Simulation Representation

In this time dependent ODE Simulation framework, a simulation is based on a template class called the `Simulator` class. This class uses the **Curiously Recurring Template Pattern** (CRTP) to allow for customization of certain framework methods, while removing the runtime overhead of virtual polymorphism.

An example simulation based on the `Simulator` framework might be the following:

```cpp
#include "Simulator.hpp"
#include "Pendulum.hpp"

class PendulumSim : public Simulator<PendulumSim,RungeKutta4> {
public:

    PendulumSim(){
        std::string historyFile("history.txt");
        setSimHistoryPath(historyFile);
        state.printFrequency = 30;
        numMC = 10;
        writeSimHistory = true;
    }


    // These are all expected CRTP methods
    void _linkModelsToSim( SimState & state );
    void _connectModelsTogether();
    bool _finishedSimulation( SimState & state ) const;
    void _finalizeMonteCarloRun();
    void _finalize();

private:

    TimeStep tstep;
    PendulumModel pendulum;

};
```

As one can see, the following methods need to be defined in a new class based on the `Simulator` class:

- `void _linkModelsToSim( SimState & state )`
    - Method meant to connect Discrete or Dynamic models to the simulation back-end
- `void _connectModelsTogether()`
    - Method is meant to connect models together in the event they need each other
- `bool _finishedSimulation( SimState & state ) const`
    - Method is meant to return whether a simulation has completed or not
- `void _finalizeMonteCarloRun()`

    - Method allows user to do something after a given Monte Carlo run
- `void _finalize()`
    - Method allows user to do something after the simulation completes

Models

Dynamic Models

When going about doing a simulation, a common model is one based on a dynamical system of equations described by differential equations. These models are the ones that require numerical integration in time with approaches like Euler or Runge-Kutta integration.

Discrete Models

These models represent discrete events in time. This can be, for example, a discrete GPS update for a missile, a discrete time step at which to force dynamic models to integrate, and even a discrete event to write data to a file at a certain time interval. These models have their own unique importance in most simulation environments and can be exploited in this framework.

Time

Time Representation

Within the framework, time has been designed to be kept and incremented in the most precise fashion possible. It is common knowledge that working with floating point numbers will result in round off errors, especially when one floating point number is much larger than the other. In the simulation, since it is likely we may be doing operations with large times against small time steps, round off errors are sure to occur.

To ensure a precise time step is taken, time will be kept in terms of a *Mixed Fraction*. Then when precise, floating point values for time are needed, the Mixed Fraction will be converted to a floating point value for use in computation.

Event Tracking in Simulation

Currently in the framework, the items that are required to be tracked in time are all discrete time events. At each discrete event, any and all `DynamicModel`s are updated via numerical time integration. Thus, it is important to efficiently and accurately keep track of these discrete time events.

This framework automatically handles the discrete event handling via a class called a Scheduler, which is essentially a Minimum Heap data structure popping the next time event everytime a simulation time step needs to be taken.

For a user of the framework, however, you just need to add the discrete models to the simulation class, and it will do the hard work for you.

Monte Carlo Simulation Capability

Random Number Generation

This framework is built to have a centralized random number generator within the `Simulator` class structure. And, in result, each `DynamicModel` and `DiscreteModel` maintains a reference to the centralized Random Number Generator.

Monte Carlo Runs

This framework also has a simple layer built into it to allow for multiple runs of the same simulation, allowing a simulation to utilize the variation created from the random number generator to generate a Monte Carlo set of varying simulation results based on different initial seeds for each Monte Carlo simulation.

While the default simulation structure sets the number of Monte Carlo runs to 1, since not all simulations need to be run in a Monte Carlo fashion, a user is able to modify this variable in their **Simulator** class.

Simulation Data Output

Simulation History Data Output

When one generates a simulation, it may be desireable to capture all the time history for a set of variables through he lifetime of each simulation, even when doing multiple Monte Carlo simulation runs. Fortunately, the framework has a solution built for this problem.

A given model, whether it is a `DynamicModel` or `DiscreteModel`, has a virtual method that can be defined when subclassing your own model known as `setupPrintData()`. This method is used by the simulation's backbone simulation loop to connect up data a user wishes to have outputted into a sim history data file. A simple example of this method might be the following for a `DynamicModel`:

```
virtual void setupPrintData() {
  simState->dataPrinter.addVariableToPrint(&state[0], "X-Position"); //save particle position
  simState->dataPrinter.addVariableToPrint(&state[1], "X-Velocity"); //save particle velocity
}
```

The `simState` variable is already a part of each model, and the `addVariableToPrint` method, defined in the `dataPrinter` object, is the method to use when adding a new variable to the sim history output file. Once you define this virtual method for a given model, you will obtain the resulting output file with the data that you desire!

End Game Simulation Output

There is no predefined code to allow for you to just save off data or print information to the terminal at the end of a simulation run, but pieces are in place that can help a user achieve these functions.

There are two methods in a `Simulator` subclass that you might want to edit. The first method is the `_finalizeMonteCarloRun()` method. By using this method, you are specifying actions to take place after a given Monte Carlo run has completed. An example of something you might do is print a simple status update to the terminal or save off some info to a file.

The second virtual method is the `_finalize()` method. This method is run after all of the simulations, Monte Carlo sims included, has completed. Often, there may be no need to overwrite this method. But in the event there is some computation that a user wants to be run just before the sim finishes, this method is the place to do it.

Examples of these two methods might be the following:

```
void _finalizeMonteCarloRun(){
  printf("Finished the %ith Monte Carlo run!\n",static_cast<int>(getCompletedMC()));
}

void _finalize(){
  printf("Finished the Simulation!\n");
}
```

Defining the Completion of the Simulation

Not all simulations are created equal and often simulations have different completion criteria. Some simulations use simple completion criteria, such as stopping the simulation when a time limit is hit. Others, such as simulating a missile flight or robot maneuver, might have the completion of a simulation depend on where the objects being simulated end up spacially.

Due to these potentially very different completion criterias, the framework has defined a method that can be used to define what will make a given simulation be done. The method to overwrite in the `Simulator` subclass is called `_finishedSimulation()` and it returns a boolean. `true` means the simulation is complete, `false` means it isn't.

A simple example of this method might be the following:

```
bool _finishedSimulation() const {
  return getTime() >= 10.0;
}
```

Adding Models to a Simulator

Now the approach to adding a given model to a subclassed `Simulator` is practically the same for both `DynamicModel` and `DiscreteModel`. The general idea is that this work will need to be done in the `_linkModelsToSim()` method, which is a part of the `Simulator` subclass. To help you see how it might look, here is an example:

```
void _linkModelsToSim(){
  addDiscrete(&discreteModel, evalsPerSecond);
  addDynamics(&dynamicModel);
}
```

This piece of code shows how simple it is to add models to the `Simulator`. One just needs to use the built in `addDiscrete` and `addDynamics` methods and then everything will be put together as desired. One note is that the `addDiscrete`'s second input is the frequency, in Hz, a `Discrete` model is expected to be run in the simulation.

Getting Input from External Files

Add Configuration File to Simulation

At the moment, there is a simple system in place to setup configuration files that can be read in at runtime for a given program. All one needs to do is provide the path to some configuration file and call the `addConfigFile` method of the `Simulator` class. An example of this might be the following:

```
Simulator sim;
sim.addConfigFile("someFile.cfg");
sim.addConfigFile("../heresAnotherFile.cfg");
```

The `Simulator` then ensures the necessary variables are parsed and brought into a hash table for quick look up. Currently, the config files are expected to only have numeric inputs, though this will be changed down the road. The following is an example of a working config file:

```
# Here's the start of the Config File
# Config file #1
variable1: 12.3
variable2: 4.0     # comments after a variable is okay
variable3: 9.27    # make sure to use a : after the variable name

# Here's the last variable
willDance: 1.0
willEat:   0.0

# Here's some vectors
vec1: [1, 2, 3, 4] #equates to [1, 2, 3, 4]
vec2: [1:5, 6, 8 ] #equates to [1, 2, 3, 4, 5, 6, 8]
vec3: [12:10, 5,2] #equates to [12, 11, 10, 5, 2]
vec4: [0:1.1:5.5,7]#equates to [0, 1.1, 2.2, 3.3, 4.4, 5.5, 7]
vec5: [3.3:-1.1:0] #equates to [3.3, 2.2, 1.1, 0]

# Here's some words
name: Christian
filepath: /some/path/here/file.txt
```

By writing a config file with the appropriate syntax, the `Simulator`'s parser will take care of getting the variables parsed.

Retrieve Configuration Variable Value

To retrieve the value of a given variable, one just needs to use the `Parser` object in the `SimState` stored in the `Simulator` class. Since a given model stores a pointer to this `SimState`, an example of how to grab some variable's data can be seen below:

```cpp
typedef std::vector<double> Vector;
typedef std::String String;

class SomeModel : public DynamicModel {
public:

virtual void operator()( double time , ModelState & dqdt ){
  double variable1  = simState->get<double>("variable1");
  Vector vec        = simState->get<Vector>("vec4");
}

}
```

Since a given model, be it `DynamicModel` or `DiscreteModel`, has a reference to its parent `Simulator`'s `SimState`, any custom model's can easily call the `get` method to use input variables for the models! It is recommended you add config files to the sim before you call the sim's `initialize()` method to ensure the data exists before any models need to grab it for use.

Examples

Examples can be found in the directory **ODE_SimFramework/Examples**. The first basic example one will find is for a pendulum with the following dynamics:

$$m\ddot{\theta} + c\dot{\theta} + \frac{mg}{l}\sin(\theta) = 0$$

This can be represented in state space form as the following:

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = -\left(\frac{c}{m}x_2 + \frac{g}{l}\sin(x_1)\right)$$

where $[x_1, x_2] = [\theta, \dot{\theta}]$ and where we have uncertainty about the exact values of m, c, and l.

The dynamics can be written as the following:

```cpp
/*!
 * These dynamics represent a simple 1D nonlinear Pendulum model
 */
```

```cpp
class PendulumModel : public DynamicModel {

public:

    // Constructor
    PendulumModel(){
        model_name = "pendulum";
    }

    // initializer for a given Monte Carlo sim
    virtual void initialize(){
        mass            = generator->rand() * 50 + 50;
        dampening       = generator->rand() * 15;
        lengthPendulum  = generator->rand() * 10;
        gravity         = 9.81;
        state[0]        = Constants::pi/3;  // initial condition for theta
        state[1]        = Constants::pi/12; // initial condition for theta-dot
    }

    // data to be printed to file
    virtual void setupPrintData(){
        simState->dataPrinter.addVariableToPrint(&state[0], "Theta");
        simState->dataPrinter.addVariableToPrint(&state[1], "ThetaDot");
    }

    // number of dimensions being integrated
    virtual int numDims() const { return 2; }

    // dynamics model
    virtual void operator()( double time , ModelState & dqdt ){
        double theta    = state[0];
        double thetaDot = state[1];
        dqdt[0] = thetaDot;
        dqdt[1] = -( dampening*thetaDot/mass + gravity * sin(theta)/ lengthPendulum);
    }


private:
    double mass;            // in kg
    double dampening;       // dampening factor
    double gravity;         // m/s^2
    double lengthPendulum;  // in meters


};
```

The corresponding simulation can then be, as an example, written to be the following:

```cpp
#include "Simulator.hpp"
#include "Pendulum.hpp"


class PendulumSim : public Simulator<PendulumSim,RungeKutta4> {
public:

    PendulumSim(){
        std::string historyFile("history.txt");
        setSimHistoryPath(historyFile);
        state.printFrequency = 30;
        numMC = 10;
        writeSimHistory = true;
    }


    void _linkModelsToSim( SimState & state ){
        addDiscrete(&tstep, 100);
        addDynamics(&pendulum);
    }
    void _connectModelsTogether(){

    }
    bool _finishedSimulation( SimState & state ) const{
        return getTime() > 5;
    }
    void _finalizeMonteCarloRun(){
        printf("Finished the %ith Monte Carlo run!\n",
                    static_cast<int>(getCompletedMC()));
    }
    void _finalize(){
        printf("Finished!\n");
    }

private:

    TimeStep tstep;
    PendulumModel pendulum;

};
```

This simulation can then be run by doing the following in your main function:

```cpp
#include "PendulumSim.hpp"

int main(int argc, const char * argv[]) {

    PendulumSim psim;
    psim.setSimHistoryPath("simhistory.txt");
    psim.runSim();

    return 0;
}
```