Constellation - An Algorithm for Finding Robot Configurations that Satisfy Multiple Constraints

Peter Kaiser[1] Dmitry Berenson[2] Nikolaus Vahrenkamp[1] Tamim Asfour[1]
Rüdiger Dillmann[1] Siddhartha Srinivasa[3]

***Abstract*— Planning motion for humanoid robots requires obeying simultaneous constraints on balance, collision-avoidance, and end-effector pose, among others. Several algorithms are able to generate configurations that satisfy these constraints given a good initial guess, i.e. a configuration which is already close to satisfying the constraints. However, when selecting goals for a planner a close initial guess is rarely available. Methods that attempt to satisfy all constraints through direct projection from a distant initial guess often fail due to opposing gradients for the various constraints, joint-limits, or singularities. We approach the problem of generating a constrained goal by searching for a configuration in the intersection of all constraint manifolds in configuration space (C-space). Starting with an initial guess, our algorithm, *Constellation*, builds a graph in C-space whose nodes are configurations that satisfy one or more constraints and whose cycles determine where the algorithm explores next. We compare the performance of our approach to direct projection and a previously-proposed cyclic projection method on reaching tasks for a humanoid robot with 33 DOF. We find that Constellation performs the best in terms of the number of solved queries across a wide range of problem difficulty. However, this success comes at higher computational cost.**

I. Introduction

The structure of humanoid robots and the tasks they are expected to perform can severely restrict the set of feasible robot configurations. Humanoids must commonly obey simultaneous constraints on balance, collision-avoidance, and end-effector pose, among others. The feasible set of configurations is actually the intersection of the constraint manifolds corresponding to each of these constraints. In order to plan for humanoid motion, a planner must be given a goal configuration within this intersection.

Finding such a configuration is challenging because the constraints we consider are not given in closed form and there is no explicit representation of them in configuration space (C-space). Furthermore, the manifolds of feasible configurations for these constraints are known to be non-convex and disjoint [1] and some constraints, such as those on end-effector pose, induce lower-dimensional constraint manifolds. Though random sampling can be used to find configurations that meet some constraints, lower-dimensional manifolds have no volume in C-space, thus the probability of generating a random sample on such a manifold is 0.

[1]Institute for Anthropomatics, Karlsruhe Institute of Technology, Karlsruhe, Germany peter.kaiser@student.kit.edu, {vahrenkamp, asfour, dillmann}@kit.edu [2]EECS Deparment, University of California, Berkeley, Berkeley, CA, USA berenson@eecs.berkeley.edu [3]The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, USA siddh@cs.cmu.edu

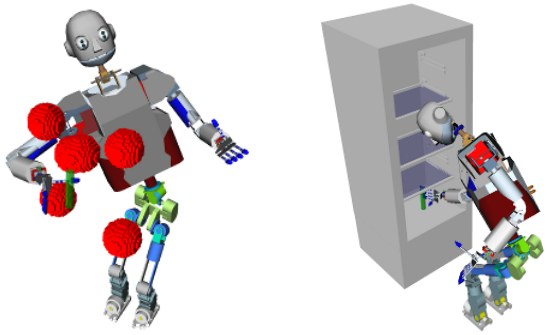


Fig. 1. The humanoid ARMAR-III [3], virtually enhanced with two 6 DOF legs (33 DOF total), reaching in cluttered environments while satisfying multiple constraints. The legs are a scaled-up version of the iCub legs [4].

Sentis and Khatib [2] showed that, given an initial guess close to the intersection of many constraint manifolds, projecting the guess to one of the constraint manifolds while projecting to the others in the null-space of the first constraint is often sufficient to generate a feasible configuration. This strategy, which we term *direct projection*, can be used to meet a variety of simultaneous constraints for humanoid robots. However, for many tasks, such as a humanoid reaching for an arbitrary target in a cluttered environment, a close initial guess is rarely available. Direct projection from a distant guess can fail due to opposing gradients for the various constraints, joint-limits, or singularities.

To generate a configuration near the intersection set our algorithm, *Constellation*, first uses direct projections from an initial guess to generate configurations that meet at least one constraint. These configurations are then used as nodes in a graph. Two nodes are connected by an edge depending on which constraints they satisfy and the distance between them. At each iteration of Constellation, we find the shortest cycle in this graph that contains a node on each constraint manifold. We then use that cycle's nodes as input to a method based on downhill-simplex [5] that outputs a new guess configuration. This guess configuration is then projected to all the constraint manifolds and the nodes resulting from these projections are added to the graph along with appropriate edges. This process is repeated until a node satisfying all constraints is found. By focusing on short cycles of nodes that, together, meet all constraints, the algorithm is able to explore areas of C-space where the intersection of all constraint manifolds is likely to occur. Also, since Constellation uses direct projection as its underlying projection operator, all problems that can be solved by a single direct projection

from a given initial guess will be solved by Constellation.

In the remainder of this paper we first describe the Constellation algorithm and how we compute displacements and Jacobians for the various constraints. We then compare the performance of our approach to direct projection and a previously-proposed cyclic projection method [6], which attempts to find the intersection by iteratively projecting to a repeating sequence of constraints. We also compare to a new variant of the cyclic projection method that uses the null-space when projecting. The four approaches are evaluated on reaching tasks for a 33 DOF humanoid, which was created by combining the upper body of ARMAR-III [3] with the scaled-up legs of iCub [4] (see Fig. 1). We find that Constellation performs the best in terms of the number of solved queries across a wide range of problem difficulty. However, this success comes at higher computational cost.

II. Related Work

The projection methods in Constellation are derived from iterative inverse-kinematics techniques [2], [7] and gradient-descent controllers [8], [9], [2].

Previous methods in motion planning and configuration optimization have concentrated on generating configurations that satisfy a specific class of constraints, such as balance [10], closed-chain kinematics [11] [12] [13], and task space constraints [14] [15] [16]. While these methods are effective, each is restricted to constraints of a certain type and/or dimensionality. We seek an approach that can generalize over a wide range of constraints relevant to humanoid robots, i.e. a unified method that is able to satisfy constraints of varying types and dimensionalities.

Randomized Gradient Descent (RGD) [17] can be used to iteratively project a configuration toward an arbitrary constraint [18]. Though RGD is quite general, it requires significant parameter-tuning and can take quite a long time to converge to within a small tolerance of the constraint, especially in high-dimensional spaces, thus it is not appropriate for our application.

Another approach to finding configurations that satisfy multiple constraints is to frame the task as a general optimization problem and to use a solver like FSQP, as in [19]. An overview of such global optimization methods can be found in [20]. While global optimization algorithms can be applied to many problems, they are often quite time-consuming and require that constraints meet certain criteria such as differentiability. Constraints such as collision-avoidance for complex geometries are difficult to encode in a way that is consistent with such criteria.

A method that is related to Constellation is cyclic projection [6], which iteratively projects a configuration to a repeating sequence of constraints. This algorithm does not take advantage of the null-space of a constraint, which we show is quite useful for guiding the search toward the intersection of the constraints. We compare Constellation to cyclic projection in Section V.

Algorithm 1: projectToConstraint($q_s, c_{prim}, \mathcal{C}_{sec}$)

while *NotStalledOrFailed()* **do**
  $\Delta x \leftarrow$ getDisplacement($q_s, c_{prim}, \delta_{prim}$)
  $\Delta x_{sec} \leftarrow$ getDisplacement($q_s, \mathcal{C}_{sec}, \delta_{sec}$)
  if $||\Delta x|| < \varepsilon$ *and* $||\Delta x_{sec}|| < \varepsilon$ **then**
    return q_s
  end
  $J \leftarrow$ getJacobian(q_s, c_{prim})
  $J_{sec} \leftarrow$ getJacobian($q_s, \mathcal{C}_{sec}$)
  $\Delta q_{error} \leftarrow J^{\#}\Delta x + (I - J^{\#}J)J_{sec}^{T}\Delta x_{sec}$
  $q_s \leftarrow q_s - \Delta q_{error}$
end
return *NULL*

III. The Constellation Algorithm

Our approach to finding a configuration that obeys a given set of constraints $\mathcal{C}$ is based on the idea of choosing a guess configuration q in each iteration and projecting it to each constraint $c \in \mathcal{C}$. Before discussing the algorithm in detail, we briefly explain how to project a configuration to a constraint.

A. Projecting a configuration to a constraint

A configuration q is projected to a constraint c by first computing the configuration's displacement Δx to c. This displacement can exist in an arbitrary space (commonly task space) as long as a Jacobian is available to map displacements in C-space to this space. For instance, for a task space constraint, the displacement is the difference vector between the current end-effector pose and the target pose.

Direct projection to meet a constraint c is then implemented as a gradient descent (as shown in Alg. 1).

Apart from the primary constraint c_{prim} that is the target for projection and the start configuration q_s, the gradient descent is given a set $\mathcal{C}_{sec}$ of secondary constraints, which should be satisfied if possible. $\mathcal{C}_{sec}$ consists of all constraints in $\mathcal{C}$ except the primary one. We attempt to satisfy the secondary constraints in the null-space of the primary constraint [2]. The Jacobians and displacements of the secondary constraints are stacked in J_{sec} and Δx_{sec}, respectively.

When obtaining the displacements Δx and Δx_{sec}, the *getDisplacement* function is given the step sizes δ_{prim} and δ_{sec}, respectively. The displacements are clamped to the step size if they are longer. The projection fails if $||\Delta x||$ increases and stalls if the change in Δx is smaller than a certain threshold. Both conditions are accounted for by the *NotStalledOrFailed* Function in Alg. 1.

B. The Constellation algorithm

A common approach to finding a configuration in the intersection of several constraint manifolds is to perform direct projection as described above. However, if the initial configuration is far from the intersection of constraint manifolds, the projection can fail to find a solution that satisfies all constraints due to opposing gradients for the various constraints, joint-limits, or singularities.

Instead of trying to solve the problem using a single direct projection, the Constellation algorithm (Alg. 2) iteratively grows a graph G out of the projections of guess configurations. The graph is then used to find a promising next guess, whose projections again extend the graph.

The Constellation algorithm (Alg. 2) begins by generating k random configurations in the D dimensional C-space and applying direct projections to those configurations using a different c as the primary constraint for each projection. A direct projection is only considered successful if it results in a configuration that meets the primary constraint (note that the resulting configuration may satisfy other constraints as well). The configurations generated by successful direct projections are added as nodes to a graph G (Alg. 3). After inserting these nodes, the algorithm creates edges between nodes using the method described in Alg. 4, the main idea being to connect nodes that satisfy different sets of constraints. The algorithm then finds the cycle of minimum length in the graph such that each constraint in $\mathcal{C}$ is met by at least one node in the cycle. This cycle marks a region where the constraint manifolds lie closer together than at any other place (according to the algorithm's current knowledge of the space), thus we would like to explore this area further. Constellation then computes a guess configuration using the nodes of this cycle (Alg. 5) and repeats the above process to generate more nodes. This procedure repeats until a direct projection produces a configuration that satisfies all constraints, the algorithm has considered all cycles in the graph, or the time limit is reached. A flowchart of the algorithm is shown in Fig. 2 and an example run of the algorithm is depicted in Fig. 3.

C. Balancing Exploration and Exploitation

As in most search algorithms, we need to balance exploration and exploitation; i.e. how much Constellation searches unexplored areas of the space vs. how much it searches near promising areas that have been explored previously. Our method for balancing these two aspects is encoded in the *getShortestCycle* function.

To avoid over-exploitation (i.e. choosing the same or similar cycles repeatedly), the *getShortestCycle* function maintains two blacklists: a cycle blacklist and a node blacklist. After a cycle is processed by Constellation, it is added to the cycle blacklist and its nodes are added to the node blacklist. The *getShortestCycle* function then selects the shortest cycle that is not in the cycle blacklist and does not contain a node in the node blacklist. If there is no cycle that contains no blacklisted nodes, the node blacklist is cleared (this allows Constellation to explore near promising previously-explored areas). If all cycles in G have been blacklisted, Constellation returns failure.

D. Generating the next guess

The method that generates the next guess configuration is a key component of Constellation and heavily influences the performance of the algorithm. All calculations that are necessary to derive the next guess from a cycle are encapsulated in the *generateNextGuess* function and thus can be replaced easily.

The method presented here is based on the Downhill Simplex Algorithm (or Nelder-Mead Algorithm) for the optimization of multi-dimensional non-linear functions [5]. To apply this algorithm, we treat the configurations of the nodes of a given cycle as samples of a cost function. The cost function value at a node's configuration is the sum of the displacements to each constraint[1], thus the cost at a configuration which satisfies all constraints would be zero.

Algorithm 2: Constellation($\mathcal{C}$, k)

```
G ← {}
i ← 1
while TimeRemaining() do
    if i ≤ k then
        q ← randomSample()
        i ← i + 1
    end
    else
        Z ← getShortestCycle(G)
        if Z = NULL then
            return NULL
        end
        q ← generateNextGuess(Z, C)
    end
    q_res ← addSampleProjections(G, C, q)
    if q_res ≠ NULL then
        return q_res
    end
end
return NULL
```

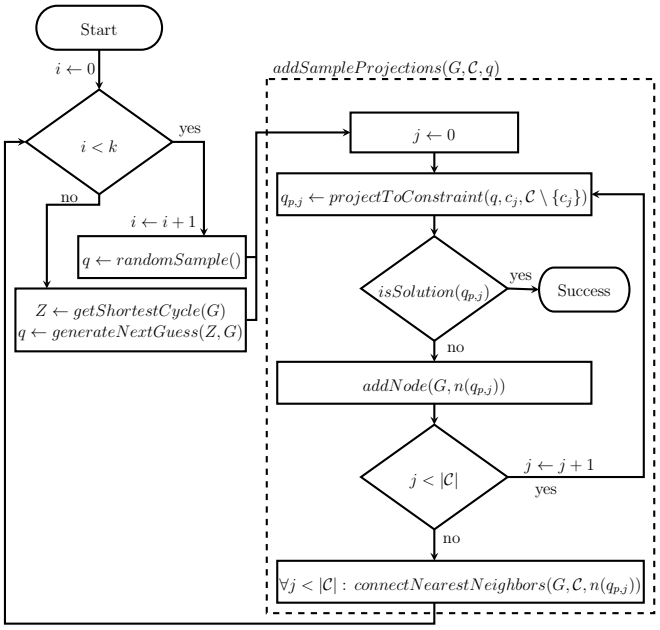
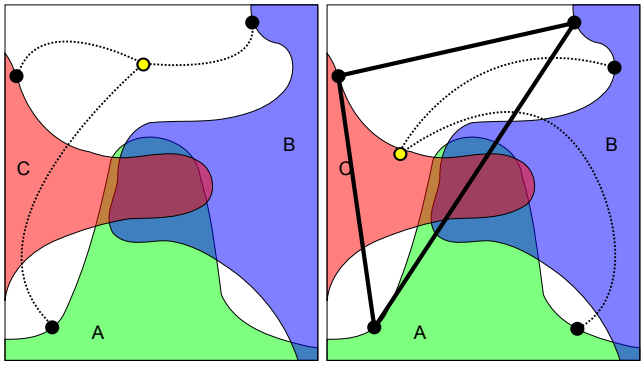
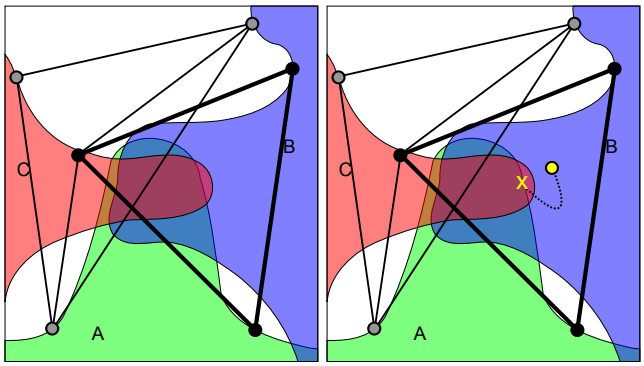


Fig. 2. Flowchart of the Constellation algorithm.

[1]These displacements are calculated in the space of each constraint, for instance in task space for end-effector pose constraints.



(a) (b)



(c) (d)

Fig. 3. The Constellation algorithm attempting to find the intersection of the constraint manifolds A, B and C in C-space. Note that there is no guarantee that projections find the closest point on the manifold to the starting configuration. (a) The initial guess configuration is projected to the manifolds A, B and C. The guess configuration is shown in yellow, and graph nodes are shown in black. (b) Each node connects to the closest node on each constraint it does not satisfy. The graph has only one cycle, which is now used to determine the next guess. (c) The resulting nodes of projections from the guess configuration are inserted into the graph and connected as in (b). The previous cycle and its nodes are blacklisted (shown in gray). The new cycle of minimal length that contains no blacklisted nodes is highlighted. (d) A new guess is generated and projected to the manifold C, which produces a configuration in the intersection of all manifolds, thus solving the problem.

An iteration of the downhill simplex algorithm can then be applied to try to generate a new lower-cost configuration, i.e. one closer to the intersection of all constraints.

This method (shown in Alg. 5) reflects the configuration n_q of the worst node n (i.e the node with the highest cost, as computed by Alg. 6) of the cycle Z through the weighted average of the configurations of the other cycle nodes $Z \backslash \{n\}$. An example of how a next guess configuration is generated is depicted in Fig. 4.

The last component of the *generateNextGuess* method to discuss is the way the weighted average of a set of nodes is computed. If we simply average the nodes' configurations without any weighting, each node would have the same influence on a joint-value in the average, regardless of this joint being used in projecting to the given constraint.

To see why this is a problem, let us consider an example problem where we have two simultaneous constraints: A pose constraint for the left hand and a pose constraint for the right hand. The left arm's joints are not important to project to

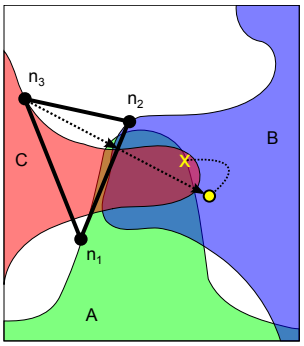


Fig. 4. Determining the next guess from a cycle consisting of the nodes n_1, n_2 and n_3. Node n_3 is the worst node according to the sum of the displacement lengths to the constraints A, B and C. To determine the next guess configuration, the worst node n_3 is reflected through the weighted average of the cycle's other nodes n_1 and n_2. Projecting the guess configuration to constraint A then solves the problem.

Algorithm 3: addSampleProjections($G, \mathcal{C}, q$)

```
1  N ← {}
2  for c ∈ C do
3      q_p ← projectToConstraint(q, c, C \ {c})
4      if q_p ≠ NULL then
5          if isSolution(q_p) then
6              return q_p
7          end
8          addNode(G, n(q_p))
9          N ← N ∪ n(q_p)
10     end
11 end
12 for n ∈ N do
13     connectNearestNeighbors(G, C, n)
14 end
15 return NULL
```

the right hand constraint. Thus the corresponding columns in the Jacobians used in the projection would be zero and hence the left arm's joint values would be unchanged during the projection to the right hand constraint (ignoring the null-space component). The next guess configuration would be biased toward these unchanged joint values if we simply averaged all configurations within the cycle. This would be a problem because the left arm's joints would be pulled back toward their initial values by the node that satisfies the right arm constraint and the right arm's joints would be pulled back similarly by the node that satisfies the left arm constraint. To prevent this problem, we try to decrease the influence of unchanged joint-values by using a weighted average, as shown in Alg. 7.

The *weightedAverage* method considers the sum of all Jacobians that were necessary to project a configuration to a certain constraint. This Jacobian sum is then stored in the resulting node's meta-information. Note that the calculation of the Jacobian sum is not shown in Alg. 1 to improve readability.

To compute the weighted average, we consider each node of the given set $\mathcal{N}$ and retrieve the corresponding Jacobian sum. A column of the Jacobian sum $\mathbf{J}_{*i}$ captures the con-

Algorithm 4: connectNearestNeighbors($G, \mathcal{C}, n$)

```
for c ∈ C do
    if not satisfiesConstraint(n, c) then
        m ← getClosestSatisfyingNode(G, c, n)
        addEdge(G, e(n, m))
    end
end
```

Algorithm 5: generateNextGuess($Z, \mathcal{C}$)

```
n ← getWorstNode(Z, C)
q ← weightedAverage(Z \ {n})
return q + (q − n_q)
```

Algorithm 6: getWorstNode($Z, \mathcal{C}$)

```
w ← {}
d_max ← 0
for n ∈ Z do
    d ← 0
    for c ∈ C do
        d ← d+getDisplacement(n, c)
    end
    if d > d_max then
        w ← n
        d_max ← d
    end
end
return w
```

tribution of one joint while the norm of this column $||\mathbf{J}_{*i}||$ is considered as a measure for the importance of joint i for the projection resulting in node n. We then weigh each joint value by its importance and the reciprocal of the sum of all importances of the current node before adding it to the accumulator b. *weightedAverage* then returns the weighted average after all nodes' contributions have been considered.

IV. Constraints

To apply the Constellation algorithm, we must be able to perform Alg. 1 on a given constraint. Thus, for each type of constraint, we must have a method to calculate 1) a displacement to the constraint and 2) a Jacobian that maps displacements in C-space to displacements in the space of the constraint.

In this section, we will discuss three example constraints that we implemented and used together with the Constellation algorithm: a modified TSR constraint, a balance constraint, and a collision constraint.

A. The Modified TSR Constraint

A Task Space Region (TSR) is a volume in the task space that the robot's end-effector has to be placed in to satisfy the constraint [1]. We briefly summarize how to compute the displacement to a TSR below.

Algorithm 7: weightedAverage($\mathcal{N}$)

```
b ← 0_{1×D}
s ← 0_{1×D}
for n ∈ N do
    J ← getJacobianSum(n)
    a ← Σ_{i=1}^{D} ||J_{*i}||
    for i ∈ {1, ··· , D} do
        s_i ← s_i + (1/a) · ||J_{*i}||
        b_i ← b_i + (1/a) · ||J_{*i}|| · n_{q_i}
    end
end
for i ∈ {1, ··· , D} do
    b_i ← b_i / s_i
end
return b
```

Throughout this section, we will be using transformation matrices of the form $\mathbf{T}_b^a$, which specifies the pose of b in the coordinates of frame a.

A TSR consists of three parts:

- $\mathbf{T}_w^0$: transform from the origin to the TSR frame w
- $\mathbf{T}_e^w$: end-effector offset transform in the w frame
- $\mathbf{B}^w$: 6×2 matrix of bounds in the coordinates of w:

$$\mathbf{B}^w = \begin{bmatrix} x_{min} & x_{max} \\ y_{min} & y_{max} \\ z_{min} & z_{max} \\ \psi_{min} & \psi_{max} \\ \theta_{min} & \theta_{max} \\ \phi_{min} & \phi_{max} \end{bmatrix} \quad (1)$$

The first three rows of $\mathbf{B}^w$ bound the allowable translation along the x, y, and z axes (in meters) and the last three bound the allowable rotation about those axes (in radians), all in the w frame. The Roll-Pitch-Yaw (RPY) Euler convention is used for the rotational bounds.

Given a TSR, [1] describes how to compute the task space displacement $\Delta\mathbf{x}$ to that TSR. The Jacobian for the TSR constraint is simply the Jacobian of the end-effector it corresponds to.

If we project to a TSR from a configuration which is close to satisfying the constraint and we are not concerned with a rotation dimension of the task space (for instance, the hand's yaw around a cylinder), we can set the rotation bounds to $-\pi$ and π. The task space displacement for this component would then always be zero, which forces a projection to keep the yaw value of the starting pose.

However, since Constellation projects to TSRs from distant configurations, we have found that allowing more freedom yields better results. When a dimension of task space is totally unconstrained, we take the corresponding components out of the task space displacement and the Jacobian, thus allowing full freedom in the unconstrained dimension.

B. The Balance Constraint

The second type of constraint we consider is the balance constraint, which is satisfied if the robot's center of gravity—projected to the ground—lies in the support polygon of the

robot. For a humanoid, the support polygon would be the region under and in-between the feet. A detailed discussion of the balance constraint can be found in [8].

Let l_i represent the ith link of the robot, and let m_i and x_{cog_i} represent that link's mass and center of gravity, respectively. The robot's center of gravity is

$$x_{cog} = \frac{1}{\sum_{i=1}^{N} m_i} \sum_{i=1}^{N} m_i \cdot x_{cog_i}, \tag{2}$$

where N is the total number of links. Let S represent the robot's support polygon and let the function $P : \mathbb{R}^2 \rightarrow S$ be the projection of a point $x \in \mathbb{R}^2$ to its closest point in S.

The displacement Δx for the balance constraint is the negative vector from x_{cog} to its closest point in the support polygon (Note that this operation is done in the two dimensional ground plane, thus the z-component of x_{cog} is not considered):

$$\Delta x = [x_{cog_x}, x_{cog_y}]^T - P([x_{cog_x}, x_{cog_y}]^T). \tag{3}$$

The Jacobian for the balance constraint is computed similarly to the center of gravity:

$$J = \frac{1}{\sum_{i=1}^{N} m_i} \sum_{i=1}^{N} m_i \cdot J_{l_i}(x_{cog_i}), \tag{4}$$

where $J_{l_i}(x_{cog_i})$ represents the Jacobian for the center of gravity of the i-th link.

C. The Collision Constraint

Our approach to computing the displacement and Jacobian for collision constraints is similar to that used in CHOMP [21], which attempts to maximize a configuration's distance to obstacles.

Let $R(q) \subseteq \mathbb{R}^3$ be the space occupied by the robot at the configuration q. Let $O \subseteq \mathbb{R}^3$ represent the space occupied by obstacles in the scene. Let $C(q) = R(q) \cap O$. A displacement which, when applied, will satisfy this constraint will move all points in $C(q)$ outside of O. We use a voxel grid along with a Signed Distance Field (SDF) to approximately compute such a displacement:

Before running Constellation, we voxelize the environment (producing a voxel grid V) and sample a set of points X within the volume of the robot (see Fig. 5). We also compute an SDF over V, which produces a value $S(v)$ for each voxel grid cell $v \in V$. $S(v)$ stores the distance from the center of cell v to the boundary of the nearest obstacle. Values of S are negative inside obstacles, positive outside, and zero at the boundary.

When the robot is placed in a configuration q, we use forward kinematics to determine where the points X are located. For each point $x_i \in X$, we compute the voxel grid cell v_i that contains the point. If $S(v_i)$ is negative, we compute the displacement Δx_i as the difference in S value between voxels neighboring v_i along each spatial dimension. We can then stack the displacements to produce

$$\Delta x = \left[\Delta x_i^T, \Delta x_{i+1}^T, ...\right]^T. \tag{5}$$

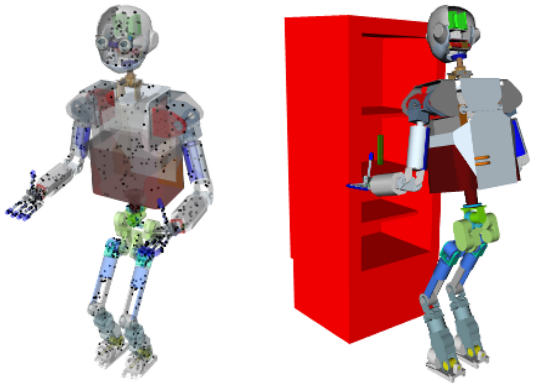


Fig. 5. Left: The set of collision points. Right: Voxelized refrigerator.

To compute the Jacobian for the collision constraint, we first compute a Jacobian $J(q, x_i)$ for each x_i that yields a negative $S(v)$ value. We can then stack Jacobians to produce

$$J = \begin{bmatrix} J(q, x_i) \\ J(q, x_{i+1}) \\ \vdots \end{bmatrix}. \tag{6}$$

V. Results

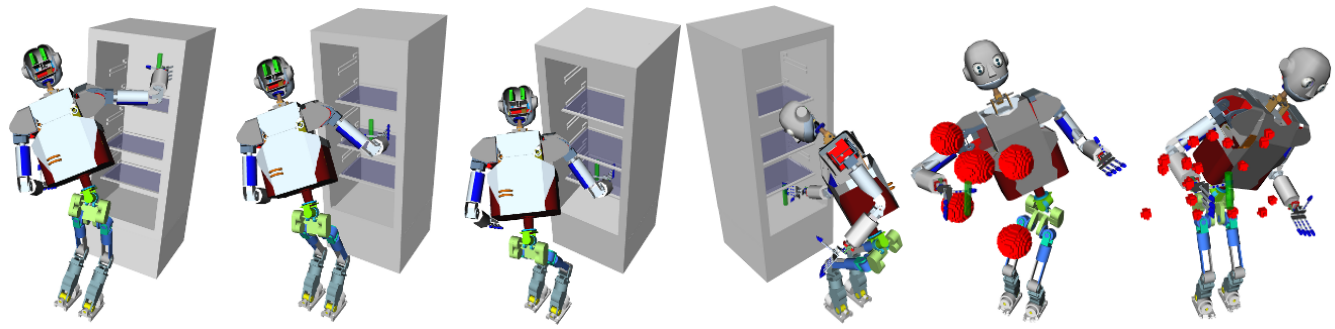
We now present the results of several test scenarios in which we compare Constellation to direct projection, cyclic projection, and cyclic projection with null-space. In the evaluations, we use an enhanced model of the humanoid ARMAR-III with 33 DOF as shown in Fig. 1 performing reaching tasks in two types of scenes. All results for Constellation were generating using $k = 10$ initial random configurations on an Intel i7 870 2.93 GHz CPU with 3.5 GB RAM.

A. Test Setup

We try the *direct projection* approach with each constraint as the primary target while attempting to satisfy the other constraints in the null-space. To be fair in comparing direct projection to Constellation, we try this method using each constraint as the primary target for all k random configurations that are used to initialize Constellation. The approach is considered to be successful if any of these attempts results in a configuration that meets all constraints.

The *cyclic projection* method iteratively projects a starting configuration to a repeating sequence of constraints. This projection does not use the null-space. The constraint order is randomly determined at the beginning of a run and then kept unchanged until the run is finished. The null-space variant of cyclic projection (*cyclic-ns*) differs from the cyclic method in that it uses the projection method of Alg. 1, which uses the null-space.

Table I lists and explains the constraints that are used in the test scenarios. The tolerance for meeting a constraint ϵ was set to 0.001 for all examples. The step sizes required by the projection algorithm (Alg. 1) were set as follows: $\delta_{prim} = 0.015$ for the collision constraint and 0.2 for other constraints. $\delta_{sec} = 0.015$.



(a) (b) (c) (d) (e) (f)

Fig. 6. Example configurations found by Constellation for each type of test scene. (a)-(d) Reaching tasks for shelves 1-4, respectively. (e) Randomly-generated scene with 7cm spheres (f) Randomly-generated scene with 2cm spheres.

Name	Definition
Balance	The robot must be in balance, i.e. its center of gravity must be above the support polygon.
Collision	The scene contains obstacles that must not intersect the robot.
Grasp	A cylinder in the scene has to be grasped with the right hand. The TSR for this constraint allows the grasp to rotate freely around the z-axis of the cylinder.
Stand	The robot kinematics are rooted at the right foot. The TSR for this constraint requires the left foot to be placed near the right foot with the same orientation.

TABLE I

CONSTRAINTS USED IN THE TEST SCENARIOS

B. Reaching into a Refrigerator

First, we compare the performance of the algorithms on a scene where ARMAR must reach into a refrigerator to retrieve an object placed randomly on one of four shelves. Different shelves of the refrigerator induce problems of varying difficulty. For instance, reaching to retrieve an object on the bottom shelf requires the robot to crouch and extend its arm, which causes opposing gradients for the balance and grasp constraints (see Fig. 6d). However, reaching for an object on the top shelf is fairly straightforward because there is a great deal of space for the arm and the balance constraint can be easily satisfied (see Fig. 6a).

We ran all four algorithms with 100 random seeds for each shelf. We used a timeout of 1200s for all methods. All algorithms run until they fail, succeed, stall, or reach this timeout value. To initialize each algorithm we use the start configuration depicted in Fig. 5 and add a small random offset in C-space with a length of 0.1rad.

The percent success and average runtimes are shown in Table II. These results confirm that there are many cases (shelves 3 and 4) where direct projection rarely succeeds or does not succeed at all. Cyclic and cylic-ns perform better in terms of the success rate but Constellation has the best success rate for all shelves. Note that Constellation's runtime for shelves 3 and 4 was quite high compared to the others. This is because some of the instances of these problems were very difficult and, while the other methods failed, Constellation simply took longer to find a solution.

Shelf	Direct	Cyclic	Cyclic-ns	Constellation
1	84% (11.8)	99% (19.3)	100% (8.61)	100% (34.8)
2	69% (6.91)	67% (14.6)	93% (10.6)	100% (33.2)
3	8% (13.0)	57% (8.67)	89% (11.1)	96% (239)
4	0% (-)	19% (6.21)	89% (45.6)	93% (476)

TABLE II

TEST RESULTS FOR THE REFRIGERATOR EXAMPLE: PERCENT SUCCESS AND RUNTIME IN PARENTHESES (IN SECONDS). ONLY RUNTIMES OF SUCCESSFUL RUNS ARE USED IN COMPUTING THE AVERAGE.

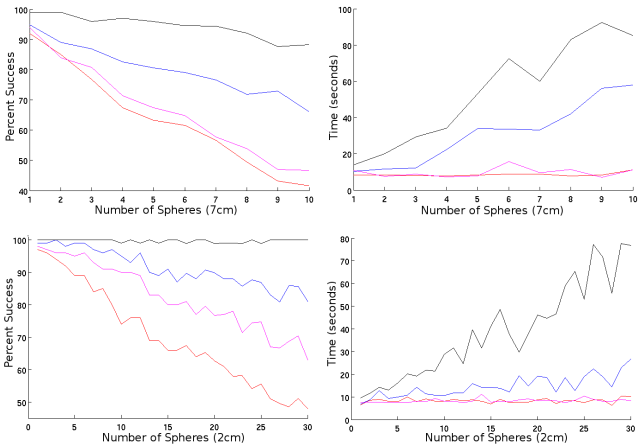


Fig. 7. Percent success and average runtime for solvable scenes of the sphere tests. Runtime averages were computed over successful runs of each algorithm. Black: Constellation, Blue: Cyclic-ns, Magenta: Cyclic, Red: Direct projection.

C. Reaching in Randomly-Generated Scenes

We also compare the performance of the algorithms on randomly-generated scenes of varying complexity. In these scenes the robot's task is to reach for an object while avoiding spherical obstacles placed around its body (see Fig. 6). We tested each algorithm over scenes containing varying numbers of obstacles. Our goal with these experiments is to see how the algorithms scale with the complexity of the collision constraints.

Each level of complexity, determined by the number of obstacle spheres in the scene, is tested with 100 random seeds. One run, i.e. a certain level of complexity together with a certain random seed, is then performed for each of the four algorithms. Initial configurations are computed using

the same method as the previous example. We used a timeout of 600s for all methods.

To test the algorithms' performance with different kinds of obstacle distributions, we evaluated two types of randomly-generated scenes: one with spheres of radius 7cm and the other with radius 2cm. The spherical obstacles were placed at random positions between the robot and the object to grasp. Obstacles were not allowed to intersect with a certain area around the robot and the object to avoid unsolvable scenes as much as possible. Nevertheless unsolvable scenes are possible and likely. We consider a scene to be solvable if any of the tested methods succeeded.

Fig. 7 shows the results of the tests described above. Since a scene which cannot be solved by any of the considered methods gives no information on which method is superior, the percentages given in the Fig. 7 are relative to the number of solvable scenes, not to the total number of tested scenes.

The results show that Constellation consistently outperforms the other approaches in terms of percent success. Making use of the null-space in cyclic projection considerably improves the results of the cyclic approach. However, though cyclic-ns runs faster than Constellation, Constellation achieves a higher success rate in all tested scenes. For the highest number of 7cm spherical obstacles, the percentage of scenes that are solved by Constellation, but not by cyclic-ns reaches 17%. Cyclic projection without null-space and direct projection solve about half of the scenes Constellation solves at the highest level of complexity for the 7cm test.

In the 2cm test, Constellation achieves between 97% and 100% success in all scenes, while the success rate of cyclic-ns decreases significantly as the complexity of the scenes increases. Cyclic-ns only achieves 81% success in the most difficult scene for the 2cm test.

VI. Conclusion

The structure of humanoid robots and the tasks they are expected to perform can severely restrict the feasible robot configurations to a small or even lower-dimensional set. We approach the problem of generating configurations in this set by searching for a point in the intersection of all constraint manifolds in C-space. Starting with an initial guess, our algorithm, *Constellation*, builds a graph in the C-space whose nodes are configurations that satisfy one or more constraints and whose cycles determine where the algorithm explores next. Since Constellation uses direct projection to generate nodes, all problems that can be solved by a single direct projection from the initial guess will be solved by Constellation. We evaluated four approaches, including Constellation, on reaching tasks for a 33 DOF humanoid. We find that Constellation performs the best in terms of the number of solved queries across a wide range of problem difficulty. However, this success comes at higher computational cost.

VII. Acknowledgements

This work was partially conducted within the German Humanoid Research project SFB588 funded by the German Research Foundation (DFG: Deutsche Forschungsgemeinschaft) and the International Center for Advanced Communication Technologies (interACT).

References

[1] D. Berenson, S. Srinivasa, and J. Kuffner, "Task space regions: A framework for pose-constrained manipulation planning," *International Journal of Robotics Research (IJRR)*, vol. 30, no. 12, pp. 1435–1460, October 2011.

[2] S. Sentis and O. Khatib, "Synthesis of whole-body behaviors through hierarchical control of behavioral primitives," *International Journal of Humanoid Robotics*, vol. 2, pp. 505–518, December 2005.

[3] T. Asfour, P. Azad, N. Vahrenkamp, K. Regenstein, A. Bierbaum, K. Welke, J. Schröder, and R. Dillmann, "Toward Humanoid Manipulation in Human-Centred Environments," *Robotics and Autonomous Systems*, vol. 56, pp. 54–65, January 2008.

[4] N. Tsakarakis, G. Metta, G. Sandini, D. Vernon, R. Beira, F. Becchi, L. Righetti, J. Santos-Victor, A. Ijspeert, M. Carrozza, and D. Caldwell, "iCub - The Design and Realization of an Open Humanoid Platform for Cognitive and Neuroscience Research," *Journal of Advanced Robotics*, vol. 21, no. 10, pp. 1151–1175, 2007.

[5] J. A. Nelder and R. Mead, "A simplex method for function minimization," *Computer Journal 7*, pp. 308–313, 1965.

[6] P. Combettes, "The foundations of set theoretic estimation," *Proceedings of the IEEE*, vol. 81, no. 2, pp. 182–208, Feb. 1993.

[7] L. Sciavicco and B. Siciliano, *Modeling and Control of Robot Manipulators*, 2nd ed. Springer, 2000, pp. 96–100.

[8] T. Sugihara and Y. Nakamura, "Whole-body cooperative balancing of humanoid robot using COG jacobian," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2002.

[9] O. Khatib, "A unified approach for motion and force control of robot manipulators: The operational space formulation," *IEEE Transactions on Robotics and Automation*, vol. 3, no. 1, pp. 43–53, 1987.

[10] J. Kuffner, S. Kagami, K. Nishiwaki, M. Inaba, and H. Inoue, "Dynamically-stable motion planning for humanoid robots," *Autonomous Robots*, vol. 12, no. 1, pp. 105–118, 2002.

[11] X. Tang, S. Thomas, and N. M. Amato, "Planning with Reachable Distances : Fast Enforcement of Closure Constraints," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2007.

[12] J. Cortes and T. Simeon, "Sampling-based motion planning under kinematic loop-closure constraints," in *Proc. Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2004.

[13] L. Han, L. Rudolph, S. Dorsey-Gordon, D. Glotzer, D. Menard, J. Moran, and J. R. Wilson, "Bending and Kissing : Computing Self-Contact Configurations of Planar Loops with Revolute Joints," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2009.

[14] X. Tang, S. Thomas, P. Coleman, and N. M. Amato, "Reachable Distance Space: Efficient Sampling-Based Planning for Spatially Constrained Systems," *The International Journal of Robotics Research*, vol. 29, no. 7, pp. 916–934, Jan. 2010.

[15] K. Yamane, J. Kuffner, and J. Hodgins, "Synthesizing animations of human manipulation tasks," in *SIGGRAPH*, 2004.

[16] M. Stilman, "Task constrained motion planning in robot joint space," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2007.

[17] J. H. Yakey, S. M. LaValle, and L. E. Kavraki, "Randomized path planning for linkages with closed kinematic chains," *IEEE Transactions on Robotics and Automation*, vol. 17, no. 6, pp. 951–958, 2001.

[18] Z. Yao and K. Gupta, "Path planning with general end-effector constraints: using task space to guide configuration space search," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2005.

[19] A. Escande and A. Kheddar, "Contact planning for acyclic motion with tasks constraints," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2009.

[20] A. Neumaier, "Complete Search in Continuous Global Optimization and Constraint Satisfaction," *Acta Numerica*, pp. 1–94, 2004.

[21] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "CHOMP: Gradient optimization techniques for efficient motion planning," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, May 2009.