Architectural Assembly: Simulation and Optimization

Jan Schneider Boris Belousov

***Abstract*— Construction is an industry that could benefit significantly from automation, yet still relies heavily on manual human labor. Thus, we investigate how a robotic arm can be used to assemble predefined structures from complex building blocks autonomously. We propose to split the task of adding a building block to an existing structure into three subtasks: *grasping*, *transporting*, and *placing*. For the *grasping* and *transporting* subtasks, we apply the sampling-based motion-planning algorithm RRT-connect to plan collision-free trajectories in a simulated scene. These trajectories are then executed on a real UR10 robot arm. Since *placing* a block involves complex contact dynamics, we propose to use reinforcement learning for this task. We use the model-free reinforcement learning algorithm TD3 to train such a controller. In our evaluation, we show that the learned controller is able to place a block on another one with an average accuracy of around 8mm.**

I. Introduction

Nowadays, automation is an integral component in many parts of the world-wide economy. However, working in the construction industry requires solving complex manipulation tasks. Hence, automation in this field is still an active area of research (e.g., [1, 2, 3]). To date, the construction industry relies heavily on manual human labor.

Working in the construction industry, however, is linked to a series of health and safety issues. For example in the UK, a report from 2020 found that the rate of fatal injuries in the construction industry is almost four times as high as the national average [4]. Furthermore, heavy lifting and working in cramped positions for long durations, which is typical in the construction industry, also increases the risk of musculoskeletal disorders [4]. Therefore, construction workers are often forced to quit their job early due to health problems. Furthermore, there are construction tasks in locations that are inherently inaccessible or unsafe to human workers, such as areas of recent natural disasters, deep sea, or outer space [5]. Many of these problems can be avoided by using robots instead of human workers to fulfill harmful and dangerous tasks on construction sites.

The Robotic Assembly project is part of an interdisciplinary cooperation with the Digital Design Unit[1] of the architecture department at TU Darmstadt with the goal of automating the construction of modular structures. These structures consist of predefined building blocks. In this work we take a look at two different kinds of blocks, the block in figure 1a, we call *rectangular block* and the one shown in figure 1b, we call *SL block*. From the architecture point of view, these shapes are interesting since they interlock when stacked. Thus, these blocks allow to construct stable structures, while needing less mortar. In this work we aim to tackle the problem of constructing small structures from these blocks using a UR10 [6] robot arm with a Robotis RH-P12-RN [7] gripper. Example structures that can be built from the blocks are visualized in figure 2.

[1]https://www.dg.architektur.tu-darmstadt.de/fachgebiet_ddu/index.en.jsp

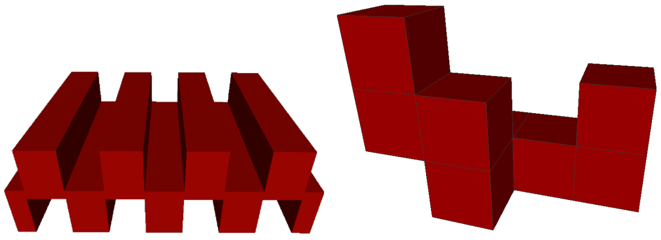

(a) Rectangular block (b) SL block

Fig. 1. The types of blocks that are used to construct modular structures from. The blocks are designed in such a way that they interlock during construction. Hence, less mortar is needed to construct stable structures from these blocks.

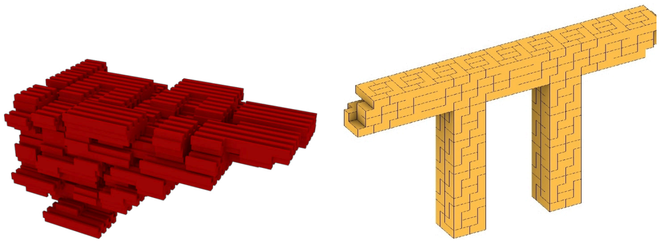


Fig. 2. Example structures built from rectangular blocks and SL blocks

A. Combining planning and reinforcement learning

To complete a single step of the construction process, the robot has to perform three steps: *grasping* a block at the pickup location, *transporting* it to the vicinity of the target location, and *placing* it. During construction, there is a multitude of sources of inaccuracies, like inaccurately placed blocks, incorrectly placed grasps, and slippage. In the contact-rich manipulation task of placing a block on the existing structure, very small offsets of a couple of millimeters can already have a significant impact on the task performance. For example, a small error in the estimated block position can make the difference between placing the block correctly and hitting and potentially destroying the present structure. Thus, it is clear that a controller needs to adapt to external feedback to solve the task with high accuracy in spite of these inaccuracies. Furthermore, for complex shapes, like SL blocks, the motions for joining two blocks are complicated and depend on the direction from which the blocks should be joined. While we focus on placing rectangular blocks for now, we strive for a method that is also applicable for these very complex blocks. Therefore, we propose to use reinforcement learning to solve the *placing* subtask. That is, we train a controller to place a block on the existing structure, with the block already starting close to the target location. Learning such a controller for rectangular blocks is described in section III. For the subtasks of *grasping* and *transporting*, we use open-loop motion planning, as described in section IV.

An alternative to this approach would be to learn all three of these steps in sequence with a single controller. However, this would cause very long training sequences, which are generally hard to learn. This is especially the case if the rewards are sparse. A general way of dealing with such problems is reward shaping [8], where expert knowledge about the solution is incorporated into the reward function to guide the learner towards a good solution. Yet, methods that require reward shaping are usually less general as the reward function is solution specific and they might result in a lower final performance as the agent is prevented from exploiting strategies that the expert did not consider [8].

II. Related Work

A. Task and motion planning

Similar to our work, Hartmann et al. [1] investigate the problem of constructing structures from a given set of building blocks with autonomous robots. They tackle the task by bringing sampling-based motion planners into the Logic-Geometric Programming framework [9]. This framework allows to formulate task and motion planning (TAMP) problems as continuous mathematical programs with constraints formulated in first-order logic. The problems are then solved by alternating search over symbolic action sequences and solving the corresponding path planning problems. Driess et al. [10] propose to use a neural network to predict promising sequences of symbolic actions directly from images of the initial configuration of the scene.

Lozano-Pérez and Kaelbling [11] tackle TAMP problems by converting them to constraint satisfaction problems (CSPs). A generic CSP solver, e.g. [12], is then used to obtain a solution.

A comprehensive overview of other approaches for task and motion planning is given in [13].

B. Reinforcement learning for assembly tasks

Thomas et al. [14] tackle the task of assembling small objects autonomously. The authors first apply motion planning to compute trajectories for solving the task. Due to inaccuracies in the trajectory-tracking controller and the localization of the objects in the scene, this approach often does not succeed. Therefore, the idea of the authors is to warm-start a policy search with a policy that follows these planned trajectories. They show that the policy resulting from this optimization is able to outperform the original trajectory-tracking controller.

Sartoretti et al. [15] tackle the problem of building a structure with many tiny autonomous robots. They use reinforcement learning to learn a common collaborative policy for all robots. The fact that the policy is shared across all robots allows to vary the number of robots without much additional effort.

III. Learning to place rectangular blocks

In the following section, we describe how the *placing* subtask, described in section I-A, is framed as a reinforcement learning problem. We then describe the techniques that we use to tackle the task for the rectangular blocks (see figure 1a) in simulation.

In the report of last semester, we showed how reinforcement learning can be used to place a single block in an otherwise empty scene with an accuracy of $1cm$. This semester, we tackled the task of stacking two blocks in different configurations. We achieved a slightly higher accuracy in this more challenging task, while requiring less training samples. These improvements were achieved mainly through hyperparameter tuning and changes in the observations and cost function. The changes in the observations included adding the current time step as observation. Further improvements were achieved by changing the scaling and clipping of different components of the cost function, described in section III-B. Additionally, the controller was made more robust to changes in the environment by introducing further randomizations in the initial state of the scene and robot during the training, which is highlighted in section III-A.

A. Task Definition

At the beginning of each episode, the starting block is placed at random in a square with $60cm$ side length. This block is assumed to be placed correctly already, e.g., by the controller that was trained last semester. The robot then has to place a second block on top of the starting block in a given stable configuration. Examples of these stable configurations are shown in figure 3. While placing the second block, the robot has to ensure that the first block is not displaced.

The spawning location of the second block is sampled uniformly in a cube of side length $2cm$ that is located $30cm$ above the center of the starting block's spawning area. The spawning orientation is obtained by perturbing the goal orientation slightly. This perturbation is achieved by sampling Euler angles between $-5°$ and $5°$ and rotating the block accordingly. Each episode, the robot starts with the block in its gripper. To avoid overfitting with respect to the exact grasp location, the position of the grasp on the block is also randomized along the ridges of the block and along the block's up-axis. Figure 4 displays a possible starting configuration of the training environment. All these randomizations are included to make the controller more robust with respect to deviations in the positioning of the block and gripper that might occur on the real robot due to inaccuracies in the localization of the blocks and controller deviations.

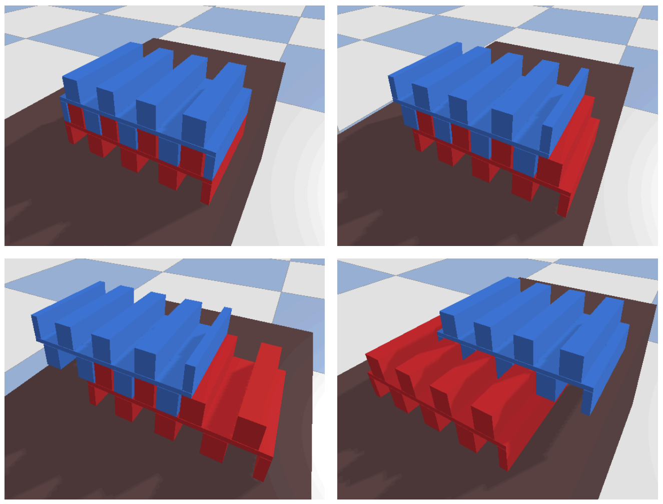


Fig. 3. Examples of stable configurations. The red block is already placed at the beginning of the episode. The robot has to place the blue block without displacing the red block.

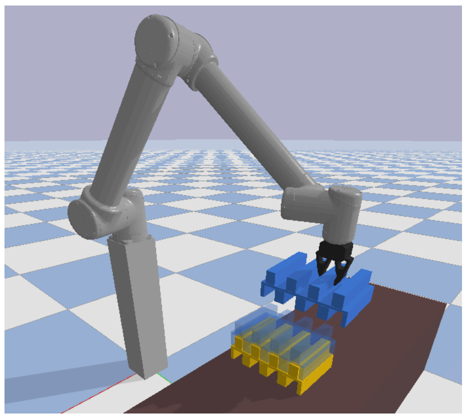


Fig. 4. Example of a starting configuration simulated in PyBullet [16]. The blue silhouette marks the target pose in which the block has to be placed.

An episode terminates if the gripper moves more than $10cm$ away from the current block or if the time limit of 3 seconds is reached. With the controller frequency of $20Hz$, this yields a maximum sequence length of 60 steps.

B. Optimization Objective

The objective is to learn a policy $\pi_\theta(\mathbf{a} \mid \mathbf{o})$ that minimizes the expected cost of an episode $\tau = (\mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{a}_{T-1}, \mathbf{s}_T)$:

$$\min_\theta \mathbb{E}_{\tau \sim p(\tau \mid \pi_\theta)} \left[\phi_f(\mathbf{s}_T) + \frac{1}{T} \sum_{t=1}^{T} \phi_i(\mathbf{s}_t, \mathbf{a}_t) \right]$$

where $\mathbf{s}_t$ is the full system state at time step t, $\mathbf{o}_t$ is the observation the policy gets of $\mathbf{s}_t$, $\mathbf{a}_t$ is the chosen action, $\phi_f(\mathbf{s}_T)$ is the final cost, $\phi_i(\mathbf{s}_t, \mathbf{a}_t)$ is the intermediate cost at step t, and

$$p(\tau \mid \pi_\theta) = p(\mathbf{s}_1) \prod_{t=1}^{T-1} p(\mathbf{s}_{t+1} \mid \mathbf{a}_t, \mathbf{s}_t) \pi_\theta(\mathbf{a}_t \mid \mathbf{o}_t) p(\mathbf{o}_t \mid \mathbf{s}_t)$$

is the trajectory distribution induced by policy π_θ.

The action vector is composed of two components: $\mathbf{a} = (\mathbf{a}_{arm}, a_{gripper})$, where the entries of $\mathbf{a}_{arm} \in \mathbb{R}^6$ are the target joint velocities of the 6 arm joints and $a_{gripper} \in [0, 1]$ is the target closure state of the gripper. Concerning the gripper control signal, a 1 indicates that the gripper shall be fully closed, while a 0 indicates that the gripper shall be fully opened. The signal is realized by a position controller on the gripper motor.

Our main objective is that both blocks of the structure are placed in their respective target pose. Hence, we chose the final cost function as follows:

$$\phi_f(\mathbf{s}_T) = \alpha_{pose} \phi_{pose}(\mathbf{s}_T) \tag{1}$$

where $\alpha_{pose} \in \mathbb{R}$ is a weighting factor. ϕ_{pose} is a penalty for the pose error of the blocks. To ensure that the robot does not displace the block that is already placed at the beginning of the episode, this penalty includes the pose errors of both blocks. Note that "pose" here refers to a combination of position and orientation. It is challenging to define a distance metric on poses directly since it is not straightforward to weight position and orientation error up in a useful manner. Thus, we chose a different approach to measuring pose error. To measure the error of the pose, we compute the average positional error of the eight vertices of the bounding box of each block:

$$d_k^2 = \frac{1}{8} \sum_{j=1}^{8} \| \mathbf{m}_{k,j} - \tilde{\mathbf{m}}_{k,j} \|_2^2$$

where $\mathbf{m}_{k,j}$ is the target position of vertex j of block k at the final time step and $\tilde{\mathbf{m}}_{k,j}$ is the actual position, respectively.

While it would be possible to use the mean of the distances d_k^2 directly as a cost function, this approach comes at a major drawback: the closer a block is to its target pose, the flatter the squared penalty becomes and the less incentive the learner has to place the block even more precisely. In construction tasks, even tiny deviations from the correct position can have a huge impact on whether a structure can be assembled properly. Due to the complex shape of the building blocks that we consider, even tiny deviations from the target poses might cause that future blocks cannot be stacked on the existing structure properly. We therefore adapt the work of Levine et al. [17] and use

$$\delta_{\log}(d^2) = d^2 + 0.01 \left(\ln(d^2 + \epsilon) - \ln(\epsilon) \right)$$

as cost function, where $\epsilon = 10^{-5}$. Due to the function's concave shape close to 0, even small positioning errors are punished significantly.

The final pose cost is then computed as

$$\phi_{pose}(\mathbf{s}_T) = \min \left\{ \frac{\frac{1}{B} \sum_{k=1}^{B} \delta_{\log}(d_k^2)}{\phi_{pose}^{\max}}, 1 \right\}$$

where $B = 2$ is the number of blocks in the scene and $\phi_{pose}^{\max}$ is a hyperparameter that controls the clipping of this term. The intuition behind the clipping is that if the block is placed too far away from the target location, we consider the task as failed and simply assign a maximum cost. Hence, the reward becomes more sparse and the Q-function in regions far away from the target state (e.g., if the robot threw the block away from itself) becomes easier to learn. The scaling of this term by $\frac{1}{\phi_{pose}^{\max}}$ simply ensures that $\phi_{pose}(\mathbf{s}_T) \in [0,1]$, which makes it easier to choose the weights α later since all cost terms will share the same domain.

The intermediate cost $\phi_i(\mathbf{s}_t, \mathbf{a}_t)$ consists of a constant term, a penalty for movements of the current block while it is not grasped, and a penalty on the acceleration of the end-effector:

$$\phi_i(\mathbf{s}_t, \mathbf{a}_t) = \alpha_{time} + \alpha_{rel}\phi_{rel}(\mathbf{s}_t) + \alpha_{acc}\phi_{acc}(\mathbf{s}_t, \mathbf{a}_t) \quad (2)$$

The first term encourages the learner to complete the episode quickly. ϕ_{rel} is a penalty for the movement of released blocks, i.e., blocks that are not grasped. Previously, the controllers often learned a strategy of throwing the block to the target location to finish the task quickly. While this strategy works well in simulation, it is not sensible for the real setup. In reality the dynamics of the blocks are more complex and less predictable due to effects that are not simulated with high accuracy, like friction. Hence, we assume that this strategy does not work as well in reality as it does in simulation. Furthermore, this strategy potentially damages the blocks and can be dangerous to humans that might work in the vicinity of the robot.

As long as the block is in the gripper, the agent should be able to move it around freely, without additional cost. The cost ϕ_{rel}, thus, is activated only if the block is released from the gripper. Whether the block is grasped is determined by checking if the distance from the block to the gripper's fingers $d_{fingers}$ is larger than $1cm$.

$$\phi_{rel}(\mathbf{s}_t) = \begin{cases} \phi_{vel}(\mathbf{s}_t) & \text{if } d_{fingers} > 0.01m \\ 0 & \text{otherwise} \end{cases}$$

The cost for the movement of a released block is realized as a quadratic penalty on the block's linear and angular velocity.

$$\phi_{vel}(\mathbf{s}_t) = \min\left\{ \frac{\|\mathbf{v}_{lin}\|_2^2 + \|\mathbf{v}_{ang}\|_2^2}{\phi_{vel}^{\max}}, 1 \right\}$$

Here, $\mathbf{v}_{lin}$ is the linear velocity of the block's center, $\mathbf{v}_{ang}$ the angular velocity, respectively, and $\phi_{vel}^{\max}$ is again a hyperparameter.

The penalty on the accelerations of the end-effector is realized as a quadratic cost on the linear and angular velocity differences between the current and the last time step.

$$\begin{aligned}\phi_{acc}(\mathbf{s}_t, \mathbf{a}_t) = \frac{1}{\phi_{acc}^{\max}} \Big(& \left\| (\mathbf{v}_{lin}^{ee})_t - (\mathbf{v}_{lin}^{ee})_{t-1} \right\|_2^2 \\ & + \left\| (\mathbf{v}_{ang}^{ee})_t - (\mathbf{v}_{ang}^{ee})_{t-1} \right\|_2^2 \Big)\end{aligned} \quad (3)$$

where $(\mathbf{v}_{lin}^{ee})_t$ and $(\mathbf{v}_{ang}^{ee})_t$ are the linear and angular velocities of the end-effector at time t, respectively. The purpose of this term is to encourage smooth movements in order to reduce energy consumption and decrease the strain on the robot's motors and gears.

C. Observation space

The observation $\mathbf{o}_t$ of the agent consists of the joint angles and velocities of both the robot arm and gripper, as well as the end-effector pose and velocity in cartesian space. For both blocks in the scene, the agent observes the center positions and orientations, as well as their linear and angular velocities. Furthermore, the target poses are included in the observations as center position and orientation.

While it is straightforward to find vector representations of all the state attributes mentioned above, some of them need a bit of tweaking to work well with neural network policies. For the joint positions of the robot, we use the two-dimensional representation $(\sin(\varphi), \cos(\varphi))^T$. In comparison to using the angles directly, this representation has the advantage that there is no discontinuity between the representations of $-\pi$ and π. To represent the 3D orientations of the end-effector and blocks, we use two columns of the respective rotation matrices. As shown by Zhou et al. [18], this representation has the advantage that it is unique and the mapping of the representation to the rotation space is continuous. Hence, the authors found that this representation is better suited for neural networks. While both these representations increase the observation dimensionality, we found that using them leads to a more stable training.

Lastly, we also add the current time step to the observations. The importance of observing the time step in tasks with time limits is shown by Pardo et al. [19]. The authors argument that the time step is inherently part of the system's state since the termination of the environment at the time limit can be seen as a transition to a terminal state that only occurs if the final time step is reached. Thus, to avoid violating the Markov property of the system, the time step needs to be included in the system state. In our case this problem can also be seen by the fact that the cost at the last time step, defined in equation (1), is very different from the intermediate cost, defined in equation (2). Hence, the agent needs a notion of time to estimate the Q-function properly. In practice, we found that adding the time step as observation significantly improved the rate of convergence, as well as the overall performance of the controller.

D. Algorithm

We decided to use the model-free reinforcement learning algorithm Twin Delayed DDPG (TD3) [20] to learn the controller since it supports continuous state and action spaces and there already exists a stable implementation [21]. Furthermore, TD3 is said to require relatively little hyperparameter tuning [22]. TD3 is a successor of the well-known algorithm Deep Deterministic Policy Gradients (DDPG) [23].

Note that the following section is formulated with rewards r instead of costs ϕ to match the formulation in the original paper [20]. The use of TD3 with our cost formulation from section III-B is straightforward since costs are nothing but negative rewards: $r = -\phi$.

TD3 learns two Q-functions Q_{θ_1} and Q_{θ_2} and a policy π_ψ, which are parameterized by neural networks. To make learning more stable, TD3 uses separate target Q-functions

$Q_{\theta'_1}$ and $Q_{\theta'_2}$ and a target policy $\pi_{\psi'}$ to update the current Q-functions Q_{θ_1} and Q_{θ_2}. The parameters of the target networks are updated to slowly track the parameters of the current networks:

$$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$
$$\psi' \leftarrow \tau\psi + (1-\tau)\psi'$$

where $\tau \in (0,1)$ is a hyperparameter.

TD3 maintains a replay buffer, in which it stores previously seen transitions as tuples (s, a, r, s'). For each update step, TD3 samples a random minibatch of N transitions (s_j, a_j, r_j, s'_j) from the replay buffer. Both Q-functions Q_{θ_1} and Q_{θ_2} are then trained to minimize the error of the Bellman equation. To counter the overestimation bias that is present in DDPG, both target Q-functions $Q_{\theta'_1}$ and $Q_{\theta'_2}$ are evaluated and the smaller of the two values is used as target. This results in the following loss function:

$$L(\theta_i) = \frac{1}{N}\sum_j (y_j - Q_{\theta_i}(s_j, a_j))^2$$
$$y_j = r_j + \gamma \min_{i=1,2} Q_{\theta'_i}(s'_j, \tilde{a}_j)$$

where $\tilde{a}_j$ denotes the action chosen by the target policy $\pi_{\psi'}$ plus Gaussian noise.

$$\tilde{a}_j = \pi_{\psi'}(s'_j) + \epsilon$$
$$\epsilon \sim \text{clip}(\mathcal{N}(0,\sigma), -\epsilon_{max}, \epsilon_{max})$$

The noise ϵ is added to enforce the notion that in a given state similar actions should result in similar values.

The policy π_ψ is trained to maximize the agent's performance J.

$$J(\psi) = \mathbb{E}\left[\sum_{t=1}^{T} \gamma^t r(s_t, \pi_\psi(s_t))\right]$$

This optimization is done by estimating the policy gradient with Q-function Q_{θ_1}:

$$\nabla_\psi J(\psi) \approx \frac{1}{N}\sum_j \nabla_a Q_{\theta_1}(s,a)|_{s=s_j, a=\pi_\psi(s_j)} \nabla_\psi \pi_\psi(s)|_{s=s_j}$$

E. Results

For training the controller, we used following values for the hyperparameters introduced in section III-B: $\alpha_{pose} = 0.6$, $\alpha_{rel} = 0.39$, $\alpha_{acc} = 0.005$, and $\alpha_{time} = 0.005$, as well as $\phi^{\max}_{pose} = 0.2$, $\phi^{\max}_{vel} = 0.1 + \frac{\pi}{4}$, and $\phi^{\max}_{acc} = 100$.

The controller was trained with 10,000 randomly generated structures as training data. Every 100,000 steps, the controller is evaluated on 1,000 unseen test structures. Figure 5 shows the evolution of the cost and the position error of the block that is placed by the robot. The controller reaches a block center position error of around $8mm$ after 12 million steps or 48 hours of training. However, in the graph it is visible that the controller reaches similar accuracy already after 4 million steps (16 hours). This clearly shows that the training could be stopped much earlier without a significant drop in task performance.

While an accuracy of $8mm$ is not perfect and could likely be improved with further hyperparameter tuning, we are confident that these results validate the feasibility of our approach and move on to the subtasks of *grasping* and *transporting* blocks.

IV. Grasping and transporting SL blocks

From now on, we consider assembling structures from SL blocks (see figure 1b). These blocks are even more challenging than the rectangular blocks since they interlock more tightly. At the moment, other students of the Robotic Assembly project are working on *placing* SL blocks on existing structures based on tactile feedback. Thus, we now tackle the *grasping* and *transporting* subtasks, described in section I-A. That is, the blocks start fixed on the table as shown in figure 6. The robot then has to grasp a block, lift it up and move it close to its placing location. For now, we assume that each block is already in the correct orientation.

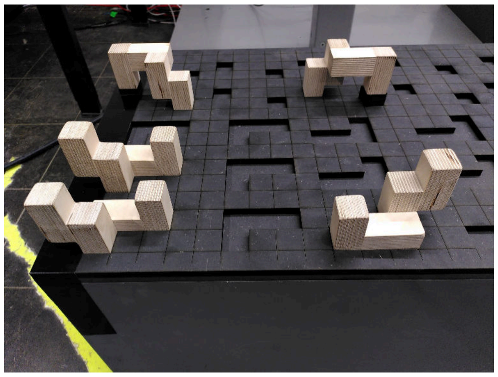
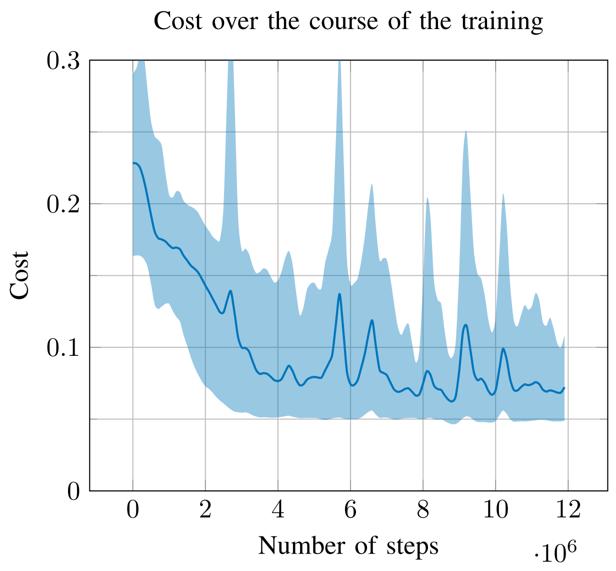


Fig. 6. In the real setup the blocks are fixed with holes in the table. This allows to provide blocks in different orientations.

We approach both *grasping* and *transporting* with sampling-based motion planning. For planning, we use the library `pybullet-planning` [24] since it allows to plan directly in the scene simulated in PyBullet. Furthermore, it supports a wide range of planning algorithms.

A. RRT-connect

We use the planning algorithm RRT-connect [25]. The algorithm is based on Rapidly-exploring Random Trees (RRTs) [26]. An RRT is a scheme for sampling in the configuration space $\mathcal{C}$ that is biased towards regions that are unexplored. Let $\mathcal{C}_{\text{free}} \subseteq \mathcal{C}$ be the set of configurations that are free, i.e., not blocked by obstacles. Each node in an RRT is a free configuration. Initially, the RRT contains only the initial node q_{init}. In each iteration, a configuration q is sampled uniformly from $\mathcal{C}$. Then the node of the RRT closest to q is selected. We call this node q_{near}. From q_{near} the algorithm takes a small step of fixed size ϵ towards q. The resulting configuration q_{new} is added to the RRT if $q_{\text{new}} \in \mathcal{C}_{\text{free}}$. This extend operation is visualized in figure 7.



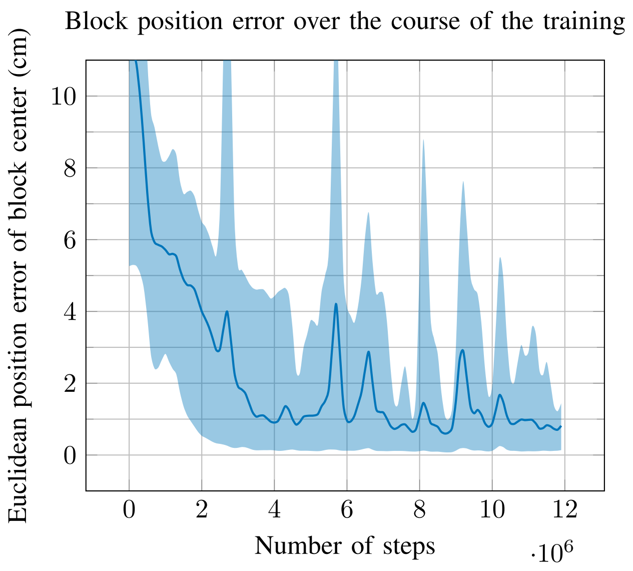


Fig. 5. Evolution of the cost and the final center position error of the block to be placed over the course of the training using TD3. *Number of steps* refers to the total number of simulator steps made at the respective point in the training. The data was obtained by evaluating the model on 1,000 unseen test configurations every 100,000 steps. The lines show the mean over the test configurations and the shaded areas show the percentiles 5 to 95. We started with a learning rate of $10^{-4.5}$ and multiplied it by a factor of $\frac{1}{\sqrt{10}}$ after 40%, 60%, 80%, 90%, and 95% of the total training time of 48 hours.

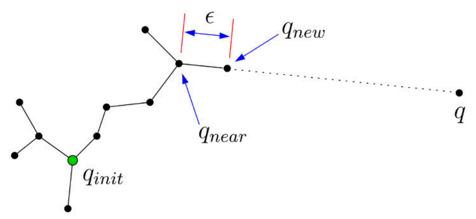


Fig. 7. Extending an RRT (image taken from [25]). First, q is sampled uniformly from the configuration space. From the closest node of the RRT a step of length ϵ is taken towards q. The resulting configuration q_{new} is added to the RRT if it is not blocked by an obstacle.

For RRT-connect [25], two RRTs are extended in an alternating fashion. One of the trees is rooted at the initial configuration q_{init} and one at the goal configuration q_{goal}. After one of the trees is extended by q_{new} as described above, the algorithm tries to connect the two trees. This is done by taking multiple steps from the other tree towards q_{new} until either that node or a blocked configuration is reached. If connecting the two trees is successful, the algorithm returns the path that connects the root nodes of the two trees. Since only unobstructed configurations are added to the RRTs, this is a collision-free path from q_{init} to q_{goal}.

B. *Planning sequence*

For grasping the planner checks both for self-collisions and collisions of the arm and gripper with obstacles, like the blocks on the table and the table itself. As soon as the block is grasped, collisions of the grasped block with the robot and obstacles are checked as well. In practice, the joint controllers exhibit tiny deviations from the target trajectory. Furthermore, the actual position of a grasped block might differ slightly from the expected position due to slippage in the gripper and inaccuracies in the localization. Both these effects potentially lead to collisions if the planned trajectory leads too close around the obstacles. Therefore, during planning, we define a margin of width d_{min} around the obstacles. If the arm or the current block penetrates into the margin of one of the obstacles, this is counted as a collision. In our experiments we found that a margin width of $d_{\text{min}} = 5mm$ is sufficient to ensure safe movements around the obstacles. For grasping the block, however, the gripper has to move closer than $5mm$ to the block, which would be counted as a collision. To solve this problem, we split the planning for grasping in two stages. First, a motion is planned with a margin width of $d_{\text{min}} = 5mm$ that brings the gripper directly above the grasp location. In the second stage, a motion is planned from this pre-grasp location to the grasp location. During the motion planning for the second stage, a reduced margin width of $d_{\text{min,reduced}} = 2mm$ is used. Since now the gripper just has to move down for a short distance, the controller deviations are not as large as for the motions of the first stage. Thus, we found that this reduced margin is sufficient for the motion.

Then the gripper has to lift up the block from the table and transport it close to its target location. As the block is fixed in a hole on the table (see figure 6), it is naturally in collision with the table. This collision would cause the planning to fail. To avoid the problem, we first move the block straight up for a short distance without checking for collisions. This is always possible since there are no obstacles above the pickup location. From this new position, we start the collision-aware motion planning to obtain a trajectory to the location from which the block can be placed.

C. *Transitioning to the real system*

After validating our approach for *grasping* and *transporting* SL blocks in simulation, we now move on to the real system. Note, that we could not use the Robotis gripper in

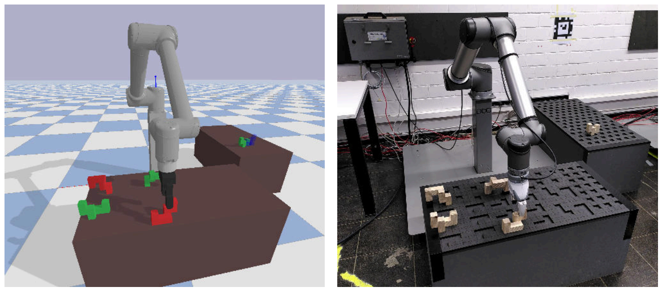


Fig. 8. The state of the real system is copied to the PyBullet simulation. Motions are then planned in the simulated scene. The resulting trajectories are visualized in simulation before the execution on the real robot.

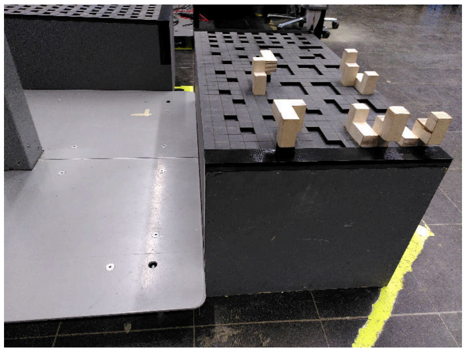


Fig. 9. The position of the table is constrained in one direction. Therefore, knowing the precise position of one of the corners is sufficient for calibration.

the real setup since it is needed in another experiment at the moment. Instead, we use a Schunk EGH gripper [27]. We use the Real-Time Data Exchange (RTDE) interface to control the UR10 robot arm. A Python API is provided by the `ur_rtde` library [28]. For the communication with the gripper, a proprietary Schunk server is executed on the control box of the UR10 arm. This server communicates with the gripper via Modbus. The gripper is then controlled with remote procedure calls to the server.

To plan motions for the real robot with `pybullet-planning`, we first extract the state of the robot, i.e., the current joint positions. This state is then copied to the robot simulated in PyBullet. Afterwards, the trajectories are planned in the simulated scene. To ensure safe operation of the robot, the trajectories are first visualized in simulation before they are executed on the robot, as shown in figure 8.

One issue that arises in the real setup is that the tables for picking up and placing the blocks are not fixed to the ground. Hence, the location of the tables do not match the simulation precisely. As displayed in figure 9, the position of the tables is constrained in one direction by the base of the robot. This ensures that both the position in this direction and the table orientation are sufficiently precise. To match the real setup, we use the robot to calibrate the positions of the tables for the simulation. For the calibration, the robot is moved by hand, so that the tool center point of the gripper is directly above the corner of the table. Since the orientation of the table is assumed to be precise, the center position of each of the tables can be determined from the location of this corner point. With this information the tables and blocks can be moved in the planning scene to the locations of their real counterparts.

As visualized in figure 10, we are able to grasp blocks on the pickup table and move them to their respective target locations on the placing table autonomously. For now, placing the blocks is done by an open-loop controller as well. As mentioned earlier, in the future placing will be done by a controller that utilizes tactile feedback.

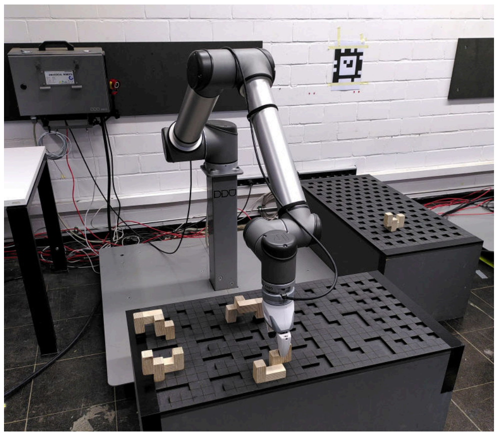
V. Conclusion and Future Work

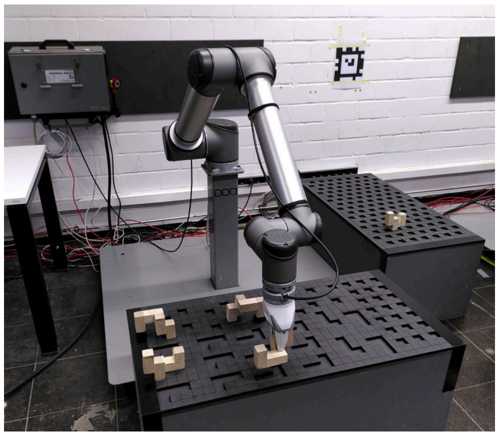
In this work we tackled the problem of autonomously constructing modular structures. We separated the problem of adding a block to a structure into three subtasks: *grasping*, *transporting*, and *placing*. We regarded two different kinds of building blocks, the *rectangular block* and the *SL block*. For the *rectangular block* we tackled the *placing* subtask with reinforcement learning. We showed that the learned controller is able to stack two blocks with an average accuracy of around $8mm$ in simulation. For the *SL block*, we used motion planning to solve the *grasping* and *transporting* subtasks for the real system.

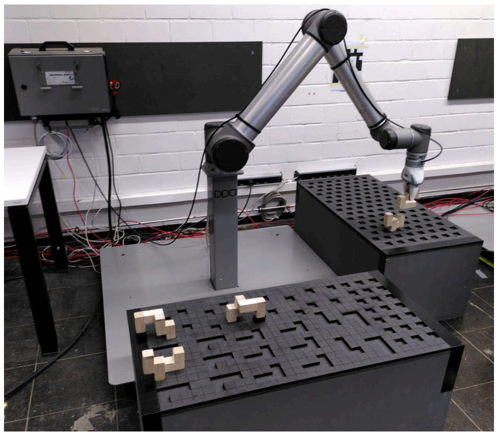
At the moment, the learned placing controllers are able to build structures consisting of exactly two blocks. In order to construct arbitrary structures, a controller is needed that is able to deal with structures consisting of a variable number of blocks. Since every structure is different, the robot needs a way of sensing where other blocks of the structure are in order to place the next block without collisions. Here, a major challenge is to process an arbitrary number of block poses and achieve invariance to permutations. Both of these challenges could be tackled with Graph Convolutional Networks [29]. Alternatively, these challenges could be approached by using images of an RGB or depth camera as input to the policy instead of the block poses. However, using camera images introduces a series of new challenges, as some blocks might be occluded and the robot additionally has to learn to interpret complex visual signals.

Furthermore, only the planning-based controller for *grasping* and *transporting* has been transferred to the real robot. For transferring the learned *placing* controller there are still some open challenges. First of all, especially when contact is happening, simulators tend to be fairly inaccurate, which necessitates fine-tuning the controller with the real system dynamics. Additionally, some measurements are not easy to obtain in the real system. One such example is the pose of the blocks, which is used in the cost function and as observation for the policy. In the real setup, these poses can only be obtained from camera images. There are currently two student groups in the Robotic Assembly project working on extracting block poses from camera images.

Using RRT-connect as planning algorithm sometimes leads to motions that go very far around obstacles. While this is a valid solution to the planning problem, the solution is far from optimal. This means that the robot consumes more time


(a) The gripper moves to the next block


(b) The gripper picks up the block


(c) The block is moved above the target location

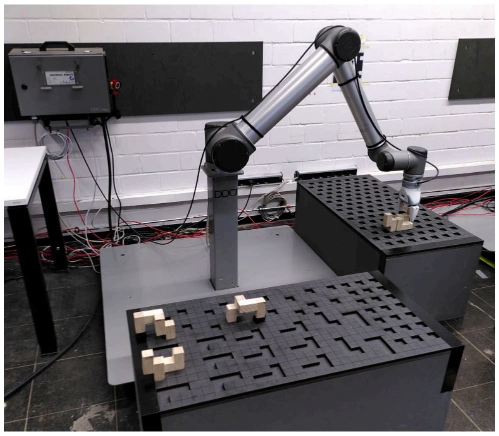

(d) The block is placed at the target location

Fig. 10. Assembly with the real setup. The robot arm picks up a block and places it at a predefined location in the existing structure on the second table. The trajectories for these motions are planned in a simulated scene that is calibrated to match the real scene.

and energy than necessary to solve the problem. It also means that the movements of the robot are less predictable, which could lead to problems should the robot work with humans in a shared workspace. These suboptimal paths are a problem that is inherent to RRTs. In fact, Karaman and Frazzoli [30] proved that even in the infinite steps limit, planning with standard RRTs does not find the shortest possible path. The authors propose a modified version of the algorithm, which is called RRT*. In RRT* new nodes are not necessarily connected to the closest node, but to the node with the smallest cumulative path length in a neighborhood around the new node. Furthermore, existing nodes in the neighborhood are rewired if the path over the new node is shorter than the previously shortest path. The authors prove that this modified algorithm converges to the optimal solution in the infinite steps limit. In future work, experiments with RRT* could be conducted to see whether this planner avoids the problem described above.

For now, we assumed that all blocks are already in the correct orientation. To remove this limitation, the robot needs to be able to re-orient blocks. Since blocks on the table cannot be grasped from every direction and the gripper must be in a specific orientation for placing a block, such a re-orientation cannot always be achieved by only rotating the gripper. Ali and Lee [31] solve this problem by planning a sequence of stable poses in which the block is laid down on the table and re-grasped from a different angle. Alternatively, two separate robot arms could be used for picking and placing. In this scenario one robot picks up the block and hands it over to the other robot in a desired orientation. Since the block is lifted up from the table during the hand-over, the second robot can grasp it more freely in order to then place it in the target orientation.

At the moment, the order in which the blocks should be placed needs to be specified manually. To facilitate the task of specifying the target structure, it is sensible to instead plan this construction order automatically. For a construction order to be feasible, the assembled structure needs to be stable at every intermediate step. Furthermore, there must be a valid trajectory for the robot to place each block without being obstructed by previously placed blocks. Determining such a feasible construction order requires solving a task and motion planning problem similar to that tackled by Hartmann et al. [1].

References

[1] V. N. Hartmann, O. S. Oguz, D. Driess, M. Toussaint, and A. Menges, "Robust task and motion planning for long-horizon architectural construction planning," *arXiv preprint arXiv:2003.07754*, 2020.

[2] A. Adel, A. Thoma, M. Helmreich, F. Gramazio, and M. Kohler, "Design of robotically fabricated timber frame structures," 2018.

[3] V. Bapst, A. Sanchez-Gonzalez, C. Doersch, K. Stachenfeld, P. Kohli, P. Battaglia, and J. Hamrick, "Structured agents for physical construction," in *International Conference on Machine Learning*. PMLR, 2019, pp. 464–474.

[4] Health and Safety Executive, "Construction statistics in Great Britain," https://www.hse.gov.uk/statistics/industry/construction.pdf, 2020, Accessed: 2021-03-08.

[5] H. Ardiny, S. Witwicki, and F. Mondada, "Are autonomous mobile robots able to take over construction? a review," 2015.

[6] "Universal Robot UR10," https://www.universal-robots.com/products/ur10-robot, Accessed: 2021-03-03.

[7] "Robotis hand RH-P12-RN," http://www.robotis.us/robotis-hand-rh-p12-rn, Accessed: 2021-03-08.

[8] A. D. Laud, "Theory and application of reward shaping in reinforcement learning," Tech. Rep., 2004.

[9] M. Toussaint, "Logic-geometric programming: An optimization-based approach to combined task and motion planning." in *IJCAI*, 2015, pp. 1930–1936.

[10] D. Driess, J.-S. Ha, and M. Toussaint, "Deep visual reasoning: Learning to predict action sequences for task and motion planning from an initial scene image," *arXiv preprint arXiv:2006.05398*, 2020.

[11] T. Lozano-Pérez and L. P. Kaelbling, "A constraint-based method for solving sequential manipulation planning problems," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014, pp. 3684–3691.

[12] X. Chen and P. Van Beek, "Conflict-directed backjumping revisited," *Journal of Artificial Intelligence Research*, vol. 14, pp. 53–81, 2001.

[13] C. R. Garrett, R. Chitnis, R. Holladay, B. Kim, T. Silver, L. P. Kaelbling, and T. Lozano-Pérez, "Integrated task and motion planning," *arXiv preprint arXiv:2010.01083*, 2020.

[14] G. Thomas, M. Chien, A. Tamar, J. A. Ojea, and P. Abbeel, "Learning robotic assembly from CAD," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 3524–3531.

[15] G. Sartoretti, Y. Wu, W. Paivine, T. S. Kumar, S. Koenig, and H. Choset, "Distributed reinforcement learning for multi-robot decentralized collective construction," in *Distributed Autonomous Robotic Systems*. Springer, 2019, pp. 35–49.

[16] "Bullet real-time physics simulation," https://pybullet.org, Accessed: 2021-03-03.

[17] S. Levine, N. Wagener, and P. Abbeel, "Learning contact-rich manipulation skills with guided policy search," *CoRR*, vol. abs/1501.05611, 2015. [Online]. Available: http://arxiv.org/abs/1501.05611

[18] Y. Zhou, C. Barnes, J. Lu, J. Yang, and H. Li, "On the continuity of rotation representations in neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[19] F. Pardo, A. Tavakoli, V. Levdik, and P. Kormushev, "Time limits in reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2018, pp. 4045–4054.

[20] S. Fujimoto, H. Van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," *arXiv preprint arXiv:1802.09477*, 2018.

[21] "TD3," https://github.com/sfujim/TD3, Accessed: 2021-03-09.

[22] "Reinforcement learning tips and tricks," https://stable-baselines.readthedocs.io/en/master/guide/rl_tips.html, Accessed: 2021-03-08.

[23] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[24] "pybullet-planning," https://github.com/caelan/pybullet-planning, Accessed: 2021-03-03.

[25] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, vol. 2. IEEE, 2000, pp. 995–1001.

[26] S. M. LaValle *et al.*, "Rapidly-exploring random trees: A new tool for path planning," 1998.

[27] "SCHUNK EGH Gripper," https://schunk.com/de_en/homepage/egh/, Accessed: 2021-03-10.

[28] "ur_rtde," https://gitlab.com/sdurobotics/ur_rtde, Accessed: 2021-03-03.

[29] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016. [Online]. Available: http://arxiv.org/abs/1609.02907

[30] S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning," *Robotics Science and Systems VI*, vol. 104, no. 2, 2010.

[31] A. Ali and J. Y. Lee, "Integrated motion planning for assembly task with part manipulation using re-grasping," *Applied Sciences*, vol. 10, no. 3, p. 749, 2020.