Chapter 5

Path planning for physical effects

In the previous chapters two different planning problems have been investigated, namely: *path planning* and *planning under uncertainty*. As we discussed, path planning concerns the problem of constructing actions (typically trajectories in free space) for a robot in deterministic, but continuous domains. In contrast, planning under uncertainty concerns the problem of finding action-selection strategies that are more robust in the face of uncertainty, but these techniques are typically adequate only for discrete domains.

This chapter formalises the problem of planning a sequence of push operations (action-selection), by which a robotic arm equipped with a single rigid finger can move an object to be manipulated towards a desired goal pose. This is a challenging problem, because of the complex relationship between pushing actions and resulting object motions. A plan must be built in the action space of the robot, which is only indirectly linked to the motion space of the object through a complex interaction for which inverse models may not be known.

I will present a two stage approach to planning pushing operations. A global RRT path planner is used to explore the space of possible object configurations, while a local push

planner makes use of predictive models of pushing interactions, so as to plan sequences of pushes to move the object from one RRT node to the next. The effectiveness of the algorithm is demonstrated by simulation experiments in which a robot must move a rigid body through complex 3D transformations by applying only a sequence of single finger pushes.

5.1 Organisation

This chapter proceeds as follows. Section 5.2 provides an introduction to the problem of robotic pushing and addresses the problem of planning in domains where it is hard to derive inverse models. Additional information is required of what pushes should be applied to move the object along the sequence of planned poses. There is often no way of knowing what the outcome of a push will be, except by trialling it in a predictor (here for proof of principle we use a physics simulator (Nvidia Physx), however learned predictors may also be useful [Kopicki et al., 2011]). Instead, what is needed is a planner, which uses a physics engine to span the working space with a branching tree-like structure of possible object motions. Note that all predictive models are not exact, thus we need to overcome somehow such a source of uncertainty. In this chapter, the planner overcomes this source of uncertainty by re-planning: if the observed object pose deviates from the planned pose at execution time, then a new plan is constructed starting from the present pose.

Section 5.3 presents the core algorithm of this chapter. This algorithm is based on the RRT algorithm presented in Sec. 3.3.2.1. A tree-like structure is constructed in the configuration space of the object to be pushed, as in the standard algorithm. However, for this problem, each extension of the tree can be thought as a MDP in continuous spaces under uncertainty on the physical effect of the action to be applied. To overcome

the problems of solving such a large MDP problem, I present an alternative formulation based on a randomised depth first search procedure.

Section 5.4 presents experimental test results in which a series of pushes are applied to objects and both learned and physics based predictors are tasked with predicting the resulting motions. Performance is evaluated through a combination of virtual experiments in a physics simulator.

5.2 Introduction

In robotics, as described in [LaValle, 2006], motion planning was originally concerned with problems such as how to move a piano from one room to another in a house without colliding with obstacles or bounding walls. The state-of-the-art algorithms to solve such problems are based on randomly sampling points or poses in the free configuration space of the object being moved, e.g [Kavraki & Svestka, 1996; Geraerts & Overmars, 2002].

While a path planning algorithm can deal with the large state space to find a sequence of intermediate poses of the object (waypoints) to move the object from an initial pose to a desired one, findind a sequence of pushing to move the object from one waypoint to another is not trivial. As mentioned in Sec. 4.3.5, this is a decision-making problem, however, the complexity for searching for a policy in continuous state and action space render such a formulation impractical. Next section explains how this problem can be transformed in an optimisation problem and solved efficiently.

5.3 Planning pushes to reach a goal pose

The method presented here breaks this process down into two components. First, a *global* path planner is considered, which uses an RRT algorithm to explore the configuration space of the object to be pushed by growing a tree of nodes towards the desired goal pose. Secondly, a *local* push planner is discussed, which uses a randomised depth first search procedure for finding local sequences of pushes to reach from a previous node towards the next candidate node suggested by the RRT. Note that, although I describe this work as a 'two-level planner', a third level of planning is also used in the sense that conventional motion planning techniques - here a state-of-the-art PRM - are applied to build any particular action that the arm will execute. However, since building simple motion actions for a conventional arm is well understood and deterministic, 'two-level' refers to the key problem of choosing which set of push actions to build.

5.3.1 Global path planner

RRT planners iteratively expand the search tree by applying control inputs which lead the system towards randomly selected points. The RRT planner considers these selected points as new *candidate* nodes and tries to extend the closest vertex (node) of its existing search tree towards them. The important point is that the RRT planner does not directly compute the actions that are required in order to extend its tree, but instead selects new candidate nodes (in our case object poses) which must be moved towards by applying an appropriate sequence of pushes.

As described earlier and in [LaValle, 1998], this kind of planning can generally be viewed as a search in a metric space, $\mathbf{X}$, for a continuous path from a given initial configuration, x_{init}, to a target configuration, x_{goal}. In conventional motion planning, a state transition function of a form $\dot{x} = f(x, u)$ is enough to encode the kinematic and dynamic constraints

Algorithm 1 BUILD_RRT

Input: $x_{init}, \Delta t, \mathbf{G}$
Output: T
  T.init(x_{init})
  repeat
    x_{rand} ←RANDOM_STATE()
    x_{near} ←NEAREST_NEIGHBOUR(x_{rand}, T)
    π ←LOCAL_PUSH_PLANNER($x_{near}, x_{rand}, \Delta t$)
    x_{new} ←NEW_STATE(x_{near}, π)
    T.add_vertex(x_{new})
    T.add_edge(x_{near}, x_{new}, π)
  until SATISFIED($x_{new}, \mathbf{G}$) or maximum number of nodes, ζ_1, reached

of the problem. The vector u is typically selected from a set of possible control inputs, $\mathbf{U}$. In the simplest case, u can be analytically computed and specifies a particular direction in the metric space in which we extend the RRT. The vector $\dot{x}$ represents the derivative of the state x with respect to time. A new state x_{new} which will become a new node of the search tree is simply computed by integration of $\dot{x}$ over a fixed time period Δt. Note that, in contrast to Chap. 4, I denote the continuous state and action spaces with, respectively, $\mathbf{X}$ and $\mathbf{U}$, instead of the more familiar notation $\mathbf{S}$ and $\mathbf{A}$ used for discrete spaces.

Since we are not able to explicitly compute the correct action to achieve a desired object motion, we make use of physics engines to predict the outcome of a possible action. Hence, an intuitive approach is to try some pushing actions in different directions until we find one which causes the object to move in a desirable direction. More specifically, in our scenario the space $\mathbf{X} \subseteq SE(3)$ represents the set of all possible configurations of the object with respect to a global frame of reference o. Any point $x \in \mathbf{X}$ is expressed as $x = [R_o, p_o^T]$, where $R_o \in SO(3)$ is a rotation matrix and $p_o \in \mathbb{R}^3$ is a translation vector, both over the x, y and z axis with respect to o. Hereafter we also refer to this set as the *configuration space*.

Unlike with more conventional uses of RRTs, we need to specify the control inputs of

Algorithm 2 LOCAL_PUSH_PLANNER

Input: $x_{near}, x_{rand}, \Delta t$
Output: π

```
x ← x_near
π ← ∅
repeat
  Û ←SAMPLE_RANDOM_ACTIONS(x, N)
  x_best, u_best, ρ_best ←SELECT_ACTION(x, Û, x_rand, Δt)
  if u_best = ∅ then
    store as failure and goto repeat
  end if
  π.add_element(x_best, u_best, ρ_best)
  x ← x_best
until ρ_best < ϵ or maximum iterations, ζ_2, reached or maximum number of failures, ζ_3, reached
```

the system in a different space that should not be confused with the configuration space of the object. Motion planning for the manipulator occurs in the *joint space*, $\mathcal{C}$, which defines the set of all possible configurations that the robot arm can assume. In particular, we want to select the change in the joint space, Δu, which produces a desirable change in the object configuration space, Δx, such that the reduction of the distance between the current object pose and the next candidate node, selected by the RRT algorithm, is maximised as follows:

$$\bar{u} = \underset{u \in \hat{\mathbf{U}}}{\arg\min}\, \rho(x_{rand} - f(x, u, \Delta t)) \tag{5.1}$$

where $\hat{\mathbf{U}}$ is a set of randomly selected actions (as described in the next section); x_{rand} is the next candidate node, randomly selected by the RRT algorithm for expansion of its tree; $f(\cdot)$ is the state transition function which applies the action u in the current state x for a fixed time interval Δt; and ρ is a metric distance in $\mathbf{X}$ which denotes the distance between two configuration states $x, x' \in \mathbf{X}$ in terms of rotational and translational displacement.

The rotational or angular displacement between two object poses is evaluated using the quaternion representation. Let q, q' be points on the 3D unit sphere about the origin in 4D quaternion space, which represent the rotation matrices $R, R' \in SO(3)$ respectively, then it is possible to evaluate the difference in orientation between the two poses as:

$$\|q - q'\|_{rot} = \frac{2}{\pi} min(\cos^{-1}(qq'), \cos^{-1}(q(-q'))) \tag{5.2}$$

The rotation space is isomorphic to the 3D unit sphere in quaternion space, with diametrically opposite points identified [Mason, 2001]. The distance is computed as the smaller of the two possible angles between the images of the two rotations on this sphere. The distance thus is never greater than $\frac{\pi}{2}$. For simplicity we normalise the value of the angular distance into the range $[0, 1]$.

We denote the translational distance between two different positions $p, p' \in \mathbb{R}^3$ as the Euclidian norm in $\mathbb{R}^3$:

$$\|p - p'\|_2 = \sqrt{(p_1 - p'_1)^2 + (p_2 - p'_2)^2 + (p_3 - p'_3)^2} \tag{5.3}$$

Now, given two object poses $x, x' \in \mathbf{X}$ we are able to evaluate the "distance" between them as a combination of the two aforementioned metrics as follows:

$$\rho(x, x') = \frac{1}{2}\|Q(x) - Q(x')\|_{rot} + \frac{1}{2L}\|p - p'\|_2 \tag{5.4}$$

where $p, p' \in \mathbb{R}^3$ are the translational components of x, x' respectively; $Q(x)$ is an operator which transforms the orientation of a pose x into the corresponding point on the surface of a 3D unit sphere in quaternion space; and L is a critical parameter of the workspace (e.g. the diameter of the region in which the robot manipulations take place). The purpose of L is to ensure that both rotational errors and translational errors

Algorithm 3 SELECT_ACTION

Input: $x, \hat{\mathbf{U}}, x_{rand}, \Delta t$
Output: $x_{best}, u_{best}, \rho_{best}$
  $\rho_{best} \leftarrow \infty$
  $x_{best}, u_{best} \leftarrow \varnothing$
  for all $u \in \mathbf{U}$ **do**
    $x' \leftarrow$ SIMULATE_ACTION$(x, u, x_{rand}, \Delta t)$
    $\rho' \leftarrow \rho(x_{rand}, x')$
    if $\rho' < \rho_{best}$ **then**
      $x_{best}, u_{best}, \rho_{best} \leftarrow x', u, \rho'$
    end if
  end for

Algorithm 4 SIMULATE_ACTION

Input: $x, u, x_{rand}, \Delta t$
Output: x'
  $x' \leftarrow x$
  repeat
    $x \leftarrow x'$
    $x' \leftarrow f(x, u, \Delta t)$
  until $\rho(x_{rand}, x') \leq \rho(x_{rand}, x)$

occupy ranges between 0 and 1, so that the summation of (5.2) results in a meaningful cost function, in which rotational and translational displacements result in costs of similar magnitude. Note that alternative methods of computing a net pose error are also possible, such as that adopted in [Kopicki, 2010].

Algorithm 1 shows the pseudo-code of our global path planner which iteratively build an RRT, T, given an initial pose of the object. The main different between our implementation from the standard RRT planner described in [LaValle, 1998] consists of the choice of the control inputs. In our case, the variable π identifies a sequence of pushes instead of a single vector.

5.3.2 Local path planner

Algorithm 2 shows the implementation of a *local* push planner which estimates a sequence of motor commands which will move a object to the next candidate node x_{rand} (which has been randomly generated by the RRT), from a pose at the closest vertex x_{near} of the existing tree structure. The RRT planner supplies both inputs: the candidate node x_{rand} and the closest vertex x_{near} to it, determined according to the metric ρ described above.

The local push planner now randomly selects N possible finger trajectories in $\mathcal{J}$ which ensure that the end effector of the manipulator will collide with the object. Each trajectory N is generated as follows in the procedure SAMPLE_RANDOM_ACTIONS(x, N). First a pair of points are randomly generated, each from a uniform distribution on two different surfaces of the object, neither of which is in contact with any other surface. These are then linked along a straight line path, which is extended away from the object in space by a fixed distance d from each surface point. Thus a pair of points in configuration space have been defined that define the beginning and end of a straight line path through the object. These are then converted into joint space via an inverse kinematics solver, and a conventional PRM based path planner [Kopicki, 2010] with optimisation is used to generate the trajectory in joint space. All the N trajectories are simulated using a physics engine. Algorithms 3 and 4 define how the planner selects the next manipulative action by making use of a simulator for prediction. The physics simulator provides a prediction of the next object configuration x' which will result, given the current configuration x (initialised as the vertex x_{near}) and the selected action u for a fixed time interval Δt, $x' \leftarrow f(x, u, \Delta t)$.

During the simulation the actual object's configuration state is checked at each fixed time period Δt. After each period Δt, if the distance ρ to the candidate node x_{rand} has been reduced, then action u is continued for an additional period Δt. Once the distance

ρ begins to rise again, rather than continue falling, the push is interrupted and the configuration which minimised the distance is stored. Note that the pose at which the push is interrupted may often be an unstable pose in which the object is balanced along an edge while resting against the finger. For this reason we instead find the closest stable pose by withdrawing the finger and letting the object settle into a stable configuration. This stable configuration is the one that is stored as the new node on the RRT.

If the final stored configuration is identical to the initial configuration, x_{near}, then it means that this action is not useful and the planner discards it. The simulator then resets the initial position of the object at x_{near} and repeats the procedure by executing each of the remaining N pushing actions. After all N push trajectories have been simulated, the planner selects the most efficient push which minimises the distance ρ to the candidate node x_{rand}. If no trajectory reduces ρ, then the planner stores this iteration as a failure and selects another N random pushing actions. If a preset limiting number of failures is reached without reducing ρ, then the RRT planner will not extend the tree, and will instead randomly choose a new candidate node x_{rand}.

Once a candidate node, x_{rand}, and a useful pushing action u have been found (where u moves the object nearer to x_{rand}, thus diminishing ρ), it is still unlikely that u alone will bring the object sufficiently close to x_{rand} (satisfying a pre-defined maximum distance). As suggested by the results of this study (described in Section 5.4), it is more likely that a single action will move the object to an intermediate configuration which is still close to the initial one, x_{near}. If we settled for the single push u alone, and this intermediate configuration was added to the RRT as a new node, we would lose the desirable behaviour of the RRT planner that yields a biased exploration towards unexplored regions. Applying a naive random action-selection behaviour to the RRT planner would transform its behaviour into a simple naive random tree, where randomly selected vertices are selected to be extended by random actions in order to generate new

vertices. In this case, as described in [LaValle, 1998], "Although one might expect the tree to randomly explore the space, there is actually a very strong bias toward places already explored". Consequently most of the new generated nodes would remain in the same Voronoi region as their parent nodes.

To avoid this overlapping or clustering of nodes, it is necessary that the local push planner be iterated to extend these 'intermediate' nodes until a configuration pose is found which comes close to the candidate pose, x_{rand}, which has been requested by the RRT planner (specified by some maximum threshold distance). The intermediate nodes are encoded as part of a control sequence which defines a *series* of pushes (in the experiments presented here, typically 2-3 pushes) to extend the vertex x_{near} towards the new node x_{rand}.

Once a series of pushes has been found that achieves a configuration x_{best} which comes suitably close to x_{rand}, then the series of pushes is recorded and x_{best} is added as a new vertex to the RRT tree structure. This process continues until an RRT node is found that comes sufficiently close to the overall goal state of the manipulative operation. Figures 5.2(a)-5.2(b) compare our algorithm and a naive RRT which iteratively extends the tree using a single push within the $N = 8$ randomly generated options. The results clearly show that, although the naive RRT is iteratively less expensive in terms of computational time, it converges very slowly with respect to the solution computed by the approach presented in this chapter.

Note that using a physics simulator as a forward model for pushing can be problematic, since it will not perfectly predict the results of pushes on real objects. Furthermore, reasonable predictions require careful tuning of a large number of physical properties (e.g. friction coefficients) which will not generalise to new objects or scenarios. Nevertheless, we employ a physics engine here for proof of principle, as a simple and convenient means of forward modelling. Other work, [Kopicki et al., 2011], suggests that superior

predictions (with superior generalisation capabilities) can be enabled by *learning* forward models, and we intend to substitute these prediction methods instead of the physics engine in our future work.

Additionally, note that *all* forward models (either physics-based, [Duff et al., 2011], learned, [Kopicki et al., 2011], or a combination of both, [Kopicki et al., 2010]) make predictions that can have significant errors. The planner overcomes this source of uncertainty by re-planning: if the observed object pose deviates from the planned pose for the present node by more than a predefined threshold value, then a new plan is constructed starting from the present pose.

5.4 Results

This section presents the empirical experiments and the results collected to evaluate the presented algorithm.

5.4.1 Experiments

The push planning algorithm was tested using a simulation environment based on the NVIDIA PhysX physics engine. The test environment features a simulation model of a five axis manipulator (modelled after the Neuronics Katana 320 robot, [Neuronics AG, 2004]), equipped with a single rigid finger with a spherical finger-tip, see Fig. 5.3. While PhysX and other physics simulators do not perfectly replicate real-world rigid body motions, [Kopicki et al., 2011], the simulated motions are physically plausible, and important physical properties are modeled, including static and dynamic friction coefficients (which for proof of principle we have hand-tuned in our system), collisions, and gravity.

In each experiment, the robot is tasked with pushing an L-shape object (referred to as a "polyflap" [Sloman, 2006]) from a randomised starting pose, towards a randomly chosen goal pose. Polyflaps are interesting objects for testing push manipulation planners, because they can occupy a variety of different stable configurations, as well as a range of unstable modes. They are free to tip and topple, as well as slide and rotate upon a plane. In particular, we distinguish between the two flanges of each polyflap (shown as different colours in the figures), so that it is often necessary for the robot to 'flip' the polyflap in order to move it into the desired goal configuration. This choice of object enables us to illustrate the 3D manipulations planned, in contrast to the majority of related literature in which the manipulanda are effectively 2D and are constrained to planar motions of sliding and rotating upon a flat surface.

Small five-axis manipulators, such as the Neuronics Katana, have a very limited envelope of effective operation. By coding this constraint into the planning algorithm, we see that the system plans pushes, both towards and away from the robot, such that the object is never pushed out of the effective reach of the arm.

5.4.2 Examples

Figure 5.3 shows an example of experiments in which a sequence of pushes has been planned which successfully manipulates a polyflap into a desired goal configuration. In each image sequence, the wire framed polyflap (towards the right of each image) is a 'phantom' which is merely used to denote the desired goal configuration. The two faces of the polyflap are distinguished, one in pink and the other in grey, so that the reader can understand the orientation of the object following each successive push by the robot.

A detailed explanation of the sequence of pushes and resulting object motions is provided in the caption under each figure. Note (by observing the pink and grey faces of the

polyflap) that the robot successfully re-orients the object by causing it to flip its resting base from one face to the other, in order to achieve the desired goal configuration.

5.4.3 Trends and evaluation data

For proof of principle, 30 experiments were carried out in which the robot had to plan and execute a series of multiple pushes. In all experiments, the robot successfully maneuvered the polyflap to within the specified threshold distance from the desired goal configuration.

During these experiments, the global path planner described in Algorithm 1 was set with a maximum number of nodes $\zeta_1 = 2000$ and the method SATISFIED(x_{new}) returns *true* only for configurations within 6 cm and 9 degree[1] of translational and rotational displacement (respectively) from the goal configuration. The local push planner described in Algorithm 2 was set with a threshold value $\epsilon = 0.01$, a maximum number of iterations $\zeta_2 = 3$ and a maximum number of failures $\zeta_3 = 3$. The only parameter varied in the experiments was the number N of randomly trialled pushing trajectories at each iteration (N is defined in section 5.3.2). Ten experiments were performed each for N equal to 4, 8 and 12 pushes respectively.

For each experiment, data was collected on the total number of iterations, the total number of generated RRT nodes, and the rotational and translational distances and the combined cost function values each time a new RRT node has been generated. This data is charted in Figures 5.1 and 5.2. Figures 5.2(a)-(a) show how the error between the achieved object pose and the goal pose decreases as successive nodes are added to the RRT tree. Figure 5.2(b) shows how the computational cost increases as additional nodes are added to the RRT tree. The convergent properties of the results suggest that an

[1] In the experimental results, the rotational error s computed on the unit hypersphere in 4D quaternion space, thus the displacement between two orientations is the length of the arc of the geodesic which connects them. In Fig. 5.1(b), 9 degrees are equivalent to 0.01.

arbitrary degree of positional accuracy is achievable, with an expected trade-off between desired end-state accuracy and computational burden.

The results in Figures 5.1 and 5.2 reveal trade-offs between the amount of effort expended in high-level planning (generating additional RRT nodes) versus effort expended in low-level planning (computing the pushes required to move between RRT nodes). Additionally, a naive implementation of the RRT planner is also shown, which only generates a single push ($\zeta_2 = 1$) to connect one RRT node to the next (in contrast the other methods use multiple pushes, typically three, between RRT nodes). The naive RRT results (labelled nRRT, Figures 5.1 and 5.2) show that this approach damages the useful exploratory properties of RRT planners, since new nodes requested by the RRT planner are often not reachable with a single straight-line push. This causes the RRT to generate excessive numbers of nodes, which tend to be closely clustered rather than reaching out to explore the search space. Similar sub-optimal behaviour is also observed in the "RRT with 4 pushes". This is because, even though this method allows multiple inter-node pushes, the space of selectable pushing actions is very sparse, resulting in crude and inefficient plans. This explains why poorer accuracy and a large number of nodes result from both "RRT with 4 pushes" and the naive "nRRT with 8 pushes".

In contrast, RRT "with 12 pushes" explores 12 different possible push directions at each low-level iteration, with the benefit that comparatively few pushes are required to move from one RRT node to the next. From Figures 5.1 and 5.2 it appears that there is an optimal trade-off "sweet spot" between these extremes, with RRT "with 8 pushes" proving the most efficient.

As we expected the number of trialled pushing trajectories available at each iteration affects the accuracy and computational time of the local push planner. A planner with a larger set of actions is supposed to spend more time in exploring the outcome of all possible actions, whilst a more accurate exploration allows the planner to find the goal

configuration in fewer steps. In particular, the results shows that the RRT with 4 pushes generate a poor exploration which generally upper bounds the performances of the other planners in terms of reducing the cost function (Chars 5.2(a)- 5.1(a)) and computational time (Chart 5.2(b)) suggesting that it requires a larger threshold of maximum iterations, ζ_2, in order to move the polyflap from x_{near} to x_{rand} within the given threshold value ϵ. That obviously affects also the entire solution path which requires more nodes and longer computational time to achieve the same accuracy of the other planners.

5.5 Conclusion

This chapter has addressed the issue of how to plan a series of actions of a robot manipulator to achieve a transformation of an object. Previous work was restricted to 2D motions [Cappelleri et al., 2006]. My approach, by contrast, splits the planning problem into two parts a global RRT planner, and a local planner which performs a randomised depth first search procedure in the action space of the robot. The push plans are iteratively refined by using a physics simulator to evaluate them. I have shown the ability of this two level approach to produce plans for a push manipulation scenario. Empirical experiments suggest convergence of the planned action sequences on arbitrarily accurate approximations to the desired goal state.

As a remark, two aspects of the algorithm presented should be discussed: i) the set of parameters and ii) the choice of the cost function. First, I present the set of parameters divided in two categories:

fGlobal planner parameters:

- ζ_1: maximum number of nodes in the tree;
- problem constrains.

Local planner parameters:

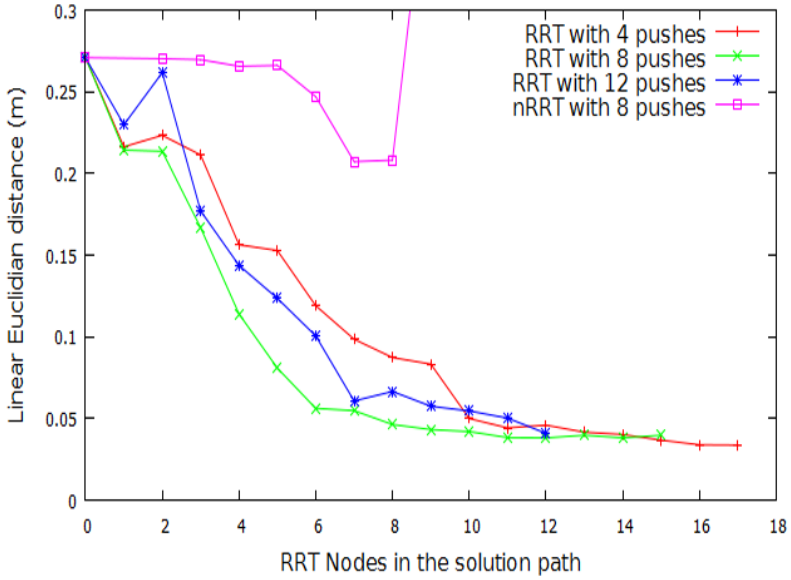
- ζ_2: maximum number of iterations for the local planner;
- ζ_3: maximum number of failures for the local planner;
- ϵ: tolerance for the local push planner;
- N: number of pushes tested at each iteration;
- L: normalising factor for the cost function;

The former set of parameters are typical terminal conditions for an RRT-like planner, to avoid that the algorithm run forever. The first terminal condition is not of particular interest for the scope of this discussion, however a special note should be made for the second one. In many applications an analytic solution of the problem at hand is not available, however a set of (sufficient) constrains can be defined such that: i) any given state that satisfies these constrains is a solution, and ii) this procedure can be performed in polynomial time w.r.t. the number of bits used to described a state. In this thesis, I define such constrains as translational and rotational tolerance w.r.t. a given goal state. This choice has also another advantage: "entire path planning algorithms can be constructed without requiring the ability to steer the system between two prescribed states, which greatly broadens the applicability of RRTs" [LaValle, 1998].

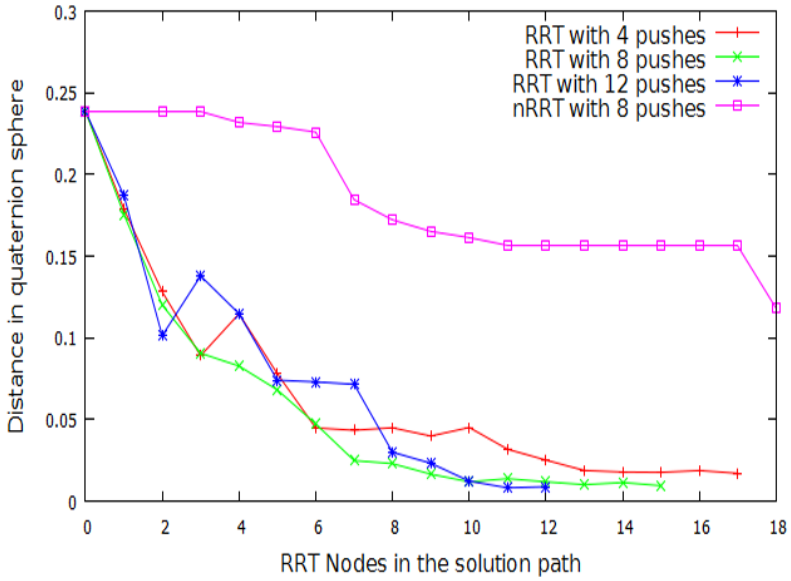
On the other hand, the set of parameters for the local planner is critical for the behaviour of the proposed algorithm, especially the maximum number of failures (ζ_3) and the size of the action set used to extend the tree (N). In Sec. 5.4, I have empirically shown how accuracy and computational expense vary with different choices for such parameters. As already mentioned, ζ_3 affects the ability of the local push planner to reach a candidate node x_{new}. When $\zeta_3 = 1$ the exploration is biased towards already visited regions of the configuration space of the object, which critically penalises the performance of the planner (see Figures 5.1 and 5.2). The same figures shown that varying the size of

options in terms of possible actions affects the ability of the algorithm to converge to the goal region (see Sec. 5.4).

A special note should also be made for the normalising factor L. I compute L as the maximum pairwise translational distance, such that the rotational and translational terms in the cost function have the same magnitude (see Eq. 5.4). This choice biases the exploration strategy in favour of reducing the rotational error. In the experimental results, the terminal conditions mirror this bias, allowing us to relax the tolerance on the (final) translational error: 6 cm versus 9 degrees of tolerance for rotational errors. However, Fig. 5.1(a) shows that these are not the asymptotically achieving errors. In fact the planner reaches the required translational accuracy terminal condition on average with half of the total nodes in its final path, after this, even though the planner is not anymore encouraged to reduce the translation error, the final error achieved is around 4 cm. Additionally, the convergence properties of RRTs to a goal region with an arbitrary tolerance have been intensively studied (see e.g. [LaValle, 2006]), showing that the algorithm is probabilistic complete: "It is not difficult to prove that the vertices will become uniformly distributed. As the RRT initially expands, the vertices are clearly not uniformly distributed; however, the probability that a randomly-chosen point lies within Δt of a vertex of the tree eventually approaches one." [LaValle, 1998].

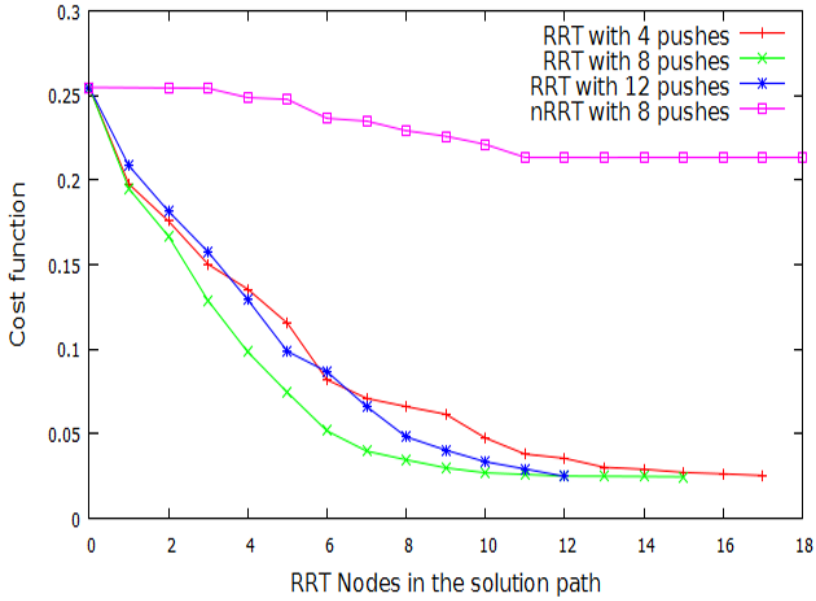


(a)

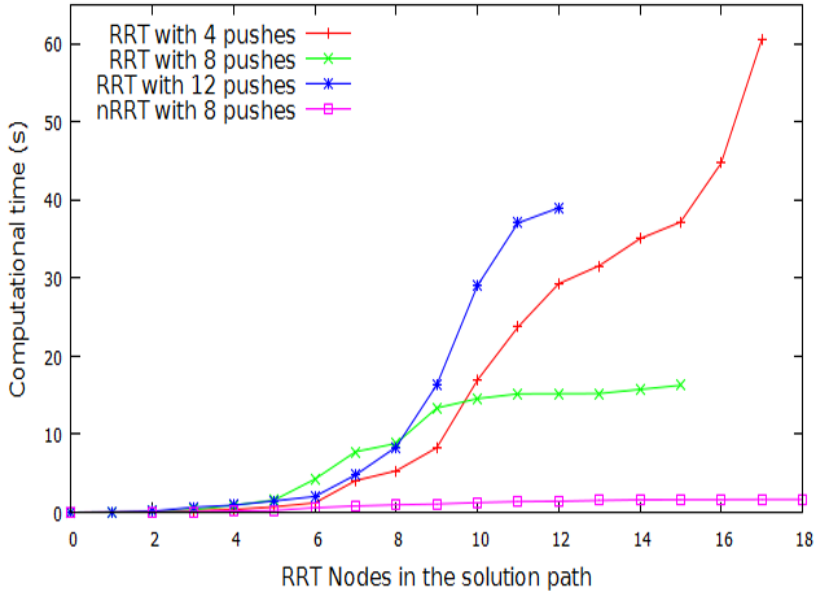


(b)

Figure 5.1: Charts (a) and (b) plot the final translational and rotational errors for solution plans with three different numbers N of random push choices. The graphs show how the mean costs (over ten trials) vary as successive additional nodes are added to the RRT trees. The x-axis shows the number of RRT nodes which compose the solution path. The results are compared with a "naive RRT" which executes only a single push to move from one RRT node to the next ($\zeta_2 = 1$). We let the "naive RRT" explore $N = 8$ possible push directions, since this appears optimal compared to 4 or 12 direction choices. The rotational error in quaternion sphere is bounded in $[0, 1]$ where 1 means 90 degrees of error, thus the terminal condition of 9 degree is equal to 0.01 in 5.1(b).



(a)



(b)

Figure 5.2: Charts (a) show overall cost (combined rotation and translation error), (b) shows computation time for solution plans with three different numbers N of random push choices. The graphs show how the mean costs (over ten trials) vary as successive additional nodes are added to the RRT trees. The x-axis shows the number of RRT nodes which compose the solution path. The results are compared with a "naive RRT" which executes only a single push to move from one RRT node to the next ($\zeta_2 = 1$). We let the "naive RRT" explore $N = 8$ possible push directions, since this appears optimal compared to 4 or 12 direction choices.

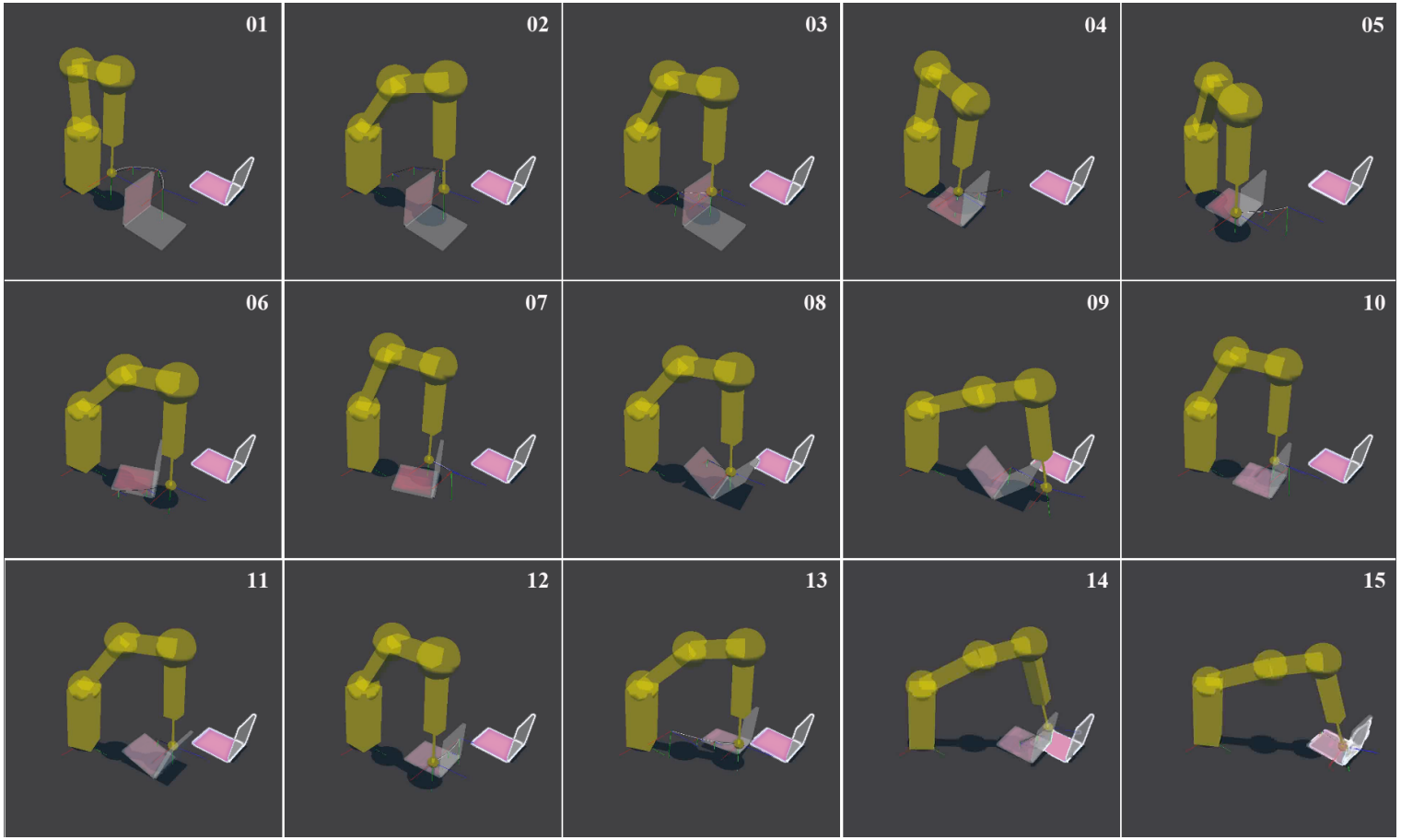


Figure 5.3: The image sequence shows a solution path computed by the planning algorithm in which 8 different pushes were randomly selected at each iteration. Image 01 shows the initial pose. The wire-framed L-shaped object (or polyflap) is a 'phantom' to indicate the desired goal state. The goal pose is translated from the initial pose by 28 cm and rotated by 90 degrees. In this trial the algorithm has planned a distinctly different strategy from the one shown in Fig. 1.3. Image 02 shows the collision-free trajectory to bring the end effector to the start pose of the first push. Images 01-04 show the first push which makes the polyflap to tip over. Images 05-09 show a series of pushes which culminate in the polyflap resting in an unstable equilibrium pose along its folded edge. Images 12-13 show a sideways push. Images 14-15 show the final frontal push.