Simulating Artificial Muscles for Controlling a Robotic Arm with Fluctuation

Max Braun[1,2]

[1] Institute for Computer Science,
University of Koblenz-Landau, Koblenz, Germany
maxbraun@uni-koblenz.de
[2] Department of Adaptive Machine Systems,
Graduate School of Engineering, Osaka University, Osaka, Japan
max.braun@ams.eng.osaka-u.ac.jp

Abstract. Understanding and utilizing fluctuation effects is the motivation behind the interdisciplinary *Yuragi Project*. One specific application of this principle is the control of a complex, biologically inspired robotic arm based on the *Attractor Selection Model*. A simulator for the robot's artificial muscles has been developed in order to evaluate this approach. The *Artificial Muscle Simulator* introduces a model of the flexible pneumatic actuators which accurately reproduces their behavior. It has been designed to be highly customizable, allowing for other robot models and control methods.

Table of Contents

1 Introduction

In engineering, random noise is usually an undesired phenomenon, which one tries to suppress at great cost. At normal temperatures, Brownian motion always causes an unpredictable and irregular fluctuation. This effect is particularly strong on a microscopic level and therefore influences many biological processes. Yanagida et al. could show that motor proteins use fluctuation for directional movement [14] and Kashiwagi et al. describe the gene expression in bacteria for adaption to unknown environments as a utilization of fluctuation effects [5].

This biological principle inspired an interdisciplinary project at *Osaka University* by the name of *Yuragi*, the Japanese word for fluctuation [15]. In the four sections biology, materials science, information science, and robotics, scientists try to gain control of the general phenomenon of fluctuation and apply it as a solution to a number of different problems. In the robotics section, *Yuragi* is used for improving human-computer interaction with Mitsubishi's household robot *Wakamaru* [8] – with algorithms for task assignment and path planning that mimic biological processes – and developing small swimming robots imitating the movement of bacteria. Professor Ishiguro's Androids [3] use *Yuragi* as well in order to generate movements as lifelike as possible.

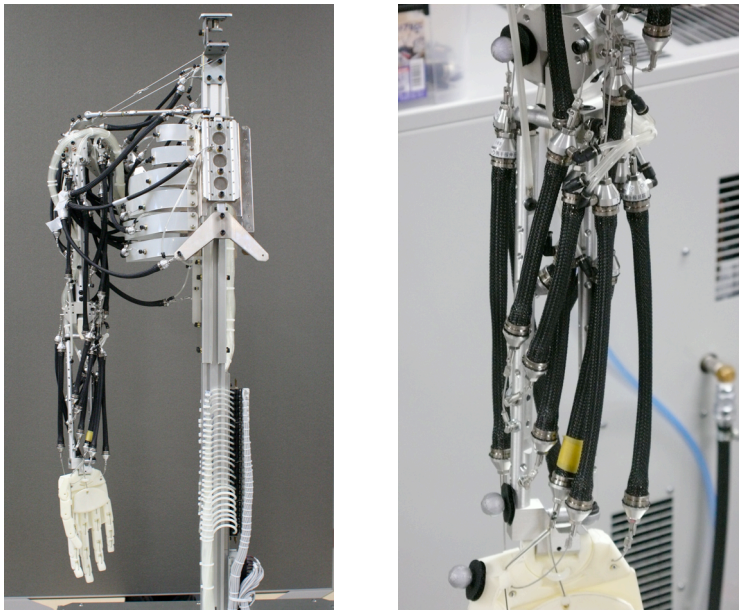
The focus of this thesis, however, will be on one specific application of the *Yuragi* principle to a biologically inspired robotic arm, which is described in Section 2. In order to evaluate a control method based on the *Attractor Selection Model* (as introduced in Section 5) a simulator for the robotic arm has been developed. The underlying model for the simulated artificial muscles is described and justified in Section 3 and the simulator software itself in Section 4. The results of the simulation experiments are explained in Section 6.

2 A biologically inspired robotic arm

The robot shown in Figure 1(a) mimics the anatomy of a human upper limb. Its bones are made of metal and plastic, while air pressure controls the 30 flexible pneumatic actuators or "artificial muscles". The complexity of the system is caused by the large number of degrees of freedom, the elasticity of the actuators and their redundant arrangement as seen in Figure 1(b).

These factors render it difficult to calculate the correct air pressure distribution for a desired posture of the robot. Solutions with inverse kinematics [7] would be very complicated and would require detailed understanding of the system's geometric relationships. Machine learning techniques like reinforcement learning [4] at this level of complexity are CPU-intensive [12] and, moreover, need to be trained in advance.

In the *Yuragi Project*, techniques to control the robotic arm are developed which – by exploiting fluctuation – can do without a priori information about the arrangement or the behavior of the actuators and, furthermore, do not require any previous training as in machine learning. Here, the search for the optimal air pressure distribution is driven by a stochastic fluctuation and controlled by



(a) Total (b) Detail

Fig. 1. The biologically inspired robotic arm

comparing the current external state of the robot with the desired goal state. The *Attractor Selection Model*, originally a model for the change of genetic programs in bacteria, provides the mathematical foundation.

3 Simulating artificial muscles

The pneumatic actuators ("artificial muscles") used for the robotic arm are controlled by adjusting the air pressure inside them. An increase in air pressure leads to an inflation of a bag inside the actuator and thus, due to the structure of the enclosing fabric, to a contraction of the artificial muscle along its main axis. Figure 2(a) shows such an actuator in its short, contracted state. Decreasing the air pressure again causes the muscle to relax, i.e. becoming longer and losing stiffness.

As a result of their structure and behavior, the actuators are able to exert a strong force while pulling, but practically none while pushing. Therefore each joint of the robotic arm had to be equipped with more than one artificial muscle in order to cover its full angle, mostly some redundant form an agonist-antagonist pair.

The model of the artificial muscles has two components, one qualitative and one quantitative. The qualitative model mimics the described structure and be-

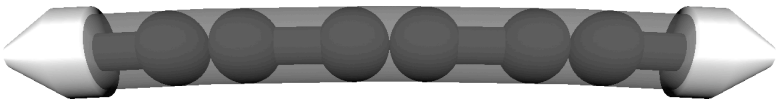
havior of the actuator whereas the addition of the quantitative refinement allows for accurate numerical measurements on the simulated actuators.

3.1 Qualitative model

The qualitative model depicted in Figure 2(b) is simply a chain with links of variable length. This captures the two main characteristics of an artificial muscle: The chain models the flexibility of the actuator whereas the variable lengths of the links can be adjusted to represent different air pressures inside the actuator. This model can be easily expressed with the physics engine *ODE* that is used for the simulator (compare Section 4).

For the *Open Dynamics Engine*, each link consists of two bodies (spheres) connected by a slider joint. The chain is formed by connecting each pair of links by a hinge joint with one axis. During the simulation, a force that corresponds to the given air pressure is exerted on each slider joint. The calculation of this force depends on several factors and a number of empirically determined constants, which were derived from the actual artificial muscles.

(a) A pneumatic actuator



(b) Illustration of the qualitative model

Fig. 2. An artificial muscle and its model in the simulator

3.2 Quantitative refinement

The desired force function $F(p, l)$ for one actuator in a given time step depends on the air pressure p inside the artificial muscle and the length l of the muscle itself. This relationship has been determined through a series of experiments and an interpolation of the obtained data. The final experiment setup and procedure went as follows:

1. The analyzed actuator was mounted in a frame as shown in Figure 3(a).
2. The air pressure inside the muscle was set to a value between 0 and 255.[3]
3. The actuator was weighted with a load (Figure 3(b)) between 0 and 8.7 kg.[4]
4. The length of the actuator was measured and recorded together with the current values for air pressure and weight (i.e. force).

Steps 1 through 4 were repeated to cover the whole range of air pressures, producing all possible lengths (by attaching all appropriate loads), and including three different kinds of actuators (short, medium-sized, and long).

(a) Analyzed actuator (b) Weights

Fig. 3. Experiment setup to obtain the force function

The obtained three-dimensional data points (plotted in Figure 5) represent the desired relationship between air pressure, length, and force. To be used during the simulation, however, this data has to be expressed as a function interpolating the holes between the data points, while also being simple to calculate.

The approach was a force function which is linear in length for high air pressures and exponential in length for low air pressures:

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

[3] The control software for the compressor used an 8-bit value for the air pressure.
[4] Of interest, however, is the corresponding force of $8.7 \cdot 9.81$ N.

The variable p denotes air pressure, l the length of the actuator, and their respective minimum and maximum values are $l_{\min}$, $l_{\max}$, $p_{\min}$, and $p_{\max}$. The parameters a, b, and c were determined using nonlinear least squares fitting [2] in Mathematica [13] on the obtained data and result to:

$$a = 7.687724845623249 \cdot 10^{-5} \tag{2}$$

$$b = 9.203893778016578 \tag{3}$$

$$c = 9.799574209874333 \tag{4}$$

A plot of the final force function, which is used to calculate the force exerted on the slider joints in each simulation step, is shown in Figure 4.

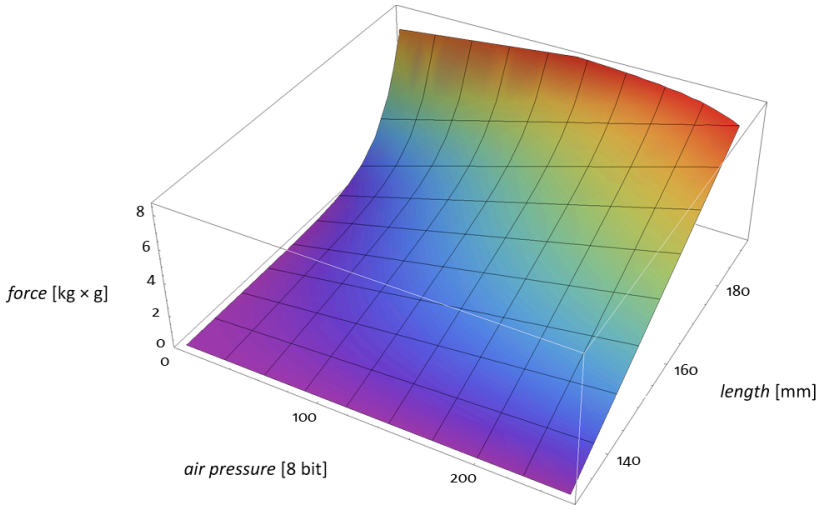


Fig. 4. The interpolated force function

3.3 Justification

To justify the proposed model for simulating artificial muscles, the experiments introduced in Section 3.2 were accurately reproduced in the simulator. The data from corresponding experiments was compared (Figures 5 and 6) and showed a sufficiently small error: $5.7 \cdot 10^{-4}$ for a long muscle, $4.2 \cdot 10^{-4}$ for a medium-sized muscle, and $8.9 \cdot 10^{-4}$ for a short muscle.[5]

The combination of the qualitative model and the force function in the simulator also lead to a notable effect that supports the justification of the proposed

[5] The error function used was: $\frac{\sum_i (\text{measured}_i - \text{simulated}_i)^2}{\sum_i \text{measured}_i^2}$

model: Although the kink seen in the upper part of the experimental data plot of Figure 5 is not present in the abstract force function (Figure 4), it reappears in the plot for the simulated experiments shown in Figure 6.

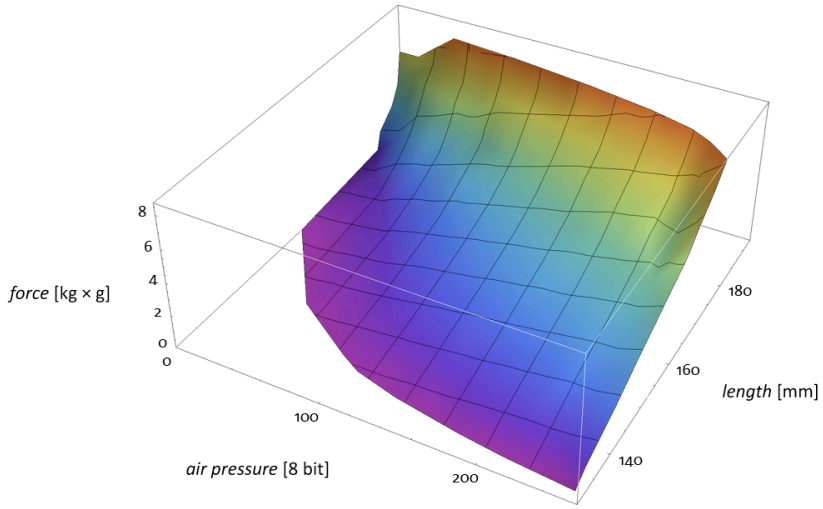


Fig. 5. Data points for a medium-sized artificial muscle

4 The simulator software

Although the *Artificial Muscle Simulator*[6] had originally been designed and used to evaluate the *Attractor Selection Model* control method for a specific biologically inspired robotic arm, the core of the simulator is independent of the original robot design and can be customized to suit different tasks. The *Artificial Muscle Simulator* can in fact be used to simulate any robot that uses pneumatic actuators and can be modeled with the available primitives and joints.

4.1 Architecture

The simulator is an abstraction of the physics simulation engine *ODE* [9]. The *Open Dynamics Engine* simulates rigid body dynamics with collision detection, gravity, and joints between bodies. The simulator is 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++*.

Figure 7 illustrates the architecture of the simulator. The central parts robot, controller, and simulation interact in each step of *ODE*'s simulation run loop.

[6] Download here: `http://www.uni-koblenz.de/~maxbraun/musclesim.zip`

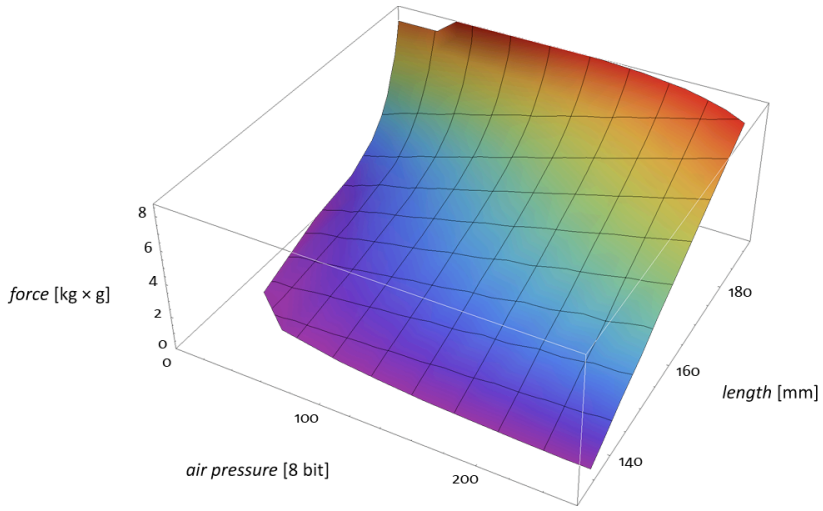


Fig. 6. Data points for a simulated medium-sized artificial muscle

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 this action. Displaying the animated robot and interacting with it are both optional components, which can be switched off.

Files and classes The main folder `musclesim 1.2` contains a `main.cpp`, the `Makefile`, the binary `musclesim`, and a `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`.
- `controller` holds the class `Controller`, where the control method for the robot is implemented.

While the classes in the second folder provide the basic functionality of the simulator, those in the last two folders may realize optional customization by the user. Detailed class diagrams for the latter can be found in Appendix A. Before describing ways of customizing the robot and the controller in Sections 4.3 and 4.4, respectively, the next section explains how to operate the simulator application itself.

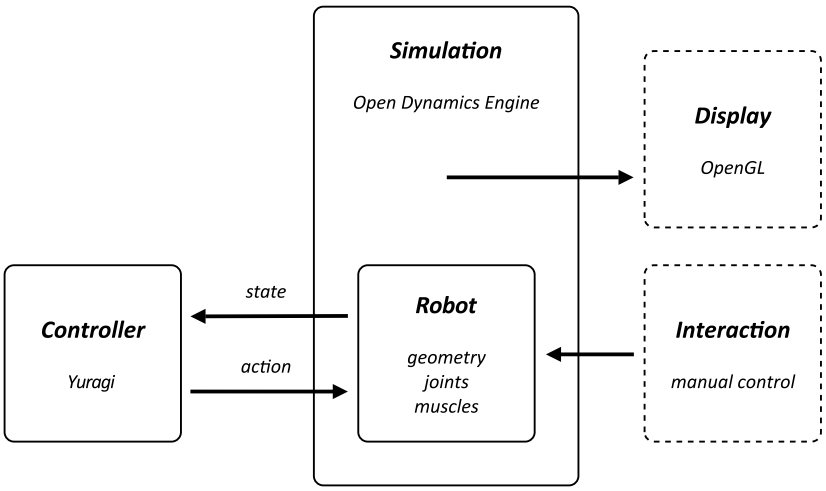


Fig. 7. Architecture of the *Artificial Muscle Simulator*

4.2 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.

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.

These command line arguments can be combined arbitrarily:

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

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 8. The air pressure in the selected actuator can be increased with `+` and decreased with `-`.[7] To quit the simulation, close the window in visualization mode or press `Ctrl-C` in simple mode.

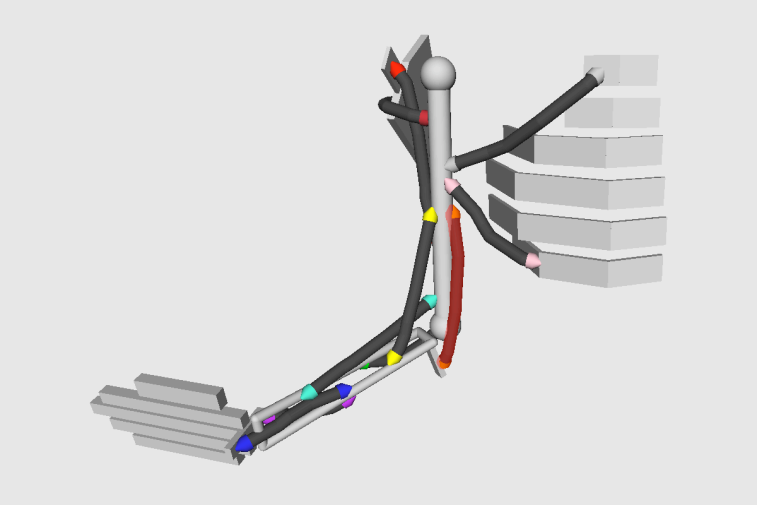


Fig. 8. The simulator in interactive mode

4.3 Building a robot

To set up a new robot in the simulator, create a subclass of `Robot` such as the existing `RoboticArm`.[8] In its constructor, the structure of the robot is defined through *primitives* which are connected by *joints* and *actuators*.[9]

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

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

[9] 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.

Primitives There are three different types of primitive objects.[10] 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
);
```

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:

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

```
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
```

Joints There are five joint types available: *ball*, *hinge* (1-axis and 2-axis), and *universal* (1-axis and 2-axis).[11] Connecting two primitives is realized by calling the corresponding methods on one primitive with the other as the first parameter[12] 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(
```

[11] For more information about the different joint types, see the *ODE User Guide*. [11]
[12] Passing `NULL` will create a connection to the environment.

```
  elbow, elbow_position, x_axis,
  -90, // minimum angle
  0 // maximum angle
);
```

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 `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.

Actuators There are two types of actuators included in this simulator: The `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.[13] 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 3rd-degree Bézier curve [1] 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};
```

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

```
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
```

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);
```

4.4 Controlling the robot

A subclass of `Controller` is mainly responsible for automatic control of the robot.[14] Two controllers are included: `ASMController` implementing the attractor selection model and `AASMController` for the adaptive attractor selection model.[15]

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.

Simulation run loop The run loop introduced in Section 4.1 uses the active instances of `Robot` and `Controller` as well as the `Simulation` class:

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

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

[15] The implementations can be found in `controller/asm/` and `controller/aasm/` respectively.

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.

4.5 Notes

Units Due to *ODE*'s problems with small numbers, the units do not comply with SI [10]. Instead, all lengths are in millimeters and masses in gram.

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 9 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 $r(30) = 442$.

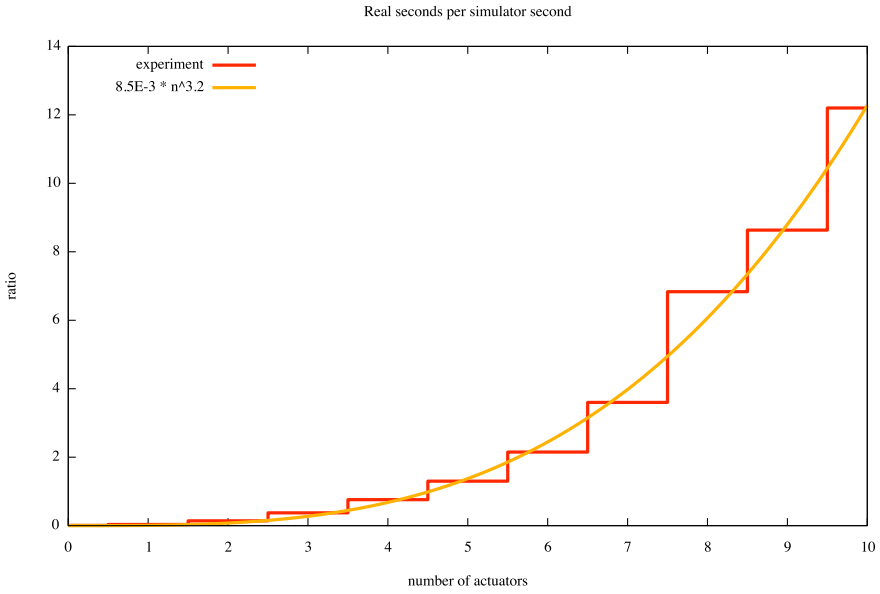


Fig. 9. Simulation time ratio over the number of actuators

5 The Attractor Selection Model

A specific task illustrating the general idea behind the *Attractor Selection Model* and its use of fluctuation is to calculate the optimal air pressure distribution for the actuators of the robotic arm in order to move the robot's hand to a certain position in space.

The fluctuating movement takes place in $\mathbb{R}^n$, where in the simplest case n is the number of degrees of freedom. Some attractors $\mathbf{x}_i$ are distributed in this space, forming a potential P: (width σ appropriate)

$$P(\mathbf{x}) = -\sum_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{1}{2\sigma^2}|\mathbf{x} - \mathbf{x}_i|^2) \tag{5}$$

A state vector $\mathbf{x}$ is now drawn to the attractors and at the same time disturbed by a random noise η. The strength of the noise's influence is determined by the scalar activity a. It is adjusted externally to high values in good situations (when the hand is near the target position) and low values otherwise. The temporal development of the state $\mathbf{x}$ is a noisy gradient descent in P and can be described with a Langevin equation:

$$\dot{\mathbf{x}} = a \cdot \nabla P(\mathbf{x}) + \eta \tag{6}$$

The state $\mathbf{x}$ is translated into a control signal for the actuators in every time step, e.g. just by scaling the domain appropriately. Then, the state $\mathbf{x}$ is changed according to Equation 6, now with a new activity a, which is calculated from the distance of the hand's current position $\mathbf{z}(t)$ to the target position $\hat{\mathbf{z}}$: ("forgetting factor" $\gamma \in [0, 1]$ appropriate)

$$a(t) = d(t) - \bar{d}(t-1) \tag{7}$$

$$d(t) = |\hat{\mathbf{z}} - \mathbf{z}(t)|^{-1} \tag{8}$$

$$\bar{d}(t-1) = \frac{\sum_{\tau=1}^{t-1} \gamma^\tau d(t-\tau)}{\sum_{\tau=1}^{t-1} \gamma^\tau} \tag{9}$$

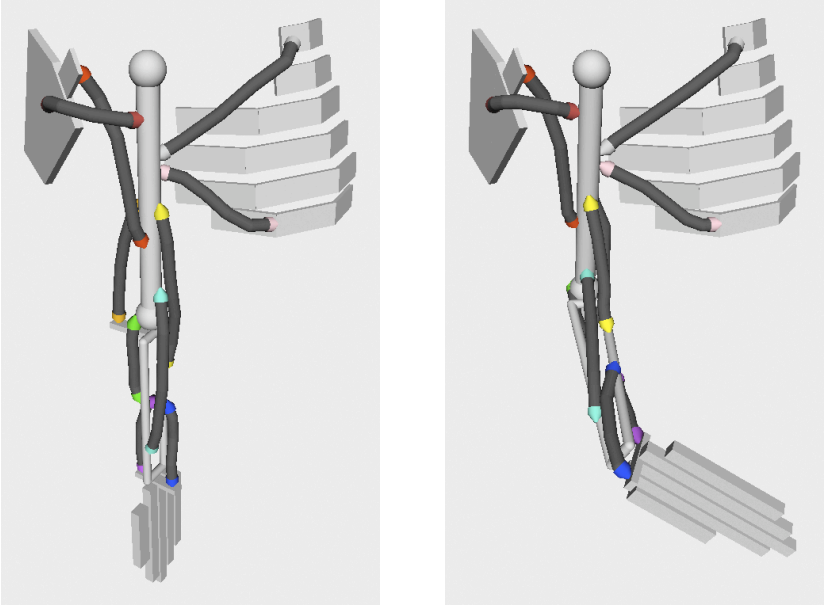
The result of this procedure is that over time the one air pressure distribution among the attractors is chosen which is most likely to move the hand to the desired target position.

An extension of the principle is the *Adaptive Attractor Selection Model*, where attractors are rearranged dynamically, depending on the activity, in order to avoid inconvenient attractor distributions in the search space.

There are similarities of this optimization method to simulated annealing [6], which also imitates a natural phenomenon and uses randomness. But the dynamically adjusted activity of the *Attractor Selection Model* plays a more complex role than the continuously decreased temperature in SA.

6 Simulation and results

Despite the generality of the *Artificial Muscle Simulator*'s design, its initial purpose was the evaluation of the *Attractor Selection Model* as a control method for the introduced biologically inspired robotic arm. Therefore, a simplified version of the robot was rebuilt in the simulator, omitting geometric details and reducing the number of actuators[16] to 10, while preserving the characteristics complexity and redundancy. Two frames of the simulated robotic arm during the experiment are shown in Figure 10.



(a) t = 0 s (b) t = 80 s

Fig. 10. Two frames of the simulated robotic arm

The implemented task for the robot was to raise its hand to a previously defined height and hold it there. Being a special case of the general reaching task explained in Section 5, the activity was designed to be inversely proportional to the difference between the hand's current height and the target height. The number of the 10-dimensional attractors in this experiment had been set to 400. To take the oscillations into account which arise due to the flexibility of the pneumatic actuators, the state of the *Attractor Selection Model* and the activity

[16] Compare the note on performance in Section 4.5.

have been updated every 6 seconds instead of every simulation step. Figure 11 shows the results of the simulation.[17]

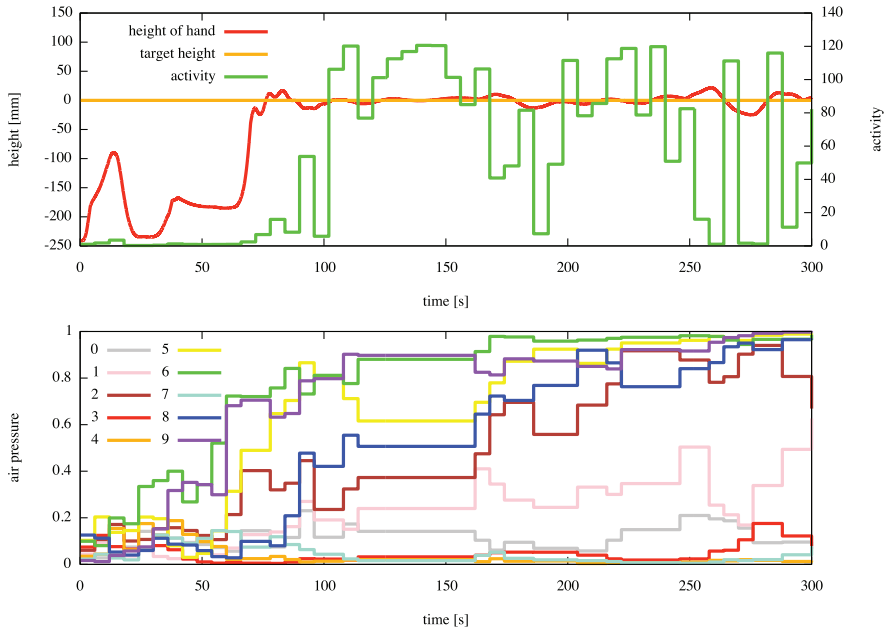


Fig. 11. Results of the height reaching task

The upper part of the graph shows how, after some time, the height of the hand (red line) reaches the target height (orange line) and stabilizes there. The activity (green line) is obviously a measure for the quality of the hand's position, reaching high values whenever the current height is near the target height and low values otherwise. The activity's function as a regulator for the degree of fluctuation also becomes evident when the initial intense fluctuation is dampened as soon as the activity rises.

The evolution of the state of the *Attractor Selection Model* is shown in the lower part of the graph. Each line represents the normalized amount of air pressure in one actuator. (The lines' colors correspond to the colors of the artificial muscles in Figure 10 and the video.) Here, the degree of fluctuation can be observed in a direct way as opposed to the resulting fluctuation of the hand's height. Again, the correlation to the magnitude of the activity becomes evident and suitable attractors can be identified by locating stable periods as the one between 110 s and 160 s.

[17] Download a video of the simulated robotic arm during the first half of the experiment here: `http://www.uni-koblenz.de/~maxbraun/musclesim.mov`

These results show that the *Attractor Selection Model* can in principle be used to control a complex and redundant biologically inspired robotic arm. For simple tasks like the introduced height reaching, this control method shows the intended behavior. Although scalability and robustness of the method yet have to be improved, there are notable advantages to comparable methods, such as design-independence and the lack of need for training, which support the *Yuragi* approach.

7 Conclusion

Research in the *Yuragi Project* is based on the observation that some biological systems are not disturbed by fluctuation and utilize it for problem solving instead. The discovered concepts are transferred and applied to engineering problems, which might not be related at first sight. One such application of a mechanism found in bacteria is the control of a complex and redundant robotic arm. The control method for the robot, which is based on a human upper limb, uses the *Attractor Selection Model* to link the external posture of the robot with its internal actuator state through controlled fluctuation.

To better evaluate the performance of the proposed method, a simulator for the robot's pneumatic actuators or "artificial muscles" has been developed. The underlying model is the combination of a qualitative physical approximation and a quantitative refinement. The core of the refinement is a force function, which interpolates data from experiments with the actual artificial muscles. The *Artificial Muscle Simulator* implements the introduced biologically inspired robotic arm and offers means for evaluation. The results of the simulation show that the *Attractor Selection Model* is a promising basis for a controller of a robotic arm with the introduced properties.

The *Artificial Muscle Simulator* itself, however, is not limited to the discussed task, but has been designed as a versatile tool to evaluate future improved versions of the *Attractor Selection Model* or completely different control methods, while also supporting other robot designs.

8 Acknowledgements

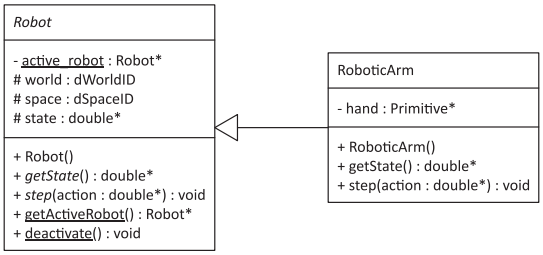
The author would like to thank the *German National Merit Foundation* (*Studienstiftung des deutschen Volkes*) and the *German Academic Exchange Service* (*DAAD*) for their respective scholarships as well as the *Intelligent Robotics Laboratory* at *Osaka University* and the *Artificial Intelligence Research Group* (*AGKI*) at the *University of Koblenz-Landau*.

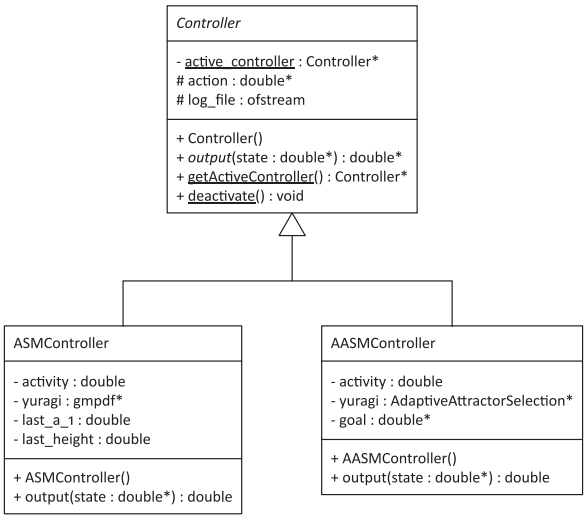
References

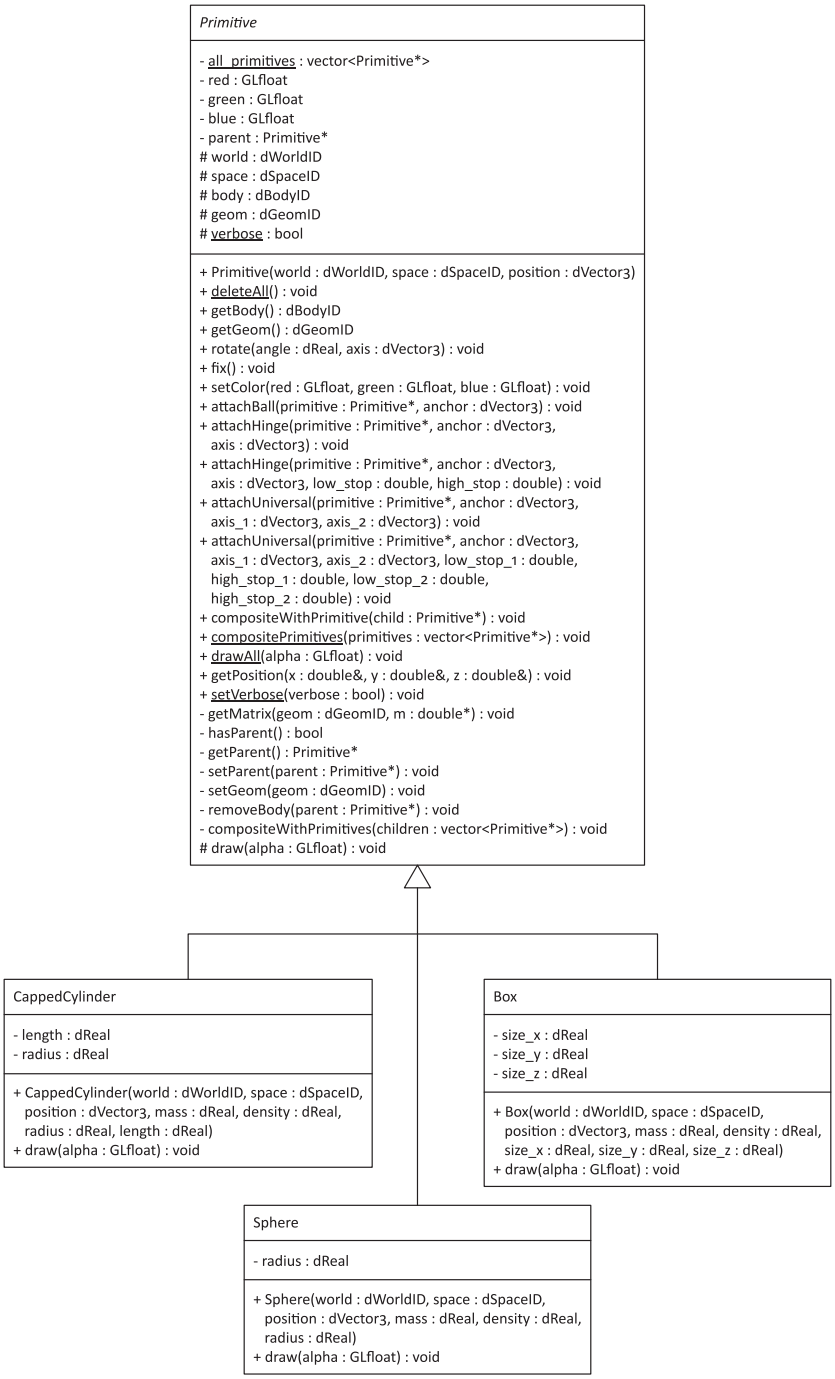
1. R.H. Bartels, J.C. Beatty, B.A. Barsky: Bézier Curves. An Introduction to Splines for Use in Computer Graphics and Geometric Modelling. Morgan Kaufmann (1998) 211–245
2. D.M. Bates, D.G. Watts: Nonlinear Regression and Its Applications. Wiley, New York (1988)
3. H. Ishiguro: Android Science: Toward a new cross-interdisciplinary framework. Robotics Research **28** (2007) 118–127
4. L.P. Kaelbling, M.L. Littman, A.W. Moore: Reinforcement Learning: A Survey. Journal of Artificial Intelligence Research **4** (1996) 237–285
5. A. Kashiwagi, I. Urabe, K. Kaneko, T. Yomo: Adaptive Response of a Gene Network to Environmental Changes by Fitness-Induced Attractor Selection. PLoS ONE (2006)
6. S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi: Optimization by Simulated Annealing. Science **220** (1983) 671–680
7. K. Kondo: Inverse kinematics of a human arm. Journal of Robotic Systems **8** (1991) 115–175
8. Mitsubishi Heavy Industries, Ltd.: Wakamaru. Life with a Robot. `http://www.mhi.co.jp/kobe/wakamaru/english/`
9. R. Smith: Open Dynamics Engine v0.5. `http://www.ode.org/`
10. Bureau International des Poids et Mesures: The International System of Units (SI) brochure. 8$^{\text{th}}$edition (2006)
11. R. Smith: ODE User Guide. `http://www.ode.org/ode-latest-userguide.html`
12. J. Peters, S. Schaal: Policy learning for motor skills. In Proceedings of ICONIP (2007)
13. Wolfram Research, Inc.: Mathematica, Version 6.0. Champaign, IL (2007).
14. T. Yanagida, M. Ueda, T. Murata, S. Esaki, Y. Ishii: Brownian motion, fluctuation and life. Biosystems **88** (2007) 228–242
15. Osaka University: Yuragi Project. `http://www.yuragi.osaka-u.ac.jp/`

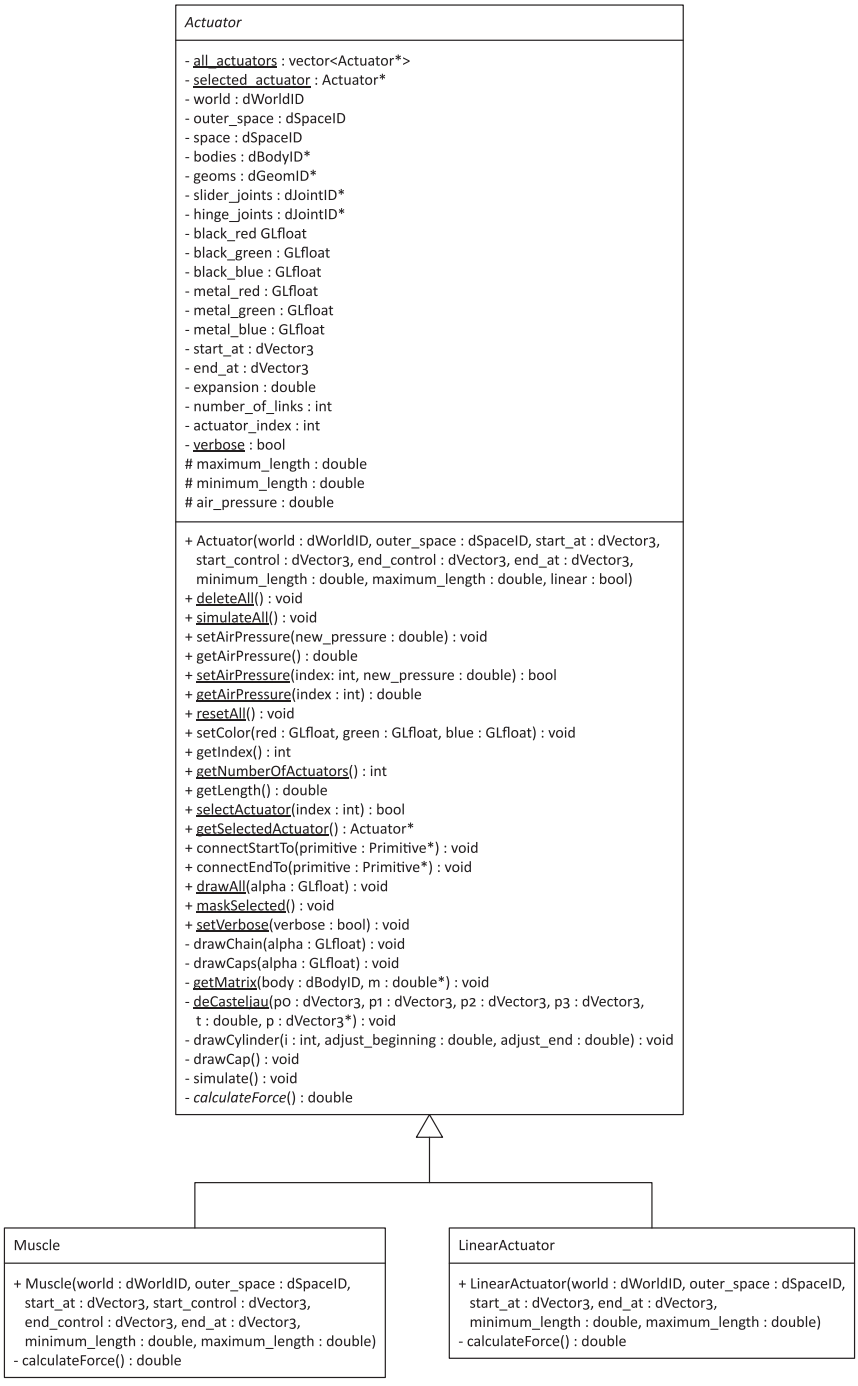
List of Figures

A Main classes






Primitive
- all_primitives : vector<Primitive*>
- red : GLfloat
- green : GLfloat
- blue : GLfloat
- parent : Primitive*
world : dWorldID
space : dSpaceID
body : dBodyID
geom : dGeomID
verbose : bool
+ Primitive(world : dWorldID, space : dSpaceID, position : dVector3)
+ deleteAll() : void
+ getBody() : dBodyID
+ getGeom() : dGeomID
+ rotate(angle : dReal, axis : dVector3) : void
+ fix() : void
+ setColor(red : GLfloat, green : GLfloat, blue : GLfloat) : void
+ attachBall(primitive : Primitive*, anchor : dVector3) : void
+ attachHinge(primitive : Primitive*, anchor : dVector3,
axis : dVector3) : void
+ attachHinge(primitive : Primitive*, anchor : dVector3,
axis : dVector3, low_stop : double, high_stop : double) : void
+ attachUniversal(primitive : Primitive*, anchor : dVector3,
axis_1 : dVector3, axis_2 : dVector3) : void
+ attachUniversal(primitive : Primitive*, anchor : dVector3,
axis_1 : dVector3, axis_2 : dVector3, low_stop_1 : double,
high_stop_1 : double, low_stop_2 : double,
high_stop_2 : double) : void
+ compositeWithPrimitive(child : Primitive*) : void
+ compositePrimitives(primitives : vector<Primitive*>) : void
+ drawAll(alpha : GLfloat) : void
+ getPosition(x : double&, y : double&, z : double&) : void
+ setVerbose(verbose : bool) : void
- getMatrix(geom : dGeomID, m : double*) : void
- hasParent() : bool
- getParent() : Primitive*
- setParent(parent : Primitive*) : void
- setGeom(geom : dGeomID) : void
- removeBody(parent : Primitive*) : void
- compositeWithPrimitives(children : vector<Primitive*>) : void
draw(alpha : GLfloat) : void
CappedCylinder
- length : dReal
- radius : dReal
+ CappedCylinder(world : dWorldID, space : dSpaceID,
position : dVector3, mass : dReal, density : dReal,
radius : dReal, length : dReal)
+ draw(alpha : GLfloat) : void
Box
- size_x : dReal
- size_y : dReal
- size_z : dReal
+ Box(world : dWorldID, space : dSpaceID,
position : dVector3, mass : dReal, density : dReal,
size_x : dReal, size_y : dReal, size_z : dReal)
+ draw(alpha : GLfloat) : void
Sphere
- radius : dReal
+ Sphere(world : dWorldID, space : dSpaceID,
position : dVector3, mass : dReal, density : dReal,
radius : dReal)
+ draw(alpha : GLfloat) : void


Actuator
- all_actuators : vector<Actuator*>
- selected_actuator : Actuator*
- world : dWorldID
- outer_space : dSpaceID
- space : dSpaceID
- bodies : dBodyID*
- geoms : dGeomID*
- slider_joints : dJointID*
- hinge_joints : dJointID*
- black_red GLfloat
- black_green : GLfloat
- black_blue : GLfloat
- metal_red : GLfloat
- metal_green : GLfloat
- metal_blue : GLfloat
- start_at : dVector3
- end_at : dVector3
- expansion : double
- number_of_links : int
- actuator_index : int
- verbose : bool
maximum_length : double
minimum_length : double
air_pressure : double
+ Actuator(world : dWorldID, outer_space : dSpaceID, start_at : dVector3, start_control : dVector3, end_control : dVector3, end_at : dVector3, minimum_length : double, maximum_length : double, linear : bool)
+ deleteAll() : void
+ simulateAll() : void
+ setAirPressure(new_pressure : double) : void
+ getAirPressure() : double
+ setAirPressure(index: int, new_pressure : double) : bool
+ getAirPressure(index : int) : double
+ resetAll() : void
+ setColor(red : GLfloat, green : GLfloat, blue : GLfloat) : void
+ getIndex() : int
+ getNumberOfActuators() : int
+ getLength() : double
+ selectActuator(index : int) : bool
+ getSelectedActuator() : Actuator*
+ connectStartTo(primitive : Primitive*) : void
+ connectEndTo(primitive : Primitive*) : void
+ drawAll(alpha : GLfloat) : void
+ maskSelected() : void
+ setVerbose(verbose : bool) : void
- drawChain(alpha : GLfloat) : void
- drawCaps(alpha : GLfloat) : void
- getMatrix(body : dBodyID, m : double*) : void
- deCasteljau(p0 : dVector3, p1 : dVector3, p2 : dVector3, p3 : dVector3, t : double, p : dVector3*) : void
- drawCylinder(i : int, adjust_beginning : double, adjust_end : double) : void
- drawCap() : void
- simulate() : void
- calculateForce() : double
Muscle
+ Muscle(world : dWorldID, outer_space : dSpaceID, start_at : dVector3, start_control : dVector3, end_control : dVector3, end_at : dVector3, minimum_length : double, maximum_length : double)
- calculateForce() : double
LinearActuator
+ LinearActuator(world : dWorldID, outer_space : dSpaceID, start_at : dVector3, end_at : dVector3, minimum_length : double, maximum_length : double)
- calculateForce() : double