldsymbol{d}_{12}) \\ \boldsymbol{d}_{11}^T(\boldsymbol{d}_{22} \times \boldsymbol{d}_{23}) + \boldsymbol{d}_{12}^T(\boldsymbol{d}_{23} \times \boldsymbol{d}_{21}) + \boldsymbol{d}_{13}^T(\boldsymbol{d}_{21} \times \boldsymbol{d}_{22}) \\ \det(\mathbf{D}_1) \end{pmatrix}. \tag{10}$$

In the special case that either of $\mathbf{D}_1$ or $\mathbf{D}_2$ are semi-definite, or indefinite with one eigenvalue equal to zero, we get $c_0 = 0$ or $c_3 = 0$. In this case, is not necessary to form and solve this cubic, we simply set $\mathbf{D}_0 = \mathbf{D}_1$ or $\mathbf{D}_0 = \mathbf{D}_2$. In the general case, we may still want to avoid being close to a case where $c_3 \approx 0$, as the cubic then becomes numerically unstable. In this case the real root of the inverse polynomial, i.e. $\mu = \gamma^{-1}$, is found instead. In practice these are near non-existent cases and the polynomial is well conditioned.

In general, we can assume that neither c_3 nor c_0 vanish, and the cubic polynomial is well-defined and has one, two, or three distinct roots. Diagonalizing $\mathbf{D}_0$ requires only one real root γ_0 which can be found with any of several standard methods. With γ_0 at hand, we set $\mathbf{D}_0 = \mathbf{D}_1 + \gamma_0 \mathbf{D}_2$.

We recall that $\mathbf{D}_0$ specifies two planes, each intersecting with two distinct lines that includes $\boldsymbol{\Lambda}$ solving (7). In the case of four such lines, there are *three* ways to specify the two planes. Therefore we should expect the problem of finding the degenerate cone, specified by $\mathbf{D}_0$, to have three solutions. As we have seen, finding this $\mathbf{D}_0$ can be formulated as finding a root of a cubic. Although additional roots provide more constraints on $\boldsymbol{\Lambda}$, we will in the next step see that one case of $\mathbf{D}_0$ is sufficient for finding all solutions to P3P.

3.5 Diagonalization of $\mathbf{D}_0$

An eigenvalue decomposition of $\mathbf{D}_0$ results in

$$\mathbf{D}_0 = \mathbf{E}\mathbf{S}\mathbf{E}^\top, \text{ where } \mathbf{E} = \begin{pmatrix} e_0 & e_1 & e_2 \\ e_3 & e_4 & e_5 \\ e_6 & e_7 & e_8 \end{pmatrix} = \left(\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3,\right). \tag{11}$$

The matrix $\mathbf{E}$ holds an orthogonal basis of $\mathbb{R}^3$ in its columns $\boldsymbol{e}_i$, and $\mathbf{S}$ holds the corresponding eigenvalues in its diagonal. Since $\det(\mathbf{D}_0) = 0$, at least one of the eigenvalues are zero, and we can assume $\mathbf{S}$ to have the following form:

$$\mathbf{S} = \mathbf{E}^T \mathbf{D}_0 \mathbf{E} = \begin{pmatrix} \sigma_1 & 0 & 0 \\ 0 & -\sigma_2 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \tag{12}$$

and $\sigma_1 > 0, \sigma_2 \geq 0$. The fact that there is at least one non-zero eigenvalue, and that if there is more than one they have opposite sign, is a general observation. This property holds when a rigid transform relates the observations.

We exploit that we know one eigenvalue of $\mathbf{D}_0$ is zero for a efficient eigenvalue decomposition in Algorithm 2. This algorithm also reorders the eigenvectors to ensure that $|\sigma_1| \leq |\sigma_2|$. This improves numerical performance in later steps.

3.6 Solving for $\boldsymbol{\Lambda}$

We use the eigenvalue decomposition of $\mathbf{D}_0$ to solve $\boldsymbol{\Lambda}^T \mathbf{D}_0 \boldsymbol{\Lambda} = 0$ by first solving the simpler equation $\boldsymbol{p}^T \mathbf{S} \boldsymbol{p} = 0$ using the substitution $\boldsymbol{p} = \mathbf{E}^T \boldsymbol{\Lambda}$ where $\boldsymbol{p} = (p_1, p_2, p_3)^T$. With $p_1 = sp_2$ we find that $s = \pm\sqrt{\frac{-\sigma_2}{\sigma_1}}$ for any p_3.

Each solution of s gives us a linear relation between $\lambda_1, \lambda_2, \lambda_3$ using $\boldsymbol{p} = \mathbf{E}^T \boldsymbol{\Lambda}$:

$$\begin{aligned} &\boldsymbol{e}_1 \boldsymbol{\Lambda} = p_1 = sp_2, \\ &\boldsymbol{e}_2 \boldsymbol{\Lambda} = p_2, \implies \\ &s\boldsymbol{e}_2 \boldsymbol{\Lambda} = sp_2, \implies \\ &\boldsymbol{e}_1 \boldsymbol{\Lambda} - s\boldsymbol{e}_2 \boldsymbol{\Lambda} = 0, \implies \\ &\lambda_1 = \underbrace{\frac{e_3 - se_4}{se_1 - e_0}}_{w_0} \lambda_2 + \underbrace{\frac{e_6 - se_7}{se_1 - e_0}}_{w_1} \lambda_3 = w_0 \lambda_2 + w_1 \lambda_3. \end{aligned} \tag{13}$$

The two solutions for s gives two possible expressions for λ_1 in Eqn (13).

Next let: $\lambda_3 = \tau \lambda_2$. This implies $\tau > 0$ since we only seek the solutions where $\lambda_i > 0$ which satisfy the geometric feasibility constraint. Inserting $\lambda_3 = \tau \lambda_2$ and Eqn (13) in e.g. $\boldsymbol{\Lambda} \mathbf{D}_1 \boldsymbol{\Lambda} = 0$ gives

$$\lambda_2^2 \begin{pmatrix} w_0 + \tau w_1 \\ 1 \\ \tau \end{pmatrix}^\top \mathbf{D}_1 \begin{pmatrix} w_0 + \tau w_1 \\ 1 \\ \tau \end{pmatrix} = 0. \tag{14}$$

This leads to a quadratic equation in $\tau : a\tau^2 + b\tau + c = 0$ with coefficients:

$$\begin{aligned} a &= ((a_{13} - a_{12}) w_1^2 + 2a_{12} b_{13} w_1 - a_{12}), \\ b &= (2a_{12} b_{13} w_0 - 2a_{13} b_{12} w_1 - 2w_0 w_1 (a_{12} - a_{13})), \\ c &= ((a_{13} + a_{13} - a_{12}) w_0^2 - 2a_{13} b_{12} w_0). \end{aligned} \tag{15}$$

In summary, for each of the two possible values of s, we get two solutions for τ. This gives up to four solutions. Complex and negative τ are discarded at this point since they will never lead to real and geometrically feasible poses. We denote the surviving solutions as τ_k.

Next we determine λ_{2k} for each τ_k using $\lambda_{3k} = \tau_k \lambda_{2k}$ and Eqn (4). Specifically we use $\boldsymbol{\Lambda}^T \mathbf{M}_{23} \boldsymbol{\Lambda}^T = a_{12}$ because it does not depend on λ_1. The result is

$$\lambda_{2k}^2 \begin{pmatrix} 1 \\ \tau_k \end{pmatrix}^T \begin{pmatrix} 1 & -b_{23} \\ -b_{23} & 1 \end{pmatrix} \begin{pmatrix} 1 \\ \tau_k \end{pmatrix} = a_{23}, \quad \lambda_{2k} > 0, \tag{16}$$

which is solved by $\lambda_{2k} = \sqrt{\frac{a_{23}}{\tau_k(b_{23}+\tau_k)+1}}$. This gives us a λ_{2k} for each τ_k. Note that $a_{23} > 0$ and $(\tau_k(b_{23} + \tau_k) + 1) > 0$ since $|b_{23}| \leq 1$.

Finally we compute $\lambda_{3k} = \tau_k \lambda_{2k}$ and λ_{1k} from Eqn (13). To summarize, for each τ_k we get a $\boldsymbol{\Lambda}_k$. So far we have only guaranteed that λ_2 and λ_3 are positive. If $\lambda_{1k} < 0$ then $\boldsymbol{\Lambda}_k$ is discarded. This ensures that the remaining $\boldsymbol{\Lambda}_k$ are geometrically feasible.

3.7 Recovering $\mathbf{R}$ and $\boldsymbol{t}$

At this point we have a set of up to four geometrically feasible solutions $\boldsymbol{\Lambda}_k$. For each $\boldsymbol{\Lambda}_k$, it remains to determine the corresponding camera pose $(\mathbf{R}_k, \boldsymbol{t}_k)$. The rotation $\mathbf{R}_k$ can be recovered from $ij \in \{12, 13, 23\}$ in (2), but since the differences of the 3D points that appear in the right hand sides of these equations are linearly dependent, these three equations are of rank two. To increase the rank, we can take the cross product of the entries in the left hand side of the first two equations. This must equal the cross product of the corresponding entries in the right hand side:

$$\boldsymbol{z}_1 \times \boldsymbol{z}_2 = \mathbf{R}\Big((\boldsymbol{x}_1 - \boldsymbol{x}_2) \times (\boldsymbol{x}_2 - \boldsymbol{x}_3) \Big), \tag{17}$$

where $\boldsymbol{z}_1 = \lambda_1 \boldsymbol{y}_1 - \lambda_2 \boldsymbol{y}_2$ and $\boldsymbol{z}_2 = \lambda_2 \boldsymbol{y}_2 - \lambda_3 \boldsymbol{y}_3$.

The first two equations in (2) together with (17) gives:

$$\mathbf{Y} = \mathbf{R}\mathbf{X}, \tag{18}$$

$$\mathbf{Y} = \big(\boldsymbol{z}_1,\, \boldsymbol{z}_2,\, \boldsymbol{z}_1 \times \boldsymbol{z}_2\big), \tag{19}$$

$$\mathbf{X} = \big(\boldsymbol{x}_1 - \boldsymbol{x}_2,\, \boldsymbol{x}_2 - \boldsymbol{x}_3,\, (\boldsymbol{x}_1 - \boldsymbol{x}_2) \times (\boldsymbol{x}_2 - \boldsymbol{x}_3)\big), \tag{20}$$

$$\mathbf{R} = \mathbf{Y}\mathbf{X}^{-1}. \tag{21}$$

This solution only provides $\mathbf{R} \in SO(3)$ if the correspondences are exact. This is guaranteed by Eqn (1).

Finally, the translation part of the pose can be solved from any of the three equations in (1):

$$\boldsymbol{t} = \lambda_i \boldsymbol{y}_i - \mathbf{R}\boldsymbol{x}_i. \tag{22}$$

In the end, this produces one pose $(\mathbf{R}_k, \boldsymbol{t}_k)$ that solves (1) for each feasible vector $\boldsymbol{\Lambda}_k$ that is generated by the previous step.

If desired, the corresponding unit quaternions $\boldsymbol{q}_k$ can be extracted given the two 3D correspondences in (17), see Appendix A.2. We use the rotation matrix representation to make the algorithm more closely comparable to the alternatives.

Algorithm 1 Lambda Twist P3P

```
 1: function MIX($\boldsymbol{n}, \boldsymbol{m}$) $= (\boldsymbol{n}, \boldsymbol{m}, \boldsymbol{n} \times \boldsymbol{m})$
 2: function P3P($\boldsymbol{y}_{1:3}, \boldsymbol{x}_{1:3}$)
 3:     Normalize $\boldsymbol{y}_i = \boldsymbol{y}_i / |\boldsymbol{y}_i|$
 4:     Compute $a_{ij}$ and $b_{ij}$ according to (3)
 5:     Construct $\mathbf{D}_1$ and $\mathbf{D}_2$ from (5) and (6)
 6:     Compute a real root $\gamma$ to (8)-(10) of the cubic equation
 7:     $\mathbf{D}_0 = \mathbf{D}_1 + \gamma \mathbf{D}_2$
 8:     $[\mathbf{E}, \sigma_1, \sigma_2] =$ EIG3X3KNOWN0 ($\mathbf{D}_0$). See Algorithm 2
 9:     $s = \pm\sqrt{\frac{-\sigma_2}{\sigma_1}}$
10:     Compute the $\tau_k > 0, \tau_k \in \mathbb{R}$ for each $s$ using Eqn (14) with coefficients in Eqn (15)
11:     Compute $\boldsymbol{\Lambda}_k$ according to Eqn (16), $\lambda_{3k} = \tau_k \lambda_{2k}$ and Eqn (13), $\lambda_{1k} > 0$
12:     $\mathbf{X}_{\text{inv}} = (\text{mix}(\boldsymbol{x}_1 - \boldsymbol{x}_2, \boldsymbol{x}_1 - \boldsymbol{x}_3))^{-1}$
13:     for each valid $\boldsymbol{\Lambda}_k$ do
14:         Gauss-Newton-Refine($\boldsymbol{\Lambda}_k$), see Section 3.8
15:         $\mathbf{Y}_k =$ MIX($\lambda_{1k}\boldsymbol{y}_1 - \lambda_{2k}\boldsymbol{y}_2, \lambda_{1k}\boldsymbol{y}_1 - \lambda_{3k}\boldsymbol{y}_3$)
16:         $\mathbf{R}_k = \mathbf{Y}_k \mathbf{X}_{\text{inv}}$
17:         $\boldsymbol{t}_k = \lambda_{1k}\boldsymbol{y}_1 - \mathbf{R}_k \boldsymbol{x}_1$
18:     Return all $\mathbf{R}_k, \boldsymbol{t}_k$
```

3.8 Implementation Details

We find the diagonalizing γ as a root of the cubic in Eqn (9) by using Newton-Raphson's method initialized using a simple heuristic.

Once the $\boldsymbol{\Lambda}_k$ have been computed they are also refined in accordance with standard praxis[13]. Specifically, we refine the solution using a few verified steps of Gauss-Newton optimization on the sum of squares of Eqn (4). A similar refinement is used in the P3P solver by Ke et al. Note that for both algorithms the accuracy rarely improves after 2 iterations.

This concludes the Lambda Twist P3P algorithm, summarized in Algorithms 1 and 2.

4 Experiments

We have performed three experiments: one to evaluate the numerical accuracy of the proposed method, and two to evaluate the execution time. In order to demonstrate the performance of the proposed method, the experiments were performed on random synthetic data. The proposed method is implemented in C++ and is compared to the state-of-the-art P3P solver by Ke et al. [6], and the P3P solver Kneip et al. [8]. The publicly available C++ implementations of the two state-of-the-art methods have been used. The method by Ke et al. [6] is available as part of OpenCV[10] and the method by Kneip et al. [8] is available at [11]. These solvers both show that they provide superior computational and numerical performance compared to earlier work. The algorithms are compiled with gcc using the same optimization settings as part of the same program. The numerical and time benchmarks are available along with the code on the main authors

Algorithm 2 eig3x3known0

```
1: function GETEIGVECTOR($\boldsymbol{m}, r$ )
2:     $c = (r^2 + \boldsymbol{m}_1\boldsymbol{m}_5 - r(\boldsymbol{m}_1 + \boldsymbol{m}_5) - \boldsymbol{m}_2^2)$
3:     $a_1 = (r\boldsymbol{m}_3 + \boldsymbol{m}_2\boldsymbol{m}_6 - \boldsymbol{m}_3\boldsymbol{m}_5)/c$
4:     $a_2 = (r\boldsymbol{m}_6 + \boldsymbol{m}_2\boldsymbol{m}_3 - \boldsymbol{m}_1\boldsymbol{m}_6)/c$
5:     $\boldsymbol{v} = \big(a_1\ a_2\ 1\big)$
6:     Return $\boldsymbol{v}/|\boldsymbol{v}|$
7: function EIG3X3KNOWN0($\mathbf{M}$)
8:     $\boldsymbol{b}_3 = \mathbf{M}(:,2) \times \mathbf{M}(:,3)$;
9:     $\boldsymbol{b}_3 = \boldsymbol{b}_3/|\boldsymbol{b}_3|$
10:    $\boldsymbol{m} = \mathbf{M}(:)$
11:    $p_1 = \boldsymbol{m}_1 - \boldsymbol{m}_5 - \boldsymbol{m}_9$
12:    $p_0 = -\boldsymbol{m}_2^2 - \boldsymbol{m}_3^2 - \boldsymbol{m}_6^2 + \boldsymbol{m}_1\boldsymbol{m}_5 + \boldsymbol{m}_9 + \boldsymbol{m}_5\boldsymbol{m}_9$
13:    $[\sigma_1, \sigma_2]$ as the roots of $\sigma^2 + p_1\sigma + p_0 = 0$
14:    $\boldsymbol{b}_1 =$ GETEIGVECTOR($\boldsymbol{m}, \sigma_1$)
15:    $\boldsymbol{b}_2 =$ GETEIGVECTOR($\boldsymbol{m}, \sigma_2$)
16:    if $|\sigma_1| > |\sigma_2|$ then
17:        Return $\big([\boldsymbol{b}_1, \boldsymbol{b}_2, \boldsymbol{b}_3],\ \sigma_1,\ \sigma_2\big)$
18:    else
19:        Return $\big([\boldsymbol{b}_2, \boldsymbol{b}_1, \boldsymbol{b}_3],\ \sigma_2,\ \sigma_1\big)$
```

github page. The comparison is performed using a Intel Core i7-6700 3.40 GHz CPU and the code compiled using gcc 4.4 on ubuntu 14.04 with the compile options: -O3 and -march=native.

The implementations by Ke and by Kneip always output four solutions, not all of which are correct. The user must determine which of these are valid. Thus we extend these implementations with a minimal post-processing step. This removes solutions which do not satisfy the geometric feasibility constraint, i.e $\lambda_i > 0$. We also remove solutions which contains NaN or do not approximately contain rotation matrices, i.e. $|\det(\mathbf{R}) - 1| < 10^{-6}$ and $|\mathbf{R}^T\mathbf{R} - \mathbf{I}|_{L1} < 10^{-6}$. This step takes a total of 40ns per sample. This cost is included in the timings since this step must always be performed before the result can be used. Table 1 shows that there is approximately 1.7 unique valid solutions on average per sample. Note that the solutions output by our algorithm intrinsically fulfill these criterion by construction.

We time the system in two ways. First we time the computation of each sample resulting in a time distribution graph. Second we take the total runtime over the same samples providing an average time without the overhead of per sample timing. The timing is performed using high resolution stable timers.

4.1 Synthetic Data

We generate random rigid transforms and random reasonably distributed observations. There are two goals with the sampling. First, the samples should uniformly cover both the image, and the set of rotations. Second, we want substantial depth and translation direction variation within reasonable limits. This resembles data found in the reconstruction of unordered images, a challenging application. Specifically we generate a

P3P observation as follows: The rotation $\mathbf{R}$ is represented by a unit quaternion $\boldsymbol{q} \in \mathbb{R}^4$. Each sample of $\boldsymbol{q}$ is drawn from a isotropic Gaussian distribution in $\mathbb{R}^4$, before being normalized to unit norm and then converted to a matrix. This assures that the rotations have a uniform distribution in $SO(3)$[9]. The translation components $\boldsymbol{t}$ are sampled from a normal distribution with $\sigma = 1$. Combined, $[\mathbf{R}_{gen}, \boldsymbol{t}_{gen}]$ is the generating pose of the sample. Observations are generated by a uniform sampling of the normalized image coordinates (u_i, v_i) in the range [-1,1], and the corresponding 3D point is computed as $\boldsymbol{x}_i = \mathbf{R}^T(\boldsymbol{y}_i z - \boldsymbol{t})$ for a uniform random positive depth $z \in [0.1, 10]$.

Note that P3P is algebraically complete in the sense that it has the same number of constraints as there are free parameters, i.e. it is a minimal solver. Therefore there is no coherent interpretation of adding noise to the observations. We do not remove near degenerate data but instead rely on their presence to strain the algorithms. Strictly degenerate cases, i.e. exactly collinear samples, are removed. It is worth pointing out that this dataset strains the algorithms far more than the "near degenerate" samples experiment of Ke et al., which fails to find the problem cases we find.

The resulting data set consists of 10^7 samples, and the same set is used in all experiments. The experiments are performed sequentially but caching cannot occur due to the size of the dataset. Note, one sample is 104Byte and the full dataset is therefore 1GB.

4.2 Experiment 1: Numerical Accuracy

The numerical accuracy is evaluated by letting the three methods determine their solutions for each sample. Each valid solution $[\mathbf{R}_k, \boldsymbol{t}_k]$ is then compared to the generating pose $[\mathbf{R}_{gen}, \boldsymbol{t}_{gen}]$ of the sample. For each algorithm, the total pose error $\|\boldsymbol{t}_{gen} - \boldsymbol{t}_k\|_{L1} + \|\mathbf{R}_{gen} - \mathbf{R}_k\|_{L1}$ of the solution with the error are compared. We are primarily interested in the error in $\mathbf{R}$ but add the translation error to avoid hiding errors in the computation of $\boldsymbol{t}$. The matrix differences are used to avoid the angle conversion from contaminating the comparison. Similarly the L1 norm is used to reduce numerical error in the norm computation and provide a fair comparison for the smaller errors. Larger errors above 10^{-6} simply indicate failure. The numerical ranking of the algorithms does not change if the L2 norm, or the delta angle, is used. The latter case favors our solution since the $\mathbf{R}$ to quaternion conversion can be avoided in using the method in Appendix A.2.

Figure 2a shows the numerical errors of the three methods. The graph shows that lower errors are more likely with our method than either of the other methods. The number of successes, shown in Table 1, is the number of samples with an error $\leq 10^{-6}$. The table support that failures are significantly less likely with our method. It also shows that the rate of outright failure is far less likely with our algorithm than either of the other two. In particular its interesting to note that Kneip's algorithm has fewer extreme failures than the method by Ke et al. It also turns out the algorithm by Kneip et al. occasionally returns NaN rather than any correct solution. NaNs are replaced by a error of 1 and partially explain the increase at the end of Figure 2a for Kneip's algorithm. Finally we note that for every single sample our error is lower than the errors of both Ke and Kneip. In short, the numerical performance of our solver is substantially better than the other two methods.

4.3 Experiment 2: Execution Time Comparison

We provide each algorithm with identical sequences of synthetic data. Since timing a function interferes with performance, we have sampled both a total time and each sample separately. Figures 2a-2b and Table 1 show that the new method is faster than the method by Ke et al., and that both methods are substantially faster than the method by Kneip et al. Note that excluding the post processing step does not change the ranking. In order to show that the timings are stable the primary comparison has been repeated 1000 times on the same data while keeping the computer otherwise idle. The result is shown in Table 2.

Table 1. Output statistics over 10^7 samples.

Method	Mean[ns]	Successes	Failures	Unique
Lambda Twist	278	9999968	32	16934510
Ke et al. 2 iters	342	9995790	4210	16924141
Ke et al. 10 iters	435	9996105	3895	16925025
Kneip et al.	1042	9994973	5027	16909306

Table 2. Timing statistics over 10^7 samples iterated 1000 times.

Method	Median[ns]	Min[ns]	Max[ns]
Lambda Twist	277	271	289
Ke et al. 2 iters	341	335	353

4.4 Solutions and Successes

First and foremost, Table 1 shows that we have less than one hundredth the number of failures for this data. Further Table 1 shows that we find more unique solutions, i.e. where $|\mathbf{R}_i - \mathbf{R}_j|_{L1} > 10^{-6}$ and $\mathbf{R}_i, \mathbf{R}_j \in SO(3)$, than either of the other methods. This is largely, but not entirely, due to the samples for which the other algorithms do not find any accurate solution. It is worth mentioning that we do not find duplicate solutions, unlike Ke et al. Their algorithm finds about 0.3% duplicates and 36% incorrect solutions in the standard configuration with two root refinement iterations. Interestingly, if the number of root refinement iterations are increased, Ke's methods only finds slightly more solutions. Instead it refines solutions, which otherwise would not satisfy the solution criteria, into duplicates which do. This results in a duplicate ratio of 36%. Arguably, duplicates should be removed in most applications, but this step was not added to the post-processing as they are technically correct.

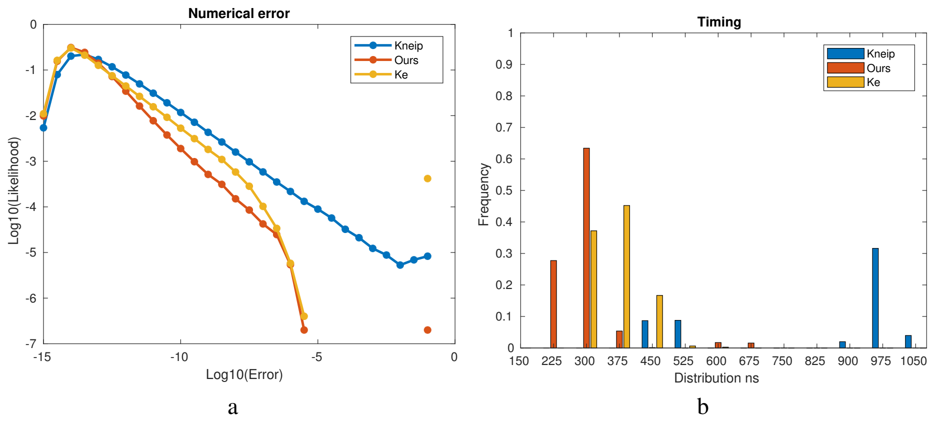

Fig. 2. a Solver numerical precision likelihoods. b Time comparison over 10^7 samples.

We make two additional observations: First, any sample which results in a failure for our method, results in a failure for both other methods, i.e. we are never worse. Second: Every failure for our algorithm and the algorithm of Ke et al. are near known degenerate cases. This is not the case for the algorithm by Kneip et al where a rare cancellation of terms causing a division by zero can occur in a nondegenerate case.

When Ke's algorithm fails, the most common reason is that the method does not ensure that the output matrix is in SO(3) due to numerical noise after the quartic root. The benchmark by Ke et al unfortunately hides this error as it normalizes/converts the resulting rotation matrix to a unit quaternion as part of the evaluation. This step is costly and not included in their timing measurements. The degradation of performance seen here compared to their work is because our metric does not hide this issue. While we understand the appeal of a geometrically interpretable error, we conclude that for the estimation of numerical accuracy, the measure which minimizes error estimation error is preferable.

5 Conclusions

We have presented a novel P3P method with substantially better numerical performance that is faster than previous state-of-the-art. We believe that both the speed and the accuracy are a result of avoiding the quartic and instead going for a cubic. We have shown the performance of the method using extensive synthetic tests. The test stresses the algorithms sufficiently to find failures. Further, the method lets the user select if a unit quaternion or $\mathbf{R} \in SO(3)$ rotation is desired as output without conversion. The method and the benchmark will be available at:
https://github.com/midjji/lambdatwist-p3p.

6 Acknowledgments

This work was funded in part by Daimler Ag, Singulareye AB and Linköping University as well as by CENTAURO: "European Union's Horizon 2020 Programme under Grant Agreement 644839

A Appendix

A.1 Degeneracy of (7)

The two equations in (7) become degenerate when $\mathbf{D}_1$ is a scalar multiplied with $\mathbf{D}_2$, in which case the set of solutions is infinite. Taking into account that $a_{ij} > 0$, a straightforward inspection of the two matrices in (5)-(6) shows that the degenerate case implies $\mathbf{D}_1 = \mathbf{D}_2$. This happens if and only if

$$b_{12} = b_{13} = b_{23} = 0, \quad \text{and} \quad a_{23} = 0. \tag{23}$$

This means that degeneracy requires $a_{23} = 0$, which in turn implies that the 3D points are collinear.

A.2 Quaternion from two 3D Correspondences

The rotation can be found as a unit quaternion q given the two 3D correspondences $\boldsymbol{a}_i, \boldsymbol{b}_i$:

$$\begin{aligned} \boldsymbol{a}_i = \mathbf{R}\boldsymbol{b}_i \iff \begin{pmatrix} 0 \\ \boldsymbol{a}_i \end{pmatrix} = \boldsymbol{q} * \begin{pmatrix} 0 \\ \boldsymbol{b}_i \end{pmatrix} * \boldsymbol{q}^c = \mathbf{Q}\mathbf{B}_i\boldsymbol{q}^c \iff \\ \mathbf{Q}^T\boldsymbol{a}_i = \mathbf{B}_i\boldsymbol{q}^c \iff \mathbf{A}_i\boldsymbol{q}^c = \mathbf{B}_i\boldsymbol{q}^c \implies \\ (\mathbf{A}_i - \mathbf{B}_i)\boldsymbol{q}^c \overset{def}{=} \mathbf{Z}_i\boldsymbol{q} = 0 \implies \\ \begin{pmatrix} \mathbf{Z}_1 & \mathbf{Z}_2 \end{pmatrix} \begin{pmatrix} \mathbf{Z}_1 \\ \mathbf{Z}_2 \end{pmatrix} \boldsymbol{q} \overset{def}{=} \mathbf{K}\boldsymbol{q} = 0 \end{aligned}$$

$\boldsymbol{q}$ is then extracted from $\mathbf{K}$ using a specialized solver which exploits that $\mathbf{K}$ has exactly one zero singular value.

References

1. Agarwal, S., Snavely, N., Simon, I., Sietz, S.M., Szeliski, R.: Building rome in a day. In: Twelfth IEEE International Conference on Computer Vision (ICCV 2009). IEEE, Kyoto, Japan (September 2009), https://www.microsoft.com/en-us/research/publication/building-rome-in-a-day/ 1
2. Gao, X.S., Hou, X.R., Tang, J., Cheng, H.F.: Complete Solution Classification for the Perspective-Three-Point Problem. IEEE Transactions on Pattern Analysis and Machine Intelligence **25**(8), 930–943 (2003) 3
3. Grafarend, E.W., Lohse, P., Schaffarin, B.: Dreidimensionaler Rckwrtsschnitt, Teil I: Die Projectiven Gleichungen. Tech. rep., Geodtisches Institut, Universitt Stuttgart (1989) 3
4. Grunert, J.A.: Das Pothenotische Problem in erweiterter Gestalt nebst ber seine Anwendungen in der Geodsie. In: Grunerts Archiv fr Mathematik und Physik (1841) 3
5. Haralick, R.M., Lee, C., Ottenberg, K.: Analysis and Solutions of The Three Point Perspective Pose Estimation Problem. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition. pp. 592–598 (1999) 3, 4
6. Ke, T., Roumeliotis, S.: An efficient algebraic solution to the perspective-three-point problem. In: Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on. IEEE (2017) 2, 3, 9
7. Klein, G., Murray, D.: Parallel tracking and mapping for small AR workspaces. In: Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07). Nara, Japan (November 2007) 1
8. Kneip, L., Scarmuzza, D., Siegwart, R.: A Novel Parametrization of the Perspective-Three-Point Problem for a Direct Computation of Absolute Camera Position and Orientation. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition. pp. 4546–4553 (2011) 2, 3, 9
9. Muller, M.E.: A Note on a Method for Generating Points Uniformly on N-Dimensional Spheres. Comm of ACM **2**(4), 19–20 (1959) 11
10. Web page: https://opencv.org/ 9
11. Web page: http://www.laurentkneip.com/software/ 9
12. Persson, M., Piccini, T., Mester, R., Felsberg, M.: Robust stereo visual odometry from monocular techniques. In: IEEE Intelligent Vehicles Symposium (2015) 1
13. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C++: The Art of Scientific Computing. Cambridge University Press (2002) 9