Fast Feature-Less Quaternion-based Particle Swarm Optimization for Object Pose Estimation From RGB-D Images

Giorgio Toscana
giorgio.toscana@polito.it
Stefano Rosa
stefano.rosa@polito.it
Politecnico di Torino, Turin, Italy

Abstract

We present a novel quaternion-based formulation of Particle Swarm Optimization for pose estimation which, differently from other approaches, does not rely on image features or machine learning. The quaternion formulation avoids the gimbal lock problem, and the objective function is based on raw 2D depth information only, under the assumption that the object region is segmented from the background. This makes the algorithm suitable for pose estimation of objects with large variety in appearance, from lack of texture to strong textures, for the task of robotic grasping. We find candidate object regions using a graph-based image segmentation approach that integrates color and depth information, but the PSO is agnostic to the segmentation algorithm used. The algorithm is implemented on GPU, and the nature of the objective function allows high parallelization. We test the approach on different publicly available RGB-D object datasets, discuss the results and compare them with other existing methods.

1 Introduction

Detection and pose estimation of 3D objects is of great importance in robotics applications for many high level tasks such as manipulation, grasping, and also localization and mapping. Affordable depth sensors, like the Kinect, have been of great interest to the robotics community. These new sensors are able to simultaneously capture high-resolution color and depth images at high frame rates (RGB-D images). We focus on the problem of object detection for robotic grasping, and in particular we are most interested in the Amazon Picking Challenge (APC)[1] scenario.

A well known approach is LINEMOD [9]. It exploits both depth and color images to capture the appearance and 3D shape of the object using a set of templates covering different views of an object. Since the viewpoint of each template is known, a coarse estimate of the object pose is available upon detection. Templates are learned online, and the pose estimates are not very precise, since a template covers a range of views around its viewpoint. 3D object models were exploited [10], improving the accuracy of pose estimation and lowering false positives. In [16] LINEMOD is adapted to be a scale-invariant patch descriptor and integrated into a regression forest, trained with positive samples only. Tests of LINEMOD

© 2016. The copyright of this document resides with its authors.
It may be distributed unchanged freely in print or electronic forms.

showed only 32% accuracy in an APC-like scenario [14]. In [11] the approach is extended with a voting procedure based on hashing for selecting candidate templates. [7] is based on training random forests with local features, while in [6] occlusion information is also added in the learning phase.

Other commonly used approaches, such as tabletop from the Point Cloud Library [4], are based on a combination of coarse detection using 3D feature descriptors and fine pose estimation using ICP. Some of these approaches rely on object textures and are not suitable for many common texture-less objects. Moreover, all these methods are limited to objects lying on a flat tabletop, and are generally not robust to occlusions.

We propose a pose estimation algorithm for simple objects which does not rely on 2D or 3D features and does not require any training phase. Given a candidate object segmented from the RGB-D image, the object's pose is estimated using *Particle Swarm Optimization* (PSO). The contributions of this work are a novel quaternion-based formulation of the standard PSO equations, the design of an objective function for pose estimation which exploits depth information only and a fast GPU implementation. A number of GPU implementations of PSO have been proposed (e.g., [13], [5], [8]). They all consider different types of particles' topology. Our own GPU implementation is loosely based upon [13], [5] and [8] as explained in Section 4.

In Section 2 we introduce the problem; in Section 3 we describe the pose estimation part; in Section 4 we explain the GPU implementation in detail; in Section 5 we present and discuss experimental results and finally in Section 6 we draw conclusions and discuss future work.

2 Background

Particle Swarm Optimization (PSO) [12] is an heuristic technique inspired by the swarming or collaborative behavior of biological populations. It is useful for exploring the search space of a problem to find the settings or parameters required to maximize a particular objective. A set of candidate solutions (particles) $\mathbf{p}_i(t) = (\mathbf{x}_i(t), \mathbf{v}_i(t))$, where $\mathbf{x}_i$ and $\mathbf{v}_i$ are the position and velocity of particle i at time t, is maintained in the search space. The algorithm consists of three steps, which are repeated until some stopping condition is met: first, the fitness of each particle is evaluated, then individual and global best fitnesses and positions are updated; finally velocity and position are updated for each particle. In the update phase the velocity is computed as follows:

$$\mathbf{v}_i(t+1) = w\mathbf{v}_i(t) + c_1 r_1 [\mathbf{x}_{pbest}(t) - \mathbf{x}_i(t)] + c_2 r_2 [\mathbf{x}_{gbest}(t) - \mathbf{x}_i(t)], \tag{1}$$

where w, c_1, c_2 (with $0 \leq w \leq 1.2$, $0 \leq c_1 \leq 2$, $0 \leq c_2 \leq 2$) are tunable parameters; r_1, r_2 are random values. $\mathbf{x}_{pbest}(t)$ is the best candidate solution for the particle $\mathbf{p}_i$ at time t and $\mathbf{x}_{gbest}(t)$ is the global best candidate solution at time t. The particle position is then computed as:

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1). \tag{2}$$

3 Pose estimation

To estimate the 6DoF object pose we use a quaternion-based formulation of the standard PSO equations (1) and (2). We use unitary quaternions to describe the orientation of an object in 3D space since they are *gimbal-lock* free and they have a well-defined interpolation

formula (SLERP) [15]. Gimbal-lock would produce wrong results when the fitness function of a particle is computed; moreover, other rotation formalisms would require the conversion to and from rotation matrix form at each step. Unitary quaternions, however, turn the optimization problem into a constrained one. We design the new PSO equations such that the explored orientations are always represented by unit-length quaternions. This means that every particle, in every time instant, holds a valid object pose hypothesis. For now on, when we talk about quaternions we refer to unit-length quaternions. Both quaternions $\mathbf{q} = [\, q_0, \vec{q}\,]$ and -$\mathbf{q} \in \mathbb{S}^3$ define the same orientation. To overcome this ambiguity we cast the quaternion to the northern hemisphere of $\mathbb{S}^3$, i.e., we ensure that the scalar part of a quaternion is always positive or equal to zero $(q_0 \geq 0)$.

3.1 Angular velocity and orientation update

The standard velocity update equation (1) describes a weighted sum of three vectors in Euclidean space. The current linear velocity of an object is expressed by the vector $w\mathbf{v}_i(t)$. The object *cognitive* linear velocity is given by vector $c_1 r_1 \left[\mathbf{x}_{pbest}(t) - \mathbf{x}_i(t)\right]$ and the *social* linear velocity acting on an object is $c_2 r_2 \left[\mathbf{x}_{gbest}(t) - \mathbf{x}_i(t)\right]$. This, along with (2), is used to optimize the position component of the object pose. In (1) velocities are computed as the difference between two position vectors. Subtraction has no meaning for unit-length quaternions so a new equation must be derived to obtain the object's angular velocity based on the current and best orientations of an object. The goal is to obtain both the cognitive and social angular velocity effecting an object, through the *quaternion inverse displacement*.

Let $\mathbf{q}_0$, $\mathbf{q}_1 \in \mathbb{S}^3$ and $t \in \mathbb{R}$ with $0 \leq t \leq 1$, the SLERP and its derivative are defined as:

$$Slerp\,(\mathbf{q}_0, \mathbf{q}_1, t) = \mathbf{q}(t) = (\mathbf{q}_1 \star \mathbf{q}_0^*)^t \star \mathbf{q}_0 \tag{3}$$

$$\begin{aligned} \frac{\mathrm{d}Slerp}{\mathrm{d}t} = \dot{\mathbf{q}}(t) =& \mathrm{Log}\,(\mathbf{q}_1 \star \mathbf{q}_0^*)\,(\mathbf{q}_1 \star \mathbf{q}_0^*)^t \star \mathbf{q}_0 = \\ =& \mathrm{Log}\,(\mathbf{q}_1 \star \mathbf{q}_0^*) \star \mathbf{q}(t), \end{aligned} \tag{4}$$

where the superscript $*$ is the quaternion conjugate operator, the symbol $\star$ defines the quaternion product and the Log operator is the *logarithmic map*. From the quaternion kinematics we can write the derivative of $\mathbf{q}$ in t as:

$$\dot{\mathbf{q}}(t) = \frac{1}{2}\mathbf{q}(t) \star \omega(t), \tag{5}$$

where $\omega(t)$ is the instantaneous angular velocity vector acting on the object. In (5), $\omega(t)$ is the augmented angular velocity with scalar part equal to zero, i.e., $\omega(t) = [0,\ \omega_x,\ \omega_y,\ \omega_z]^T$. The instantaneous angular velocity needed to rotate the object from the initial orientation $(\mathbf{q}_0)$ to the final one $(\mathbf{q}_1)$ is obtained combining (5) and (4):

$$\omega(t) = 2\mathrm{Log}\,(\mathbf{q}_1 \star \mathbf{q}_0^*) \tag{6}$$

Eq. (6) shows how the angular velocity remains constant throughout the quaternion interpolation since its value only depends on the *quaterion error* $(\mathbf{q}_1 \star \mathbf{q}_0^*)$. Moreover, we are dealing with quaternions belonging only to the northern hemisphere of $\mathbb{S}^3$. This aspect ensures that the SLERP represents the shortest arc between $\mathbf{q}_0$, $\mathbf{q}_1$. Hence, the obtained angular velocity is the optimal one. The logarithmic map for a unit-length quaternion reduces to:

$$\mathrm{Log}(\mathbf{q}) = \left[0,\ \frac{\vec{q}}{\parallel \vec{q} \parallel} \arccos(q_0)\right] \tag{7}$$

Eq. (6) can now be rewritten as:

$$\tilde{\omega} = 2\frac{\vec{\tilde{q}}}{\| \vec{\tilde{q}} \|}\arccos(\tilde{q}_0) \tag{8}$$

where $\tilde{\mathbf{q}} = \mathbf{q}_1 \star \mathbf{q}_0^* = [\, \tilde{q}_0, \vec{\tilde{q}} \,]$. The angular velocity update equation for the i-th particle is formulated as follows:

$$\begin{aligned} \omega_i(t+1) = w\omega_i(t) + \\ c_1 r_1 \left[2\mathrm{Log}\left(\mathbf{q}_{pbest_i}(t) \star \mathbf{q}^*_{current_i}(t)\right)\right] + \\ c_2 r_2 \left[2\mathrm{Log}\left(\mathbf{q}_{gbest}(t) \star \mathbf{q}^*_{current_i}(t)\right)\right] \end{aligned} \tag{9}$$

The orientation of the i-th particle is then updated by means of the discrete form of the quaternion kinematics:

$$\mathbf{q}_i(t+1) = \cos(\psi(t))\,\mathbf{q}_i(t) + \frac{1}{2}\frac{\sin(\psi(t))}{\psi(t)}\mathbf{q}_i(t) \star \omega_i(t+1)T_c, \psi(t) = \|\omega_i(t+1)\|_2 \frac{T_c}{2} \tag{10}$$

T_c represents the integration time of the discrete time quaternion kinematics. In this work T_c just collapses to a tunable parameter as we are dealing with iteration steps (t) rather than with the true definition of time. This new parameter could be employed to scale the total angular velocity obtained in (9). In this way, T_c can control the amount of perturbation to apply to the current object orientation.

3.2 Objective function

Each particles' object pose hypothesis must be checked against a fitness function to estimate how close that particle is from the true pose of the real object. The algorithm takes as inputs a set of regions $\mathcal{R}$ generated by a segmentation algorithm; each one is a cluster of pixels that represents an object. The depth map of that cluster is also extracted and it is the only source of information used in the PSO algorithm. In our work we use the graph-based RGB-D segmentation algorithm described in [17]. The segmentation algorithm also does not rely on image features nor machine learning and uses a modified Canny edge detector for extracting robust edges by combining depth and color cues. The edges are used to build an undirected graph, which is partitioned using the concept of internal and external differences between graph regions.

Each particle renders its pose hypothesis against the depth map of the cluster. The fitness value of the j-th particle is thus computed as follows:

$$\Phi_j = \frac{\alpha}{N_{R_j}} \sum_{i=1}^{N_{R_j}} \left(z_{K_i} - z_{R_{ij}}\right)^2 + \beta\,\frac{\mu_j + \kappa_j}{2} \tag{11}$$

where: N_{R_j} is the number of pixels of the depth map rendered by the j-th particle, $z_{R_{ij}}$ is the depth value of the pixel i rendered by the j-th particle, while z_{K_i} is the corresponding depth value of the cluster at pixel i. α and β are two constant parameters used to weight the two terms of the fitness function. In the fitness function, a second term along with the depth error one is needed. Depth error alone might generate ambiguity, leading to wrong pose estimation in some special cases (e.g., a box could fit one of its smaller sides against its largest size that is visible in the segmented cluster, giving a depth error close to zero even

if the particle's pose is wrong). The term μ_j models the percentage of cluster pixels that are not covered by the rendered 3D model of particle j:

$$\mu_j = \frac{N_{CW_j}}{N_{PC}} \in [0,1]. \quad \text{If } \mu_j = \begin{cases} 0 & \text{perfect match} \\ 1 & \text{the rendered object is outside the cluster} \end{cases}$$

where N_{CW_j} is the cluster's area (in pixels) that is not covered by any pixel of the rendered object of the particle j and N_{PC} is the area of the segmented cluster. The condition $\mu_j = 0$ could also hold when the rendered object has a larger area than the cluster and it is covering the entire cluster. Hence, the term κ_j is added to compensate for this problem.
κ_j is the complement of μ_j i.e., it gives the percentage of rendered pixels of particle j that are not covered by valid depth values in the cluster depth map:

$$\kappa_j = \frac{N_{RW}}{N_{R_j}} \in [0,1]. \quad \text{If } \kappa_j = \begin{cases} 0 & \text{perfect match} \\ 1 & \text{the rendered object is outside the cluster} \end{cases}$$

where N_{RW} is the rendered object's number of pixels which do not correspond to pixels of the segmented cluster.

3.3 PSO initialization

The PSO requires an initialization step in which different object pose hypotheses are assigned to each particle. The segmentation phase provides a rough approximation of the 3D centroid of a cluster. The latter is not necessarily the 3D centroid of the real object as errors in the clustering step might lead to either over-segmentation (e.g., an object is split in two or more clusters), or under-segmentation (e.g., a cluster does not enclose the whole object; thus object borders or even small parts of an object are missing). However, the 3D centroid of a cluster ($\bar{\mathbf{c}}$) can be exploited to generate the position component of the j-th particle ($\mathbf{t}_j$) as follows:

$$\mathbf{t}_j = \hat{\mathbf{t}}_{lo} + \left(\hat{\mathbf{t}}_{hi} - \hat{\mathbf{t}}_{lo}\right)\delta \tag{12}$$

$$\mathbf{v}_j = -\tilde{\mathbf{v}} + 2\tilde{\mathbf{v}}\delta \tag{13}$$

$$\hat{\mathbf{t}}_{lo} = \bar{\mathbf{c}} - \tilde{\mathbf{t}}; , \hat{\mathbf{t}}_{hi} = \bar{\mathbf{c}} + \tilde{\mathbf{t}};$$

where: $\delta \backsim \mathcal{U}(0,1)$; $\tilde{\mathbf{t}}$ is a constant relative position vector used to define the search space domain of the translation component of the object pose. Eq. (13) assigns a linear velocity, between the constant values $\pm\tilde{\mathbf{v}}$, to the particle j.

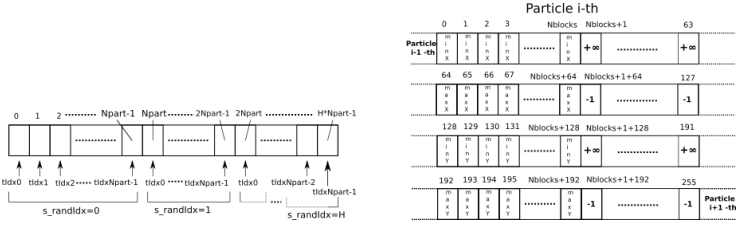
The segmentation step cannot generate an estimate of the object orientation, so the object attitude initialization and optimization are performed on the whole surface of the northern hemisphere of $\mathbb{S}^3$. Let $\mathbf{q}_{init} \in \mathbb{S}^3$ be a constant attitude quaternion lying on the surface of the northern hemisphere of $\mathbb{S}^3$, the initial orientation of the j-th particle is generated as follows:

$$\mathbf{q}_j = \cos(\psi)\,\mathbf{q}_{init} + \frac{1}{2}\frac{\sin(\psi)}{\psi}\mathbf{q}_{init} \star \widehat{\omega}_j T_c \tag{14}$$

$$\widehat{\omega}_j = \widetilde{\omega}_{lo} + \left(\widetilde{\omega}_{hi} - \widetilde{\omega}_{lo}\right)\delta \tag{15}$$

$$\psi = \|\widehat{\omega}_j\|_2 \frac{T_c}{2}$$

where Eq. (15) assigns a random angular velocity to the particle j. The dynamic range of the initial angular velocity is limited by the values $\widetilde{\omega}_{lo}$ and $\widetilde{\omega}_{hi}$. The larger the difference



(a) (b)

Figure 1: (a) ***d_randGen*** array layout. (b) Layout of ***d_AABB*** for the particle *i*-th.

$(\widetilde{\omega}_{hi} - \widetilde{\omega}_{lo})$, the greater will be the perturbation of the initial attitude quaternion $\mathbf{q}_{init}$. Experiments show that the choice of $\mathbf{q}_{init}$ has no influence on the convergence of the PSO as long as a wide dynamic range of the initial angular velocity is provided. This result corroborates the fact that our algorithm converges to the actual object pose without any prior knowledge about the object attitude.

We also experimentally determined that the final fitness value of the best particle can be used to discriminate correctly detected objects from false positives. This is necessary, since the segmentation part inevitably produces a number of false positive regions.

4 GPU-based Particle Swarm pose optimization

Our own GPU implementation is loosely based upon [18], [5] and [8], as our approach incorporates a rendering phase just before the evaluation of the fitness function. Moreover, the fitness function gathers data from the rendering process to compute the quality of a particle's object pose hypothesis. These dissimilarities from the standard PSO algorithm force us to define new data structures and new optimization techniques in order to develop an efficient quaternion-based PSO algorithm on modern GPUs.

4.1 PSO Initialization on GPU

Particles' current pose, best pose and the current velocity are set according to eqs. (12), (13), (14) and (15). These equations require a random number sampled by the uniform probability distibution (δ). We chose to generate the random numbers in CPU and load them in GPU during intialization; this guarantees backward compatibility with older CUDA versions. Let the dimensions of the search space be DIM and let the swarm size be $NPART$, with $H >> DIM$. We ensure no reduction in performances by adopting this technique since a global memory coalesced access is performed on the ***d_randGen*** array (Figure 1a). A float array of length $[H * NPART]$ is initialized in host memory with random numbers. It is then uploaded in device global memory (***d_randGen***).

Inside the $initAllParticles()$ kernel, each thread has a unique ID ($tIdx$); moreover, each thread increments a local variable (***s_randIdx***) when a new random number is needed. The coalesced memory access shown in figure 1a is achieved by reading the global memory as ***d_randGen***[*tIdx*+(***s_randIdx***++)*NPART].

4.2 Rendering of the particles pose hypotesis

Each particle holds an object pose hypothesis that must be used to render the object model onto the image plane in order to compute the particle's fitness. In our algorithm the rendering is done directly in GPU. The OpenGL rendering pipeline would have slowed down the entire optimization process, since OpenGL functions cannot be called inside a kernel function. Using OpenGL, all the particles pose should have been loaded back to CPU. OpenGL should have performed the rendering of each particle sequentially onto a new depth buffer, to avoid any depth value overwriting. Finally, the written depth buffer, allocated to each particle, should have been uploaded to GPU in order to calculate the fitness score of the pose hypotheses concurrently. The software rendering pipeline allows parallel rendering of the particles and avoids time consuming memory copy between CPU and GPU at each iteration.

Our rendering pipeline is based on the optimized version of the **edge function** as explained in [13]. This technique allows a fast and parallel processing of the object mesh triangles. Hence, the particles rendering phase offers two levels of parallelism: each particle renders its object model independently from the others and the rendering algorithm of that particle is able to process many triangles at once. This ensures high occupancy of the GPU. Achieving the two levels of parallelism inside a CUDA kernel is not straightforward due to the SIMT (Single Instruction - Multiple Threads) computing architecture offered by NVIDIA. To solve this problem we leverage the CUDA streams. Inside the rendering kernel we assign one thread per triangle mesh while we are launching 16 streams. In an ideal case the number of streams should be equal to the number of particles obtaining a complete parallelism among them. In the real case this cannot be achieved due to hardware limitations. In our experiments opening more than 16 streams does not reduce anymore the time of the rendering phase.

In the rendering phase we are also interested in finding the axis-aligned bounding box (AABB) of the rendered object onto the depth buffer. The AABBs are necessary when the fitness score of a particle is computed. To exploit all the parallelism offered by GPUs we calculate the AABB of all the particles at once using the optimized version of the parallel *reduction* technique (see [3]). During the rendering stage we save the AABB of each rendered triangle belonging to the object mesh in shared memory. Four shared memories are needed in order to store separately the x and y coordinates of both the top-left and bottom-right corners of the rectangles. A *minimum* parallel reduction is then performed on the coordinates of the top-left rectangles' corners while a *maximum* parallel reduction is executed on the bottom-right ones. The final AABB of a particle cannot be computed inside the rendering kernel without using any atomic operation. This happens because shared memories are only accessible from all the threads within the block. The shared memory allocated for a block cannot be read by other blocks. Our solution is to store in a temporary array (***d_AABB***), in global memory, all the final AABBs of each block obtained after parallel reduction. The layout of the ***d_AABB*** can be seen in figure 1b. We reconstruct the actual AABB of a particle by launching only a kernel with $Nblocks = NPART$ and $64 * 4 = 256$ threads per block. The above configuration along with the particular ***d_AABB*** layout guarantee a coalesced memory access and a very fast parallel reduction. The final AABB of each particle is stored in ***d_finalAABB*** where only 4 out of the 32 elements reserved per particle are used.

4.3 Fitness function on GPU

We compute the fitness function 11 launching *NPART* times the designate kernel through 16 streams. Each kernel handles only a particle and runs a block of 1024 threads. This kernel uses a *grid-stride loop* ([2]) to access 1024 pixels at once. We loop only within the AABB obtained by the **OR**-operation between the particle's AABB and the AABB of the segmented depth cluster. A parallel sum reduction within the same kernel is then employed to execute the final summation in 11, since only 1 block per particle is used.

4.4 Updating personal and global best on GPU

Our quaternion-based PSO employs the *global* topology where a fully-connected arrangement lets the particles share information globally. Beforehand, a minimum parallel reduction is performed among the fitness scores assigned to the particles. The particle with the minimum fitness score is elected as the best particle of that iteration. A suitable kernel spawns *NPART* threads and performs the particles' personal update concurrently. To avoid atomic operations in updating the particle global best, we test only the current best particle score against the best score found until that iteration. If the strict minimum condition is true, the current best particle is further elected as the best particle until that iteration. Finally, the update of the particles pose and velocity in Eqs. (10), (9), (1) and (2) is completely parallelized by using a thread per particle.

5 Experimental results

We tested our approach on two public datasets for 3D pose estimation [9] and [16]. The software has been developed using the CUDA library in C++ under Linux and runs on GPU. The source code will be available online. The algorithm was tested on a workstation equipped with a GeForce GTX Titan X GPU. All the 3D models were decimated to 3072 faces, which offers the best time performance with our hardware setup. This is a tradeoff, since subsampling the 3D object will have a slight effect on the accuracy of pose estimation.

[10] contains 15 registered video sequences, each with a texture-less 3D object surrounded by clutter. In [16], 6 objects are captured under varying viewpoint with lots of background clutter, scale and pose changes, and in particular foreground occlusions and multi-instance representation (three instances of the same object are present in each frame as well as other objects and clutter). We tested the approach on a subset of objects and scenes. Results are shown in Table 1. We compare the results with ground truth using the metric from [10]. For all runs we used the following fixed parameters: $\gamma = 0.001, k_x = 1.2, k_b = 0.05, \alpha = 1, \beta = 0.05, T_c = 1, \widetilde{\omega}_{lo} = [0,-10,-10,-10]^T, \widetilde{\omega}_{hi} = [0,10,10,10]^T, c_1 = c_2 = 1, w = 0.3, \tilde{\mathbf{t}} = [0.3,0.3,0.3], \tilde{\mathbf{v}} = [3,3,3]$.

The segmentation algorithm runs on CPU and the average processing time per image is 0.4s; the pose estimation part runs on GPU. In our experiments we used 1024 particles for the PSO and run 10 PSO iterations for each segmented cluster and the total time is 85ms for each cluster. We use the global topology in all the experiments. By comparison, [10] requires a training stage of 17-50s for each object and 119ms for detecting an object, but with position and rotation constraints to achieve this speed (0-90° for tilt, ±45° for in-plane rotation, 65-115cm for scaling). Figure 2 shows some examples of the algorithm on different datasets. In Figure 3 we show some failure cases and discuss the probable causes.

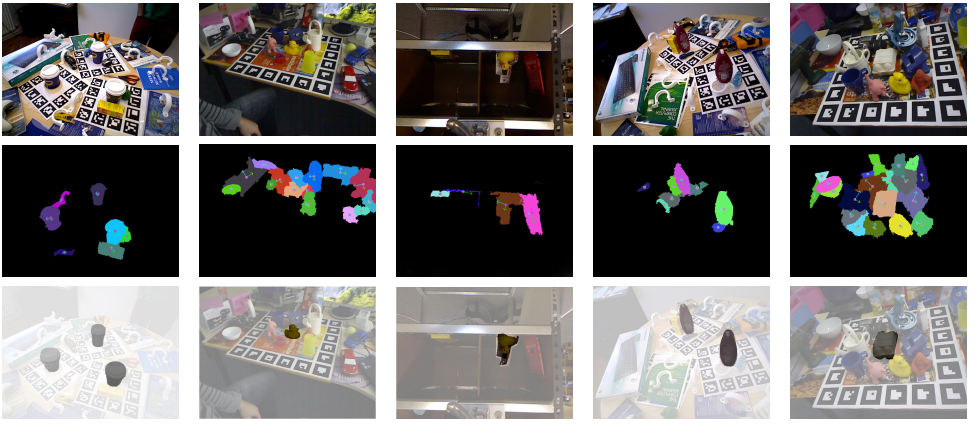


Figure 2: Examples of the approach on different images. First row: RGB images; second row: segmented objects; third row: detected object superimposed.

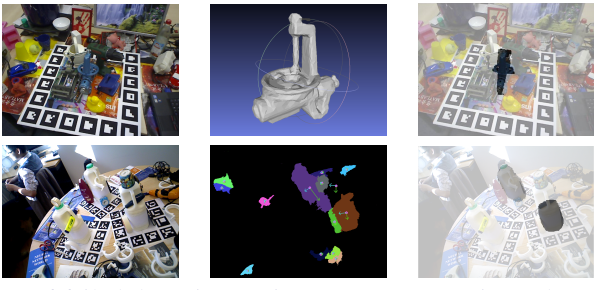


Figure 3: Examples of failed detections. First row: wrong estimated pose probably due to the complex model and its over-simplification after subsampling; second row: a case with a true positive (top object), a false positive (left object) and a wrong pose estimation (right object) both due to bad segmentation.

The algorithm is also adaptable to articulated objects. In Figure 4 we show an example of the PSO running on an object (laptop) composed by two parts (laptop base, laptop screen) joined by an revolute joint.

6 Conclusion

We presented a fast approach for estimating the pose of simple objects from RGB-D images for robotic grasping tasks.

A Particle Swarm Optimization algorithm, with a novel quaternion-based kinematics formulation, is run on candidate object regions, extracted from the input image by a segmentation algorithm, using a 3D CAD model of the object. The objective function is based on raw depth information, as well as the contour of the object. Moreover, the final fitness value can be used to further discard false positive regions from the segmentation part. The PSO exploits the parallelization offered by GPU architecture and is able to run 10 iterations at more than 10 fps on a modern GPU. Future work will be devoted to the full extension of the algorithm to articulated objects with arbitrary number of links and experiments with

Approach	[9]	[16]	Our Appr.
Sequence			
Bench Vise	0.85	0.96	0.72
Driller	0.69	0.9	0.9
Phone	0.56	0.73	0.98
Duck	0.58	0.91	0.97
Eggbox	0.86	0.74	0.95
Glue	0.44	0.68	0.8

Approach	[9]	[16]	Our Appr.
Sequence			
Coffee Cup	0.82	0.88	0.85
Shampoo	0.63	0.76	0.9
Juice Carton	0.49	0.87	0.4
Milk	0.18	0.39	0.67

Table 1: (a) Comparison between different approaches on the[10] dataset. (b) Comparison between different approaches on the[16] dataset.

different segmentation algorithms.

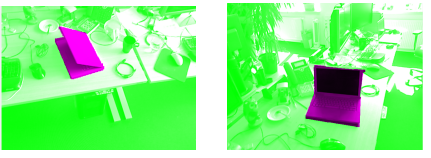


Figure 4: Examples of the algorithm running on a 2-parts articulated object.

7 Acknowledgements

This work was done in collaboration with TIM S.p.A.; the GPU used for this research was donated by the NVIDIA Corporation.

References

[1] Amazon picking challenge. Website. http://amazonpickingchallenge.org.

[2] Cuda grid-stride loop. Website. https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/.

[3] Cuda sum reduction. Presentation. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf.

[4] Aitor Aldoma, Federico Tombari, Radu Bogdan Rusu, and Markus Vincze. *Pattern Recognition: Joint 34th DAGM and 36th OAGM Symposium, Graz, Austria, August 28-31, 2012. Proceedings*, chapter OUR-CVFH – Oriented, Unique and Repeatable Clustered Viewpoint Feature Histogram for Object Recognition and 6DOF Pose Estimation, pages 113–122. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-32717-9. doi: 10.1007/978-3-642-32717-9_12.

[5] Carmelo Bastos-Filho, Débora Nascimento, and Marcos Oliveira Junior. *Running particle swarm optimization on graphic processing units*. INTECH Open Access Publisher, 2011.

[6] Ujwal Bonde, Vijay Badrinarayanan, and Roberto Cipolla. Robust instance recognition in presence of occlusion and clutter. In *European Conference on Computer Vision*, pages 520–535. Springer, 2014.

[7] Eric Brachmann, Alexander Krull, Frank Michel, Stefan Gumhold, Jamie Shotton, and Carsten Rother. Learning 6d object pose estimation using 3d object coordinates. In *European Conference on Computer Vision*, pages 536–551. Springer, 2014.

[8] L de P Veronese and Renato A Krohling. Swarm's flight: accelerating the particles using c-cuda. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 3264–3270. IEEE, 2009.

[9] S. Hinterstoisser, S. Holzer, C. Cagniart, S. Ilic, K. Konolige, N. Navab, and V. Lepetit. Multimodal templates for real-time detection of texture-less objects in heavily cluttered scenes. 2011.

[10] Stefan Hinterstoisser, Vincent Lepetit, Slobodan Ilic, Stefan Holzer, Gary Bradski, Kurt Konolige, and Nassir Navab. Model based training, detection and pose estimation of texture-less 3d objects in heavily cluttered scenes. In *Computer Vision–ACCV 2012*, pages 548–562. Springer, 2012.

[11] T. HodaÅĹ, X. Zabulis, M. Lourakis, Åă ObdrÅ¿Ãąlek, and J. Matas. Detection and fine 3d pose estimation of texture-less objects in rgb-d images. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 4421–4428, Sept 2015. doi: 10.1109/IROS.2015.7354005.

[12] James Kennedy. Particle swarm optimization. In *Encyclopedia of Machine Learning*, pages 760–766. Springer, 2010.

[13] Juan Pineda. A parallel algorithm for polygon rasterization. In *ACM SIGGRAPH Computer Graphics*, volume 22, pages 17–20. ACM, 1988.

[14] Colin Rennie, Rahul Shome, Kostas E. Bekris, and Alberto F. De Souza. A dataset for improved rgbd-based object detection and pose estimation for warehouse pick-and-place. *CoRR*, abs/1509.01277, 2016.

[15] Ken Shoemake. Animating rotation with quaternion curves. In *ACM SIGGRAPH computer graphics*, volume 19, pages 245–254. ACM, 1985.

[16] Alykhan Tejani, Danhang Tang, Rigas Kouskouridas, and Tae-Kyun Kim. Latent-class hough forests for 3d object detection and pose estimation. In *Computer Vision–ECCV 2014*, pages 462–477. Springer, 2014.

[17] Giorgio Toscana and Stefano Rosa. Fast graph-based object segmentation for rgb-d images. *CoRR*, abs/1605.03746, 2016.

[18] You Zhou and Ying Tan. Gpu-based parallel particle swarm optimization. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 1493–1500. IEEE, 2009.