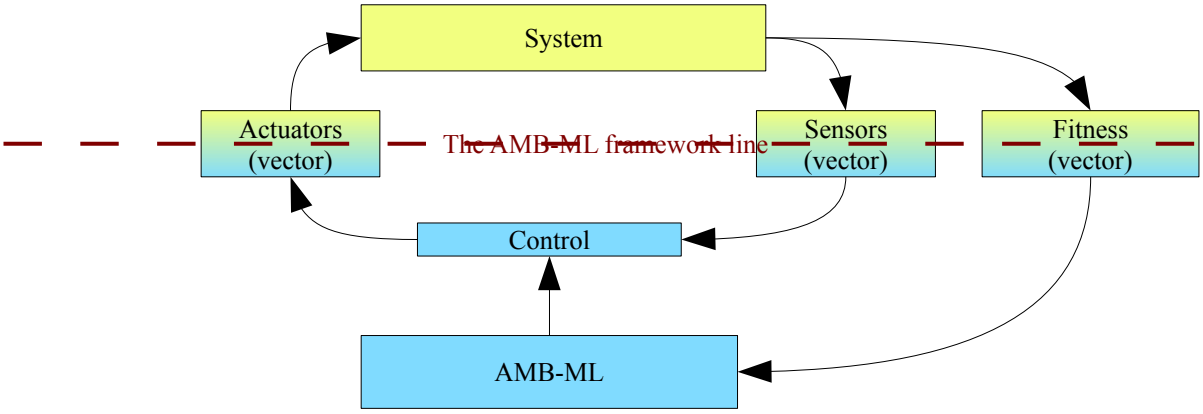
AMB-ML : Ambrosys Machine Learning Framework



Description of the software

What is it

AMB-ML is a powerful tool using Machine Learning to determine the optimal control of dynamical systems to a desired state: Natural and technical systems often show undesirable behavior. Examples include fluid turbulence which may adversely effect the forces on transport vehicles, financial crises with dramatic consequences for the world's economy, or biophysical systems with obvious impact on our own life. Control can serve to stabilize systems, as lasers, quantum systems, or delayed feedback systems or keep an industrial mixer at its best performance. Consequently, the control of complex systems is an issue of major importance.

The well-worked out classical control theory cannot address the control of the systems named above in a variety of situations. This led us to develop AMB-ML in order to find optimal control even for systems, where classical approaches do fail. We employ massively methods from artificial Lets now assume a complex system has sensors and actuators, but a useful control law is missing. The AMB-ML software uses these sensors and actuators to control the system so that it drives it and keep it into a desired state the user has specified.

Given the general outline of the software, it is clear, that the AMB-ML framework can be used either for both, numerical investigations ore experimental systems.

How does it work?

AMB-ML uses machine learning algorithms, most often genetic programming or a neural network, to learn from the system how to make a good control function for a particular system. For this purpose, it tests and improves its control methods using the system itself as a test tool. The result is a control law optimized for the user-specified needs, i.e. in terms of a fitness.

What are the advantages of AMB-ML?

AMB-ML uses a model free approach: we do not need to know the system, we only need to be able

to measure its state and actuate on it.

AMB-ML has powerful built in machine learning algorithms, like genetic programming or neural networks.

AMB-ML is flexible: it uses standardized communication between the core and the system to be controlled so that they can even be physically distant.

AMB-ML is is platform independent, being written in Java. It can run on any architecture that has a Java virtual machine an complies the minimum requirements.

AMB-ML is performant. Using the newest algorithms, the state-of-the art performance is guaranteed.

Ambrosys is also developing gpcxx, an open source high performance genetic programming library in C++. It also provides customized high performance softwares for management of complex systems.

Technical description:

General:

Language: Java

Minimum JVM: 1.5

Recommended JVM: 1.7

Basic principle:

AMB-ML is composed of two main parts: the server and the client. The server contains the machine learning algorithms and a server to receive requests from the clients.

The clients can be either integrated in the AMB-ML client, written in Java or a wrapper provided by AMB-ML to wrap in Java any hardware or software systems, or they can also be clients implemented in any other language as far as they follow the communication scheme required by AMB-ML.

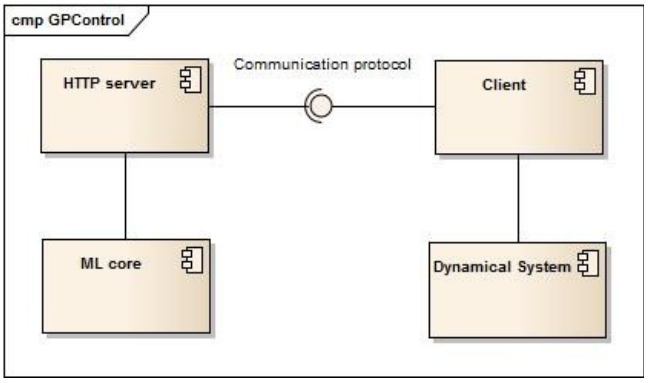


Illustration 1: General architecture of AMB-ML

In order to guarantee a high standard of quality, we use HTTP communication between the server and the clients. This ensures a simple and efficient way to communicate and makes the implementation of clients easier.

The machine learning technique used in AMB-ML is genetic programming. In artificial intelligence, genetic programming (GP) is an evolutionary algorithm-based methodology inspired by biological evolution to find computer programs that perform a user-defined task. Essentially GP is a set of instructions and a fitness function to measure how well a computer has performed a task. It is a specialization of genetic algorithms (GA) where each individual is a computer program (represented by a tree in AMB-ML). It is a machine learning technique used to optimize a population of computer programs (control functions) according to a fitness landscape determined by a program's ability to perform a given computational task (control the system).

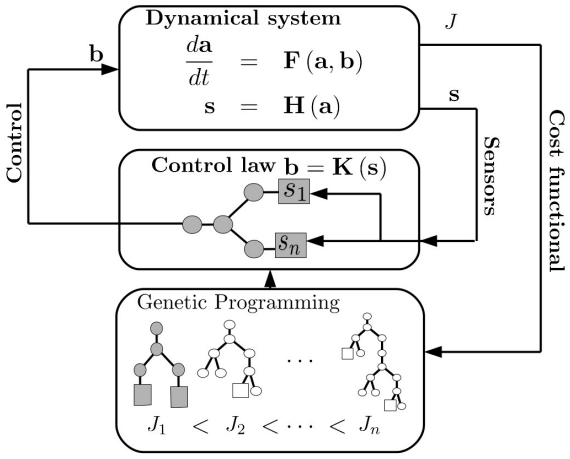


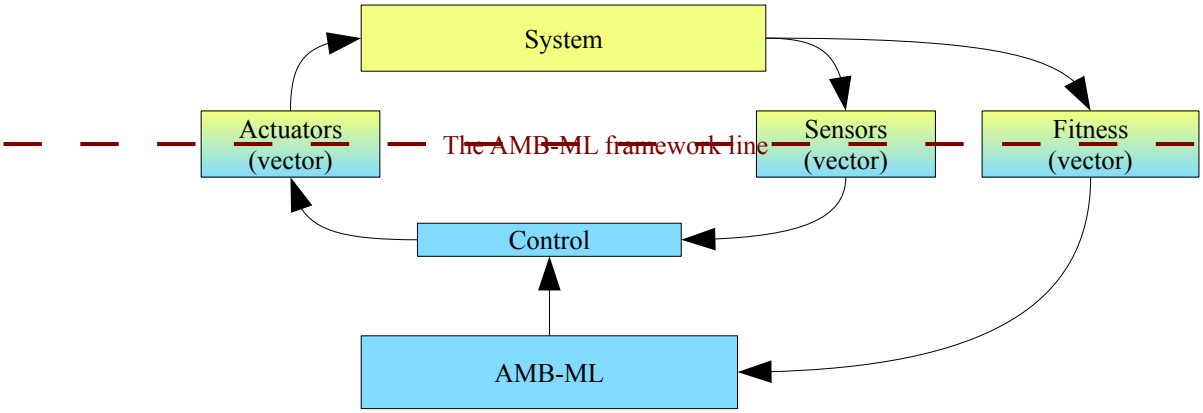
Illustration 2: Scheme of genetic programming for control

Input/output:

A typical situation where AMB-ML is used is the following: a system (hardware or software) has sensors and actuators.

The input a client gives to the server consists of informations about the state of the system described by the sensors. The actual implementation in AMB-ML is a vector of doubles. It can be of course adapted to the customers' needs.

The output that the server gives consist of actuation values computed from the sensor values provided before. Again, this is implemented as a vector of double in AMB-ML and can be adapted to the customers' needs.



Communication with AMB-ML

The AMB-ML components communicate using the HTTP protocol, encoding the messages in the Json format. This means that one of the AMB-ML server's component is actually a HTTP server. This HTTP server can be started on any port and will answer to HTTP request which format complies with the AMB-ML format.

Inside an HTTP request is a Json message. The type of Json message that AMB-ML expects in a HTTP request is the following: a first value called “type” and that contains a keyword indicating the

operation to be performed, and then an array of values called “values” containing the data to be processed.

An example of Json message is the following:

```
{
  “value”:“sensors”,
  “values”:[
    {
      “value”:0.1
    },
    {
      “value”:0.2
    }
  ]
}
```

This example message sends to the server a vector of two sensor values. The expected answer from the server in this case is the corresponding vector of actuation values.

Performance

The performance of AMB-ML decides about its useability in high-speed experiments and numerics. Therefore, we measure the performance in terms of evaluation of the control law, which is represented as a lexical tree in order to speed up its evaluation. There are two core questions we asnwer with the tests and grafics shown below:

how long is the evaluation of a typical tree (control law) in dependence on the complexity of the function. The latter is measured in terms of the depth of the tree, or the (logarithm of the) number of operations involved, respectively. Both are basically equivalent, as we show in Illustration 3.

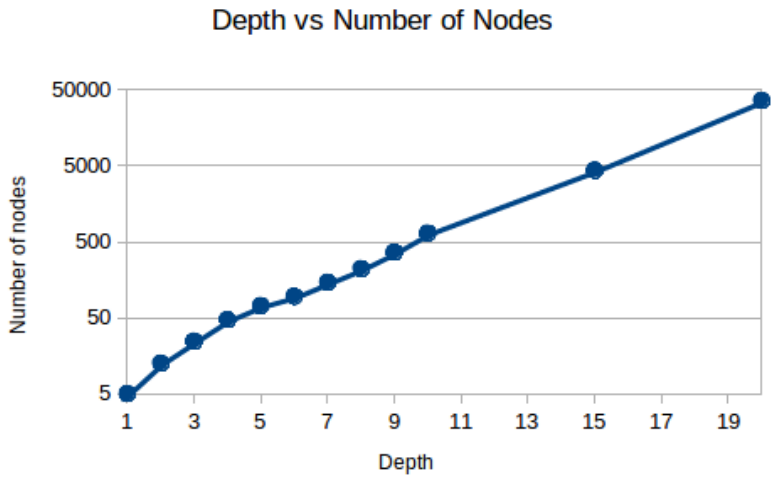


Illustration 3: The relation between the depth and the number of operations of a tree is an exponential one (note the logarithmic scale of the ordinate).

Now, we have measured the evaluation time per operation for a large range of participating operations or depth of the tree, respectively. The result is encouraging: we find a linear scaling of the evaluation time with depth of the trees; this in turn means a constant evaluation time per operation, no matter how large the tree! This is possible due to the very careful design of the function representation by AMB-ML. The result is shown in Illustrations 4 and 5: per operation in

the control law, 1.44 ns are needed. The CPU had a 3.6 GHz processor, such that we conclude that our evaluation is optimal (you have to remember that complex functions need more than 1 FLOP for their evaluation).

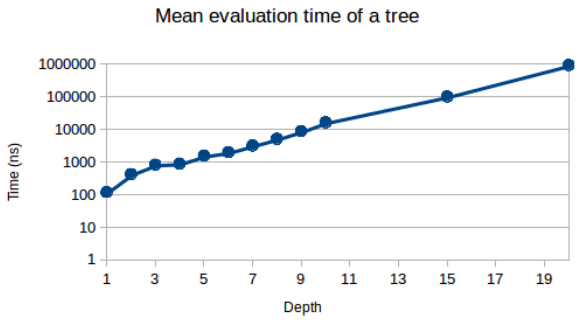


Illustration 4: Mean evaluation time of a tree depending on its depth

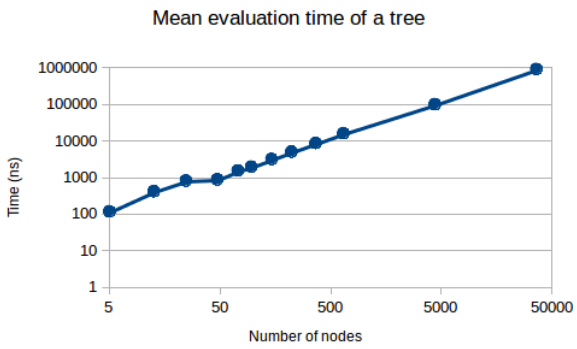


Illustration 5: mean evaluation time of a tree depending on its number of operations

Eventually, it is important how long the whole procedure runs, this involves system calls, pointer calls, links, blocking processes etc. Hence, we evaluated trees which consisted only of the same operation: +,-,:,*, and log, exp, sin, cos. This time, however, we started the evaluation from outside the program, this involves then all the "hidden" processes one usually does not care about. One notices that the evaluation is much longer, on the order of microseconds. To calm down the users: this microsecond is spent only once, at the start of the program.

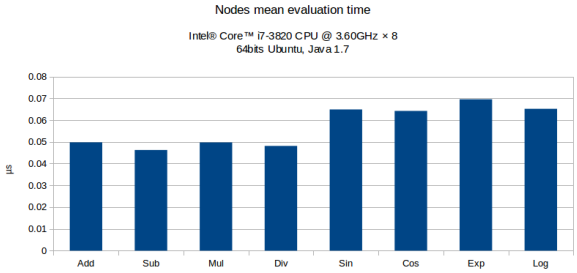


Illustration 6: Evaluation time of the nodes, including method calls and memory access

The above statistics are obtained by evaluating 5 randomly generated trees on a 20000 randomly generated set of input values. Illustrations 3, 4 and 5 show the core timings, not taking into account procedure calls. Illustration 6 shows real costs of evaluation of a node, that is to say, starting before the method call and ending after the method returns. This result implies that AMB-ML can digest sensors up to a sampling of 1 kHz, for faster devices, we need to interface directly with the software.