Artificial Muscle Simulator 1.2

Max Braun*

March 21, 2008

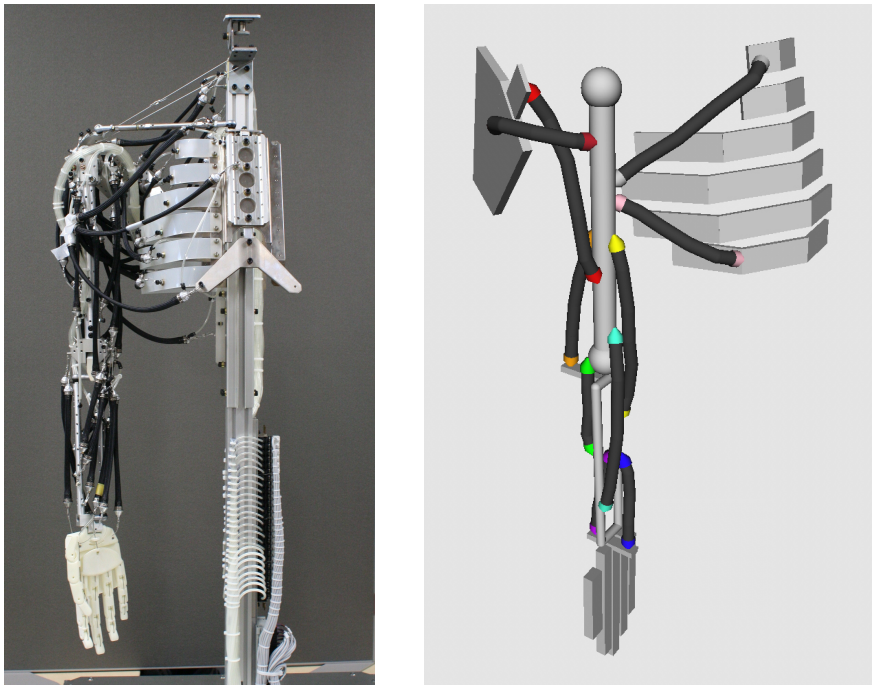
Contents

*max.braun@ams.eng.osaka-u.ac.jp

1 Introduction

The *Artificial Muscle Simulator* was originally designed and used to evaluate a novel control method for a robotic arm with 30 pneumatic actuators (or *artificial muscles*) and 6 metal bones imitating the anatomy of a human upper limb. Figure 1(a) shows the robot and figure 1(b) a still from the simulator. The control method called *attractor selection model* is described and evaluated in Nakamura et. al. [3].

The core of the simulator, however, is independent of the original robot design and can be altered to suit different tasks. This document explains how the *Artificial Muscle Simulator* can be used to simulate any robot that uses pneumatic actuators and can be modeled with the available primitives and joints.



(a) Human-like robotic arm (b) Simplified simulator model

Figure 1: The original robotic arm and its simulated counterpart

2 Architecture

The simulator is an abstraction of the rigid body dynamics simulation engine ODE [1]. It is entirely written in C++ and uses OpenGL/GLUT for 3D graphics and interaction. Possible user customization is also realized through object-oriented concepts on top of C++.

2.1 Files and classes

The main folder `musclesim 1.2` contains a `main.cpp`, the `Makefile`, and the binary `musclesim` and this `documentation.pdf`. There are four subfolders:

- `log` holds all recordings of the simulator intended for future analysis.
- `simulation` contains files for the classes `Simulation` and `Display`, which are used by the system.
- `robot` includes the classes `Primitive` and `Actuator` and their respective subclasses, which are used to define the structure of the simulated robot in a subclass of `Robot` as described in section 3.
- `controller` holds the class `Controller`, where the control method for the robot is implemented. This is explained in section 4.

3 Building a robot

To set up a new robot in the simulator, create a subclass of `Robot` such as the existing `RoboticArm`.[1] In its constructor, the structure of the robot is defined through *primitives* which are connected by *joints* and *actuators*.[2]

3.1 Primitives

There are three different types of primitive objects.[3] They all have in common a `position` in three-dimensional space, a `mass`, and a `density`. The `world` and `space` parameters are usually handled by the system.

Boxes are furthermore defined by three lengths `size_x`, `size_y`, and `size_z` along the respective coordinate axes.

```
dVector3 spine_position = {0, 0, 0};
Box* spine = new Box(
  world, space,
  spine_position,
  80, // mass
  1, // density
  50, // size in x
  1000, // size in y
  50 // size in z
);
```

[1] If you have several subclasses of `Robot`, make sure to define the one you want to use by returning the appropriate instance in `getActiveRobot()`.

[2] Most examples in this section are (possibly modified) excerpts of the constructor of `RoboticArm`, which is defined in `robot/robot.cpp`, and it is recommended to refer to this context.

[3] Their interfaces are defined in `robot/primitive.h`.

Spheres only need a `radius` in addition to the common parameters.

```
dVector3 shoulder_position = {-210, 410, 0};
Sphere* shoulder = new Sphere(
  world, space,
  shoulder_position,
  30, // mass
  1, // density
  20 // radius
);
```

Capped cylinders are defined with additional values for `radius` and `length`.

```
dVector3 humerus_position = {-210, 260, 0};
CappedCylinder* humerus = new CappedCylinder(
  world, space,
  humerus_position,
  200, // mass
  1, // density
  13, // radius
  300 // length
);
```

Each primitive has a default rotation, which can be adjusted by calling `rotate(dReal angle, dVector3 axis)`, possibly repeatedly.

```
dVector3 x_axis = {1, 0, 0};
dVector3 y_axis = {0, 1, 0};
humerus->rotate(90, x_axis); // angles in degree
humerus->rotate(-45, y_axis);
```

To fix a primitive's position and rotation to the environment, call `fix()`.

```
spine->fix();
```

The default color can be changed with `setColor(GLfloat red, GLfloat green, GLfloat blue)`.

```
humerus->setColor(1.0, 1.0, 0.0); // yellow
```

3.2 Joints

There are five joint types available: *ball*, *hinge* (1-axis and 2-axis), and *universal* (1-axis and 2-axis).[4] Connecting two primitives is realized by calling the corresponding methods on one primitive with the other as the first parameter[5] and an anchor point as the second.

Ball joints do not require any further information, *hinge* joints the definition of one axis, and *universal* joints of two axes.

```
dVector3 shoulder_position = {-210, 410, 0};
humerus->attachBall(NULL, shoulder_position);
```

```
dVector3 elbow_position = {-210, 110, 0};
dVector3 x_axis = {1, 0, 0};
ulna->attachHinge(elbow, elbow_position, x_axis);
```

```
dVector3 shoulder_position = {-210, 410, 0};
dVector3 x_axis = {1, 0, 0};
dVector3 z_axis = {0, 0, 1};
humerus->attachUniversal(
  NULL, shoulder_position, x_axis, z_axis
);
```

Both methods `attachHinge(...)` and `attachUniversal(...)` are overloaded to accept additional parameters for limiting the joint angle range of each axis.

```
ulna->attachHinge(
  elbow, elbow_position, x_axis,
  -90, // minimum angle
  0 // maximum angle
);
```

3.3 Composite primitives

If the robot exhibits shapes which differ strongly from the three simple primitives, one might want to combine a number of these to approximate more complex forms. Composited primitives have a fixed position and rotation relative to each other. Rotating or fixing a primitive after a composition will affect the whole composite primitive it belongs to.

The easiest way to create a composite of only two primitives is to call

[4] For detailed information about the different joint types, see the ODE User Guide: [2]

[5] Passing `NULL` will create a connection to the environment.

`compositeWithPrimitive(Primitive* child)` on one of them with the other as the parameter.

```
radius->compositeWithPrimitive(radius_head);
```

If there are more than two primitives involved in the composition, one must use the static `compositePrimitives(vector<Primitive*> primitives)`.

```
vector<Primitive*> humerus_parts;
humerus_parts.push_back(humerus);
humerus_parts.push_back(shoulder);
humerus_parts.push_back(elbow);
Primitive::compositePrimitives(humerus_parts);
```

Note that joints involving at least one composite primitive must be set up after the compositions.

3.4 Actuators

There are two types of actuators included in this simulator: `LinearActuator` and the flexible and eponymous `Muscle`, which we will focus on. They are both subclasses of `Actuator`, which can also be used to derive new types of actuators with user-defined behavior. [6] The behavior of an actuator is defined through its force function `calculateForce()`, which takes into account the current length of the actuator and its air pressure.

The path of a newly created artificial muscle is defined by a 3^{rd}-degree Bézier curve and thus through four control points in space. Furthermore, a minimum and maximum length of the muscle must be given. A custom color for the caps can be set optionally.

```
dVector3 muscle_start_at = {-223, 360, 0};
dVector3 muscle_start_control = {-293, 410, -40};
dVector3 muscle_end_control = {-280, 360, -100};
dVector3 muscle_end_at = {-230, 360, -120};
Muscle* muscle = new Muscle(
        world, space,
        muscle_start_at, muscle_start_control,
        muscle_end_control, muscle_end_at,
        75, // minimum length
        200 // maximum length
);
muscle->setColor(0.73, 0.09, 0.15); // brown
```

[6] See `robot/actuator.h` for details.

To connect the two ends of the artificial muscle to other parts of the robot, i.e. primitives, use the methods `connectStartTo(Primitive* primitive)` and `connectEndTo(Primitive* primitive)`.

```
muscle->connectStartTo(humerus);
muscle->connectEndTo(scapula);
```

4 Controlling the robot

A subclass of `Controller` is mainly responsible for automatic control of the robot.[7] Two controllers are included: `ASMController` implementing the attractor selection model and `AASMController` for the adaptive attractor selection model.[8]

The active subclass of `Robot` (currently `RoboticArm`) is responsible for providing the controller with the state of the robot and executing commands from the controller.

4.1 Simulation run loop

Robot, controller, and simulation interact in a simple run loop. First, the robot is queried for its state, then the controller generates an action based on its internal state and the state of the robot, and finally the robot executes the action. The physics simulation also advances one step in each iteration.

```
while (true) {
  double* state = robot->getState();
  double* action = controller->output(state);
  robot->step(action);
  Simulation::doStep();
}
```

So, to use a custom control method for the robot, implement the `output(double* state)` method of `Controller` as well as the `getState()` and `step(double* action)` methods of `Robot` accordingly.

In the current implementation, `state` points to 3 coordinates denoting the position of the robot's hand. The command `action` holds 10 values, each corresponding to the air pressure in one artificial muscle. But both can be used in any suitable way.

[7] If you have several subclasses of `Controller`, make sure to define the one you want to use by returning the appropriate instance in `getActiveController()`.

[8] The implementations can be found in `controller/asm/` and `controller/aasm/` respectively. The method is described in [3].

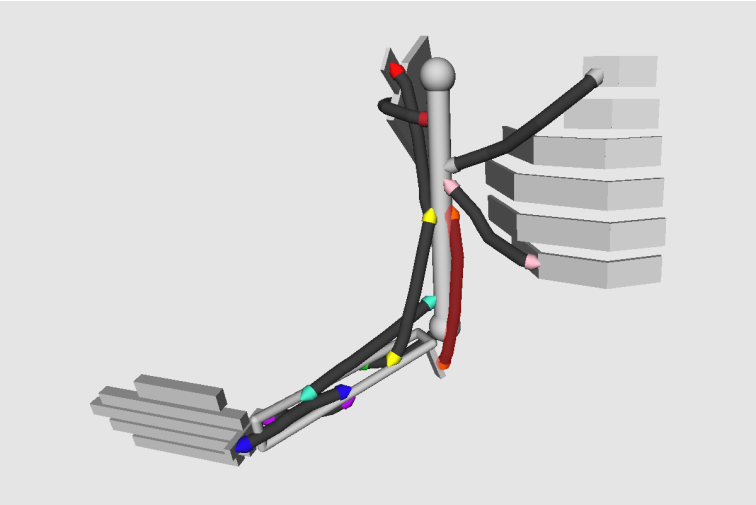


Figure 2: The simulator in interactive mode

5 User interface

Per default, the *Artificial Muscle Simulator* starts in a visualization mode while running the automatic simulation implemented in the controller.

```
$ ./musclesim
```

The user can then rotate around the robot and examine it. It is possible to switch to an interactive mode, which overrides the controller and where one can control each actuator manually. If the controller logs all important data and no graphical feedback is necessary, simple mode can disable all display and interaction entirely to increase the simulation speed.

5.1 Command line arguments

Adding the command line argument `--simple` causes the simulator to start without display or interaction, resulting in a faster simulation.

The default simulation step size of 0.01 seconds can be altered by appending `--step` followed by the desired step size. A higher step size will increase the simulator's speed, but also make it more unstable.

If the simulator starts with `--verbose`, detailed information about the used primitives and actuators will be printed to the standard output.

The command line arguments can be combined arbitrarily.

```
$ ./musclesim --simple --step 0.001 --verbose
```

5.2 Controls

The point of view in visualization mode can be rotated by moving the mouse while pressing a mouse button. The up and down arrow keys cause the camera to zoom in and out respectively.

The key `i` switches between automatic and interactive mode and `p` pauses or unpauses the whole simulation. In interactive mode, actuators are selected by pressing their index key from `0` to `9` or, alternatively, selecting the next or previous actuator with the keys `.` or `,` respectively. The selected actuator is highlighted in red as seen in the center of figure 2. The air pressure in the selected actuator can be increased with `+` and decreased with `-`.[9]. To quit the simulation, close the window in visualization mode or press `Ctrl-C` in simple mode.

6 Notes

6.1 Units

Due to ODE's problems with small numbers, the units are not SI. Instead, all lengths are in millimeters and masses in gram.

6.2 Performance

The ratio between real time and simulation time is roughly $\mathcal{O}(n^3)$, with n denoting the number of actuators used. This should be considered when simulating a robot with many actuators.

The data shown in figure 3 was acquired from a simulation with a step size of 0.01 seconds and without display or interaction. The extrapolated ratio function is $r(n) = 8.5 \cdot 10^{-3} \cdot n^{3.2}$ with $r(5) = 1.5$ and $r(10) = 13.5$, but $r(30) = 442$.

6.3 Force function

The force function implemented in the `Muscle` class was derived by interpolating data from experiments with the pneumatic actuators used in the actual robotic arm. A plot is shown in figure 4.

$$F(p,l) = c \cdot \left(\frac{p}{p_{\max}} \cdot \frac{l - l_{\min}}{l_{\max} - l_{\min}} + a \cdot \left(1 - \frac{p}{p_{\max}}\right) \cdot \exp\left(b \cdot \frac{l - l_{\min}}{l_{\max} - l_{\min}}\right)\right) \quad (1)$$

for air pressure p and length l of the muscle and the respective minimum and maximum values $l_{\min}$, $l_{\max}$, $p_{\min}$, $p_{\max}$ and the parameters

$$a = 0.00007687724845623249 \quad (2)$$

$$b = 9.203893778016578 \quad (3)$$

$$c = 9.799574209874333 \quad (4)$$

[9] The air pressure is normalized to the real interval $[0, 1]$.

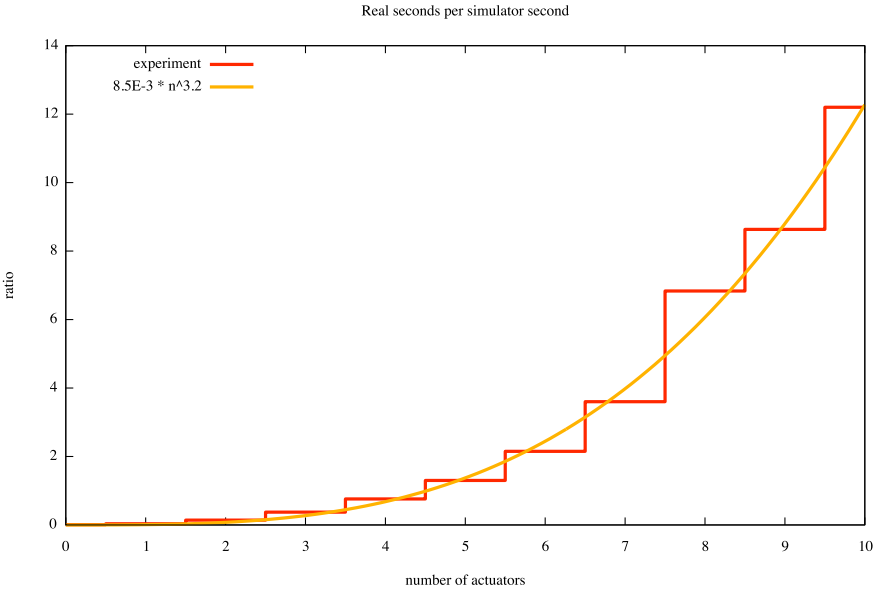


Figure 3: Simulation time ratio over the number of actuators

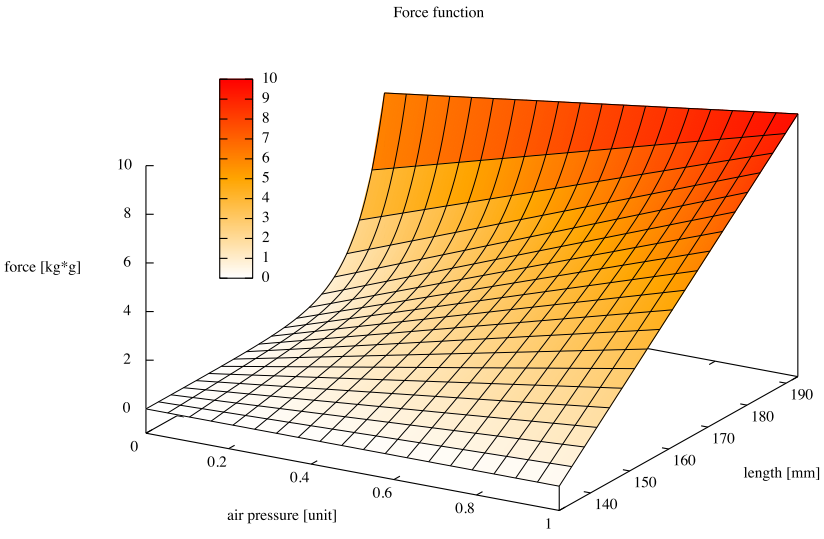


Figure 4: Plot of the force function used in the `Muscle` class

7 Acknowledgements

The *Artificial Muscle Simulator* was developed in 2007/2008 by *Max Braun*, a student of computational visualistics at the University of Koblenz-Landau, Koblenz, Germany, during a one-semester stay at the *Yuragi* project[10] of Osaka University, Osaka, Japan, initiated by the Artificial Intelligence Research Group (AGKI)[11] in Koblenz and supported by scholarships from the German Academic Exchange Service (DAAD)[12] and the German National Academic Foundation (Studienstiftung des deutschen Volkes)[13].

References

[1] Smith, R.: Open Dynamics Engine. `http://www.ode.org/`

[2] Smith, R.: ODE User Guide. `http://www.ode.org/ode-latest-userguide.html`

[3] Y. Nakamura, I. Fukuyori, M. Braun, Y. Matsumoto, H. Ishiguro: Control of human-like robotic arm based on biological fluctuation. In IROS, 2008 (submitted)

[10] `http://www.yuragi.osaka-u.ac.jp/`
[11] `http://www.uni-koblenz.de/FB4/Institutes/IFI/AGKI`
[12] `http://www.daad.de/`
[13] `http://www.studienstiftung.de/`