Assignment 2 - 3D Rigid Body Dynamics and Contact

In this assignment you will use the rigid body equation of motion to simulate the motion of rigid bodies floating in space, under the influence of gravity, and under contact forces. You will be able to observe the Dzhanibekov effect, and observe stability problems with fast rotation symplectic Euler integration.

Getting Started

You'll have libigl already from the first assignment, but you'll need polyscope for this assignment:

```
python -m pip install polyscope
```

Documentation can be found at the following link:

- https://polyscope.run/py/

Provided Code

The `a2comp559.py` file parses a json scene file and sets up a basic interface for time stepping the rigid body simulation in polyscope. A ground plane is set at y=0, and any simulation parameters not set in the scene file are set to defaults. Look at the `main_display_loop` and see that there is some polyscope interface code (e.g., running, stepping, setting parameters), and a few lines at the bottom for stepping the simulation (zeroing force accumulators, adding external forces, stepping velocities, checking and processing collisions to update velocities, stepping position, and updating the visualization).

The `rigidbody.py` file maintains loads the initial conditions and geometry definition. You will need to write code to step the velocity state and the position state. See that the mesh and mass properties are loaded from separate files, which can be created from your first assignment code. For instance, you can make other mesh files by modifying or A1 code to export inertia properties for a loaded mesh translated to have the center of mass at the origin with something like the following:

```
if args.out:
    V = V - com
    # compute J for mesh moved to the center of mass
    data = {}
    data['mesh_file'] = args.out + ".obj"
    data['mass'] = mass
    data['J'] = J.tolist()
    with open(args.out + '.json', 'w') as outfile:
        json.dump(data, outfile,indent=4)
    igl.write_obj(args.out + ".obj", Vmod, F) # mesh translated to have COM at origin
```

The `contact.py` file holds information for a single contact. There is code provided to compute a contact frame given contact point with given normal and interpenetration. The vectors are all in coordinates of the world coordinate frame. You will want to compute other information for the contact (i.e., Jacobian and inverse effective mass).

The `collision.py` file is where you will do the most work for this assignment. It provides very basic collision detection code which mostly works. While collisions with the ground plane are trivial, collisions between meshes are done with a signed distance computation of the vertices of one mesh with the definition of the other. Note that this can miss collisions, and likewise you may observe that igl can produce strange results in special configurations (at least this was my experience). There is code to visualize the contact frames and forces, which can help you identify if (when?) the basic collision detection code fails. Otherwise, see that there is a `process` function where you will need to solve for contact forces and update the body velocities.

Objectives

This assignment has several steps where the marks associated with different steps are below. Use the Erleben 2007 paper as a reference for your collision processing solution.

1. (3 marks) Finish implementing the rigid body velocity and position stepping code. You need to update velocity and angular velocity from torques and forces acting on the body, and remember to use the omega cross J omega term. In stepping position, be careful to update your rotation matrix correctly. You will also want to update the spatial frame rotational inertia matrix (and its inverse) with the current rotation of the body. You should use the spatial frame rigid body equations (though it would be a good exercise to use the body frame equations instead).

2. (1 mark) Display energy and momentum properties in the polyscope user interface (see the main display loop in `a2comp559.py` . Compute and display scalar quantities for the energy due to linear velocity, angular velocity, gravitational potential, and the total. You should also compute and display linear and angular momentum vectors. Monitor these quantities to understand that your implementation of step 1 is correctly preserving energy and linear and angular momentum in simulations where you would expect conservation (i.e., zero gravity simulations without contact or friction).

3. (2 marks) Compute the contact Jacobian. You should check your computation by testing with simple scenes that produce trivial contact geometry (e.g., a sphere dropping on the plane to generate a contact with a normal in the world z that goes through the center of mass, because this will give a Jacobian matrix with easy to interpret entries, i.e., you should print it out or look at in the debugger to make sure that it makes sense).

4. (1 marks) Compute the inverse effective mass matrix for each contact. For a pair of bodies, a single frictional contact constraint will give you a 3x3 matrix for the Delassus operator, in this case, known as the inverse effective mass. For the PGS solve you'll only need the diagonal entries, but you may find it easiest to create the full 3x3 matrix.

5. (1 mark) Prepare for the contact solve. You now have everything necessary to start implementing the `process` method in the `collision.py` class. Before you can start the PGS loop, you must compute the side b vector for each contact. This is simply the contact Jacobian times the velocities of the two bodies involved in the contact (stacked to form a 12 component vector).

6. (3 mark) Compute normal forces due to contact. You may have computed all 3 rows of the Jacobian of each contact, but here you'll only use the normal direction. Write a PGS solver that does 100 iterations where each iteration loops over all contacts. Compute the new Lagrange multiplier for the normal direction in the Gauss-Seidel manner, then clamp it to zero if it is negative (i.e., acting like glue). Use Erleben's method of maintaining a delta velocity of the bodies, that simplifies the computation of the new Lagrange multiplier, and is cheap to update with any change of the Lagrange multiplier. Note that the easiest way to store this information is in each body, rather than forming a vector for all bodies. Likewise, store the Lagrange multipliers for each contact in the

contact objects, rather than making a big vector of all Lagrange multipliers. See that you can set the simulator to stop on first contact and display the contact forces, which will let you carefully examine the results of your contact solve to see that they are correct.

7. (3 mark) Compute tangential forces due to friction. This will be almost identical to what you did for normal forces, except you need to clamp friction forces based on the friction coefficient mu and the current normal force of the contact.

8. (1 marks) Implement constraint stabilization and compliant contacts. Create parameters in the scene file simulation parameters to control these added features, and document your efforts in the readme.txt file. You can test compliance by seeing that you can produce equal contact forces at all four corners of the scene involving the brick falling on the ground.

9. (2 marks bonus) Control PGS iterations with a parameter in the scene file and implement a warm start for Lagrange multipliers. This will greatly speed up convergence of the PGS solver. You can often easily associate a contact at the current frame with one at the previous frame, for instance, in the case where you have the same vertex penetrating the ground plane.

Out of the scope of this assignment are a few other interesting things.

- Recognize that the omega cross J omega term introduces instability at higher velocities (over time, energy grows, and so does angular momentum), and use a simple technique to stabilize this term.
- Mentioned previously, one could compare with a solution using the body frame equations of motion.
- Try implementing a rattle back top by combining an ellipsoid and a rotational inertia computed for a slightly rotated shape (e.g., pretending that the mass was distributed differently internally)
- Use different techniques for collision culling, for instance, handle mesh-mesh collisions using sphere trees built with medial axis sphere approximations.

Finished?

Great! Submit a zip file using the same structure as the zip you were provided. Be sure to add your name and student number into the comments at the top of each file and in the UI. Add any other information you want in a readme.txt file (e.g., request to use your late penalty waiver). Use only zip format, i.e., do not use any other format for your submitted file.

Note that you are encouraged to discuss assignments with your classmates, but not to the point of sharing code and answers. All code and written answers must be your own. Please see the course outline and the fine print in the slides of the first lecture.