Practical 2: Constraints

Handout date: 27/Feb/2023
Deadline: 21/Mar/2023 08:59

1 Introduction

The second practical generalizes and extends the first practical, by working with constraint-based velocity and position resolution. We still only work with rigid bodies. The objectives of the practical are:

1. Implement impulse-based velocity resolution by constraints (Lecture 6).
2. Implement position correction by constraints (Lecture 7).
3. Generalize collision resolution to work with the implemented constraint-resolution framework.
4. Extend the framework with some chosen effects.

Similarly to the first practical, this zip file contains the repository for the skeleton on which you will build your second practical. The environment is, for the most part, identical to the 1^{st} practical. **However**, it is not advised to use your practical 1 implementation as a starting point for practical 2, since there are some important differences between the two templates.

2 Background

The practical is mostly about implementing procedures for velocity and position resolution to satisfy constraints. This resolution integrates into the game-physics engine as follows:

In each scene iteration:

1. Integrate the accelerations into velocities, and the velocities into positions and orientations (as in Practical 1)
2. Detect collisions
3. Resolve Collisions through a generalized method that uses constraints, instead of the Practical 1 collision handling procedure.
4. Correct velocities to be tangent to user constraints (= constraints loaded from a file).
5. Correct positions to satisfy user constraints

The constraints we deal with in this practical are purely *holonomic* and bivariate. That is, each constraint C is of the form $C(x_1, x_2)$, where x_1 and x_2 are two positions on two meshes (can be the same mesh). We distinguish between *user constraints*, read from a file, and *collision constraints*, created on-the-fly in each iteration. Moreover, we distinguish between equality constraints $C = 0$ and inequality constraints $C \geq 0$. The difference between the two constraint types only matters for determining whether the constraint is *violated*, and otherwise should not affect velocities or positions. In practice, we use some tolerance τ that defines what we deem an 'acceptable' validity (opting for $|C| < \tau$ for equality constraints), rather than trying to achieve $|C| = 0$ which is unattainable numerically.

2.1 Working with Rigid Bodies

Constraints can be defined between any two points on a body (which happen to be vertices in user constraints). Just as in practical 1 the bodies are rigid, and therefore the only degrees of freedom for a mesh are its COM position p, orientation quaternion q, linear COM velocity v and angular velocity ω. Your implementation of constraint resolution should only work with and change these variables, and not touch any individual vertex. Specifically, never alter the vertex positions `currV` directly; these are updated for you by the template based on the current COM and orientation (as in practical 1).

2.2 Velocity Resolution

For equality constraints, the total velocities $\overline{v}_1$ and $\overline{v}_2$ should always satisfy $Jv = 0$, where J is the gradient of the constraint, and v is a vector comprising $v_1, \omega_1, v_2, \omega_2$ in order (note: this vector contains 12 variables). If $Jv \neq 0$, you will be computing Δv to satisfy $J(v + \Delta v) = 0$, using the Lagrange multiplier method learnt in class (Topic 5). This requires setting up an 12×12 (inverse) mass matrix, with the two objects' masses and their (inverse) inertia tensors in order. For fixed bodies, use 0 for inverse mass and inverse inertia tensor. This will simulate the correct effect (as if they have infinite resistance). The part in the mass matrix corresponding to the linear velocity has the scalar masses m_1 and m_2 repeated 3 times each in the diagonal of the matrix, for the x, y, z components of the respective velocities. Note that the inertia tensor should transform like in the first practical; essentially your constraint-based collision resolution should be almost equivalent to what you implemented explicitly before, just

in this more general framework.

NOTE: You should not implement a solution to collision that is separate from the constraints mechanism. Your constraint resolution implementation should handle all collisions.

The coefficient of restitution is given for collisions constraints in order to induce an elastic velocity bias; you should use it as instructed in class.

The user constraints that are read from a constraints file attach two vertices from two meshes, in the sense that the distance between them has to be maintained. That is, the constraint is $C(x_1, x_2) = |x_1 - x_2| - d_{12}$, where d_{12} is computed for the position at time $t = 0$. You should derive J for that constraint on your own (see Topic 6; for intuition, you are supposed to get that the velocities of both vertices should only move in a way that doesn't change this distance, like it's a fixed rod).

2.3 Position Correction

Position correction is similar to velocity correction, except that we take the easy route (in the basic practical requirements), and only correct *linearly*. That is, we do not change q, only p of every body. This means that the mass matrix has a size of just 6×6 and contains only body masses, without any inertia tensor components. The Jacobian only contains derivatives relating to linear movement. This generalizes the linear-interpenetration resolution for collisions. Note that position correction uses totally different J, M, λ , which do not relate to those computed in the velocity correction stage! The theoretical details are in Topic 6. We do not employ stiffness in this practical.

2.4 Extensions

Position- and velocity resolution of equality- and inequality constraints (including collision constraints) will earn you 80% of the grade. To get a full 100%, you must choose ***a single*** extension out of these 3 extension options, and augment the practical with it. Some choices will require minor adaptations to the GUI or the function structure which are easy to do. The extension will earn you 20%, and the exact grading will be commensurate with the difficulty. Note that this means that all extensions are equal in grade; if you take on a hard extension, it's your own challenge to complete it well.

1. Make the fixed-distance user constraint more flexible to some extent, and therefore a two-sided inequality constraint (for instance, the fixed rod could then compress or stretch up to 20% from the original d_{12}). **Level: easy**.
2. Fix the linear position-correction hack by adding q orientation correction to constraints. For this you will need the derivatives of the position of a point w.r.t. q, which are not trivial; look here for inspiration. **Level: intermediate-hard**.
3. Add another type of original constraint, which has to be concretely exemplified. For instance, bending or some limitation on rotation. **Level: intermediate**.

You may invent your own extension as substitute to **one** in the list above, but it needs approval on the Lecturer's behalf **beforehand**.

3 Installation

This installation is exactly like that of Practical 1, repeated here for completeness

The skeleton uses the following dependencies: libigl, and consequently Eigen, for the representation and viewing of geometry, and libccd for collision detection. The libigl viewer uses or the menu. Everything is bundled as submodules that automatically install with cmake, so that installation should be straightforward.

3.1 Windows

On a Windows machine you can use cmake-gui to compile the skeleton. Create a folder `build` inside the practical root directory, i.e. the INFOMGP-Practical2-master folder in which this readme file is stored. After downloading cmake-gui, enter the path to the root directory in the field labelled `Where is the source code`. In the field `Where to build the binaries` paste the path to the build folder you created. Pressing `Configure` twice and then `Generate` will generate a Visual Studio solution in which you can work. After opening the solution, remember to set the startup project to `Practical2_bin`, or the project will not run. Note: it only seems to work in 64-bit mode. 32-bit mode might give alignment errors.

3.2 MacOS / Linux

To compile the environment, go into the 'practical2' folder and enter in a terminal:

```
bash
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ../
make
```

4 Using the dependencies

You do not need to acquaint yourself much with any dependency, nor install anything auxiliary not mentioned above. For the most part, the dependencies are used in components of the code that are not a direct part of the practical. The most significant exception is Eigen for the representation and manipulation of vectors and matrices. However, this package has quite a shallow learning curve. It is generally possible to learn most necessary aspects (multiplication of matrices and vectors, intialization, etc.) just by looking at the existing code. However, it is advised to go through the "getting started" section on the Eigen website (reading up to and including "Dense matrix and array manipulation" should be enough).

5 The coding environment for the tasks

You will find the environment almost identical to the first practical, with these main differences:

1. A constraint file is being read, and has to be given as the third argument to the executable. For example, you can load the 'tower-chain' scene by providing `../data tower_chain-scene.txt tower_chain-constraints.txt` as command arguments.
2. The function `handleCollision()` needs to be written by you to work with constraints; see comments within.
3. The function `updateScene()` is already written to work with the game-engine loop. You don't need to change it.
4. Most of the work is in `Constraints.h`, where you have to fill in the velocity and position correction functions. This file implements a Constraint class that you can use in the Scene class.

The code you have to complete is always marked as:

```
/***************
TODO
***************/
```

The description of the function will tell you what exactly you need to put in. In some functions, you will have to complete parts you already did in the first practical (to avoid "spoilers")—it's a simple copy and paste (if you did it correctly the last time. Let the teaching assistant know if you were unable to get practical 1 working correctly; in this case you will be given the solution code to practical 1 after 7 March).

5.1 Input

The TXT file that describes the scene, where you have several examples in the `data` subfolder, is the same as in practical 1. For completeness, the format of the file is:

```
    #num_objects
object1.mesh  density1  youngModulus1 PoissonRatio1 is_fixed1    COM1     q1
object2.mesh  density2  youngModulus2 PoissonRatio2 is_fixed2    COM2     q2
......
```

Where:

1. `objectX.mesh` - a MESH file (automatically assumed to be in the `data` subfolder) that describes the geometry of a tetrahedral mesh. The original coordinates are translated automatically to have $(0, 0, 0)$ as their COM.
2. `density` - the uniform density of the object. The program will automatically compute the total mass using the volume and density.
3. `is_fixed` - if the object should be immobile (fixed in space) or not.
4. `COM` - the initial position in the world where the object would be translated to. That means, where the COM is at time $t = 0$.
5. `q` - the initial orientation of the object, expressed as a quaternion that rotates the geometry to $q * object * q^{-1}$ at time $t = 0$.
6. `youngModulus1` and `PoissonRatio1` should be ignored for now; we will use them in the 3^{rd} practical.

The user attachment constraints file, given as the third argument, has to have the following format:

```
#num_constraints
mesh_i1 vertex_i1 mesh_j1 vertex_j1
mesh_i2 vertex_i2 mesh_j2 vertex_j2
.....
```

Each row is a constraint attaching the vertex `vertex_i1` of mesh `mesh_i1` to `vertex_j2` of mesh `mesh_j2`. You can find TXT files in the data folder with a similar name to the scenes they accompany. You can, of course, write new ones. Note that the meshes start indexing from 1—if you put a constraint to mesh 0, it will get attached to the platform (which should still work).

5.2 User interface

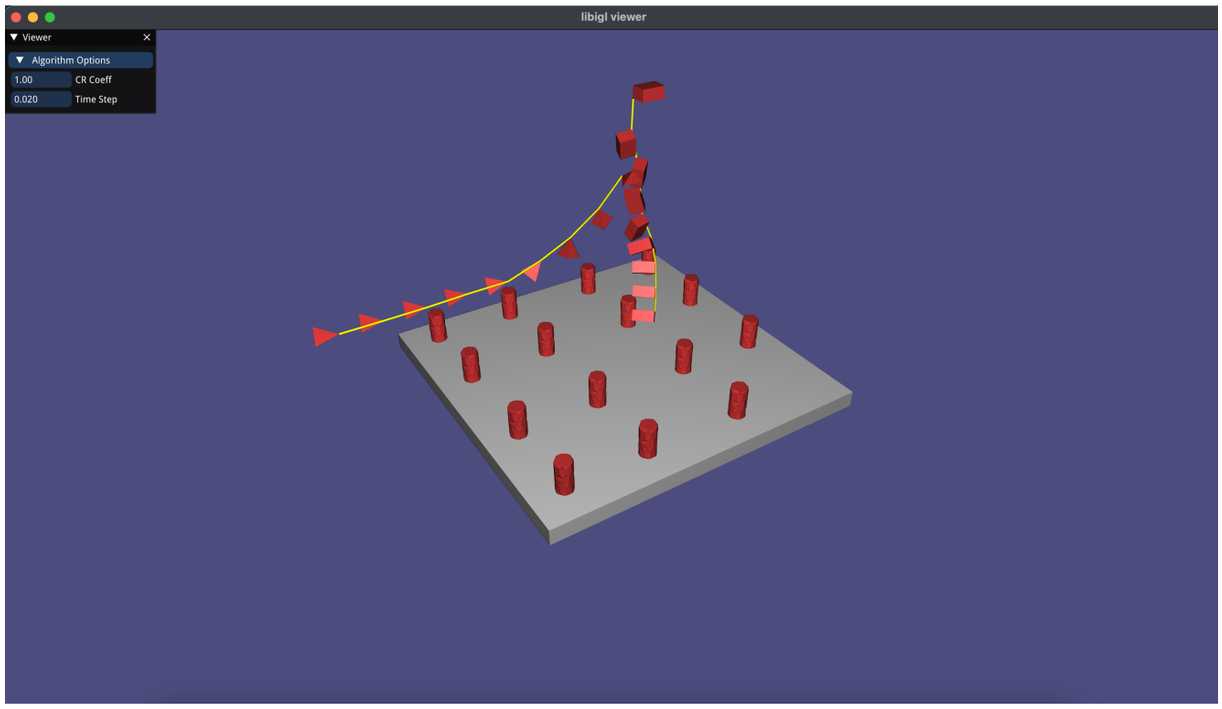

Figure 1: Practical 2 interface

The viewer presents the loaded scene, and you may interact with it using the mouse: rotate by clicking and dragging the left mouse button (the "[" and "]" buttons change the behaviour of the trackball). You can zoom with the mousewheel, and translate with the right button pressed and dragging. Some other options are printed to the output when the program starts.

The menu controls the setting of the coefficient of restitution and the time step. They can be updated at any point in the simulation. You might add more parameters with some extensions. Everything is set up in 'main()'.

The simulation can be run in two modes: continuously, toggled with the 'space' key (to stop/run), and step by step, with the 'S' key. This behavior is already encoded. The visual update of the scene from the objects is also already encoded.

The main interface difference between practical 1 and 2 is that user attachment constraints are displayed as yellow cylinders. These cylinders are simply markers to indicate the presence of a constraint, and are not real physical objects in the scene (so don't worry if they don't collide with other things; this is normal).

Note that the `demo` folder contains compiled demos for Windows and OSX; they are to be used as inspiration, because every solution can be a bit different (due to the butterfly effect).

6 Submission

All the files of the practical that you have modified to achieve your solution are to be submitted in a single zip file on blackboard. Note that this does not include any visual studio solution- or project files; only the c++ header files are to be submitted. The deadline is Tuesday 21 March at 09:00 AM. Students who have not submitted the practical by that time will not be checked in the session.

You are highly encouraged to do the practical in pairs. Doing it alone requires permission beforehand by the lecturer, and should come with the warning that the work load for a one person team may be quite significant.

The practical will be checked during a checking session on Tuesday 7 March. There will be no lecture that day. Every pair will have 10 minutes to shortly present their practical, and be tested by the lecturer with some fresh scene files. In addition, the lecturer will ask each team member a short question that should be easy to answer if this person was fully involved in the practical.

For the demonstration please bring a computer with an operating executable of your practical solution, compiled beforehand in release mode, and working on all given scene files. **Note**: In order to be able to give everyone sufficient time to demonstrate their code, the checking times will be strict. If you cannot come with your own computer, tell the teaching assistant in advance, and your code will be compiled on the lecturer's computer beforehand.

The registration for time slots is to be done in a spreadsheet that will be posted on Teams well before the deadline. You are not obligated to write your own explicit names—if you do not wish to do so, just write "occupied" and tell the teaching assistant via e-mail or Teams PM who you are and in which slot you want to present. Please do not change other people's time choices without their consent.

7 Frequently Asked Questions

Here are detailed answers to common questions. Please read through whenever you have a problem, since in most cases someone else would have had it as well.

Q: I am getting "alignment" errors when compiling in Windows.
A: Delete everything, and re-install using 64-bit configuration in 'cmake-gui' from a fresh copy. If you find it doesn't work out-of-the-box, contact the teaching assistant. Do not install other non-related things, or try to alter the cmake.

Good luck!

If you have questions about the practical you can post these in the Practicals channel on Teams. The lecturer and teaching assistant will try to respond as quickly as possible, but you are also encouraged to respond to each other's posts if you think you can help out.