Cartesian Path Planning for Arc Welding Robots: Evaluation of the Descartes Algorithm

Jeroen De Maeyer, Bart Moyaers, Eric Demeester
Departement of Mechanical Engineering
KU Leuven
Technology Campus Diepenbeek, Belgium
Email: jeroen.demaeyer@kuleuven.be

***Abstract*—Many industrial robot applications require fewer task constraints than the robot's degrees of freedom. For welding robots, for example, rotations of the welding torch around its axis do not negatively impact welding quality. Furthermore, the tool center point's Cartesian position and desired orientation as a function of time is often determined by the (manufacturing) process. Nevertheless, programming these robots can be time consuming. Reducing or eliminating this programming cost will allow robots to be used for producing small series. Recently, a promising software package for Cartesian path planning with the name *Descartes* was released by the ROS-Industrial community. To the authors' knowledge, an in-depth description of this algorithm and an experimental evaluation is lacking in literature. This paper describes the path planning approach used by the Descartes package. Moreover, the software's performance is evaluated for several key robot welding tasks and the encountered limitations are discussed. In addition, we show that the planner's performance can be improved by changing the cost function that the planner's graph search algorithm minimises.**

I. Introduction

In 2016, more than two million industrial robots were in operation around the world [1]. Programming these robots can be time consuming. This is a limiting factor if robots are to be adopted for the production of small batch sizes. Automating the programming process will increase the cost efficiency of robots in those applications, provided that collision-free robot programs are generated that adhere to the manufacturing process's requirements. Common tasks for industrial robots result in kinematic redundancy. For example, when using a 6 degrees of freedom (DOF) robot arm for arc welding, the rotation of the tool (welding torch) is not completely defined by the process. The task only restricts 5 out of the 6 available DOF. Many other applications have such kinematic redundancy, including grinding, deburring and painting. Mounting these robots on a rail further increases the degrees of freedom of the robot, thereby also increasing the kinematic redundancy. Moreover, in many of these applications, the position of the tool center point (TCP) and often also the TCP's desired orientation is known as a function of time in order to optimally execute the task. In welding applications, for example, the welding torch's Cartesian position is determined by the position of the pieces that need to be welded together, and its velocity and desired orientation are prescribed by the welding process and relative location of the work pieces. Often, acceptable deviations from these optimal Cartesian positions, velocities and orientations are dictated by the manufacturing process as well. This paper focuses on the automatic generation of collision-free programs for arc welding robots, but most conclusions are valid for similar applications.

For applications with these characteristics, path planning[1] in operational space, often called *Cartesian path planning*, is required. As mentioned above, Cartesian path planning needs to deal with kinematic redundancy, which can be exploited to avoid collisions or stay away from singular configurations. Two main approaches exist. The first, called task augmentation [3], adds constraints to reduce or even avoid the redundancy. Most industrial robots have software included to control end effector motion in operational space. When using task augmentation to resolve the kinematic redundancy, the Cartesian path can be directly executed by the robot controller. This approach is used by [4] to calculate a robot program based on an analytical description of objects involved in a welding task. The second approach formulates a secondary objective in addition to the kinematic task constraints. This results in an optimisation problem that can be solved locally for each trajectory point [5] [2] [6] or globally for the whole trajectory [3]. Both methods can be combined. The optimization problem can also be formulated as a graph search problem. A formal description of this approach is given in [7].

Path planning algorithms can be implemented and tested in different open source software environments. Examples are the Robotics Library (RL) [8], OpenRave [9] and Robotic Operating System (ROS) [10]. Within the latter one, there is an industrial consortium, called ROS-Industrial, which recently published a software package, Descartes, that implements a graph based path planning algorithm, and which is specifically designed for applications such as robot welding. Where a path is given in operational space and the pose for the end effector is under-defined. The source code is available on the Github page of ROS-Industrial [11]. No formal description of this software has been found by the authors and few example applications are available for evaluation. This paper addresses this lack of description and evaluation. In addition, we propose a new

978-1-5090-6505-9/17/$31.00

[1] In this paper, the term *path* refers to a sequence of end effector poses in operational space. In general, a path is a geometrical description of a robot's motion, which can also be specified in joint space as described in [2].

cost function to improve the planner's performance for robotic welding.

This paper is organised as follows. Section II describes the path planning problem for robot welding and the adopted performance criteria more formally. Next, Section III describes the approach used to solve this problem, as implemented in the Descartes package. In robot Section IV, the Descartes approach is applied to some key welding tasks and its performance is evaluated. Based on this evaluation, an improved cost function for the planning algorithm is proposed, implemented and evaluated in Section V. Conclusions are presented in Section VI.

II. Path planner criteria for arc welding robots

This section describes several characteristics and requirements for the automatic generation of collision-free paths for arc welding robots. The notation is based on [5]. The dynamics of the robot are not considered in the path planning approach discussed in this paper.

Kinematic redundancy and end effector tolerances. Suppose a 3R planar robot gets the task to follow a path with its end effector in a two-dimensional operational space. For this example we assume that only the position of the end effector is specified, the orientation can be freely chosen. When using graph search methods, a discretised version of this task is formulated. The path is specified in N Cartesian points $\boldsymbol{x_i}$:

$$\boldsymbol{x_i} = (p_{ix}, p_{iy})^T, \text{ with } i = 1 \dots N \tag{1}$$

Solving the discrete path planning problem means finding feasible joint positions $\boldsymbol{\theta}_i$ for each Cartesian point $\boldsymbol{x_i}$:

$$\boldsymbol{\theta}_i = (\theta_{i1}, \theta_{i2}, \theta_{i3})^T, \text{ with } i = 1 \dots N \tag{2}$$

Feasibility means satisfying the robot's kinematics $\mathbf{f}$ for each Cartesian point:

$$\boldsymbol{x_i} = \mathbf{f}(\boldsymbol{\theta_i}), \text{ with } i = 1 \dots N \tag{3}$$

Notice that in this example there are an infinite number of joint solutions $\boldsymbol{\theta}_i$ for every Cartesian point $\boldsymbol{x_i}$, because the end effector's orientation is not specified. Notice also that, even if the orientation is specified, there are still two solutions to the inverse kinematics, the *arm up* and *arm down* configuration. Ideally, the planning algorithm should be able to exploit these properties in order to find feasible paths.

Furthermore, the motion between consecutive joint positions has to be physically executable by the robot: it should be collision-free and should adhere to the robot's maximum velocity and acceleration limitations. In addition, the path should be discretised fine enough to ensure that the end effector constraints are also satisfied in between the specified path points.

As a variation on this problem, we could also place a tolerance on the x-position of the end effector, p_x. In that case, the path planning algorithm should be able, for example, to prioritize the orientation freedom over the p_x tolerance, i.e. the planner should be able to prefer deviations from the desired end effector's orientation above deviations from the desired x-position when searching for a collision-free path. For arc welding robots, such prioritization is imperative: whereas the orientation around the welding torch axis (z) is of lesser importance, deviations from the ideal welding torch orientation for the other two angles (x and y) is not equal. Depending on the welding type, a deviation in one angle (x) is often preferred over the other (y), or vice versa.

The following section extends this problem to a three-dimensional operational space. When evaluating a path planning algorithm for arc welding robots, the following performance criteria will be valued:

- **Computation time**: how long does it take to solve the planning problem.
- **Memory requirements**: can the algorithm run on a computer without running out of memory?
- **Correctness**: is the path collision-free and does it respect the joint limits?
- **Singularities**: can the algorithm handle paths that require the robot to move close to singularities? Does the planning algorithm avoid these singularities in any way?

The Descartes planning approach explained in the following section will be evaluated in Section IV w.r.t. the above criteria.

III. The Descartes path planning approach

This section describes the Cartesian path planning approach of the Descartes package [12], a ROS-Industrial project [11]. The Descartes package was first presented at ROSCON and in a ROS-Industrial community meeting in 2015 [12] [13]. However, a formally published description of the software has not been found by the authors. Therefore, this section describes the path planning approach used in the package. Some details are available in the ROS-Industrial training session slides [14]. This section is partially based on those slides; missing details were completed through a thorough analysis of the source code.

Before introducing the planning algorithm, the axis conventions used in the software are shown in Fig. 1. The end effector frame $\{ee\}$ is positioned at the welding tip. The z-axis is chosen along the welding torch, the y-axis in the welding direction and the x-axis completing the right hand frame. Every trajectory point gets a local frame $\{\mathrm{p}\}$ assigned that indicates the preferred position and orientation of the end effector in that trajectory point. These coordinate frames are expressed in a common reference frame $\{\mathrm{B}\}$.

In the software, every path point is called a *trajectory point*. However, these points do not contain speed nor acceleration information. Only an upper limit on the travel time between two successive trajectory points is specified in the software, called the *TimingConstraint*.

The planning algorithm can be divided in three important phases, see Fig. 2. In phase one, the given trajectory is converted to joint space, if necessary. In phase two, the joint solutions created in phase one are organised in a graph. Finally, in phase three, a graph search is applied to the graph of

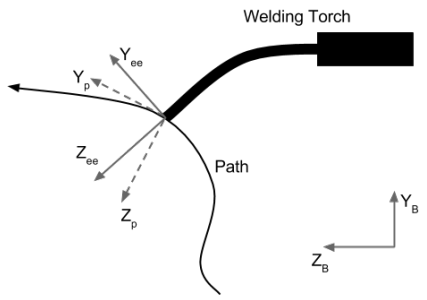

Fig. 1. Definition of path frame (p) and end effector frame (ee) for a Cartesian trajectory point. The difference between the two frames must lie within the specified tolerance limits. The frames are expressed in a reference frame (B).

phase two to get a joint trajectory that minimizes a given cost function. These three phases will now be discussed in more detail.

Phase 1. The algorithm starts from a list of so-called *trajectory points*, which can either be expressed as desired Cartesian poses for the end effector or as desired joint positions. When a Cartesian end effector pose is specified, this pose needs to be converted to feasible joint positions before the graph search can start. Fig. 2 illustrates this procedure. For each trajectory point, every parameter of the preferred pose (position or Euler angle) can be assigned a tolerance range. For a non-zero tolerance, the trajectory point is uniformly sampled in the tolerance range, with a user-specified resolution. This leads to multiple joint or Cartesian points. Every Cartesian point is then converted into one or more joint positions according to the inverse kinematics of the robot. Note that the tolerance on the end effector orientation is defined as a deviation from the desired Euler xyz angles that express the trajectory path point in the base frame. A specific implementation of the Cartesian trajectory point, a so-called *axial-symmetric point*, which allows free rotation around one axis and no tolerance on the others, is available in the software. This makes the implementation more convenient for some tasks. When a trajectory point is given in joint space, only the tolerances need to be processed for this point as shown in Fig. 2. These different types of trajectory points allow the user to specify a hybrid trajectory, containing both Cartesian and joint space trajectory points, with or without tolerances.

Phase 2. The resulting joint positions are organised as nodes (vertices) in a graph. Multiple nodes can belong to the same trajectory point, as schematically shown in Fig. 3. When collision checking is enabled, a joint position resulting in collision is not added to the graph. Edges are constructed between all nodes of two successive trajectory points. Before adding an edge between two nodes, the software checks the time constraint *TimingConstraint*, if specified. The angle that each joint has to travel is compared with its velocity limit times the maximum travel time. If the edge is added, the cost is set proportional to joint motion. For an edge going from joint position $\boldsymbol{\theta}_n$ to $\boldsymbol{\theta}_{n+1}$, the cost is defined as the L_1-norm

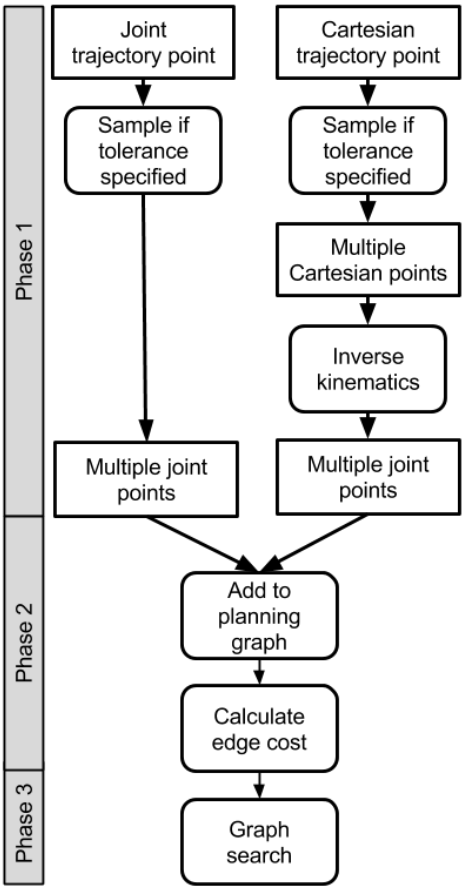

Fig. 2. Descartes' approach to convert different types of trajectory points to joint positions (phase 1), to create a planning graph (phase 2), and to search this graph for a feasible path (phase 3).

of the difference of the two joint positions:

$$\text{edge cost} = \|\boldsymbol{\theta}_{n+1} - \boldsymbol{\theta}_n\|_1. \tag{4}$$

The use of an alternative cost function is discussed in Section V.

Phase 3. The resulting directed graph enables the use of standard graph search algorithms to find the shortest path. In Descartes' case, Dijkstra's algorithm [15] is used from the Boost C++ library [16].

The final result is a sequence of joint positions that, when followed by the robot, result in the desired Cartesian path within the given tolerances, if such a path exists and is found by the inverse kinematic solver. The sequence of joint positions should allow easy execution on standard industrial robot controllers. However, as discussed in the next section, this last step is not always possible. The joint trajectory can contain large accelerations and even collisions in certain cases. When these problems occur, re-planning, or post processing of the joint trajectory is necessary.

A sparse version of the planning software exists, which allows faster execution by only calculating the inverse kinematics for a subset of the trajectory points. The intermediate

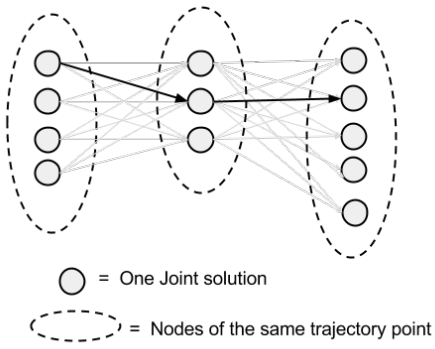

Fig. 3. Schematic representation of the planning graph in the Descartes package.

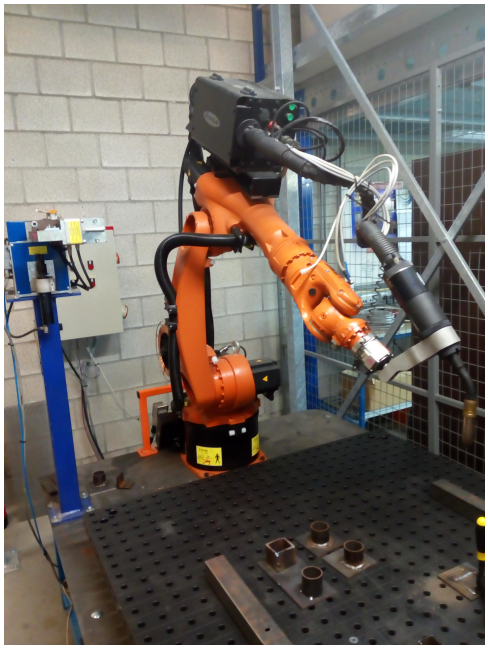

Fig. 4. Our Kuka kr5-arc robot with welding torch.

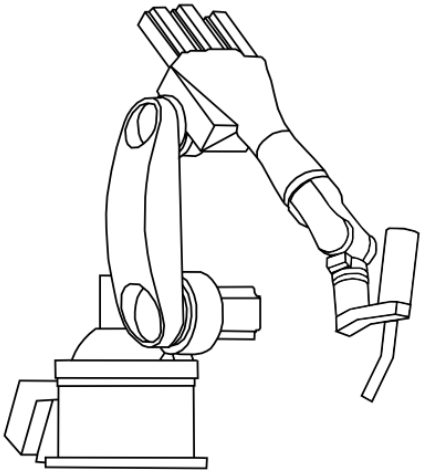

Fig. 5. Kuka kr5-arc robot model used in simulations. The wire feeder and cables are not included in the model, because they do not have a big influence on the welding tasks that are simulated, neither are they the focus of the Descartes software package.

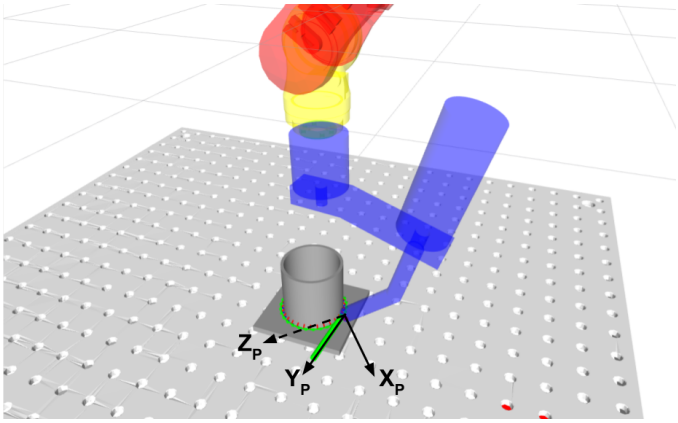

Fig. 6. Welding task A: "cylinder on a plane".

points are calculated by interpolating the joint motion of the sparse points. The resulting error on the Cartesian path is checked using forward kinematics. If the error is too big, an extra point is added to the sequence of sparse points to decrease the error, and the path is re-planned.

IV. Evaluation of the Descartes package

This section evaluates the Descartes package with respect to the requirements given in Section II. For this, the robot model (shown in Fig. 5) of our Kuka kr5 arc welding robot (shown in Fig. 4) is used. We have adapted the software to enable collision detection with objects based on the source of the Godel [17] project, another ROS-I project. Our source code is available at [18]. Based on this code we also made a new tutorial for Descartes available at [19] and [20].

In order to evaluate Descartes' performance, Cartesian trajectory planning is executed for three welding tasks. Task A and B are shown in Fig. 6 and 7, task C is discussed later in this section. For task A, welding the cylinder, a tolerance of $\pm\pi$ rad is allowed around the tool's z-axis, which does not influence the welding process. For task B, welding the L-profile, the torch is allowed to rotate around an axis along the path Y_p with $\pm\pi/18$ rad ($\pm 10°$). This allows the robot to avoid collisions with the obstacle, a small plate mounted at the inside of the L-profile. The planning results in a joint motion profile that the robot should execute, shown in Fig. 8.

Computation time. When planning these Cartesian paths, the inverse kinematics of the robot are solved multiple times for every trajectory point. Solving the inverse kinematics can be executed with the standard solver which uses the KDL library [21]. In this case, the solution is calculated with a numerical method. Alternatively, IKFast is a library that enables to create C++ code to solve the kinematics analytically. This method can reduce the calculation time drastically. For example, path planning for task B with 50 trajectory points

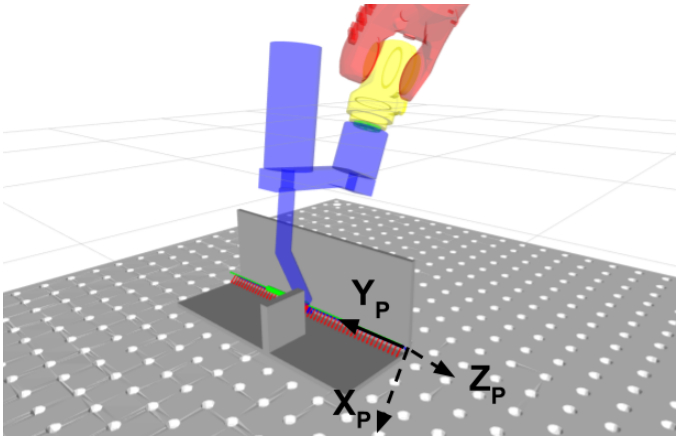

Fig. 7. Welding task B: "L-profile with obstacle".

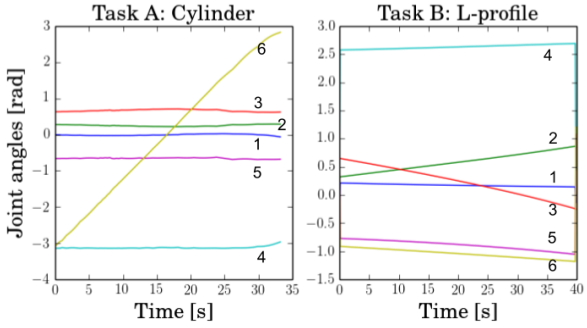

Fig. 8. Joint motion profile for the two welding tasks shown in Fig. 6 and 7. Joints are numbered from base to tip.

results in the computations times [2] shown below, given for the different phases described in Section III, where phase 1 includes the inverse kinematics calculation (and collision detection). The KDL solver is executed with 20 different stating conditions for each trajectory point to get enough different joint solutions.

	phase 1	phase 2	phase 3
KDL	731.1 s	8.6 s	1.6 s
IKFast	1.4 s	0.8 s	0.2 s

The reason that the computation times for phase 2 and phase 3 are also longer for KDL is that it returns more inverse kinematic solutions and therefore results in a bigger planning graph. There are two reasons why KDL returns more joint solutions for the inverse kinematics than the IKFast solver. First, some of the solutions returned by the KDL solver are almost identical, apart from small errors caused by numerical approximations. Second, the IKFast solver does not return solutions with joint values outside the interval $[-\pi, \pi]$, whereas the joint limits on joint 4 and 6 go up to $[-2\pi, 2\pi]$.

[2]Simulations were executed on a laptop with Intel® Core™ i7-6820HQ CPU @ 2.70GHz and 16 GB RAM.

The KDL solver does return solutions with joint values outside $[-\pi, \pi]$, but obeying the robot joint limits. Even though IKFast is considerably faster than KDL, IKFast is limited to robots with 6 DOF or less. However for robots mounted on a rail, the IKFast solver could be adapted to handle this.

Memory requirements. When using the IKFast solver, the computation time is not a limiting factor any more, but increasing the path complexity will cause excess memory usage. For example, a tolerance on one axis of 360° sampled in intervals of 1° results in 360 nodes per trajectory point in the graph. An edge cost is calculated and saved between every node of two successive trajectory points (except when the joint speed limit dictates that the motion is infeasible). The memory requirements to save all the edges can quickly escalate to more than a standard computer has available. Suppose there are N trajectory points and M joint solutions for each trajectory point. M corresponds to the number of sampled poses in the discretised tolerance region. In general M is not constant, the different robot configurations increase M even more, typically a factor 2 for our robot. The number of nodes and edges in the graph roughly equals:

$$\text{number of nodes} \approx \; 2NM \tag{5}$$

$$\text{number of edges} \approx (N-1)(2M)^2 \tag{6}$$

For the simple task, with only a tolerance on the z-axis (360° sampled in intervals of 1°) and 30 trajectory points, this results in 2.16×10^4 nodes and 1.50×10^7 edges, corresponding to around $1\,\mathrm{G}B$ of required RAM. Adding tolerances to the other tool axes will quickly increase the memory usage to unmanageable values. The use of the sparse planner can allow increased complexity of the welding task. Only adding a subset of points to the planning graph results in a lower memory demand. However, the total calculation time can become longer when it has to do a lot of re-planning when the resulting end effector path is not accurate enough. Fig. 9 shows the memory usage for the two planners for welding task B. Given were 50 trajectory points with a tolerance of $\pi/4$ rad around the welding path and $\pi/20$ rad around the torch axis. The sparse planner used 9 trajectory points and computed the other 41 points by interpolation. The planner started with 6 points but had to add a new point three times. Other options to decrease memory usage are, only adding tolerances to specific trajectory point where they are excepted to be useful and adding fewer trajectory points on a sections of the path where less complicated robot motion is excepted.

Correctness. The created joint profile does not necessarily result in a collision-free path. For example, for welding task B, a tolerance is added around the path's x-axis (perpendicular to the path) and no tolerance is allowed around the other axes. The obstacle cannot be avoided if the rotation of the welding torch around the path is fixed as shown in Fig. 11. Nevertheless, Descartes still generates a motion plan. The torch pose jumps from one end of the tolerance space to the other when crossing the object, as shown in Fig. 10 and 11. No collision checking is performed in between consecutive poses

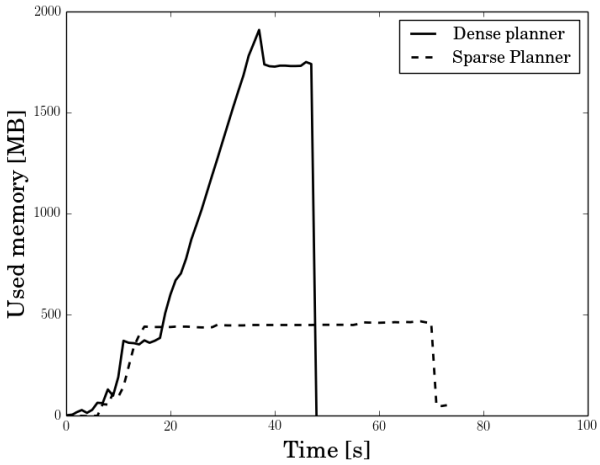

Fig. 9. Memory usage during execution of the planning algorithm for welding task B.

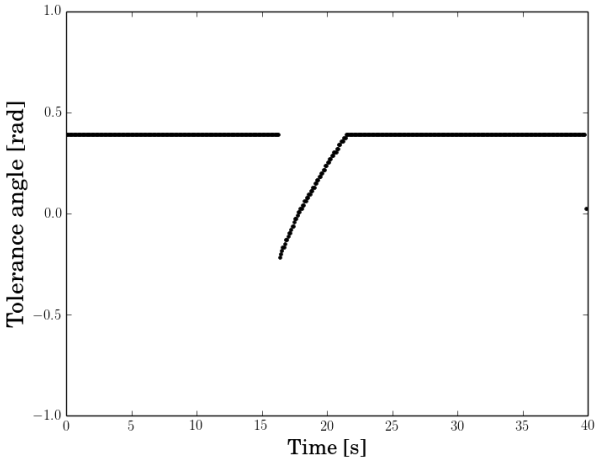

Fig. 10. Angle between path frame and end effector frame around path x-axis. The torch *jumps through* the obstacle and then restores its initial orientation. The collision is not checked in between the two trajectory points around the jump.

of trajectory points. Therefore the planner does not detect this collision. A limit on the allowed change in end effector pose, in addition to the joint motion limit, could avoid this problem. As a final note on correctness, the acceleration limits of the robot joints are not considered by the planning algorithm, which is another way the planner could generate infeasible trajectories.

Singularities. For another task C, the L-profile is placed as shown in Fig. 12. A tolerance is specified identical to task B, the robot can turn the end effector around the path Y_P axis to avoid the obstacle. As shown in Fig. 13, joint 5 will have a negative angle when the torch is moving along the straight line. This keeps the wrist away from the singularity. When moving around the obstacle, the angle of joint 5 comes close to zero This causes joint 4 and 6 get aligned. Moving joint 4 or 6 will then cause identical end effector motion. The robot wrist gets

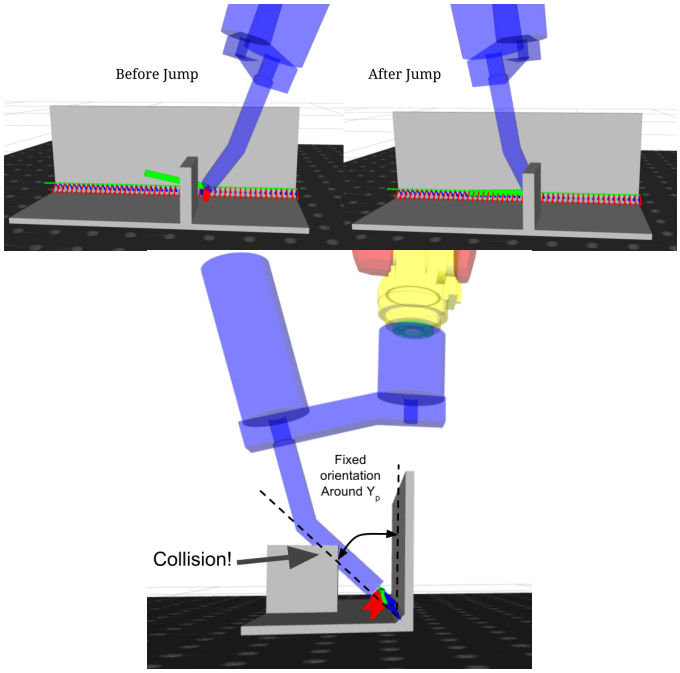

Fig. 11. The torch *jumps through* the obstacle. The planning algorithm does not check for collisions in between the two trajectory points around the jump.

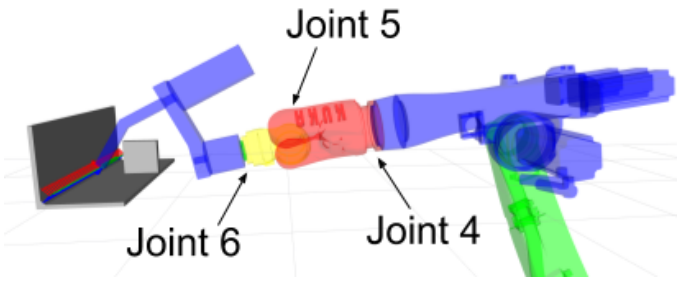

Fig. 12. Task C. Position of the robot where singularity occurs in the wrist. Joint 4 and 6 are aligned and cause identical end effector motion.

in a singular position, shown in figure 12. In this position the angles of joint 4 and 6 grow large and in opposite directions as is visible in figure 13 at 22 s. This short example shows that the cost function that minimizes joint motion also implicitly avoids singularities, where typically large joint motion occurs. Hoverer it is also clear that this does not always works, as showed in this example. The behaviour around singularities can be examined in more detail, but this is not the focus of this paper. We can usually move the workpiece to a better position.

V. Improved Cost Function

When a tolerance is specified around a nominal pose, the default Descartes planner is indifferent to deviations from this nominal pose, as long as they stay within the tolerance (i.e. all joint positions of a trajectory point are assigned the same cost). However, in robot welding, a rotation of the torch around an axis other than the torch itself is undesirable and

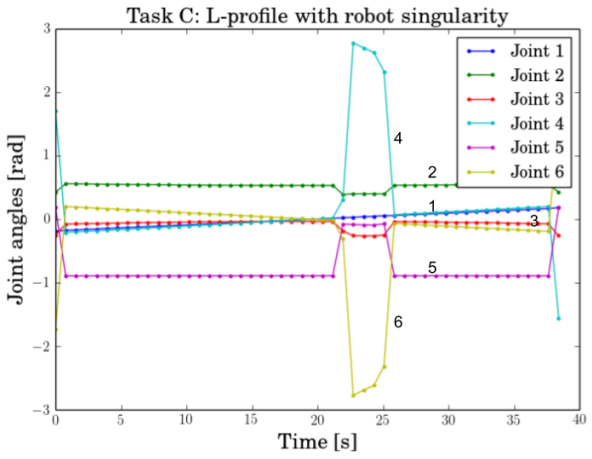

Fig. 13. Path generated close to the robot wrist singularity.

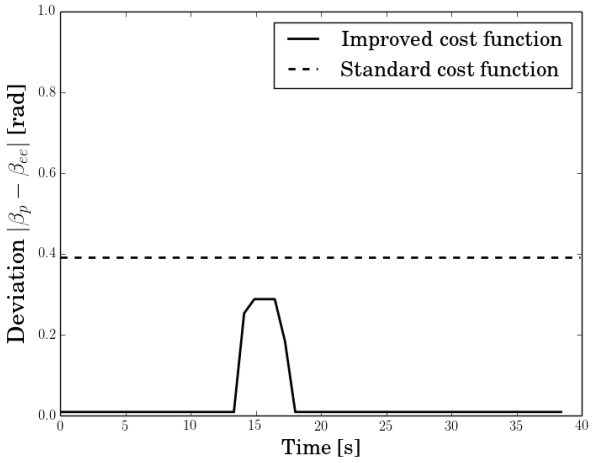

Fig. 14. Deviation from preferred pose around the path's y-axis along the path. Constant deviation of 0.4 rad with the standard cost function. Smaller deviation with the improved cost function.

those deviations should receive a higher cost. Sometimes a small deviation on the desired torch angle is allowed to avoid collision with objects, for example in welding task B. For this reason, we have added a new cost to the nodes of the planning graph. Since the orientation tolerance on the nominal pose is given in xyz Euler angles, the deviation from this pose is also expressed in xyz Euler angles for the cost function. In this paper we do not consider position tolerance, those are more straightforward to add than orientation tolerances. Therefore, only the rotation matrices of the end effector ${}^{B}_{ee}R$ and the trajectory point on the path ${}^{B}_{p}R$ are used, expressed in a base frame B as implemented in the package. For some applications it could be more convenient to express the end effector frame in the local path frame and add tolerance to the Euler angles describing this transformation ${}^{P}_{ee}R$. This is suggested as future work. The xyz Euler angles are defined based on three elementary rotations around moving axes, as defined in [22] and used in the Descartes package.

$$ {}^{B}_{ee}R = R_X(\alpha_{ee})R_Y(\beta_{ee})R_Z(\gamma_{ee}) \quad (7) $$

$$ {}^{B}_{p}R = R_X(\alpha_{p})R_Y(\beta_{p})R_Z(\gamma_{p}) \quad (8) $$

Where the notation ${}^{B}_{p}R$ stands for the rotation matrix of a path frame $\{p\}$ expressed in the base frame $\{B\}$ and $R_X(\alpha)$ for a rotation with an angle α around the (moving) x-axis. Ignoring the rotation α around the z-axis, which is not specified for the welding task, the cost function is defined as follows:

$$ \eta|\alpha - \alpha_{ee}| + |\beta_p - \beta_{ee}|. \quad (9) $$

The factor η allows to prioritize one of the two rotations. The Euler angles are calculated using the Eigen library [23], which ensures the α, β and γ lie within the ranges $[0, \pi]$, $[-\pi, \pi]$, and $[-\pi, \pi]$ respectively. Therefore the angles are unique, except in the singular case.

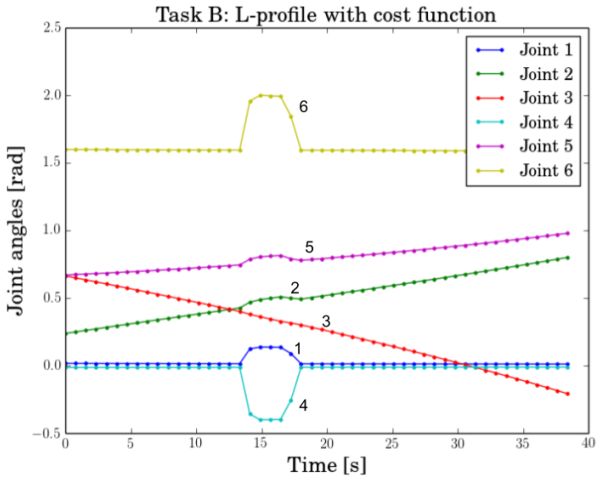

Fig. 15. Joint trajectory plan for the L-profile (task B) when using the new cost function.

Since there is no functionality in the package to add a cost to a node, this cost is added to the edge cost of all the edges leading to the node. This way the standard implementation of the graph search can still be used.

Executing the path planning with this new cost function for task B results in less deviation from the preferred pose. This is shown in Fig. 14. The deviation of the preferred pose around the path y-axis stays zero, when no deviation is necessary for collision avoidance. The dashed line represents the same orientation angle when using the standard Descartes cost function. The resulting joint motion profile for the new cost function is shown in Fig. 15. There are some brisk changes in the angles of joint 4 and 6. This could be avoided by adding a cost function that penalizes large joint speed, acceleration or jerk.

VI. Conclusion

To decrease programming time for industrial robots, reliable path planning algorithms are needed. This paper described the Descartes path planning package, a ROS-Industrial project intended for use in applications such as welding, painting or grinding. The approach is promising in that it computes globally optimal, collision-free paths for the discretised problem. Furthermore, it allows for a flexible specification of desired robot paths as sequences of Cartesian end effector poses or joint points, with specification of the allowed tolerance on the various degrees of freedom. Descartes is specifically designed to handle kinematic redundancy in path planning problems. This redundancy follows amongst others, from the tolerances on the desired path (under-defined end effector poses).

In order to verify these promises, we have evaluated this package on specific welding tasks. Some of the limitations of the Descartes package and its graph search method are described. For robots with more than 6 DOF, or tolerances on multiple degrees of freedom, the computational load can quickly escalate due to the discretisation of the redundant DOF. Furthermore, acceleration limits are not taken into account and collisions are not adequately verified. As future work, we suggest to have a tighter coupling between the graph search algorithm, the calculation of the inverse kinematics and the tolerances, to further improve the computational efficiency. Another improvement could be defining the tolerance in a local path frame instead of a global base frame. In addition, taking some dynamics in to account in the cost function could improve the resulting joint trajectory.

Finally, this paper also proposed, implemented and evaluated a new cost function to ensure that the welding torch stays close to a preferred orientation. This new cost function has been applied to a welding task that clearly illustrates its advantage.

Acknowledgment

The authors would like to thank the ROS-Industrial consortium and in particular Jorge Nicho for providing us with valuable feedback on this work. The authors also gratefully acknowledge the financial contribution of the KU Leuven Impulse fund IROM.

References

[1] International Federation of Robotics, "How robots protect the EU in global competition," Presentation on IFR EB Meeting, 2016 September.

[2] B. Siciliano, L. Sciavico, L. Villani, and J. Oriolo, *Robotics: Modeling, Planning and Control*. Springer, 2010.

[3] S. Seereeram and J. T. Wen, "A global approach to path planning for redundant manipulators," *IEEE Transaction on Robotics and Automation*, vol. 11, no. 1, 1995.

[4] L. Shi, X. Tian, and C. Zhang, "Automatic programming for industrial robot to weld intersecting pipes," *International Journal of Advanced Manufacturing Technology*, vol. 81, no. 9-12, pp. 2099–2107, 2015.

[5] D. Whitney, "Resolved Motion Rate Control of Manipulators and Human Prostheses," *IEEE Transactions on Man Machine Systems*, vol. 10, no. 2, pp. 47–53, 1969.

[6] L. Huo and L. Baron, "The joint-limits and singularity avoidance in robotic welding," *Industrial Robot*, vol. 35, no. 5, pp. 456–464, 2008. [Online]. Available: http://dx.doi.org/10.1108/01439910810893626

[7] R. M. Holladay and S. S. Srinivasa, "Task Space Trajectory Planning via Graph Search," Senior Thesis Prospectus.

[8] M. Rickert, "Efficient motion planning for intuitive task execution in modular manipulation systems," Dissertation, Technische Universität München, Munich, Germany, 2011. [Online]. Available: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20110719-981979-1-6

[9] R. Diankov, "Automated construction of robotic manipulation programs," Ph.D. dissertation, Carnegie Mellon University, Robotics Institute, August 2010. [Online]. Available: {http://www.programmingvision.com/rosen_diankov_thesis.pdf}

[10] Open Source Robotics Foundation. Robot Operating System (ROS). [Online]. Available: {http://www.ros.org/}

[11] ROS-Industrial. [Online]. Available: {http://rosindustrial.org/}

[12] S. Edwards. (2015) The descartes planning library for semi-constrained cartesian trajectories. ROSCon 2015. Last accessed april 2017. [Online]. Available: {http://wiki.ros.org/descartes}

[13] ——. (2015, january) Shaun edwards presents descartes planner at ros i community meeting jan 2015. Video of ROS-I Community Meeting. Last accessed april 2017. [Online]. Available: {https://youtu.be/60Uuu5cplVo}

[14] Ros-industrial github. [Online]. Available: {https://github.com/ros-industrial/industrial_training/tree/kinetic/slides}

[15] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[16] Boost C++ libraries. [Online]. Available: {http://www.boost.org/}

[17] ROS-Industrial. Godel: application for demonstrating surface blending with ros. [Online]. Available: {https://github.com/ros-industrial-consortium/godel}

[18] B. Moyaers and J. D. Maeyer. Source code for simulations described in this paper. [Online]. Available: {https://github.com/Bart123456/lasrobot_ws}

[19] ——. Tutorial contributed to descartes. [Online]. Available: {https://github.com/JeroenDM/descartes_tutorials}

[20] ——. Changes made to descartes software for tutorial. [Online]. Available: {https://github.com/JeroenDM/descartes}

[21] Orocos. Orocos kinematics and dynamics. [Online]. Available: {http://www.orocos.org/kdl}

[22] J. J. Craig, *Introduction to Robotics: Mechanics and Control Third Edition*. Pearson, 2014.

[23] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.