ADMM-Elastic Quick Start Guide

Matthew Overby, George E. Brown, and Rahul Narain

August 2016

1 Introduction

ADMM-Elastic is a parallel implicit solver for elastic deformation. This code was used for the paper "ADMM $\supseteq$ Projective Dynamics: Fast Simulation of General Constitutive Models" by Narain, Overby, and Brown [1].

2 Using the code

ADMM-Elastic uses CMake and has been tested on Linux and Mac. Dependencies are included in the `deps/` directory with the exception of OpenMP. The ADMM-Elastic repository does not contain any graphics or rendering code. For that, see ADMM-Elastic-Samples.

The `admm::System` class is the context of the solver, it houses the data and acts as the interface. A scene is described by a list of nodes/vertices, and the forces that act upon them. Setting up the solver proceeds with the following steps:

1. Create an instance of `System`
2. Add nodes and their initial positions
3. Add forces to act on the nodes
4. Initialize `System`
5. Take a time step

2.1 A Simple Spring

Below, we show an example of a spring connecting two nodes.

1. Creating an instance of `System` is simple enough:

```
admm::System system;
```

2. Nodes can be added to the system through the `System::add_nodes` function. You pass an `Eigen::VectorXd` of initial positions and masses. Note that these vectors are scaled by three (for the x, y, and z coordinates). So the first node's position are indices 0, 1, 2, while the second node's position are indices 3, 4, 5, and so on.

```
Eigen::VectorXd x(6), m(6);
m.fill(1); // set node mass to 1
x.segment(0,3)=Eigen::Vector3d(0,0,0); // 1st node position is (0,0,0)
x.segment(3,3)=Eigen::Vector3d(1,0,0); // 2nd node position is (1,0,0)
system.add_nodes( x, m );
```

3. There are two kinds of forces to differentiate between explicit and implicit integration. Simple forces (e.g., gravity and wind) derive from `admm::ExplicitForce` and elastic/collision/anchor forces derive from `admm::Force`. Internal forces like springs require the indices of the nodes they affect.

```
double stiffness = 10;
int idx0 = 0; // 1st node id
int idx1 = 1; // 2nd node id
std::shared_ptr<admm::Spring> sf(
    new admm::Spring( idx0, idx1, stiffness )
);
system.forces.push_back( sf );
```

4. Finally, we initialize the system. This creates the global matrices of the solver and should only be called once. The argument passed is the simulation time step (seconds). If something goes wrong, this function returns false. *Note: The rest lengths and shapes are set when* `System::initialize` *is called.*

```
double timestep_s = 0.04;
if( !system.initialize(timestep_s) ){ ... }
```

5. Now we can proceed with the simulation by calling `System::step`. The number of solver iterations to be used is passed as an argument. This function returns false on a failure.

```
bool simulating = true;
while( simulating ){
    if( !system.step( 20 ) ){ ... }
    ...
```

2.2 Other Forces

The code is extensible in that as long as a new force derives from the base class, it can be added to the system. To see all of the forces already implemented, take a look at the header files in `src/system/`. Some of the more complex forces are described here.

2.2.1 Collision Force

Collision detection and resolution can be simulated through *one* force with a given list of collidable shapes. Self-collisions are not handled at the time. For a list of shapes, see `src/collision/`. Only one force that affects all nodes and contains all collidable shapes is needed. An example of adding a sphere is shown below:

```
using namespace admm;

// Add all of your collidable shapes to one vector:
std::vector< std::shared_ptr<CollisionShape> > shapes;
Eigen::Vector3d center(0,0,0);
double radius = 1.0;
std::shared_ptr<CollisionShape> shape( new CollisionSphere(center,radius) );
shapes.push_back( shape );

// Create a collision force and add it to system:
std::shared_ptr<Force> force( new CollisionForce( shapes ) );
system.forces.push_back( force );
```

2.2.2 Moving Anchors

An anchor (or pin) is a constraint that holds a node in place. `MovingAnchors` are bound to a `ControlPoint`, which can be moved around and enabled/disabled to allow interaction such as mouse control. An example of a moving anchor that moves over two seconds is shown below:

```
using namespace admm;

// Create and add the moving anchor:
std::shared_ptr<ControlPoint> p( new ControlPoint() );
p->pos = Eigen::Vector3d(0,0,0);
int idx = 0; // node index
std::shared_ptr<MovingAnchor> f( new MovingAnchor( idx, p ) );
system->forces.push_back( f );

// Later on in the game loop...

while( simulating ){
    // ...
    double start_dt = 1.0; // start movement at 1 second
    double end_dt = 3.0; // end movement at 3 seconds
    Eigen::Vector3d startPos(0,0,0);
    Eigen::Vector3d endPos(1.0,1.0,1.0);
    p->pos = helper::smooth_move( elapsed_s, start_dt, end_dt, startPos,
        endPos );
    elapsed_s += timestep_s;
```

References

[1] Rahul Narain, Matthew Overby, and George E. Brown. ADMM ⊇ projective dynamics: Fast simulation of general constitutive models. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2016.