Optimizing Actuator Network Positions

Robert L. Read *

Founder, Public Invention, an educational non-profit.

September 19, 2017

1 Introduction

The Tetrobot Project builds robots that are networks of linear actuators connected with concentric joints. A fundamental problem of controlling 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 $G = (N, E)$.
- A mapping from from node index i to the square of the minimum length $Y(i)$ and maximum length $Z(i)$ It is convenient to treat these as the square of the length, or quadrance.
- A set of goal points H h_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:

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

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 Formulation as Inverse Problem

In some ways it may be helpful to formulate our problem as an inverse modelling problem.

Let a "model" be a Cartesian position of nodes. Let "observed data" be the lengthe between connected nodes as defined in the graph. Then the "forward problem" is to compute lengths from node positions, which is computationally easy. The "inverse problem" is

to compute a model that is close to a set of lengths. In otherwords, given a set of lengths d compute a model m that satisfies:

$$d = G(m)$$

where G is an operator that just computes the Euclidean distance between connected nodes.

It would be nice to be able to instantly solve the forward and the inverse problem.

However, our deeper problem (as formulated as a QCQP problem above) is to find a model that minimizes a somewhat arbitrary penalty function. This function is conveniently thought of as operating both on positions (e.g., we want a node to move as far to the North as possible) and on lengths (e.g., we want all constraints met and for actuators to be close to the same lengths.)

Because generally speaking every set of lengths produces a positional model, we conjecture that it will be easier to search the space of lengths than to search the space of positions.

In fact, we can imagine a brute-force technique: compute the penalty for every actuator position at 10% length increments, and find the one with the lowest penalty. Such a method requires 11^n evaluations, where n is the number of actuators.

4 Why we may use Powell's Method

This is fundatmentally a constraint optimization problem. However, we can model it as a continuous optimization problem with heavy penalties for violating the constraints.

Given that we have a relatively small number of nodes, it is possible that a global optimization system, such as Powell's Method, might work just fine for what we are trying to acheive here.

I do not anticipate the space to have many local minima. Although using Powell's method is probably wasteful from a scientific point of view, coding it has a number of advantages:

- This will be a contribution to open source systems in general. (This is no longer true, as fmin appares to implement it!)
- It can be used as a subroutine in the Strainfront algorithm below.
- It may simply solve our problem without much additional effort.

Powell's method has been implemented many times in C and C++. However, I am contemplating implemeting it in JavaScript. This could be considered a forward-thinking action in the vanguard, or it could be considered a limitation. Since Public Invention leans very heavily on browser-based functionality to be as accessbile to as many people as possible, a JavaScript implemention is potentially useful because it can be run in the browser.

Certainly at present my immediate goals are to be able to work on the Ammo simulation which is all browser-based. From our emacs-lisp control structure, we can call a C function or a Node function with inter-process communication equally easily.

If I can't do this in JavaScript, it will be inaccessible to all of my other browser-based code.

5 Open source project: fmin

There is an open source javascript project called fmin that looks really awesome and appears to be what we want. The only limitation is that it requires the computation of the gradient at each point. I wanted to avoid this, but in fact it is quite calculatable for this situation, so I think the best thing is for me to actually work out the gradient and use these methods, which is undoubtably a better approach than relying purely on a non-gradient approach.

So here begins my attemtp to define the objective function in a differentiable way.

$$f(\mathbf{x}) = r(\mathbf{x}) + p(\mathbf{x}); \tag{1}$$

$$r(\mathbf{x}) = \sum_{i \in H} w_i d(\mathbf{x}_i, Targ_i)^2 \tag{2}$$

$$r(\mathbf{x}) = \sum_{i \in H} w_i \big((x_{ix} - h_{ix})^2 + (x_{iy} - h_{iy})^2 + (x_{iz} - h_{iz})^2 \big) \tag{3}$$

where $\mathbf{x}$ represents the vector of 3-dimensional node positions, $p(\mathbf{x})$ represents a penalty function defined by:

$$p(\mathbf{x}) = \sum_{e_{i,j} \in E} max(0, (y(i,j) - \|\mathbf{x}_i - \mathbf{x}_j\|^2)) + max(0, (\|\mathbf{x}_i - \mathbf{x}_j\|^2 - z(i,j))) \tag{4}$$

$$p(\mathbf{x}) = \sum_{e_{i,j} \in E} max\Big(0, \Big(y(i,j) - (\sum_{d \in D} (X_{id} - X_{jd})^2)\Big)\Big) + max\Big(0, \Big(\sum_{d \in D} (X_{id} - X_{jd})^2) - z(i,j)\Big)\Big) \tag{5}$$

To use the fmin software, we need to compute the derivative of f. In parcticular it requires the computation of the partial derivative for each variable x_i.

$$f'(\mathbf{x}) = r'(\mathbf{x}) + p'(\mathbf{x}) \tag{6}$$

$$\frac{\partial r}{\partial x_i} = 2w_i(x_i - h_i) \tag{7}$$

$$\frac{\partial p}{\partial x_i} = \sum_{j:e_(i,j)\in E} -2(\sum_{d\in D}(x_{id}-x_{jd}))\big[\|\mathbf{x}_i-\mathbf{x}_j\| < y(i,j)\big] + 2(\sum_{d\in D}(x_{id}-x_{jd}))\big[\|\mathbf{x}_i-\mathbf{x}_j\| > z(i,j)\big] \quad (8)$$

$$\frac{\partial p}{\partial x_i} = \sum_{j:e_(i,j)\in E} \big(\sum_{d\in D}(x_{id}-x_{jd})\big)\big(-2\big[\|\mathbf{x}_i-\mathbf{x}_j\| < y(i,j)\big] + 2\big[\|\mathbf{x}_i-\mathbf{x}_j\| > z(i,j)\big]\big) \quad (9)$$

where the square brackets represent Iverson notation, which evalutes to 1 if the condition inside the brackets is true and 0 if false.

It seems likely that we can compute the derivatives of these functions in code relatively easily. My plan is to complete that and then try to work this into the existing actoptmin framework.

We must decide whether to treat the x,y,z coordinates as a vector or as independent variables in this system. I believe mathematically, it is 100% possible to treat them as independent variables.

The functions as I have defined them here appear to violate the "Wolfe Condition". Fmin doesn't seem to work. I switched to using "optimiize.js" and have been able to obtain solutions. However, it is a little tricky to develop weighting funcitons that correctly penalize the bounds being broken.

After switching to optimize.js, this is working somewhat better. However my attempts to increase the penalty for violating a constraint seem to fail pretty badly, even if I just do a linear increase. I suspect this means that it would be best to use optimize.js locally inside the STRAINFRONT algorithm to find good solutions, which was my original intention. I would like to understand better why it fails. Possibly the problem is that the derivative makes jumps which are too drastic—that appears to be the case. I could possibly solve this problem by examining each iteration.

5.1 Evaluation

Well, I could not get optimize.js to work very well either. It is possible that I am not using it correctly. Possibly I should talk to Martin about this. However, it seems to fail completely to find a good global solution.

I vaguely remember when I first started the numerical approaches that I was going to use them to improve the "intersection only" algorithm in strainfront. I think that still may be possible.

I also must consider if I should try to integrate strainfront into the interactive animation part.

Now I see that indeed the intersection algorithm is potentially a problem.
I would like to put it into the interactive mode. Not sure I can.

6 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)

7 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 $a \to b_i$ 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 $x \to y$ 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 $y \to x$ 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 $x \to y$ 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.

8 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. (this is done-ish).

Having come back to this after a month of travel and other work, it seems to be working modestly well, worth continuing. However, a number of major points remain. I believe in developing test cases I found that I did not like this algorithm, although now I am not sure why.

- The current insistence on intersections creates sub-optimal micro-solutions. We need to use the off-the shelf solutions to solve that problem. We must reach all points on small problems; it is not acceptable not to. Tha can be considered a major point. This is something we should be able to go back to testing with an automated test!
- The current algorithm does not penalize crossings. It would be a major goal to work on that.
- It is worth exploring what happens if you run the algorith several times.
- It is worth exploring what happens if you try to "equalize" energy in the system somehow.
- Why did I think it was failing based on test cases? From animation it is sort of okay.
- What really would be a true success criterion for the operation?
- I thought that intersections were a problem – can I now use the other optimize function to do a better job with that?

9 A Key Idea

The problem should be recursively reducible. For example, A triangle can be reducde to a linear segment with an apex at one end and the midpoint of the other side at the other end. In principle, this allows a 3-fold reduction in the complexity of the problem. It also may lead to more symmetric solutions, even based on the intersection-point only algorithm. In small examples this seems to work.

10 August 23: Coming back to Inverse

Coming back to this on August 23rd, after unfortunate business took me away, I am now attempting to summarize the situation as I remember it:

- I completely mistrust the software exercised by test_inverse.js. I believe the test basically show the opt.minimize_Powell, nor any of the other software, does a good job finding a solution when we are not close to the correct solution.
- This is a particular problem because the problem posed appears (in my thought) to be smooth and simple. I would expect any gradient solution to correct solve it.
- Altough the physics engine works, I would rather have a non-physical solution for something so simple—or so it seems to me to be simple.
- If I can't solve this simplest of problems numerically, I have no hope nor right to move forward.

Note: I may have discovered a signficant isse. When the distance are great, my inversion routine fails. If I can't invert the distances correctly into coords, then all hope is loss for the further optimization. This is a major insight which may explain part of the problem with the instability of the off-the-shelf routines. I need to address this next.

Note: I have now written a way to test just that the invert works, but I haven't turned it into a test. INVERT fails whenever you start to far away, no matter which version I try to use. I need to turn this into a formal test, and figure out why the heck it fails.

6:33 I finally developed a funcation test of only the recovery of coordinates from distance. It partially works with Powell's method and seems to generally fail with the "BFGS" algorithm. I need to look to see if this is a bug in that algorithm, or possibly in my code (in the gradient?) If we can't use BFGS, we may have to code our own somehow.

There are two poosible repos that will have better stuff:

Go: https://github.com/gonum/gonum

C++: https://github.com/PatWie/CppNumericalSolvers

11 August 25: New Ideas

I installed the PatWie numerical solver code in CPP. It is not too easy for me to learn C++. Additionally, I have decided that I need to do the invesion in a anchor-first topologicla sort manner, which should make it an essentially determined problem, but it has been a good experiment to get it working.

I now have high hopes for this approach — don't quite know how I will get it to work with javascript. I suppose I could make a chrome extension.

12 August 26: New Ideas

I have high hopes for the C++ code, so far it is working well and seems far better documented and used.

This C++ code (DLib) does seem to work much better. I can solve the 2-dimensional problem rather well now.

13 Using Derivatives

It is conventional wisdom that if dervivatives are analytically computable, more efficient numerical optimization techniques are available.

In a two-dimensional ladder, the length x between B and C where A and B are nodes fixed by the geometry and Z is the goal node to be moved, s is the score, we can compute the $\frac{\partial s}{\partial x}$ in a multi-stage process. Considering the case that the $\vec{AB}$ is an external (rather than internal node).

- $\frac{\partial \theta}{\partial x}$ is (probably) $\frac{1}{\|\vec{AC}\| \sin \phi}$, where ϕ is $\angle ABC$. This derivation used the fact $\arcsin x \approx x$ when $x \ll 1.0$.
- $\frac{\partial Z}{\partial \theta}$ can then be computed by a pure rotation about A. Note that this quanitity as actually a vector.
- $\frac{\partial S}{\partial Z}$ (a scalar) can then be computed analytically from this, if the S is simply a distance measure
- All of these can be chained together to produce the derivative needed by optimization work.

In other words, given a confuration, we can compute (in constant time) the $\frac{\partial s}{\partial x}$ for each variables edges.

The case when the edge is an interior is slightly more complicated, because you have to take into account the change to the next triangle in the latter BCD to compute the change to Z. In both cases, however, you model the remaining parts of the ladder essentially as stiff triangles.

In order to test this, I will use my grhapical system to render the varies stages as vectors on screen before I attempt to use an optimization technique that utilizes the derivatives.

14 References