Optimizing Actuator Network Positions

Robert L. Read *

Founder, Public Invention, an educational non-profit.

May 6, 2017

1 Introduction

The Gluss Project builds robots that are networks of linear actuators connected with concentric joints. A fundamental problem of contronlling such robots is to move one of the joints, or nodes, to a given position. More generally, we would like to leave some nodes (the feet) in position which we move another set of nodes (other feet, or graspers) to some positions that we specify. In general, this may require a motion of every actuator in the network.

Design of efficient gaits and motion depends on this abiity. The ability to crawl over obstacles depends on this ability.

Although the current robot is a tetrahelix, a structure isomorphic to the Boerdijk–Coxeter helix, we would eventually like to build robots and machines with general geometries, including non-tetrahedral geometries, and will develop the algorithm with that in mind.

This problem is dependent on the physical nature of the actuator: it can change length between and minimum and maximum length. This problem is silimar to optimization problems such as solved by linear programming. However, at a minimum the constraints are quadratic.

The mininum length and maximum length are defined by the Cartesian distance formula, which expressed as a length contains a square root, but we may always work with the square of this formula, leaving quadratic formualae. I'm pretty sure is is a subcase of Conic Quadratic optimization: http://docs.mosek.com/modeling-cookbook/cqo.html because we are dealing with Euclidean norms.

Note we are dealing with positive definite matrices.

*read.robert@gmail.com

2 Formulation

We now define the problem ACTNETOPT.

The input to our problem can be formalized as:

- An input dimension d (probably 2 or 3.) Nodes positions are elemenst of $\mathbb{R}^d$
- A model of a net of n nodes in the form of a graph.
- A mapping from from node index i to minimum length $Y(i)$ and maximum length $Z(i)$.
- A set of goal points g_j associated with node j chosen from $\mathbb{R}^d$.
- A linear weighting $w(j)$ of goal points interpreted as the cost of not placing the node j at the goal point g_j proportional to the square of that distance.
- A set of static nodes S which may not be moved by the algorithm.

We now attempt to formulate the problem as a QCQP. The quadratically constrained quadratic programming problem can be formulated:

$$\begin{aligned} \text{minimize: } & \frac{1}{2}x^T P_0 x + q_0^T x \\ \text{subject to: } & \frac{1}{2}x^T P_i x + q_i^T x + r_i \leq 0 \end{aligned}$$

In ACTNETOPT case we have no equality constraints.

In order to create this in matrix form, we create from it:

- A model of our robot represented by a symmetric matrix $M^{n\times n}$, where n is the number of joints in the robot, and a $M_{i,j} = 1$ if the nodes i and j are connected by an acutator, and is zero if not. Elements of M are real-valued.
- Similar matrices $Y^{n\times n}$ and $Z^{n\times n}$ representing respectively the minimum and quadrance (square of the distance) for each actuator i, j. If actuator i, j does not exist in the robot, then $Y_{i,j}$ and $Z_{i,j}$ are undefined. (Note that $Y_{i,j} = 0$ is an interesting case. Furthermore we are particularly interested in the case then all Y and M values are equal where they are defined.) Elements of Y and Z are real-valued.
- A set of goal points g_j associated with node j chosen from $\mathbb{R}^d$.
- A goal matrix $P_0^{n\times n}$ which is positive definite and represents a potentially weighted sum of the square of the Euclidean norm, or quadrance, of the position of nodes in our x_i from their respective goal postions g_i.

The output of the algorithm is a vertical vector X which satisifies all contraints and minimizes the objective function f.

To the author it was not obvious how to construct the matrices in question, so it is perhaps worth stating this explicity. Let us consider a single upper bound constraint, $z_{i,j}$. z_i is a scale, x_i is a 2- or 3-vector.

$$|\mathbf{x_i} - \mathbf{x_j}| \leq z_{i,j}$$
$$(x_{i_x} - x_{j_x})^2 + (x_{i_y} - x_{j_y})^2 + (x_{i_z} - x_{j_z})^2 \leq z_{i,j}$$
$$x_{i_x}^2 + -2x_{i_x}x_{j_x} + x_{j_x}^2 + x_{i_y}^2 + -2x_{i_y}x_{j_y} + x_{j_y}^2 + x_{i_z}^2 + -2x_{i_z}x_{j_z} + x_{j_z}^2 \leq z_{i,j}$$

Which can be represented in the matrix P_i, assume $i = 0$, $j = 2$, the 3-vectors are unpacked linearly.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & -2 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

This matrix has 9 entries representing the nine coefficients. In fact it is positive definite (proof?).

So: $x^T P_i x + r_i \leq 0$ represents a distance constraint when r_i is the square of the mininum distances for actuator i.

The goal matrix P_0 can be constructed similarly, multiplying each set of 9 entries by a $w(j)$ to weight that node.

Since there is one P_i for each actuator and each such matrix has only nine entries, this entire approach is very sparse. In the fact the constraint matrices are so specialized that we can expect an algorithm specific to this problem to do well. Since most of the quadratic constraint programming packages are commercial, there is a strong incentive to develop a specific algorithm.

3 First Algorithm

Although this problem is a quadratic constraint problem, is is highly specialized, and such solvers are not freely available. Because our robots are at present models (the current robot has 24 actuators), it is reasonable to assume this nut can be cracked with a small hammer.

Basic ideas for an algorithm:

- Keep a set of goals, which are nodes to move. Attempt to compute a "goodness" for each goal, which will basically be the improvement of the weighted score if the goal is reached. If the expected change is equal (which can happen?) then order by shortest move.
- Keep a set of goals and search Breadth-first. For a given goal, move the shortest distance of all limitations applied to it. Amoungst goals, move the distance that increases the goodness the most.
- If in processing a goal new new goals are added possibly take only the highest "goodness" goal in order to keep the set of goals as small as possible.
- Keep moving so long as each move improves the score by an input ϵ.

So we can try imagine the data structures used by the algotihm:

A priority Queue ordered first by Breadth from the primary goals and secondly by goodness. A goal is a node, a position, and an estimated change to the weighted score if the goal is acheived. A goal additionally has back pointers to the nodes that may improve if the node of this goal is improved.

The basic operation is: Take the goal from the top of the priority queue. Try to move the node to the goal.

If the node moves but does not reach its goal, backtrack to the nodes that are expected to improve. Move each node in the goal queue as possible as you back track.

If the node cannot be moved, find the connected nodes which are not in the fixed set. Compute a goal point for each of these nodes. Compute the expected change to the weighting if these points are are acheived. If that expected goodness is greater than epsion, add the goal to the priority queue.

What are the termination properties of this algorithm? Invariant: Every move improves the goodness.

Conjecture: The estimate of the improvement of goodness is guaranteed to improve the goodness that much.

Even if it has to visit all N nodes, the running time of O((inital score / epsilon) * N)

4 Strainfront Algorithm

The idea here is to perturb the target node, creating a *strainfront*. A strainfront is a set of directed edges. The strainfront tries to relieve strain by moving the head node of the edge. Each such move may cause new edges to be added to the strainfront. When a strainfront reaches a fixed nodes, it is reflected backwards.

This seems like a really beautiful idea to me.

Questions remain:

- in what order to we process the strainfront?
- how do we prove termination?
- can we prove that all strain is relieved by this algorithm? (yes, because it can leave things unchanged and it starts with a legal configuration.)

Here is my attempt to start the algorithm

Input: A node to move a, and target point to move it to, $\vec{v}$. A connectivity graph G, and a set of fixed nodes F, and a legal configuration C satisfying all constraints.

Output: A new legal configuration with a as close to $\vec{v}$ as possible.

Begin Algorithm STRAINFRONT:

Let S = perturb$(a, \vec{v})$.

While $S \neq \emptyset$ do:

$S = Relieve(S)$

End Algorithm

Subroutine PERTURB$(a, \vec{v})$: Input a, a node, and $\vec{v}$, a desired position for a. Create empty strainfront S.

If a is not marked "PERTURBED", Move a to G. Mark a "PERTURBED" and annonate it with the original position.

For each node b_i connected to a in G, compute $s = strain(a, b_i)$. Strain is negative if they are too far apart, positive if they are too close together. If $s \neq 0$, add $a \rightarrow b_i$ to the strainfront S. Return S.

Subroutine RELIEVE(S,C) (where S is a strainfront):

Outpout: a new strainfront and configuraiton

Choose and remove a directed edge $x \rightarrow y$ from the strainfront in order of number of edges from our start node a.

If y is a fixed node or marked "FIXED" node, add $y \rightarrow x$ to the strainfront and return.

Mark the node y "FIXED".

Let z_s = LEAST-STRAIN(y, S, C).

Return $Perturb(S, x, z_s)$.

Subroutine LEAST-STRAIN(x, y, S, C):

There are a variety of possible implemetations for this. We can evolve it rather simply I think.

If y is "PERTURBED", then we will only seek solutions that occur on a straight line between the current y position and the original y position.

First algorithm: for the N neighbors of y, compute the at most $4N^2$ intersection points of the circles corresponding to the lower bound and upper bound of each circle. Test each such intersection point for being part of the "free region" (that is, satisfying all constraints.) Return the free region intersection point that minimizes the strain on the $x \rightarrow y$ joint.

If y is PERTURBED, insure that all neighbor constraints are satisfied, by restoring to the original position if necessary.

Sketch of termination: Each node is visited at most twice.

Sketch of correctness: The algorithm proceeds as two wave fronts: a wave front generated by the first perturbation, and a wavefront reflected off the fixed nodes going in the opposite direction. The internal configurations temporarily violate the constraints. However, once the back wave has passed a node, it has all constraints obeyed, if nessary by restoration to its original position.

5 Notes

This algorithm is almost certainly not globally optimal. I have not even proved correctness nor termination.

However, it seems pretty clearly that one operation of the algorithm is extraordinarily fast.

If we kept a data structure that kept intersection points, it is possible that we would have an $O(n)$ algorithm, where n is the number of nodes, but we may very well have less than that if the net has slack to acheive the position in question.

It seems that this may be a very valuable adjunct algorithm to the algorithms that use a general purpose objective function, which could also be used.

An open question remains how the algorithm will function with repeated uses. For example, will it fine tune a solution? Possible not, due to the way the intersection points are used.

Note that the algorithm can be improved by lambda-lifting selection critera into the basic strain-front algorithm. This could allow better control of the works; for example, on a secon iteration, an algorithm that departed from the basic intersection point system could be valuable.

The current algorithm may be adding too much to the strain front.

Major TODOs at this point are:

- clean up the code!
- work out termination proof and asymptotic complexity.
- count major operation.
- shift to squares instead of square roots whenever possible.
- lambda lift
- MAKE WORK WITH 3D vectors!
- develop more awesome test cases.
- develop a highly over constrained network.
- develop interactive work.

6 References