URA Report
Isaiah Witzke
2023

This report is initially a "README" to help you get up to speed quickly. Some interesting results and findings are found later on in the report.

1 The code

1.1 Introduction to the scisim C++ code and what's important for us

This "scisim" repository was created and used years ago by the researchers that wrote the "Rosi paper" (i.e. "Reflections on Simultaneous Impact" by Rosi et al.). Scisim is a pretty large codebase, and is capable of running some cool simulations, however in the current stage of our URA project *most* of the codebase is irrelevant. I.e., scisim can handle 3d simulations, friction effects, complex geometry, ..., but we're using Scisim just to simulate comparatively simple 2d ball collisions.

The most relevant C++ code for us is found in the `ImpactOperator` classes (see the `scisim/ConstrainedMaps/ImpactMaps/` folder). More specifically, we are interested in the `ImpactOperator::flow` method. `ImpactOperator::flow` is responsible for collision response: When Scisim detects that one or more balls will collide in the next time-step, `flow` is called to determine how the colliding balls' velocities will change as a result of the collision. More verbosely, `flow` is called with the colliding balls' positions, pre-collision velocities, masses, etc, and it must find the "normalized impulse coefficients" λ. λ in turn will yield the balls' correct post-collision velocities, since $\dot{q}^+ = \dot{q}^- + M^{-1}G_{\mathbb{A}}\lambda$ (more details on this in the Rosi paper).

The goal of our research is to find a `ImpactOperator::flow` implementation which finds λ fast!

1.2 Variable name discrepancies

It's worth noting that there are some variable name differences between the Rosi paper and the code – it caused some confusion when I was first going through it.

- $G_{\mathbb{A}}$ is referred to as `N` in the code

- $\dot{q}^{-}$ is called `v0`

- λ is sometimes called `alpha`, `x`, or `solution`.

1.3 Running scisim & scene files

Let's see some balls colliding!

`cd` into the `scisim/build/rigidbody2dqt4` directory and run `make` to compile the code. Then to start a simulation, run the compiled `rigidbody2d_qt4` executable and provide a path to an xml "scene" file:

```
./rigidbody2d_qt4 ../../ura_research/scenes/simple/10_balls_in_box.xml
```

There's a lot of different scene files that you can use in the `ura_research/scenes/` directory. There's also some left-over scenes under the `ura_research/src/archive/old_simulation_data/` directory, but these probably aren't very interesting nor useful anymore. Finally, the creators of Scisim left some interesting scenes in `assets/rigidbody2d/` that are fun to play around with (although you will need to change the `linear_solver` part to be only: `linear_solvers="ma27 ma86"`, otherwise you'll get some hsl-related errors)

These scene xml files define the objects (balls, walls, etc...) in the simulation and their initial properties (positions, velocities, masses, coefficients of restitutions, etc...). The scene files also define the "impact operator" that will be used to resolve collisions over the duration of the simulation. This is done in the `<solver>` tag of the scene file, which can usually be found near the top. In the example above, we used the `10_balls_in_box.xml` scene, which specifies `"ipopt"` as the solver. This means that `scisim/ConstrainedMaps/ImpactMaps/LCPOperatorIpopt.cpp`'s `flow` implementation is used to resolve collisions (see `RigidBody2dSceneParser` line 710 for *why*; this is relevant if you want to make your own ImpactOperator).

1.4 Impact operators of interest

There are a couple different ImpactOperator implementations that we're interested in:

- `LCPOperatorIpopt.cpp` - an implementation that comes with scisim. Uses the IPOPT algorithm to find λ. Ipopt is widely used for "large scale nonlinear optimization of continuous systems" (wikipedia) and does a good job in our case - it's a very robust (i.e. converges on a correct solution every time), but not very fast. The reason that it's not super fast is because it's a general purpose solver, not tuned for our specific collision problem. Hence our goal of finding a faster solver!

- `LCPOperatorPenalty.cpp` - an implementation that a previous student, Wen Zhang wrote. Uses a penalty method to find λ. See Zhang's URA writeup for more details on it's performance. I believe this implementation is faster than Ipopt, but it's not as robust. Over the past 2 research terms we haven't returned to this method, although there's potentially some progress to be made here. Right now we don't fully understand why the current penalty method fails sometimes.

- *`LCPOperatorPI.cpp`* - an implementation that a previous student, Kevin Wan wrote. Uses policy iteration method. Not very robust in it's current state, but very fast when it does work. **This is the implementation that current efforts are focused**; it shows the most speed gain potential, we just need to make it more reliable.

- `LCPOperator3.cpp` - An amalgamation of the previous 3 implementations - it runs them all side-by-side and reports the differences in solutions after all have completed. Used for analysis/debugging purposes.

- `LCPOperatorPIv2.cpp` - an attempted improvement on LCPOperatorPI. More on this below.

- `LCPOperatorIsaiahDebug.cpp` - Similar to the LCPOperator3 in that it runs 3 operators side-by-side for comparison. This operator differs from LCPOperator3 right now as it runs LCPOperatorPIv2 instead of LCPOperatorPenalty (it still runs the LCPOperatorIpopt and LCPOperatorPI though). I've been using this operator as a bit of a sandbox, and regularly update it to get different data out of the C++ simulation. It prints out debugging info and collision data when the balls collide (i.e. Q, N, v0, ...) in json format to stdout. This allows us to ingest the simulation's state at the time of the collision into python and run

our own analysis on it. This is the implementation that we're primary using to test ideas, so this operator is used in most of the scene .xml files.

Note: this implementation will exit the program after the first call to `LCPOperatorIsaiahDebug::flow`! This is intentional since I've only ever wanted to do analysis on one collision at a time.

1.5 Python workflow

Here I'll go over some of the python source that's been built up over the past term. Much of it isn't super clean (sorry about that), but most of it won't relevant going forward anyways (the important bits should be well-documented enough ⌣). For instance, many of the notebooks like `ura_research/src/archive/<date>.ipynb` were just made to test out my "idea of the week", and then never used again! I just found it really helpful to have a python notebook workflow where I could import simulation data from Scisim/C++ and quickly test out ideas in python. Playing around with algorithms and preforming analysis tends to be easier when using Python notebooks instead of C++ (you'll find it takes a long time for the C++ code to compile). This workflow might not hold true for you though, so feel free to move away from using Python going forward.

With that being said, let's take a look at some of the Python tools you have at your disposal.

1.5.1 Loading simulation data into a Python notebook

Let's run a simulation with the `LCPOperatorIsaiahDebug` operator and pipe the output to a file:

```
./rigidbody2d_qt4 ../../ura_research/scenes/simple/2_balls.xml > \
    ../../ura_research/simulaiton_outputs/2_balls_output.json
```

This will run a simple simulation of 2 balls running into each other. You can play the simulation until the first collision occurs, at which point the collision data (i.e. N, Q, v0, ...) will be piped into the `2_balls_output.json` file and the program will terminate.

We can then import the simulation data into python and analyze it. See `ura_research/src/analyze_data.ipynb` as an example jupyter notebook that does just this. In this notebook we use the `read_file_to_pd_dataframe`

function from `util/data_import.py` to import the simulation data into a pandas dataframe. The matrices and vectors that are passed to `LCPOperatorIsaiahDebug::flow` (i.e. N, Q, v0, ...) are parsed into numpy arrays. Ipopt's solution for λ is also made available as numpy array too so that we know what a correct solution (λ) is (that is, assuming that IPOPT converged).

1.5.2 Custom impact operators in python

This means that you can now make your own implementations for `ImpactOperator::flow` in python and compare results against Ipopt's solution. This is exactly what is being done in `ura_research/src/solvers/....` An example use of how these solvers were then used can be found in `ura_research/src/archive/apr_11.ipynb`. In cell 2 we import simulation data (now under the `ura_research/old_simulation_outputs/` directory), and in subsequent cells we run the `PolicyIterationXXX`s and run their solve method. Then, since we're in python notebooks, nice graphs and analysis can be made (see the end of the notebook)

To make your own custom python impact operators, it should just be a simple matter of copying the code that's already there then altering it as you wish. This code *is* actually pretty well documented and should be easy enough to digest.

1.6 A note on the X_by_X_grid scenes

Under the `ura_research/scenes` folder you will find a couple folders named something like `5_by_5_grid`. These contain scenes of balls in a 5 by 5 (or 7 by 7, etc...) grid arrangement, each generated by `src/util/generate_balls.py`. What makes one scene different from the next is that each ball's position is slightly randomized so that they're on a sort of perturbed grid. The space between the balls are much smaller than the balls' radii, so that if you run `./rigidbody2d_qt4` with any of these scenes as an input, you'll just see a big clump of balls all overlapping each other. If you then press play, the first collision immediately occurs, lots of debug information is printed to `stdout`, and the program quits (this behavior is described in the `LCPOperatorIsaiahDebug.cpp` section of 1.4).

The reason for all these strange scene setups is because the policy iteration algorithm (and penalty method, and others...) tends to fail when the balls start to overlap each other (see Kevin Wan and Wen Zhang's URA reports for more on this). We've come to the conclusion that it would be really hard

to prevent the ball overlapping "problem", and in fact we're not even sure that the overlapping balls phenomena is *the* problem causing policy iteration to fail (in my opinion, policy iteration starts failing when collisions become sufficiently complicated, and it's easy for collisions to become complicated in ball collision scenarios when they start to overlap each other and collide not only with their neighbor balls, but their neighbors' neighbor balls). The reason for the scenes of the 5x5 overlapping balls is to small scenes that are fast to run/easily repeatable (vs the 1000 + ball scenes used in previous terms) that are also easy to analyze (matrices resulting from a scene containing 25 balls are much more digestible than matrices resulting from 1000 + ball collisions). In many of these 5x5 scenes policy iteration will fail on the first collision, so it's really easy to export the data into python, replicate the policy iteration failing in a python notebook, step through and/or preform your own analysis.

There is even a util script left (`util/runner.py`) that will run the `rigidbody2d_cli` executable against a folder of scenes (i.e. the 50 5x5 scenes) and bulk output the data into another folder (see `old_simulation_outputs/5_by_5_grid` for example). These output folders can then be bulk imported for further analysis (see `src/archive/apr_12.ipynb` for example).

2   Some Math Background/Recap

2.1   The Linear Complementarity Problem (LCP)

The primary problem for this project (when stated as an LCP problem instead of a QP problem as posed the Rosi paper), is to find a solution to the following LCP:

find $\lambda \in \mathbb{R}^n$ such that...
$Q\lambda + b \geq 0$, $\lambda \geq 0$, $\lambda^T(Q\lambda + b) = 0$ ($\geq$s are element-wise)
With $\mathbf{Q} := G_{\mathbb{A}}^T M^{-1} G_{\mathbb{A}}$, $\mathbf{b} := G_{\mathbb{A}}^T \dot{q}^-(1 + c_r)$

What this is saying is that for a system of equations $Q\lambda + b = 0$ we want to find an "almost solution". The solution λ will be a vector such that the ith row of the linear system will equate to 0 (i.e. $(Q\lambda + b)_i = 0$) while the ith λ element is greater than 0 ($\lambda_i \geq 0$), **or** $\lambda_i = 0$ while the ith row of the linear system is greater than 0, $(Q\lambda + b)_i \geq 0$. It turns out this is a lot more

difficult than just solving a linear system of equations, since even if you have a λ such that $Q\lambda + b = 0$ for all rows, it's pretty likely that there's an element of λ which is less than 0, so you'll have to throw out your λ and try again.

2.2 What is a *Policy*?

Now we're in a good place to describe the term "policy" in "policy iteration" (Caveat: the following isn't a totally correct definition of what a "policy" is. I believe it's more general than whats presented here, but for this project I've found the following definition to be useful). The policy π is the set of indices $\pi = \{\pi_1, \pi_2, ..., \pi_k\}$ which determines what rows of the linear system $Q\lambda + b = 0$ will hold. For example, if our original system of equations had 4 rows and $\pi = \{1, 3, 4\}$ it would mean that the first, third, and fourth rows of the linear system will equate to 0 with $\lambda_1, \lambda_3, \lambda_4 \geq 0$, but the second row will just be greater than 0 with $\lambda_2 = 0$.

2.3 Policy Iteration (PI) Overview

Although there's some good papers and youtube videos available online explaining PI, it's usually in the context of reinforcement learning or markov decision processes, not as a solution to our LCP problem. [1] Therefore, we'll present a pseudo-code implementation of PI here similar to how you'll see it implemented in our codebase. [2] Hopefully this will help relate PI directly to our LCP problem at hand.

---

[1] The reinforcement learning/MDP context is an example where the definition of "policy" differs a bit from the definition given above. An interesting video I recommend watching (from UWaterloo btw!) is Pascal Poupart's lecture on policy iteration: https://www.youtube.com/watch?v=mjyrRG7RD84. In it he goes over an example of how the policy iteration algorithm finds an optimal "advertising vs saving strategy" (in this context, "strategy" = "policy") for a company that wants to become rich & famous! In my opinion videos like this make it easier to initially grasp the idea of policy iteration vs trying to go learn it directly in this weird and abstract ball-collision/LCP context. Aside: I believe Poupart's preceding lecture video is on value iteration which is useful as well. You'll actually see some value iteration implementations left around in this codebase too as value iteration and policy iteration usually go together hand in hand.

[2] There's many implementations in the source code of policy iteration; see `LCPOperatorPI.cpp` for a C++ implementation or `policy_iteration.py` in the archive directory for a python version.

```
# This is the "main" PI function. All it does is flip-flop between making a guess
# for the value (lambda) then updating the policy afterwards.
# It repeats until a good policy is found that yields the correct lambda
# (or until it runs out of iterations -- in which case we say that PI
# probably doesn't converge for the given Q & b, and return FAILURE)
def policy_iteration(
    Q,
    b,
    initial_policy_guess,
    initial_solution_guess,
    max_iters,
    tol
):
    # these choices really shouldn't matter too much
    policy = initial_policy
    lambda = initial_guess

    loop max_iters times:
        lambda = update_value(policy, Q, b)
        policy = update_policy(Q, lambda, b)
        error = evaluate_error(Q, lambda, b)
        if error < tol:
            return (SUCCESS, lambda)

    return FAILURE  # PI didn't converge to a correct solution

def update_value(policy, Q, b):
    # here we first update our linear system of equations so that when we
    # try to solve the system, the new solution (lambda) will be 0
    # for each row not contained within the policy array
    augmented_Q = copy of Q
    augmented_b = copy of b
    for i = 1...len(b):
        if i in policy:
            # we want this row of the linear system to strictly equate to 0,
            # so no need to change any rows of the linear system here
            continue
```

```
        else:
            # the changes made in this block mean than when the augmented system
            # is solved later, the resulting lambda[i] will be forced to 0.
            # Hopefully, the lambda we get from solving this augmented system
            # will result in the ith row of the original linear system being >= 0
            set the ith row of augmented_Q to 0s
            set augmented_Q[i, i] = 1
            set augmented_b[i] = 0

    use linear algebra library to find solution to solve:
    augmented_Q * new_lambda = -augmented_b
    # ^^ we call the method to find the new_lambda here the "inner solver"
    # Usually we use a "least squares" or "conjugate gradient" method,
    # but ideally it *shouldn't* matter (as long as Q isn't singular!)

    return new_lambda


def update_policy(Q, lambda, b):
    policy = []
    for i = 1...len(b):
        # This is really where the magic happens!
        if (Q * lambda + b)[i] < lambda[i]:
            policy.append(i)
    return policy



def evaluate_error(Q, lambda, b):
    return the L2 norm of min(Q * lambda + b, lambda)
    # ^ the min function is element-wise
    # When the L2 norm of: min(Q * lambda + b, lambda) = 0,
    # we know that the LCP conditions from section 2.1 have been met.
    # Question/exercise for reader: Why?
```

This is a surprisingly simple algorithm considering how well it preforms when it does converge. You can see *why* PI converges in the fairly digestible proof of theorem 2.1 in the paper "Some Convergence Results For Howard's Algorithm" by Olivier Bokanowski, Stefania Maroso and Hasnaa

9

Zidani (link). The issue with the proof, and why it doesn't hold in our scenario, is because it makes the assumption that Q is monotone. In our scenario, when the collisions become complex enough, the Q matrix gets pretty ugly (i.e. singular even – see sections 6 and 7!) which is one of the reasons PI won't converge.

3 Some ideas from last term

Ok, so then how might we try to update PI to get it to work? Here's a couple ideas from last term that showed a little potential.

3.1 Randomizing the policy when stuck in a loop

One thing that we noticed last term, is that when policy iteration doesn't converge, it usually does so because it gets stuck in a loop. For example, maybe PI starts at with an initial policy guess of $\pi^1 = \{1, 2\}$, then it does an iteration which updates it's policy to $\pi^2 = \{2, 3\}$. It's possible that after another iteration we get $\pi^3 = \{1, 2\} = \pi^1$. Because the inner solver is (usually) deterministic, we'll now be stuck in an endless loop! The next λ will just yield π^2 as the next policy, which will in turn cause `update_value` to return a λ that yields π^1, ... and the cycle continues.

To counter this, we can update the algorithm to detect when a loop has occurred, and when it does, randomize the policy to something new and then continue iterations. For an implementation of this "V2 algorithm", see `policy_iter_v2.py`.

This actually helped the algorithm converge significantly times than the basic PI algorithm. For some analysis and results, see `apr_5.ipynb`. This was pretty promising progress when this was first found, because it does show that PI might not be completely dead in the water.

It didn't totally fix PI though. There were still many cases with an ugly enough Q matrix, where PI V2 would just keep going through different policies an never converge. I suspect that we do eventually reach the correct policy, but because the Q matrix is singular the inner solver chooses an incorrect lambda from the infinitely large solution space.

3.1.1 Slightly perturb a "good policy" when we are stuck

`policy_iter_v3.py` was similar to v2, however instead of completely randomizing the policy when a loop was detected, we instead look for a previous policy that yielded a λ with a small error, and just perturbed that policy a little bit. In practice, this didn't actually work too well.

3.1.2 Make Q non singular

`policy_iter_v4.py` was also similar to v2, but this additionally attempted to alter/remove rows and columns of the Q matrix to make it non-singular. In practice this also didn't work too well.

3.2 Changing the inner solver when stuck

I personally haven't tried this yet, but maybe we could try changing the inner solver when we get "stuck". For example, this might mean going from using a least squares inner solver method to conjugate gradient. Or maybe, we can use a more sophisticated custom inner solver that tries to see if the solution space includes solutions where all lambda elements are ≥ 0 if the Q matrix is singular.

4 Physical Results

Below is the result of me trying to find any physical interpretation for what our mathematical objects represent (Q, λ, b). The most significant result I was able to find was that the entries in the Q matrix were equal to the cosines of the angles between each triple of balls which are simultaneously colliding with each other. Past this result there's some non-sense ramblings that honestly I've debated just deleting, but I feel like if you understand physics better than me, you might be able to go somewhere with it.

4.1 result: interpretation of Q as cosines of angles

For now, let's simplify and assume that $M = I = M^{-1}$ so that $Q = G_{\mathbb{A}}^T M^{-1} G_{\mathbb{A}} = G_{\mathbb{A}}^T G_{\mathbb{A}}$. This isn't a totally unreasonable assumption to make – in fact, in most of my simulation scenes I've set the mass of all the balls

to "1" which in fact does result in the mass matrix just being the identity matrix.

If you look in the rosi paper and through the code, you'll see that $G_\mathbb{A} \in M_{3m \times n}(\mathbb{R})$ where n = number of collisions (size of "active set"), and m = the number of balls in the simulation. Here we assume that each ball has 3 coordinates: an x, y, and a rotation $\phi = 0$ [3] (hence $3m$ rows).

Every column of $G_\mathbb{A}$ can be written as $(G_\mathbb{A}^T)_i = \nabla g_i(q)$, where $g_i(q)$ is the collision constraint between the "i^{th} pair" of balls (say, balls **a** and **b**). Then, using a_x, a_y, b_x, b_y to represent the indices of the x and y coordinates of balls a and b respectively, we have:

$$g_i(q) = \text{dist}(a,\ b) - r_a - r_b$$
$$= \sqrt{(q_{a_x} - q_{b_x})^2 + (q_{a_y} - q_{b_y})^2} - r_a - r_b$$

Where, for example, q_{a_x} is the x coordinate of ball a, and r_a is the radius of ball a. [4]

The constraint gradient (i.e. the columns that make up $G_\mathbb{A}$) can now be written as a vector of derivatives of this function with respect to each balls' coordinates:

$$\nabla g_i(q) = \left[0,\ \ldots,\ 0,\ \frac{\partial \text{dist}(a,\ b)}{\partial q_{a_x}},\ \frac{\partial \text{dist}(a,\ b)}{\partial q_{a_y}},\ 0,\ \ldots,\ 0,\ \frac{\partial \text{dist}(a,\ b)}{\partial q_{b_x}}\ \frac{\partial \text{dist}(a,\ b)}{\partial q_{b_y}}\ 0,\ \ldots,\ 0 \right]^T$$
$$= \left[0,\ \ldots,\ 0,\ \frac{q_{a_x} - q_{b_x}}{\text{dist}(a,\ b)},\ \frac{q_{a_y} - q_{b_y}}{\text{dist}(a,\ b)},\ 0, \ldots 0,\ \frac{q_{b_x} - q_{a_x}}{\text{dist}(a,\ b)},\ \frac{q_{b_y} - q_{a_y}}{\text{dist}(a,\ b)},\ 0, \ldots 0 \right]^T$$

Let's define the sub-vector $\vec{n_{ab}} := \left[\frac{\partial \text{dist}(a,\ b)}{\partial q_{a_x}}\ \frac{\partial \text{dist}(a,\ b)}{\partial q_{a_y}} \right]^T = \left[\frac{q_{a_x} - q_{b_x}}{\text{dist}(a,\ b)}\ \frac{q_{a_y} - q_{b_y}}{\text{dist}(a,\ b)} \right]^T$ which is made up of the first 2 non-zero values of $\nabla g_i(q)$ (I believe this is what the rosi paper refers to as a "collision normal"). Similarly, the other 2

---

[3]We aren't yet concerned with friction effects, so balls with no rotation initially stay that way during and after collision. Hence $\phi = 0$ and for the most part we can just ignore it.

[4]It's worth noting that this is the only place where the radius of the balls will appear. Once we take the gradient of $g(q)$ the r constants will disappear. This is why, in the C++ code, you won't see the radii be used in the calculation of the N matrix ($G_\mathbb{A}$).

non-zero entries of $\nabla g_i(q)$ make up the opposing collision normal: $\vec{n_{ba}}$ which is just equal to $-\vec{n_{ab}}$. Notice that $||\vec{n_{ab}}|| = 1$ (exercise: why?). It's a nice surprising fact that the first 2 non-zero entries of the constraint gradient for colliding balls a and b ($\nabla g_i(q)$) is just the normalized direction vector from the center of ball a to the center of ball b (and vice-versa for the 2nd pair of non-zero entries of the constraint gradient).

Now we know what the columns of $G_{\mathbb{A}}$ consist of, we can look at the individual elements of Q:

$$Q_{ij} = G_{\mathbb{A}i}^T \cdot G_{\mathbb{A}j}^T$$
$$= \nabla g_i(q) \cdot \nabla g_j(q)$$
$$= ...$$

... 3 Cases

It's helpful to draw out a picture example and do the computations for each case.

1: $i = j$ (Collision data is only coming from "the ith pair of balls")

In this case, $Q_{ij} = Qii = \nabla g_i(q) \cdot \nabla g_i(q) = ||\vec{n_{ab}}||^2 + ||\vec{n_{ba}}||^2 = 2$. So the diagonal of the Q matrix will be all 2s.

2: $i \neq j$ with 2 separate collisions (i.e. there are 4 balls constituting 2 separate pairs of balls colliding separately)

Consider $Q_{ij} = \nabla g_i(q) \cdot \nabla g_j(q)$ where $g_i(q) = \text{dist}(a, b) - r_a - r_b$ and $g_j(q) = \text{dist}(c, d) - r_c - r_d$. i.e. we consider 2 collisions (i,j) involving 4 distinct balls (a,b,c,d). Here the 4 non-zero entries of $g_i(q)$ will occur at different indices than the non-zero elements of $g_j(q)$! [5] Therefore $\nabla g_i(q)$ and $\nabla g_j(q)$ are orthogonal so $Q_{ij} = \nabla g_i(q) \cdot \nabla g_j(q) = 0$.

---

[5]The kth element of $\nabla g_i(q) \neq 0$ means that the kth element is the partial derivate of $g(q)$ with respect to ball a or b's x or y coordinate. This implies that the kth element of $\nabla g_j(q) = \dfrac{\partial g_j(q)}{\partial q_k} = 0$ since $k \in \{a_x, a_y, b_x, b_y\}$ and constraint between balls c and d is independent of a or b's position.

3: $i \neq j$ with 2 interacting collisions (3 balls colliding with each other simultaneously)

The most interesting case! Let's assume that 3 balls a, b, and c are colliding simultaneously. Say, the ith pair of balls consists of balls a and b, while the jth pair of colliding balls consist of balls b and c. Because of this, $\nabla g_i(q)$ and $\nabla g_j(q)$ will share some indices where the elements are non-zero. For example, in $\nabla g_i(q)$ we would see that the fourth element (first 3 elements are partial derivatives wrt ball a's x, y and ϕ coordinates respectively) is $\dfrac{\partial \mathrm{dist}(a,\ b)}{\partial q_{b_x}}$ which $\neq 0$ while in $\nabla g_i(q)$ the fourth element would be $\dfrac{\partial \mathrm{dist}(b,\ c)}{\partial q_{b_x}}$ which also $\neq 0$.

Then we would get:
$Q_{ij} = \nabla g_i(q) \cdot \nabla g_j(q) = \vec{n_{ab}} \cdot \vec{n_{cb}} = |\vec{n_{ab}}||\vec{n_{cb}}| \cos \theta = \cos \theta$ [6]

Where θ is the angle formed between the 2 line segments contecting the centers of balls a & b, and balls b & c.

4.2 interpretation of b, $Q\lambda$, LCP criteria

(honestly, feel free to ignore this section)

b

b is the other constant value of interest in out LCP. If we assume once again that $g_i(q)$ is the constraint between balls a and b:

$$
\begin{aligned}
b &:= G_{\mathbb{A}}^T \dot{q}^-(1 + c_r) \\
\implies b_i &= (1 + c_r)\nabla g_i(q)\dot{q}^- \\
&= (1 + c_r)\frac{dg_i}{dt} \\
&= (1 + c_r)\frac{d(\mathrm{dist}(a,\ b))}{dt}
\end{aligned}
$$

---

[6]exercise: fill in the lines of high-school linear algebra that I was to lazy to include here.

Interpretation: the b_i represents

$-(1 + \mathbf{c_r}) \times$ (relative speed ball a is approaching ball b).

(note the negative sign since $\frac{d(\text{dist}(a,\, b))}{dt} \leq 0$)

$Q\lambda$

From the Rosi paper, we know that $\lambda \in \mathbb{R}^{|\mathbb{A}|}$ is the vector of impulse coefficients (i.e. λ_i is the force that ball a exerts on ball b ***TODO: ensure this is *technically/semantically* correct!) Now consider the ith element of $Q\lambda$:

$$(Q\lambda)_i = \sum_{j=1}^{\mathbb{A}} Q_{ij}\lambda_j$$

For Q_{ij} there are 3 cases (see above). Firstly, collision constraint i could not be affected by either of the balls involved in collision j, in which case $Q_{ij} = 0$ and we can ignore those terms. Secondly, there will be a term where $i = j$, in this case $Q_{ij} = Q_{ii} = 2$. And finally, assume that constraint i is "concerned" with balls a and b, (i.e. $g_i(q) = \text{dist}(a,\, b)$) and j is "concerned" with *either* ball a or b *and* some 3rd ball c. In this case $Q_{ij} = \cos(\angle abc)$ or $Q_{ij} = \cos(\angle cab)$.

So, if we set:

$A = \{x \in \mathbb{A} | s.t.\ g_x(q) = \text{dist}(a,\, c)\} \setminus \{i\}$

And $B = \{x \in \mathbb{A} | s.t.\ g_x(q) = \text{dist}(b,\, c)\} \setminus \{i\}$

Where in both cases c is just a ball that is in the process of colliding with ball a or b respectively, then we can rewrite above as:

$$
\begin{aligned}
(Q\lambda)_i &= \sum_{j=1}^{\mathbb{A}} Q_{ij}\lambda_j \\
&= 2\lambda_i + \sum_{x \in A} \cos(\angle ba\ \text{ball}(x))\lambda_x + \sum_{x \in B} \cos(\angle ab\ \text{ball}(x))\lambda_x \\
&= 2\lambda_i + ||\sum_{x \in A} \text{proj}_{\overrightarrow{n_{ba}}}(\overrightarrow{n_{\text{ball}(x)a}})\lambda_x|| + ||\sum_{x \in B} \text{proj}_{\overrightarrow{n_{ab}}}(\overrightarrow{n_{a\ \text{ball}(x)}})\lambda_x|| \\
&= \text{net force acting on balls a and b along direction: } \overrightarrow{n_{ab}}
\end{aligned}
$$

15

TODO: clean up the equation above... using "ball(()x)" is confusing... maybe some of the ab should be ba... theres a little more explaining that could be done. ESPECIALLY relating to our disregard of mass - really, the above should be the net $\Delta \dot{q}$ once we bring back M^{-1} and divide each term by it's balls' masses.

LCP Criteria: $Q\lambda + b \geq 0$

This is saying for each collision $i \in \mathbb{A}$, we need $(Q\lambda)_i \geq -b_i$. Let $[\dot{q}]_{\overrightarrow{n_{ab}}}$ represent the speed of ball a wrt ball b.
As we have seen before, $b_i = -(1+c_r)[\dot{q}]_{\overrightarrow{n_{ab}}}$ so we can rewrite our lcp condition as:

$$\text{net force (speed?) acting on balls a and b} \geq (1 + c_r)[\dot{q}]_{\overrightarrow{n_{ab}}}$$

Or something like that... basically, the LCP condition is enforcing our solution (λ, the impulse coefficients) will result in exiting velocities that conserve momentum.

when we factor in the complementary condition, the times 2 above makes sense!

either $(Q\lambda + b)_i = 0$, in this case $\lambda_i > 0$ and we get

$$
\begin{aligned}
(Q\lambda)_i &= \sum_{j=1}^{\mathbb{A}} Q_{ij}\lambda_j \\
&= 2\lambda_i + \sum_{x \in A} \cos(\angle ba \text{ ball}(x))\lambda_x + \sum_{x \in B} \cos(\angle ab \text{ ball}(x))\lambda_x \\
&= 2\lambda_i + ||\sum_{x \in A} \text{proj}_{\overrightarrow{n_{ba}}}(\overrightarrow{n_{\text{ball}(x)a}})\lambda_x|| + ||\sum_{x \in B} \text{proj}_{\overrightarrow{n_{ab}}}(\overrightarrow{n_{a \text{ ball}(x)}})\lambda_x|| \\
&= ||\sum_{x \in A} \text{proj}_{\overrightarrow{n_{ba}}}(\overrightarrow{n_{\text{ball}(x)a}})\lambda_x|| + ||\sum_{x \in B} \text{proj}_{\overrightarrow{n_{ab}}}(\overrightarrow{n_{a \text{ ball}(x)}})\lambda_x||
\end{aligned}
$$

basically, our LCP solution in this case requires that the relative exit velocity of balls a and b is ONLY a result of the forces of the OTHER balls colliding with balls a/b.

16

Or... wait no nvm lol

TODO: more investigation here perhaps?

4.3   result: Q is not generally an "M-matrix"

This was something we were concerned with in previous semesters of work. The off-diagonal entries of Q are $\cos\theta$ where $\theta \in [0, \pi]$! It's easy to imagine a scenario where 2 balls (a and b) simultaneously collide with a common 3rd ball (c) so that the angle connecting the center of the 3 balls $\angle acb < \pi/2$ and thus $\cos\theta > 0$, ($\implies Q$ cannot be an M-matrix). It's especially to imagine this scenario occurring when there's lots of *overlapping* occurring.

5   Overlapping and Collisions Through Balls

5.1   "K3" Example

Generally speaking, when only 2 balls collide with each other, there isn't much special going on... $G_{\mathbb{A}} \in M_{3m \times 1}(\mathbb{R})$ so $Q \in M_{1 \times 1}(\mathbb{R})$ and λ is really easy to find.

A more interesting scenario can be found when 3 balls collide with each other while all on the same axis. [7] In this case, all angles are either 0 or π, so Q is: [8]

$$Q = \begin{bmatrix} 2 & 1 & -1 \\ 1 & 2 & 1 \\ -1 & 1 & 2 \end{bmatrix}$$

Which is singular! There are an infinite number of solutions to the LCP - and IPOPT does in fact find a different solution to policy iteration!

---

[7]This does required that ball collides with ball c *through* ball b... but sadly, because of issues that arise when we discretize time, this is a case that should be considered. Imagine 2 balls are going so fast that by the time the positions have been updated for the next frame (say, in a 60th of a second), a ball has mostly passed through it's neighbor, and partially into the neighbor's neighbor. This results in a case where all 3 balls could be overlapping (i.e. colliding) with each other at the same time, on the same axis.

[8]Order of values also depend on the order of collisions in the active set.

5.2 "K4" Example

4 balls (all overlapping each other) configured in a square with velocities towards the center of the square makes a "K4" type of collision graph (i.e. vertices of the graph are the center of the balls and the edges of the graph would be the line segments connecting the centers of balls who are colliding with each other) where each of the 4 balls are colliding with the other 3 in the same instant. This also produces a singular Q matrix (rank = 5, nullity = 1), and IPOPT/PI yield different solutions.

In both K3 and K4 examples, there are 3 viable policies (control sets) that would produce a viable λ solution. One way to think about what a policy/control set means in this scenario, is "which of the collisions are material?" i.e. in order to update the balls' velocities as a result of all the collisions happening in this frame, which collisions are important to create a new final velocity, and which ones can we ignore? (See `K4_demonstrate_PI_2solns.ipynb`)

Let's show some illustrations... if we cd into the `rigidbody2dqt4` directory and run the simulation with the K4 scene: `./rigidbody2d_qt4 ../../ura_research/scenes/simple/K4.xml`, we'll see something like this:
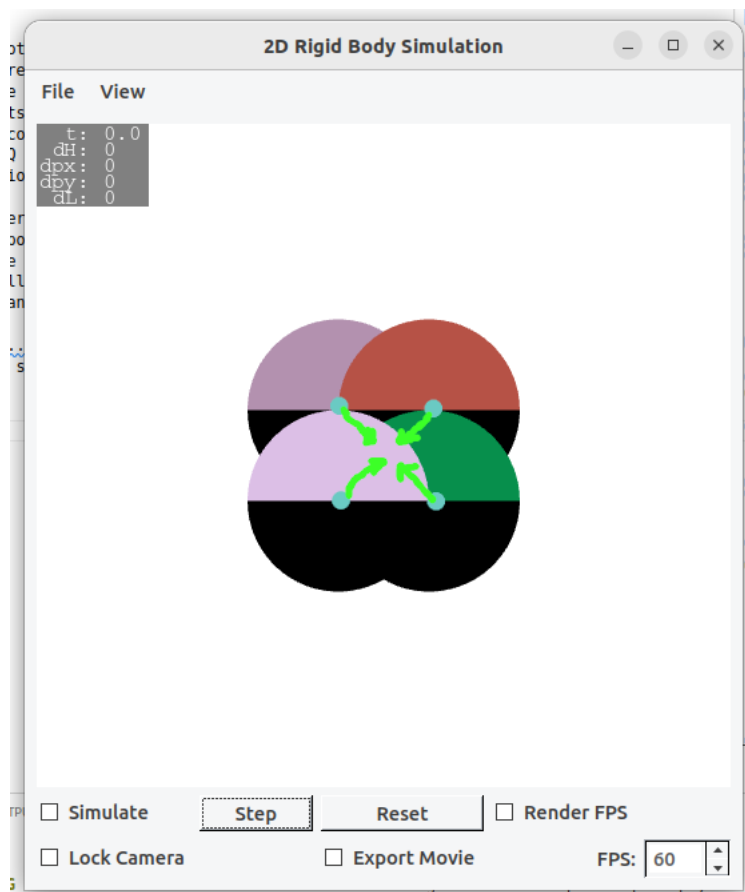
Here I've highlighted the center of the balls and drawn an arrow showing their initial velocities $\dot{q}^-$. Even though this isn't a totally realistic/physical scenario, one could instead imagine that all the something like squares instead with their corners all meeting in the middle. Each object is in a simultaneous collision with its 3 other neighbors.
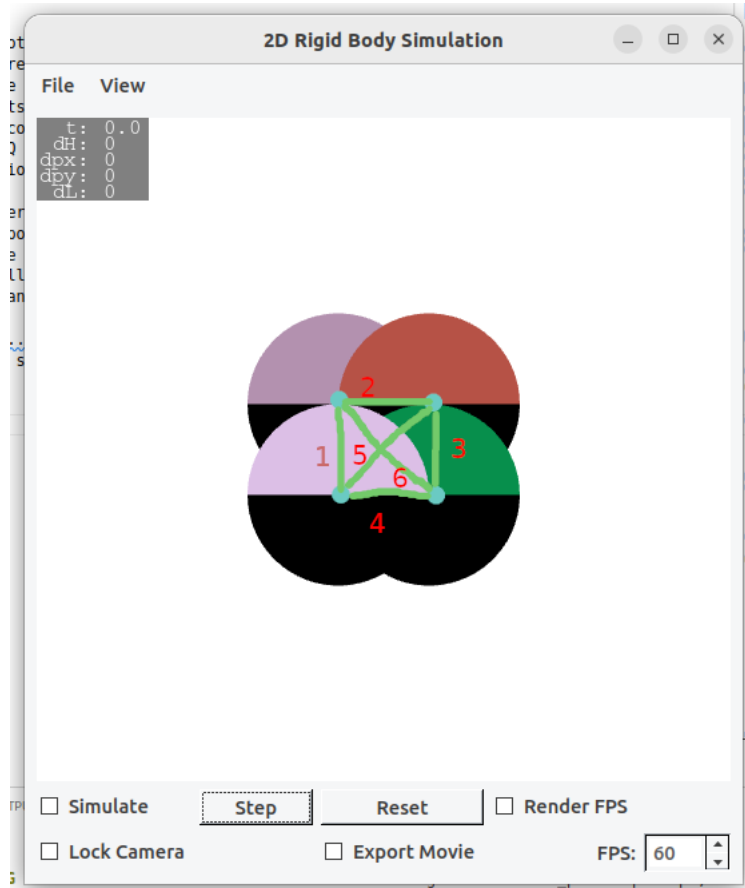
Here I've drawn all the collisions "normals" connecting each pair of colliding balls. This which shows the K4 graph structure. Each green line drawn corresponds to a column in the $G_{\mathbb{A}}$ matrix. Each green line also corresponds to an element in λ. Each lambda element is an "impulse coefficient" (see the rosi paper) that can exert force on 2 balls in the direction of each collision normal (i.e. green line i can be interpreted a like a compressed spring, whose amount of compression is proportional to the size of the corresponding λ_i. The higher λ_i is, the higher the force that those 2 balls involved in collision i will exert on each other).

There are infinite solutions (λ) to the above picture that result in the same correct exit velocity of the balls. Either (a) the 4 λ_is corresponding to the 4 outside collisions are > 0 with the 2 λ_is corresponding to the inside collisions are $= 0$ (in this case $\pi = \{1, 2, 3, 4\}$), (b) vice-versa ($\pi = \{5, 6\}$)), or (c) any linear combination of the 2 $\pi = \{1, 2, 3, 4, 5, 6\}$.

6 References

- BOKANOWSKI, OLIVIER, et al. "SOME CONVERGENCE RE-SULTS FOR HOWARD'S ALGORITHM." SIAM Journal on Numerical Analysis, vol. 47, no. 4, 2009, pp. 3001–26. JSTOR, http://www.jstor.org/stable/2786276 Accessed 2023.

- Zhang, W. 2022 A Comparison of Linear Solver Methods in Object-Collision Simulations. (See this folder for access)

- Wan, K. 2022. URA Report for Computer Simulation of Many Object Collisions. (See this folder for access)