LinearBody Class

This document serves as a description of the class, the motivations behind key decisions, limitations, and assumptions inherent to its design. The document also features an overview of the various public and private methods in the class, and can guide you through the layout of the code.

All arrays are 0 indexed unless otherwise stated.

Many of the methods are quite specific to this task, and and an abstract "Body" class may be desirable. This implementation is far from perfect. Some suboptimal design decisions are listed below and may be considered as improvements for the future:

- Switch array pairs (positions, forces, velocities, etc...) to proper vectors (use TVector class, or make a new vector class.) This way, all vector operations could be written in one statement instead of n-component statements.
- Clean up some of the temporary arrays created at instantiation. They are used to share calculated data between force application functions, but there is probably a more elegant way to do this.

A Note on Units

Units are all relative to each-other. If you decide to use SI units, then all parameters will behave as you expect them to. Length would be in meters, mass in KG, force in Newtons, and time in seconds.

External API

Setting Up LinearBody

The class constructor, **LinearBody(int segments),** takes one parameter: The number of segments you'd like the body to contain. Note the terminology. A segment is defined as a node surrounded by two other nodes. This means that if you instantiate a LinearBody with 3 segments, the model will actually simulate 3+2 nodes, two "end nodes" automatically inserted and accounted for. You don't have to worry about this discrepancy until you use the NodesPointer, which is described later in this section. For now, just remember that a model with n segments will actually have n+2 nodes.

LinearBody contains various public instance variables that correspond to the various physical constants used in simulation. These variables should be modified after instantiation and before you call Setup(), described later. These public variables (all of type double) are:

- **springK**: Linear spring constant for hooke's law spring between nodes.
- **springL**: Equilibrium length for hooke's law springs between nodes.
- **springD**: Dampening constant for hooke's law springs between nodes.
- **torsionalSpringK**: Constant for linear torsional spring at each segment.

- **torsionalSpringD**: Dampening constant for linear torsional spring at each segment.
- **nodeFriction**: Direction independent kinetic friction experienced on each node.
- **nodeMass**: Mass of each node.
- **muscleK**: Multiplier on strength of muscle.
- **fluidDragConstant**: The viscosity of the fluid. Multiplied by force vector normal to each bar between nodes.

After modifying these parameters, you must call **Setup(int timestep).** This method clears all forces, resets nodes to be at equilibrium length, and runs initial calculations (such as angles and distances) to be ready for the first simulation step. You must declare the timestep at this point. There are many simulation parameters (notably timestep and springL) that should **not** be changed after you call Setup(). To stay safe, **do not modify ANY simulation parameters after you call Setup()**. If your experiment requires this, you may need to extend or modify the class.

Running the Simulation

You can execute one simple Euler step (using the timestep declared with the Setup method) with the **EulerStep()** method. LinearBody only supports simple Euler integration. It is easy to implement any other type of integration using EulerStep() as a template. Forces are completely recalculated before the step is taken. EulerStep() will return zero upon a successful step. If the body enters a “dead” state (described later in this document), EulerStep() will immediately return as non-zero for subsequent calls. **Check to make sure EulerStep() returns 0**.

Various methods are exposed to control and observe the body during simulation.

Double GetBendingAngle(int segment) takes a segment index and returns the measure, in radians, of the angle made by sweeping an arc counterclockwise from the bar between the adjacent node towards the lower index end of the body, and the bar reaching toward the opposite node. This value represents the angle after the previous integration step. A perfectly linear body will report angles of precisely pi. Note that angles do not “wrap around” should forces get large enough to cause a segment to cross over itself. When this happens, the body will be considered dead. The public instance variable “dead” is set to true, and subsequent calls to integration methods will instantly return with nonzero value.

Void SetMuscleInput(int segment, double value) sets the muscle input of segment to value. You probably want this value bounded between -1 and 1, though feeding in higher or lower values will not cause any runtime errors. This value is multiplied by muscleK internally. muscleK*value is the torque you are applying from this segment.

Int GetNumSegments() returns the value passed to the constructor. This value is precisely two less than the number of nodes being simulated.

Double* GetCOM() returns a pointer to two doubles, the first double being the x coordinate and the second double being the y coordinate of the body's center of mass. Since all nodes have the same mass, this value is actually ignored and the positions are merely averaged. This block of memory resides on the heap and will exist until destruction of the LinearBody. Subsequent calls to GetCOM() will update the values in this memory, but will not change the memory address. The center of mass is not updated automatically on calls to EulerStep(). You must call GetCOM() every time you need to read out the center of mass.

nodeState* GetNodesPointer() exposes a (normally protected) internal array of nodeStates. This method's purpose is to expose all of the positions, velocities, and forces at each node for efficient observation. This array has GetNumSegments() + 2 length. The first and last nodes in this array are the "dummy" nodes tacked on at each end. The bendingAngle and muscleInput for end nodes are not used and will be zero. You probably shouldn't set anything in this array unless you have a really good reason. Values are updated during execution of EulerStep(). The memory will remain allocated and in the same place for the lifetime of the LinearBody. The definition of a nodeState is given in the LinearBody header file.

Implementation Details

This section describes some of the inner workings of the LinearBody class. After reading this section, you'll be familiar with naming conventions and program flow. This section is intended for readers who need to extend the class's functionality, change active forces, or implement any future improvements.

Flow

The general flow of a step is as follows:

- UpdateForces() is called.
    - f_clear is called to zero the forces from the last step.
    - f_precalc is called to update variables with values needed by multiple force applicators.
    - fa_**** are called which add forces to nodes.
- DeathCheck() is called. If the body is dead, return nonzero.
- Forces are integrated into velocity, which is integrated into position.
- Return 0

Naming Conventions

The prefix "f_" is for variables that are primarily used in force calculation or application methods.

The prefix “fa_” is for force application methods. These methods (generally protected or private) are called during the force update period of a step evaluation.

Forces

Forces are implemented as methods with an “fa_” prefix (read as “force applicator”). These methods are called from UpdateForces(), and should not depend on the previous execution of any other force applicator method. They can use positions, velocities, and bendingAngles from the nodeState array, as well as precalculated values such as those in the instance variables f_distances (length of bars between nodes), f_distances_delta (change in this length between the previous two steps), and f_bendingAngle_delta (change in bendingAngle between the previous two steps). A force application method then calculates the actual forces, and applies it to each node.

To implement your own force, create a new force applicator method that behaves as described above, and add a call to it from the UpdateForces() method. If you are implementing multiple forces that share similar calculations, consider creating a new variable in nodeState, and calculating this value once in the f_precalc method which will be called once per step, before any force applicators.

To remove a force, comment out its call in UpdateForces().

Death

The conditions upon which a body should be declared “dead” are defined in the DeathCheck() method. This method is called once per step, right before the step. DeathCheck() should set the instance variable “dead” to true if any death conditions happen. The dead variable should then be **returned** at the end of DeathCheck(). It is probably unwise to unset dead once it is true, as zombie C. Elegans are undesirable and most dangerous to civil society.