Assignment 2 - Numerical Integration

In this assignment you will use explore different numerical integration methods.

Getting Started

You should have all the modules needed for this assignment already installed.

Provided Code

The `a3comp559w.py` file parses a json scene file and sets up a basic interface for time stepping a mass-spring particle system polyscope. The `particle_system.py` file has a constructor for loading the json file, and sets up different parameters for simulation. It provides the **derivatives** method that is used for explicit numerical integration methods, i.e., this will compute the acceleration of the system at the given provided state. The `particle_system` class needs to be finished to compute the forces, and quantities needed for the backward Euler solve. See that many of the function stubs take the position and or velocity of the system, and while the class is a container for the current state of the system, these functions should work instead with the state that they are passed, i.e., the functions are called to ask "what are the forces if the position velocity in the given state"?

One note of warning. It is likely your main challenge will be to wrangle vectors and matrices and to reshape them for different reasons. See that the position (and velocity) data loaded form the json file will be n by 2, but when it comes to setting up and solving matrices you may want to `flatten` or `reshape` this data as you compute values or update state. You may also prefer the data as vectors rather than n-by-2 matrices for the explicit methods, but this could need changes to the `phase_space_wrapper` and `phase_space_unwrapper`, which is code you should examine carefully to understand how the integration methods are defined for use with our 2nd order system along with implementations of explicit method for a 1st order system.

Another note of warning with respect to your backward Euler solve. There are multiple ways for solving the Ax=b equation. One is to use `np.linalg.solve`, but you should notice that there are pinned particles that must not be part of your solve. If you assemble the matrix to solve in this manner, then you should assemble an Ax=b system that only involves the degrees of freedom you need to solve. You may find the `np.ix_` function useful for extracting sub-matrices for a given list of indices of your degrees of freedom see the docs. This will probably be useful both when building your stiffness matrix, but also when selecting the sub-matrix to solve. Otherwise, you can also solve the Ax=b system, pinned particles and all, using the conjugate gradient algorithm modified to include filtering. See the Baraff and Witkin 1998 or Ascher Boxerman 2003

Objectives

This assignment has several steps where the marks associated with different steps are below.

1. **Spring Forces (2 mark)** Finish implementing the code that computes spring deformation and viscous damping forces (note that you only need simple viscous damping and should not implement spring damping or Rayleigh damping). Do not use the spring stiffness as a material stiffness, as shown in the PBA course notes, but use it as the element stiffness as seen in class. As such, you'll be able to more easily check stability of numerical integrators at different step sizes and different mass stiffness and damping parameters with the analysis from

class. If you use `sympy` to compute the force and stiffness for springs, use `cse=True` for lambidfy to get faster code via common subexpression elimination, otherwise it is also fine to use external code-gen (e.g., matlab), or write code by hand to compute these quantities (recall that testing with finite differences is a good idea if you do so).

2. **Forward Euler (1 mark)** Finish implementing the `forward_euler` integration. See how `advance_time` calls on the particle system call your function through the `phase_space_integrator` wrapper. Your other explicit numerical integration implementations will use the same wrapper.

3. **Midpoint Method (1 mark)** Implement the `midpoint` function for midpoint method numerical integration.

4. **Modified Midpoint Method (1 mark)** Implement a `modified_midpoint` function, using the 2/3 point stepping scheme discussed in class.

5. **RK4 Method (1 mark)** Implement a fourth order Runge-Kutta integration class.

6. **Symplectic Euler (1 mark)** Implement the `symplectic_euler` function, and note that it does not use the phase_space_integrator wrapper.

7. **Backward Euler (3 marks)** Implement a semi-implicit backward Euler step computation (i.e., solve the linearized system). Assemble and solve, or use CG with filtering without assembling the matrix. If you assemble, dense matrices are fine for this assignment, but a bad idea in general as they will not work well for large systems. Similarly if you use sparse matrices, if the sparsity structure doesn't change, then you probably want a pre-allocated sparse matrix with the correct sparsity pattern. Avoiding matrix assembly is a good approach, but requires more work! That is, you'll need code to give you the answer on multiplication by a matrix, and you'll need your own implementation of the CG solve.

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.