CGGS Coursework 2 (Simulation): Soft-Body Simulation

Amir Vaxman, Thomas M. Walker

Errata

- None.

Introduction

This is the 2^{nd} coursework for the "simulation" choice. The purpose is to write Python code for a simplistic simulation of deformable bodies, with the finite-element method (FEM) approach to soft-body simulation, learned mostly in lectures 13 and 14. Visualization will be done with PolyScope as in all other Practicals. The concrete objectives are:

- implement the stiffness, mass, and damping matrices essential for the FEM,
- implement constrained discrete-time integration of the soft-body FEM equation to obtain velocities and positions,
- extension: implement the corotational approach for large deformations.

General guidelines

This coursework uses the same Python + PolyScope setup as in CW 1 (simulation). Your job is to complete several functions, given by signatures in the code, as described in the sections below. The code that you need to write is all within the script `SBSFunctions.py`. You will need to install the `meshio` Python package for loading tet meshes. You are also provided with the `SoftBodySimulation.py` script that deals with the loading of scenes and constraints. Explore its code documentation and options to see how you can modify it to your settings—it does not get graded, but rather only the functions you have to complete. Moreover, you have a script `CreateMesh.py` that allows you to create tet meshes (in `.mesh` format) from `OFF` files. You are encouraged to create scenes and meshes of your own to try! For this optional one, you will need to install the `tetgen` package (not needed for the actual CW otherwise).

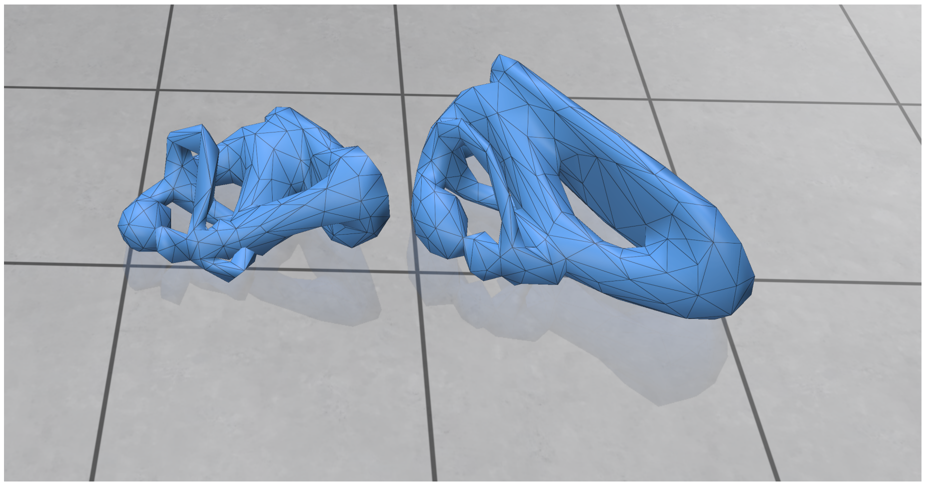

Figure 1: Results of the simulation for the `fertility-scene.txt`, where the same mesh with different material properties exhibits different behaviors.

Important note: this is a very basic simulation and moreover with some “hacks” to make the simulation much simpler to implement. Expect the results to be more or less plausible but far from what you would perceive as physical. We are also using a low-tet count since none of the elements of our algorithm are optimized, which adds to the coarseness. Note especially the artifacts of using linear FEM where objects often arbitrarily shear and scale.

1 File and Scene Formats

Soft-boy deformation is considerably more complex than rigid-body deformation, and therefore we adhere to a specific formatting of the code. There are three main classes:

- The `Constraint` class handles constraint data for collisions and distances (loaded from file), and has functionality for providing corrections for resolving positions, in the linear way you had in CW1. Since nothing much new happens here, code is for the most part already implemented for you.
- The `Mesh` class implements a single mesh in the scene. This is where you will implement the individual FEM matrices (See below).
- The `Scene` class is where the main action takes place: this is where you will integrate velocities and positions.

1.1 Global and local indexing

The FEM matrices used in the CW will work on *all vertices in the scene at once*. Furthermore, it works on them as a single vector in the vertex-dominant format $xyz, xyz, xyz, \cdots$, vertex after vertex. Thus, we keep two copies of the original and current vertices in the scene: one in the individual meshes, in the regular $|V_m| \times 3$ format (for mesh m), as `mesh.origVertices` and `mesh.currvertices`, and a *global* one in the `scene` class, as `globalOrigVertices`, `globalCurrVertices`, and `globalVelocities` for the current vertex velocities. They are all of the size $|V|$ where $|V| = \sum_m |V_m|$ over all meshes. The vertices in the meshes are mapped to the global one by order of mesh and then vertices in the mesh. The (already implemented) function `global2Mesh()` copies global values back to local ones.

Except for the definition of the individual FEM matrices, you only work with the global indexing in this CW. Integration is done on them, and constraints are defined by indexing into them. All the translation between local and global is already handled (and not very interesting in the context of what you need to do).

1.2 Scene file format

Soft-body deformation requires quite a few parameters. The loaded scene file is then of the following format:

```
#num_meshes
#name0 density0 YoungMod0 PoissonRatio0 #isfixed0 #xyz_position0, #orientation0
#name1 density1 YoungMod1 PoissonRatio1 #isfixed1 #xyz_position1, #orientation1
...
```

The position and orientation are just for the original position, and you shouldn't use them. The three important parameters are the density ρ, Young's modulus Y, and Poisson ratio ν, corresponding to the quantities learned in class. The meshes are of the `.mesh` format which encodes tetrahedral meshes.

In addition, we may have a constraint file in the same format and setting as for CW1; however, we ignore flexibility constraints, and for simplicity only do a single equality distance constraints, constraining the original distance between vertices on meshes.

2 Linear FEM Matrices

In this section you will compose the stiffness matrix K_m, mass matrix M_m, and dampening matrix D_m for every mesh m, where this should be encoded in `compute_soft_body_matrices()`. You will find details on how to compose them in the lectures and lecture notes. They are all square matrices of size $3|V_m| \times 3|V_m|$ for the vertices V_m, where they will eventually be appended to big matrices K, M, D of the entire scene, in `compute_global_matrices()` in the `Scene` class, which you should also implement. Note that they work on vertex-dominant vectors that are just the segments of the global vectors, that belong to mesh m.

Note that you will need to compute the Lamé parameters λ, μ from Y and ν, to form the stiffness tensor C. The dampening matrix will use the α, β parameters in the Rayleigh dampening: $D = \alpha M + \beta K$.

The matrices you created will be checked by the `FEMMatrixGrader.py`, and this is worth 40% of your grade directly. There is no report component in the section specifically, as it does not culminate in a visible simulation result just yet.

3 Constrained Time Integration

The soft-body loop works as follows:

1. Given (distance and collision constraints) from the previous iteration (if there are), perform integration for new velocities $v(t + \Delta t)$ and positions $x(t + \Delta t)$, which are in the scene global indices and for every vertex.

2. Collect new collision constraints and resolve their positions.

Let us first talk about the 2^{nd} step in which you have nothing new to do; this is a simple linear resolution of constraints that moves the entire body as rigid to fully resolve interpenetration. This is quite a hack, as true soft-body resolution is actually an intricate task involving constraint stabilization and penalties. Thus, the result might jitter a bit unrealistically; however, it's effective enough for our CW.

Collision constraints are expressed using the barycentric interpolation (and consequent gradient) learned in class; thus, they use $8 \times 3 = 24$ global indices per intersection of two tets; this is however transparent to what you need to do next.

Your task is to perform the first step, which is the constrained integration. You will solve the following system of linear equations for the velocities $v(t + \Delta t)$ and Lagrange multipliers β in each step:

$$\begin{gathered}(M + \Delta t D + (\Delta t^2) K) v(t + \Delta t) + J_c^T \beta = M v(t) - \Delta t K (x(t) - x(0)) + f_{\mathrm{e}} \\ J_c v(t + \Delta t) = 0\end{gathered}$$

as shown in class, with the KKT matrix. This will be done in the function `integrate_global_velocity()` found in the `Scene` class. J_c is the Jacobian of the constraints, which is computed with `compute_all_constraints_jacobian()` function (already implemented). Results should be fed into `globalVelocities`.

The next step is to update `integrate_global_position()` for integrating `globalCurrPositions`. Again, only update global positions and not individual mesh positions.

This Section will be checked by the `IntegrationGrader.py`, for 20% of your grade. Further 15% will be graded for the report detailing your insights on the results (and artifacts and advantages and disadvantages) of linear FEM over the existing scenes. You are encouraged to generate new scenes to demonstrate your insights more clearly.

4 Extension: corotational elements

This section is a more advanced extension of the practical, which is worth the final 25% of your total grade. It will not be checked automatically, but rather entirely depend on your demonstration and explanations in the report.

You will have noticed that linear FEM allows for quite a few scale and shear artifacts of deformation. To mitigate this, you should compute the co-rotational approach to computing the stiffness matrix taught in class. For this, in every iteration, you will need to compute the rotational part of the forces in each tet, and factor them in the individual element K_e before mixing it into the mesh K_m and then the big scene K. Demonstrate the difference clearly in the report, which should be evident where big deformations are involved.

5 Submission

You should submit the following:

- A report that must be at most 3 pages long including all figures, according to the instructions above, in PDF format.
- At least `SBSFunctions.py`, and potentially any other supporting files in case you made extensions that you would like to demonstrate.
- Any extra data files of scenes or constraints if you need them.
- You are encouraged to submit videos of your simulation with specific close-ups to demonstrate specific effect, and refer to them in the report.

The submission will be in the official place on Learn, where you should submit a single ZIP file of everything.