Animation & CGI Motion: Mass-Spring Systems

Theme I, Milestone I

Introduction

Welcome to Animation & CGI Motion!

Our first *theme* for this course focuses on how to *get things moving*, i.e., how to integrate a dynamical system forward in time. We will implement a particle system that supports a number of forces, and advance this system forward in time using a number of different integration methods, discovering the advantages and disadvantages of each.

This first theme is divided into three assignments called *milestones*. Prior to each milestone, you will be given starter code to serve as your foundation for the week's milestone. You will also have access to both a grading *oracle* that you can benchmark your program's output against, as well as half the examples that will be used to grade your program.

Chapter 1

Notes

Mechanics deals with the kinematics and dynamics of a mechanical system. Kinematics *describes* the motion of a system (*e.g.*, as relations among position, velocity, and acceleration) while dynamics identifies the *causes* of the motion of a system (*e.g.*, the forces).

1.1 Kinematics

Configuration

A mechanical system can typically be in one of many possible positions. We will denote a specific position, or *configuration*, by $\mathbf{q}$. We use the boldface notation in typeset documents (and an underline when writing on the blackboard) to distiguish vectors from scalars.

Since a system could be in one of many configurations, we can also refer to the set of *all* possible configurations as the *configuration space*, Q.

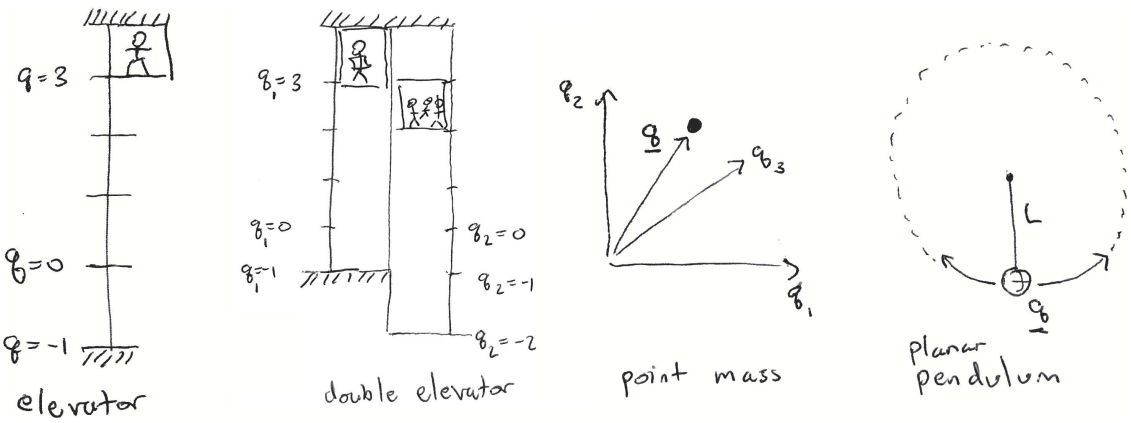

Figure 1.1: *Examples of configuration spaces.* From left to right: $Q \equiv [-1, 3]$, $Q \equiv [-1, 3] \times [-2, 3]$, $Q \equiv \mathbb{R}^3$, $Q \equiv S^1(L)$ (the circle or radius L).

Consider the examples in Figure 1.1:

- For an elevator that can travel only up and down, from floor -1 to floor 3, the configuration space is $Q \equiv [-1, 3]$, an interval subset of the real number line.
- A double-shaft elevator system has one elevator that can reach the basement (floor -1), and another that can reach all the way to the double-basement (floor -2); the configuration space $Q \equiv [-1, 3] \times [-2, 3]$

is the product of two intervals, i.e., any point $(q_1, q_2) = \mathbf{q} \in Q$ can be specified by the position of each of the elevators individually.

- For a point mass, or particle, in three dimensions, the configuration space is $Q \equiv \mathbb{R}^3$.
- For a pendulum of length L anchored at the origin and restricted to swing on the plane, the configuration space is a circle of radius L centered about the origin, each point on the circle corresponding to a possible position of the pendulum's bob.
- If the pendulum lives instead in three dimensions so that it can swing out of the plane, then its configuration space is given by the surface of a sphere.

Degrees of Freedom

We can talk about the dimensionality of a configuration space:

- The elevator has a one-dimensional configuration space Q, because (away from the limits, $q = -1$ and $q = 3$) the configuration space has the same topology as the real number line $\mathbb{R}^1$, i.e., it can be represented by a single real-valued coordinate (the height).
- The particle has a three-dimensional configuration space, because its configuration can be represented by three real-valued coordinates.
- The pendulum has a one-dimensional configuration space Q, because at any point on the circle $S^1(R)$, it can move only forwards and backwards, just like the elevator; its configuration can be represented by a signel real-valued coordinate (e.g., an angle measured relative to the positive x axis.

We say that a mechanical system with a k-dimensional configuration space has k ***degrees of freedom.***

We could say a lot more about these configuration spaces. For example, the elevator's configuration space is one-dimensional *with boundary* (it has limits $-1 \leq q \leq 3$, versus the pendulum's one-dimensional configuration space which has no boundary and is *periodic*, versus the particle's configuration space which neither has a boundary nor is periodic (it is infinite).

Trajectory

Suppose that we are interested in the motion of the system over some interval of time $[0, T] \subset \mathbb{R}$. The *trajectory* of the system is curve in configuration space, $\mathbf{q} : [0, T] \mapsto Q$, which maps any instant in time t to the configuration $\mathbf{q}(t)$ at that instant.

Referring to Fig. 1.2,

- the trajectory of our elevator, $\mathbf{q}(t) : [0, T] \mapsto [-1, 3]$, is a function that returns the height of the elevator for any given instant in time t, so that as t advances $\mathbf{q}(t)$ traces out a path that can go up and down along the interval $[-1, 3]$;
- the trajectory of a particle, $\mathbf{q}(t) : [0, T] \mapsto \mathbb{R}^3$, is a function that returns the three-dimensional position of the particle for any given instant in time t, so that as t advances $\mathbf{q}(t)$ traces out a path in three-dimensions;
- the trajectory of a planar pendulum is a function that returns a position on the circle for any given instant in time t, so that as t advances $\mathbf{q}(t)$ traces out a path along the circle, possibly doubling back on itself as the pendulum swings.
- emphthink for yourself: what does the trajectory fo the double elevator look like?
- *think for yourself:* what does the trajectory of a spherical pendulum look like?

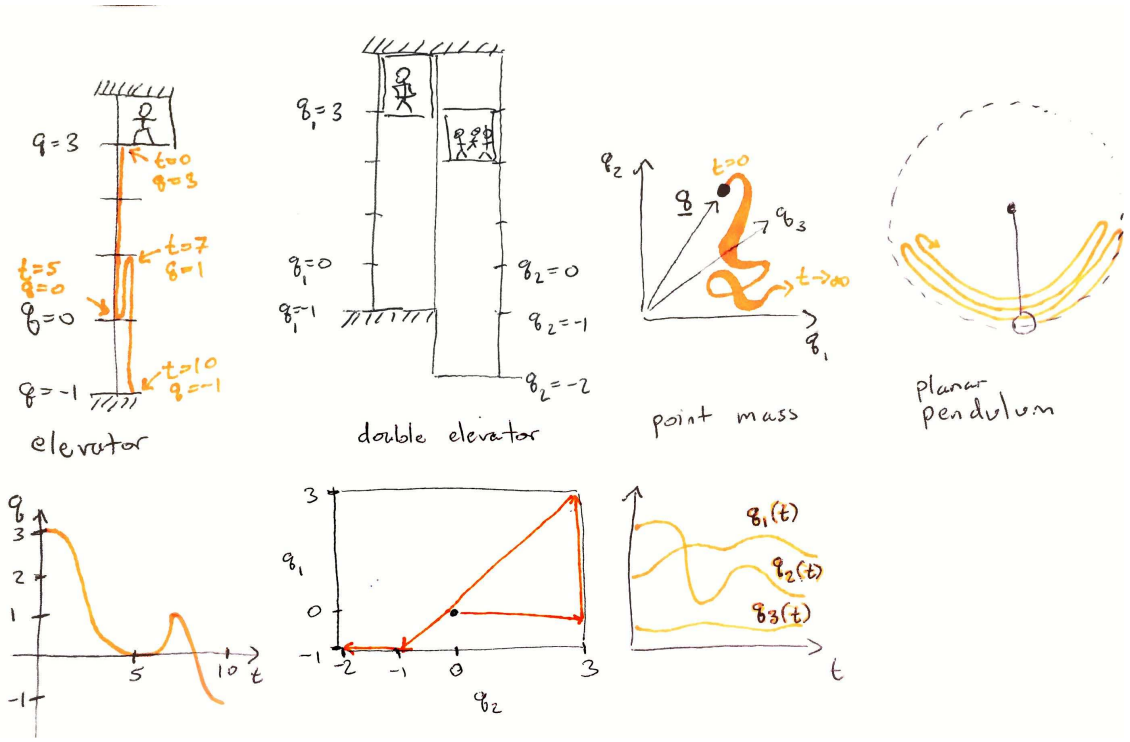

Figure 1.2: *Examples of trajectories.* From left to right: the configuration of the elevator evolves over a down-up-down trajectory *(top)*, which we can graph a real-valued function of time *(bottom)*; the configuration of the double-shafted elevator system can be plotted as a parametric curve $\mathbf{q}(t)$ in the rectangle $[-1,3]\times[-2,3] \subset \mathbb{R}^2$ *(bottom)*; *think for yourself:* can you trace out the path of the two elevators in configuration space *(above)*? The configuration of the particle evolves in a three dimensional curve *(top)*, as plotted in the graph of the three coordinates as function of time *(bottom)*; the pendulum swings back and forth in a trajectory that traverses an arc of the circle $S^1(L)$ back and forth. What coordinate system could we use to graph the trajectory of the pendulum?

Velocity

The velocity of the object, also called the *configurational velocity* or the *tangent vector* to the trajectory, $\mathbf{v}(t) = \dot{\mathbf{q}}(t) = \mathrm{d}\mathbf{q}(t)/\mathrm{d}t$, gives the instantaneous <u>*rate*</u> *and* <u>*direction*</u> of motion of the system. When the velocity is written in coordinates, the number of coefficients needed is the same as the dimensionality of the configuration space.

Referring to Fig. 1.3,

- an elevator rises and falls with velocity $\dot{q}(t) = \frac{dq}{dt}$;
- the velocity of the double elevator can be expressed in terms of two coefficients: $\dot{\mathbf{q}}(t) = (\dot{q}_1(t), \dot{q}_2(t))$.
- the velocity of a particle is given in coordinates by $\dot{\mathbf{q}}(t) = \frac{d\mathbf{q}}{dt} = (\frac{dq_1}{dt}, \frac{dq_2}{dt}, \frac{dq_1}{dt})$. Its speed is $v(t) = |\dot{\mathbf{q}}(t)|$ while its direction of motion is the unit vector $\hat{\mathbf{v}}(t) = \frac{\mathbf{v}(t)}{v(t)}$;
- for a planar pendulum, we can think of the configuration as being a radius of the circle (connecting the origin to a point on the circle), and the configurational velocity is always tangent to the circle, and therefore perpendicular to the radius;
- *think for yourself:* what are the only possible directions for the tangent vector for a spherical pendulum?

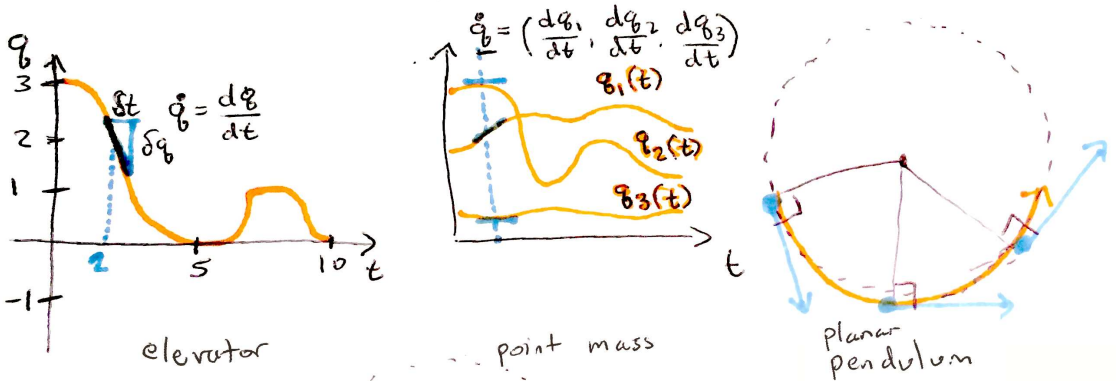

Figure 1.3: *Examples of velocities.*

State

Consider a projectile of given mass m and initial position $\mathbf{q}_0$, subject to only the force of gravity (see Fig. 1.4). How much more information do we need to predict the trajectory? Knowing the configuration is not sufficient, because different velocities will lead to different trajectories. If we know the position *and velocity*, then we can predict the trajectory under the influence of gravity (or other forces).

We define the *state* $\mathbf{y}$ of a mechanical system as a vector that concatenates the configuration and the configurational velocity, i.e., $\mathbf{y} = (\mathbf{q}, \mathbf{p})$. Because $\mathbf{q}$ and $\mathbf{p}$ have the same dimensions (they both require the same number of coefficients when written in coordinates), the state vector $\mathbf{y}$ is always of an even dimension.

Think for yourself: What is the dimensionality of the state for each of the systems surveyed in our figures above?

Sometimes, we define state in terms of *momentum* (instead of velocity); either approach is acceptable, as long as we remember to account for the mass in calculations that use the state. Let us elaborate on mass and momentum.

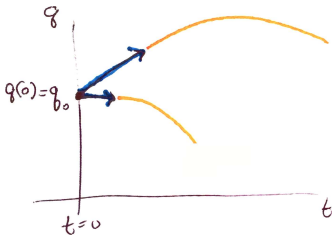

Figure 1.4: A projectile's trajectory depends on initial momentum.

Mass, Momentum, and Kinetic Energy

Closely related to velocity is *momentum* $\mathbf{p} = M\dot{\mathbf{q}}$, the velocity scaled by the mass. While we often think of mass as a real-valued scalar m, observe that we have written it here as a matrix M. Two natural questions, then, are what are the dimensions of M, and, why do we write mass as a matrix? We can answer the first question by "type-checking:" if Q is k-dimensional, then $\mathbf{p}$ and $\dot{\mathbf{q}}$ are k-dimensional vectors. Two such vectors can be linearly related either by a scalar constant of proportionality, as in $\mathbf{p} = m\dot{\mathbf{q}}$, or by a $k \times k$ square matrix, as in $\mathbf{p} = M\dot{\mathbf{q}}$. We will need the latter relation, as it is more general than the former (*think for yourself: how can you implement the former in the latter?*). For example,

- for the particle, M is a 3×3 matrix,
- for the double elevator, M is a 2×2 matrix;
- of course, for a one-dimensional system, such as our single elevator, M is equivalently a 1×1 matrix or a scalar.

Why do we opt for the generality of the matrix? For a single elevator, $p = mv$. What about for the double elevator? Let the elevators have distinct masses m_1 and m_2. Considering each elevator in isolation,

we have $p_1 = m_1\dot{q}_1$ and $p_2 = m_2\dot{q}_2$. For the ensemble as a whole, we express the relation between $\mathbf{p} = (p_1, p_2)$ and $\dot{\mathbf{q}} = (\dot{q}_1, \dot{q}_2)$ via the linear system

$$\underbrace{\begin{pmatrix} p_1 \\ p_2 \end{pmatrix}}_{\mathbf{p}} = \underbrace{\begin{pmatrix} m_1 & 0 \\ 0 & m_2 \end{pmatrix}}_{M} \underbrace{\begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \end{pmatrix}}_{\dot{\mathbf{q}}} .$$

Thus, for the double-elevator system, M is a 2×2 diagonal matrix with distinct diagonal entries.

Although we always think of a mass *matrix*, sometimes it does indeed act as a scalar. Consider a single particle in three dimensions; here $\mathbf{p} \in \mathbb{R}^3$ and $\dot{\mathbf{q}} \in \mathbb{R}^3$, and

$$\underbrace{\begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix}}_{\mathbf{p}} = \underbrace{\begin{pmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{pmatrix}}_{M} \underbrace{\begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{pmatrix}}_{\dot{\mathbf{q}}} ,$$

which is equivalent to $\mathbf{p} = m\dot{\mathbf{q}}$.

Kinetic Energy The energy stored in the motion of a mechanical system is called *kinetic energy*. By definition, the kinetic energy is

$$\text{K.E.} = \frac{1}{2}\dot{\mathbf{q}}^T M \dot{\mathbf{q}} \tag{1.1}$$

$$= \frac{1}{2}\mathbf{p}^T \dot{\mathbf{q}} . \tag{1.2}$$

Observe how the product of the momentum and the velocity gives twice the kinetic energy. Whenever the product of two related quantities gives twice the kinetic energy, we say that the two quantities are *dual* under the kinetic energy. Thus, $\mathbf{p}$ and $\dot{\mathbf{q}}$ are dual quantities.

Energy is a *scalar* quantity. In Computer Science, we often think of scalars as any real-valued quantity. In physics, however, *scalars* have a more special role: they are quantities that *do not depend on the choice of coordinate system.* Thus, the kinetic energy of a mechanical system is a real number that does not depend on the choice of coordinates used to represent the configuration space. We elaborate on this while exploring matrices further.

Off-diagonals in mass matrix In some cases, the mass matrix is not diagonal. Let us explore the two elevator system further. Suppose we select a new choice of coordinates to describe the same configuration space. Let $q_{\text{sum}} = \frac{1}{\sqrt{2}}(q_1 + q_2)$, and $q_{\text{dif}} = \frac{1}{\sqrt{2}}(q_2 - q_1)$, i.e., we capture the positions of the two elevators in terms of their (scaled) sum and difference. In matrix form, this is

$$\underbrace{\begin{pmatrix} q_{\text{sum}} \\ q_{\text{dif}} \end{pmatrix}}_{\tilde{\mathbf{q}}} = \underbrace{\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}}_{R} \underbrace{\begin{pmatrix} q_1 \\ q_2 \end{pmatrix}}_{\mathbf{q}} ,$$

where $\mathbf{q}$ and $\tilde{\mathbf{q}}$ are the coordinates of the configuration expressed equivalently in the old and new coordinate systems. Since this system is invertible, we can go back to the original coordinate system via

$$\underbrace{\begin{pmatrix} q_1 \\ q_2 \end{pmatrix}}_{\mathbf{q}} = \underbrace{\begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}}_{R^{-1}} \underbrace{\begin{pmatrix} q_{\text{sum}} \\ q_{\text{dif}} \end{pmatrix}}_{\tilde{\mathbf{q}}} \tag{1.3}$$

By definition, the momentum in the new coordinate system is proportional to the velocity, $\tilde{\mathbf{p}} = \tilde{M}\dot{\tilde{\mathbf{q}}}$. What are the entries of the matrix $\tilde{M}$? To answer this question, we make use of a fundamental principle:

the kinetic energy of a system depends on its state, but not on the choice of coordinate system. If $(\mathbf{p}, \dot{\mathbf{q}})$ and $(\tilde{\mathbf{p}}, \dot{\tilde{\mathbf{q}}})$ represent the same state in two different coordinate systems, then the two representations must compute the same kinetic energy K.E. $= \mathbf{p}^T \dot{\mathbf{q}} = \tilde{\mathbf{p}}^T \dot{\tilde{\mathbf{q}}}$. Expanding using $\mathbf{p} = M\dot{\mathbf{q}}$

$$\frac{1}{2}\dot{\mathbf{q}}^T M \dot{\mathbf{q}} = \frac{1}{2}\dot{\tilde{\mathbf{q}}}^T \tilde{M} \dot{\tilde{\mathbf{q}}}$$

and substituting (1.4) gives

$$\frac{1}{2}\dot{\tilde{\mathbf{q}}}^T \underbrace{R^{-T} M R^{-1}}_{\tilde{M}} \dot{\tilde{\mathbf{q}}} = \frac{1}{2}\dot{\tilde{\mathbf{q}}}^T \tilde{M} \dot{\tilde{\mathbf{q}}}$$

from which we obtain

$$\tilde{M} = \frac{1}{2}\begin{pmatrix} m_1 + m_2 & m_2 - m_1 \\ m_2 - m_1 & m_1 + m_2 \end{pmatrix} . \tag{1.4}$$

Thus, in the new coordinates, the mass matrix is no longer diagonal (in fact, it is dense).

The mass matrix is always a symmetric matrix; the deepest reason for this is that it is a *metric*, but I think that a more direct intuitive explanation is possible as well: *Think for yourself:* Can you examine the definition of kinetic energy and convince yourself that allowing the mass matrix to be assymetric does not "buy" any additional flexibility in defining the physical system?

1.2 Newtonian mechanics

Newton's laws Newtonian mechanics is based on Newton's three laws, which can be summarized as follows:

1. A body persists in its state of motion (at rest or moving at constant velocity) unless acted upon by an outside force.
2. The net force on a body equals the time rate of change of its momentum: $\mathbf{F} = \dot{\mathbf{p}}$. For a system with constant mass, this simplifies to $\mathbf{F} = \mathrm{d}(M\mathbf{v})/\mathrm{d}t = M\dot{\mathbf{v}} = M\mathbf{a}$.
3. When body A exerts a force on body B, body B exerts an equal (in magnitude) and opposite (in direction) force on body A.

Think for yourself: Consider three objects resting on the floor in a vertical stack. There is a constant force $\mathbf{f}_g$ on each pointing downwards due to gravity. Using Newton's laws, work out the additional forces acting on each object.

Equations of motion Assuming that we can determine the net forces acting on an object, Newton's second law allows us to determine the trajectory of that object given some initial state. The state of a particle is given by its position, $\mathbf{q}$, and momentum, $\mathbf{p}$, so that Newton's second law can be written as

$$\begin{pmatrix} \dot{\mathbf{q}}(t) \\ \dot{\mathbf{p}}(t) \end{pmatrix} = \begin{pmatrix} M^{-1}\mathbf{p}(t) \\ \mathbf{f}\left(\mathbf{q}(t), \mathbf{p}(t), t\right) \end{pmatrix} \tag{1.5}$$

This presents the equation of motion determining the particle's trajectory as two coupled first-order ordinary differential equations (ODEs). Here, we have explicitly allowed the forces to depend on time t. In certain cases, we may restrict our attention to forces that only depend on position and velocity, but not explicitly on time.

Using the relation $\mathbf{p} = M\dot{\mathbf{q}} = M\mathbf{v}$, we can rewrite the equations of motion in terms of position and velocity:

$$\begin{pmatrix} \dot{\mathbf{q}}(t) \\ M\dot{\mathbf{v}}(t) \end{pmatrix} = \begin{pmatrix} \mathbf{v}(t) \\ \mathbf{f}\left(\mathbf{q}(t), \dot{\mathbf{q}}(t), t\right) \end{pmatrix} . \tag{1.6}$$

Our first goal in physical simulation is to formulate forces that describe the physical system we want to simulate, and then to solve (1.5) numerically to obtain the trajectory of the system through time.

Chapter 2

Assignment

2.1 Policies

2.1.1 Academic Honesty Policy

You are permitted and encouraged to discuss your work with other students. You may work out equations in writing on paper or a whiteboard. You are encouraged to use the discussion board to converse with other students, the TAs, and the instructor.

HOWEVER, you may NOT share source code or hardcopies of source code. Refrain from activities or the sharing materials that could cause your source code to APPEAR TO BE similar to another student's enrolled in this or previous years. We will be monitoring source submissions for individuality. Cheating will be dealt with severely. Source code should be yours and yours only. Do not cheat. For more details, please refer to the full academic honesty policy provided with each assignment.

2.1.2 Grading

Each milestone will be tested with a fixed number of example problems. Each of these example problems is graded on a pass/fail basis, and your final grade for the milestone is the percentage of problems that passed. You will have access to half of the grading problems before the milestone is due, as well as the grading program itself.

2.2 Code and Grading Infrastructure

2.2.1 Obtaining and Building the Starter Code

New C++ starter code will be available on Codio for each assignment. This code has been designed and tested in the provided Codio box, a Unix environment with the GCC toolchain, and your final submission MUST compile and run on Codio. To obtain and build the starter code:

1. Follow the assignment Codio link that will take you to the appropriate Codio project
2. Inside the Codio box, open a new terminal
3. In the workspace directory, create a new directory called "build" (you can use a different name of course)
4. Change into the build directory, and execute the command below, which examines your system configuration and generates a makefile:

   ```
   cmake ..
   ```

5. [Optional] You can configure the build system by executing the command below:

   ```
   ccmake ..
   ```

   In the configure menu, you will have the option of setting the build mode. During development, you might find it useful to test your program with the build mode set to Debug. During grading, your program will be tested in Release mode, so please ensure your code runs in Release mode before the final submission. After setting options, press c to save the configuration, and g to update the build system and exit.

6. Build the program by executing

   ```
   make
   ```

7. The binary executable is now in the build/FOSSSim directory. Change into this directory, and execute the command

   ```
   ./FOSSSim -s assets/t1m1/StarterTests/helloworld.xml
   ```

   You should see a window that displays Hi that is made of circles and rectangles. If you run ./FOSSSim without arguments, youll see a brief description of the command line arguments.

8. You can add additional source files to your program by placing them in the FOSSSim directory under the project directory. You will have to regenerate the build system whenever you add new source files (by running cmake again)

2.2.2 Test Scene Oracle

To aid you during development, you will have access to a *oracle* implementation of this theme (of course, you don't see the source code, only the binary). This *oracle* is the same program that your submission will be graded with. As you will have access to half of the test examples, you will be privy to exactly half of your grade on the milestone before the due date. The other examples will remain hidden, but will be similar in nature to those provided. The additional examples may exploit combinations of forces or parameters not covered by the set you have access to. The intent of keeping these hidden is to encourage you to thoroughly test your code on your own example problems.

The *oracle* program can load the output of your simulation and visually highlight areas in which your simulation is incorrect. The oracle will also tell you if the particular simulation is judged a success. In addition to the example problems, the oracle will function with any scene of your design that adheres to the standard (see the XML File Format section).

2.2.3 Benchmarking Simulations Against the Oracle

1. Run your program with binary output enabled. For example, execute:

   ```
   ./FOSSSim -s scene_file.xml -d 0 -o binary_output.bin
   ```

   where binary output.bin is where you want the simulation result data to be stored. This file will then be read by the oracle to judge the correctness of your simulation.

2. Run the oracle program with the same scene file in input mode. For example, execute:

   ```
   /workspace/oracle/FOSSSimOracleT1M1 -s scene_file.xml -d 0 -i binary_output.bin
   ```

3. After executing the scene, the oracle will print the total position residual, the total velocity residual, the maximum position residual, and the maximum velocity residual. This output will indicate whether the detected residuals are acceptable.

4. You can run the oracle with OpenGL display by removing the option -d 0.

2.2.4 Grading

To receive the maximum grade on this milestone, you must:

1. submit your code
2. submit your creative submission

2.3 Milestone Specification

XML File Format

All scenes in this program are specified using an xml file. The features supported in the first milestone are defined below. XML parsing code has been provided for you.

1. The root node of the file is the scene node:

   ```
   <scene>
      ... scene contents ...
   </scene>
   ```

2. The duration of the simulation is specified with the duration node:

   ```
   <duration time="10.0"/>
   ```

   The time attribute is a scalar that specifies how long the scene should execute in 'simulation seconds.'

3. The integrator attribute specifies both the integrator to solve the system with as well as a timestep:

   ```
   <integrator type="explicit-euler" dt="0.01"/>
   ```

   For the first milestone, the integrator type will always be explicit-euler. The scalar attribute dt specifies the timestep to use with the given integrator.

4. The particle node adds a particle to the system:

   ```
   <particle m="1.0" px="1.0" py="7.5" vx="0.2" vy="-0.3" fixed="0" radius="0.04"/>
   ```

   The scalar m attribute specifies the mass of the particle. The scalar px and py attributes specify the initial position of the particle. The scalar vx and vy attributes specify the initial velocity of the particle. The boolean fixed attribute specifies whether this particle is fixed (not-simulated) or free (simulated). The optional scalar radius attribute specifies the particle's radius; if radius is not specified, a default value is used.

5. The edge node creates an edge between two particles:

   ```
   <edge i="1" j="2" radius="0.01"/>
   ```

   The integer i and j attributes specify the particles that compose the edge. The optional scalar radius attribute specifies a radius for the edge.

6. The simplegravity node defines a constant gravitational force:

   ```
   <simplegravity fx="0.0" fy="-9.81"/>
   ```

The scalar fx and fy attributes define the x and y components of gravity, respectively.

7. The maxsimfreq node defines a maximum frequency at which to step the system in interactive mode. This allows you to see simulations that would otherwise run too quickly.

   ```
   <maxsimfreq max="500.0"/>
   ```

   The scalar attribute max defines the maximum simulation frequency.

8. The particlecolor node changes a particle's color.

   ```
   <particlecolor i="2" r="0.1" g="0.2" b="0.3"/>
   ```

   The integer attribute i identifies the particle. The scalar r, g, and b attributes set the particle's color. The r, g, and b attributes must have values between 0.0 and 1.0.

9. The edgecolor node changes an edge's color.

   ```
   <edgecolor i="4" r="0.2" g="0.3" b="0.4"/>
   ```

   The integer attribute i identifies the edge. The scalar r, g, and b attributes set the edge's color. The r, g, and b attributes must have values between 0.0 and 1.0.

10. The particlepath node causes a particle to trace out a colored path during simulation.

    ```
    <particlepath i="6" duration="10.0" r="1.0" g="0.9" b="0.8"/>
    ```

    The integer attribute i identifies the particle. The duration attribute specifies how many simulation seconds each point on the path lasts. The scalar r, g, and b attributes set the path's color. The r, g, and b attributes must have values between 0.0 and 1.0.

Future milestones and themes will define additional features.

Required Features for Milestone I

2.3.1 Explicit Euler

We can discretize Newton's second law using explicit Euler, giving:

$$\mathbf{q}^{n+1} = \mathbf{q}^n + h\dot{\mathbf{q}}^n$$
$$\dot{\mathbf{q}}^{n+1} = \dot{\mathbf{q}}^n + hM^{-1}\mathbf{F}(\mathbf{q}^n, \dot{\mathbf{q}}^n)$$

Observe that both the position and the velocity update depend only on the position and velocity at the previous timestep. Edit the provided source file *ExplicitEuler.cpp* to compute the updated position and velocity using explicit Euler.

The kinetic energy of a particle is given by $T = \frac{1}{2}m\mathbf{v}^2$. Here a boldface font denotes a vector quantity. Recall that the dot product between two vectors $\mathbf{a}$ and $\mathbf{b}$ can be computed as $\mathbf{a} \cdot \mathbf{b} = \mathbf{a}_x\mathbf{b}_x + \mathbf{a}_y\mathbf{b}_y$ (here $\mathbf{a}_x$ denotes the x component of the vector $\mathbf{a}$). Furthermore, the shorthand $\mathbf{v}^2 = \mathbf{v} \cdot \mathbf{v}$ is employed. Edit the provided source file *TwoDScene.cpp* to compute the kinetic energy of the system. Newton's first law implies that in the absence of external forces, the kinetic energy will be constant. As a simple 'sanity' check of your integrator, execute the scene *InertiaTests/test01explicit.xml* and print the kinetic energy at each timestep to verify that it is indeed constant and the expected value. Write code to save the kinetic energy (in the stepper, get the kinetic energy from TwoDScene::computeKineticEnergy() and accumulate dt to track the simulation time) to a text file with the format:

```
# Time   KineticEnergy
time0     energy0
time1     energy1
 ...        ...
```

Use your favorite program or tool to verify that the kinetic energy, when plotted, is constant. These plots will not be graded, but you should get in the habit of debugging your programs both textually and graphically – it will pay off.

2.3.2 Constant Gravity

Recall from introductory physics that, sufficiently close to earth's surface, we can approximate gravity's effect as a constant acceleration $\mathbf{g}$ on all objects. Placing the 0 potential reference at the origin, this force corresponds to a potential energy of $U(\mathbf{x}) = -m\mathbf{g} \cdot \mathbf{x}$. Taking the gradient of this potential, the force is given by $\mathbf{F} = -\nabla U = m\mathbf{g}$.

Edit the provided source file *SimpleGravityForce.cpp* to compute this potential energy and its gradient.

2.3.3 Fixed Degrees of Freedom

During a simulation, it is often useful to *fix* or *kinematically script* a degree of freedom. Later in the course we will discuss methods for enforcing constraints in your simulations, but for now a simple solution is to just set the force for that degree of freedom to 0. Add this functionality to your explicit Euler implementation.

2.3.4 Creative Scene

As part of your final submission for this milestone, please include a scene of your design that best shows off your program. Based on the quality of your scene, you will have the opportunity to earn up to 15% extra credit. Your scene will be judged by a secret of committee of top scientists using the highly refined criteria of:

1. How well the scene shows off this milestone's 'magic ingredients' (a la *Iron Chef*).
2. Aesthetic considerations. The more beautiful, the better.
3. Originality.

You must generate the video using your code, and upload it to EdX for grading. You will also be asked to review other peoples videos there.

In order to generate a movie using your code, we provide a functionality to both generate pictures and make calls to stitch them into movies directly from your Codio box.

2.3.5 Making Movies

Please remember to generate this scene before submitting, as the Codio box will become read-only after you submit. The FOSSSim starter code comes with a PNG outputting utility that can help you create movies from your simulation. To enable it, create a folder named "pngs" in the directory that you are calling your executable from (i.e. your current directory) and run your code with the

```
-g 1
```

flag enabled. This will prompt your program to automatically save a PNG file for each frame of the simulation. This won't work if the "pngs" folder does not exist so you need to create it first before you run the program.

After the simulation finishes, you'll find all the frames in the pngs folder. Then you can make videos out of them with command-line tools such as mencoder and ffmpeg. Both mencoder and ffmpeg are easy-to-use tools available on the Codio boxes. You can execute this command:

```
mencoder mf://pngs/*.png -mf fps=24 -ovc lavc -lavcopts vcodec=msmpeg4v2 -oac copy -o output.avi
```

followed by:

```
ffmpeg -i output.avi -vcodec libx264 -crf 25 output.mp4
```

in order to create an mp4 video you may upload for the Creative portion of the assignment.

Note, ffmpeg can be used to create a video from the png images in one step, although sometimes this will give an error depending on the count and dimensions of input files. Nevertheless, the following command can take pngs and convert them to an mp4 movie in one:

```
ffmpeg -r 24 -f image2 -i ./pngs/frame%05d.png -vcodec libx264 -crf 25 -pix_fmt yuv420p test.mp4
```

Lastly, you may preview your movie by running the following command and looking at the virtual desktop:

```
mplayer <movie_name>
```

Please submit the mp4 file to the Peer Review Assignment portion of this week. An explanation for arguments to both mencoder and ffmpeg can be found online.

2.4 Theme 1 Milestone 1 Notes and FAQ:

Theme 1 is fairly easy and does a great job walking you through the code, here is an assortment of random supplemental tips for the theme and for the year:

Q: What is the Oracle, I don't understand.
A: The Oracle can be called by running `/workspace/oracle/FOSSSimOracle` in place of the usual `./FOSSSim` followed by the usual arguments. It is a great tool which is useful for showing you what a correct simulation looks like. It is also useful for running creative scenes if you have not yet finished implementing the physics for the current assignment. Later on in the course it will be augmented and will give you more feedback on the location and time occurrence of errors in your program.

Q: I have more questions, how should I get help?
A: Please post all questions to the discussion board so that everyone can benefit from them.

Q: Hey I have a problem in the scene with the particles dropping down with the spring and the...?
A: Please refer to scenes by their name so we can best assist you, for example,
`InertiaTests/test01explicit.xml`

Q: I'm confused, how does the code keep track of everything thats going on?
A: The code works on a frame by frame basis. Every single frame you will take the existing values and parameters, operate on them, change their values in accordance to algorithms/physics you implement, and then update the values in memory. The code will then pass that along its framework and other calls, update what you see on screen, and then repeat. Essentially all you ever worry about it the system state, meaning the positions, velocities, and masses of particles at present, and how to update them one step over. Whenever you are computing anything try to think about it at a simple Time 0 to Time 1 case, taking one step, and using only the current state to determine the next state. We will discuss more complicated things later but if you can understand that idea you will go a long way.
This is a critical way to think about things. If you throw a ball up in the air, you don't need to know the entire history of the ball since it was made in order to determine how it will behave. All you need is the current state, (X, V, M) and the forces acting on it, to determine how physics will move it in time and space. Additionally, try to think of things on a particle by particle interaction (or edge-edge, particle-edge), the code is set up to iterate through different possible combinations and take care of everything for you so you

simply have to determine how one particle or edge gets effected by other particles or edges, and never how a cluster of objects interacts with another cluster.

Q: What sort of control do I have when running a simulation?
A: As with any other program, print statements will still go to the console if you wish to use that for debugging. Furthermore, familiarize yourself with the keyboard instructions available to you from the main.cpp `keyboard` function.

Q: My code runs differently every time I try it, what is this magic?
A: Everything you do should be deterministic and physically based, there is no randomness introduced from the given framework. A common mistake is to not zero out vectors when you create them, or other variables, which then take on garbage values left in memory. Use the command `vec.setZero()` to zero out a vector you want to be zero or fresh when running a simulation. Please refer to the online Eigen Library documentation for help working with vectors and Linear Algebra functions

Q: What is a VectorXs?
A: A VectorXs is an alias for a type of vector we will use throughout the semester. A VectorXs refers to a Vector of size X (variable size) and of type s (meaning Scalar). It is not your STL vector, or standard vector, which you may be used to, but is instead derived from the Eigen library which we utilize. Eigen is a very popular Linear Algebra API and we use it for just that purpose. We use the Eigen vector and not the STL one because it comes with linear algebra and math operations, such as dot product, cross, normal, matrix multiplication, addition, etc, built right in, so you don't have to implement those. If you are ever confused about how to access/use one of these, refer to other areas in the code or to the Eigen online documentation for help.

Q: Why are the vectors of size $2N$?
A: The Vectors you interact with will often contain all the positions or masses or velocities of a system. This allows people with stronger linear algebra backgrounds to operate on them as matrices or vectors rather than iterating through them. It is important to remember however that they are simply, and consistently, in the format: $q[x_0, y_0, x_1, y_1, x_2, y_2, ...]$, $q_{\text{dot}}[x_0, y_0, x_1, y_1, x_2, y_2, ...]$, and $m[m_0, m_0, m_1, m_1, m_2, m_2...]$. The masses are done in this way so their size matches that of the other vectors which is sometimes necessary for operations.

Q: What does it mean to be a fixed particle?
A: A fixed particle is a particle that is fixed, or frozen, to the physics. It is a term used in Graphics and Animation to denote something that is scripted, or hardcoded to a value, but does not interact with the system. This means they may have velocities initially placed on them but forces on them will always be zero.

Q: What is the deal with these Creative scenes, are they for bonus points?
A: No. Creative scenes are necessary and part of your submissions every week. If your submission falls within the best of the class it may even receive bonus points.

Q: What are code stubs?
A: Code stubs are comments in the code or methods which are not yet implemented and left for you to fill in, these will often be indicated from the assignment writeup or by comments indicating 'your code goes here'.

Q: Any final tips before I get started?
A: Gradient of U is NEGATIVE force! Good luck!