Chapter 2
Machine learning control (MLC)

“All generalizations are false, including this one."

- Mark Twain

In this chapter we discuss the central topic of this book: the use of powerful techniques from machine learning to discover effective control laws. Machine learning is used to generate models of a system from data; these models should improve with more data, and they ideally generalize to scenarios beyond those observed in the training data. Here, we extend this paradigm and wrap machine learning algorithms around a complex system to learn an effective control law $\mathbf{b} = \mathbf{K}(\mathbf{s})$ that maps the system output (sensors, $\mathbf{s}$) to the system input (actuators, $\mathbf{b}$). The resulting machine learning control (MLC) is motivated by problems involving complex control tasks where it may be difficult or impossible to model the system and develop a useful control law. Instead, we leverage experience and data to *learn* effective controllers.

The machine learning control architecture is shown schematically in Fig. 2.1. This procedure involves having a well-defined control task that is formulated in terms of minimizing a cost function J that may be evaluated based on the measured outputs of the system, $\mathbf{z}$. Next, the controller must have a flexible and general representation so that a search algorithm may be enacted on the space of possible controllers. Finally, a machine learning algorithm will be chosen to discover the most suitable control law through some training procedure involving data from experiments or simulations.

In this chapter, we review and highlight concepts from machine learning, with an emphasis on evolutionary algorithms (Sec. 2.1). Next (Sec. 2.2), we explore the use of genetic programming as an effective method to discover control laws in a high-dimensional search space. In Sec. 2.3, we provide implementation details and explore illustrative examples to reinforce these concepts. The chapter concludes with exercises (Sec. 2.4), suggested reading (Sec. 2.5), and an interview with Professor Marc Schoenauer (Sec. 2.6), one of the first pioneers in evolutionary algorithms.

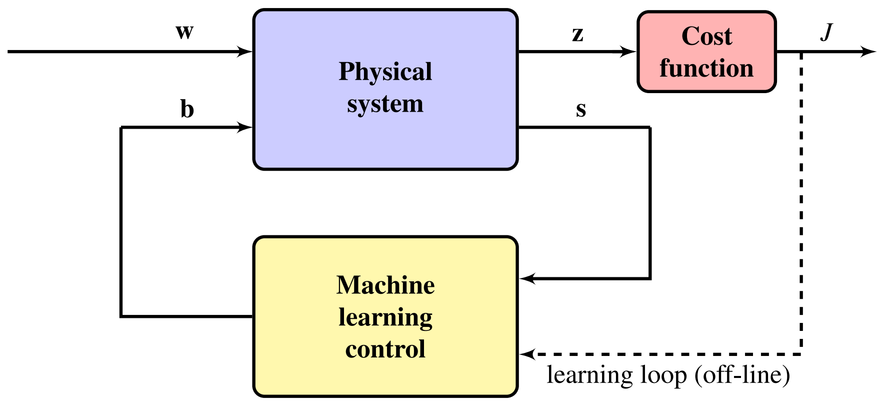


Fig. 2.1 Schematic of machine learning control wrapped around a complex system using noisy sensor-based feedback. The control objective is to minimize a well-defined cost function J within the space of possible control laws. An off-line learning loop provides experiential data to train the controller. Genetic programming provides a particularly flexible algorithm to search out effective control laws. The vector $\mathbf{z}$ contains all of the information that may factor into the cost.

2.1 Methods of machine learning

Machine learning [92, 30, 194, 168] is a rapidly developing field at the intersection of statistics, computer science, and applied mathematics, and it is having transformative impact across the engineering and natural sciences. Advances in machine learning are largely being driven by commercial successes in technology and marketing as well as the availability of vast quantities of data in nearly every field. These techniques are now pervading other fields of academic and industrial research, and they have already provided insights in astronomy, ecology, finance, and climate, to name a few. The application of machine learning to design feedback control laws has tremendous potential and is a relatively new frontier in data-driven engineering.

In this section, we begin by discussing similarities between machine learning and classical methods from system identification. These techniques are already central in control design and provide context for machine learning control. Next, we introduce the evolutionary approaches of genetic algorithms and genetic programming. Genetic programming is particularly promising for machine learning control because of its generality in optimizing both the structure and parameters associated with a controller. Finally, we provide a brief overview of other promising methods from machine learning that may benefit future MLC efforts.

2.1.1 System identification as machine learning

Classical system identification may be considered an early form of machine learning, where a dynamical system is characterized through training data. The resulting models approximate the input–output dynamics of the true system and may be used to design controllers with the methods described in Chapter 3. The majority of methods in system identification are formulated for linear systems and provide models of dubious quality for systems with strongly nonlinear dynamics. There are, however, extensions to linear parameter varying (LPV) systems, where the linear system depends on a time-varying parameter [246, 16].

There is an expansive literature on system identification, with many techniques having been developed to characterize aerospace systems during the 1960s to the 1980s [150, 174]. The eigensystem realization algorithm (ERA) [151, 181] and observer Kalman filter identification (OKID) [152, 213] techniques build input–output models using time-series data from a dynamical systems; they will be discussed more in Chapter 3. These methods are based on time-delay coordinates, which are reminiscent of the Takens embedding [259]. The singular spectrum analysiss (SSA) from climate science [38, 37, 36, 7] provides a similar characterization of a time-series but without generating input–output models. Recently SSA has been extended in the nonlinear Laplacian spectral analysis (NLSA) [117], which includes kernel methods from machine learning.

The dynamic mode decomposition (DMD) [237, 228, 269] is a promising new technique for system identification that has strong connections to nonlinear dynamical systems through Koopman spectral analysis [162, 163, 187, 47, 188, 170, 41]. DMD has recently been extended to include sparse measurements [45] and inputs and control [216]. The DMD method has been applied to numerous problems beyond fluid dynamics [237, 228], where it originated, including epidemiology [217], video processing [124, 99], robotics [25], and neuroscience [40]. Other prominent methods include the autoregressive moving average (ARMA) models and extensions.

Decreasing the amount of data required for the training and execution of a model is often important when a fast prediction or decision is required, as in turbulence control. Compressed sensing and machine learning have already been combined to achieve *sparse decision making* [278, 168, 46, 39, 215], which may dramatically reduce the latency in a control decision. Many of these methods combine system identification with clustering techniques, which are a cornerstone of machine learning. Cluster-based reduced-order models (CROMs) are especially promising and have recently been developed in fluids [154], building on cluster analysis [49] and transition matrix models [240].

In the traditional framework, machine learning has been employed to model the input–output characteristics of a system. Controllers are then designed based on these models using traditional techniques. Machine learning control circumvents this process and instead directly learns effective control laws without the need for a model of the system.

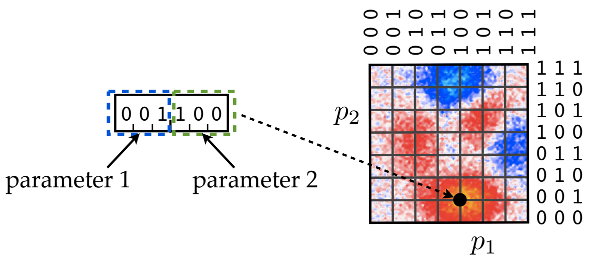


Fig. 2.2 Representation of an individual (parameter) in genetic algorithms. This binary representation encodes two parameters that are each represented with a 3-bit binary expression. Each parameter value has an associated cost (right), with red indicating the lowest cost solution. Modified from Brunton and Noack, *Applied Mechanics Reviews*, 2015 [43].

2.1.2 Genetic algorithms

Evolutionary algorithms form an important category of machine learning techniques that adapt and optimize through a process mimicking natural selection. A population of individuals, called a generation, compete at a given task with a well-defined cost function, and there are rules to propagate successful strategies to the next generation. In many tasks, the search space is exceedingly large and there may be multiple extrema so that gradient search algorithms yield sub-optimal results. Combining gradient search with Monte Carlo may improve the quality of the solution, but this is extremely expensive. Evolutionary algorithms provide an effective alternative search strategy to find nearly optimal solutions in a high-dimensional search space.

Genetic algorithms (GA) are a type of evolutionary algorithm that are used to identify and optimize parameters of an input–output map [137, 76, 122]. In contrast, genetic programming (GP), which is discussed in the next section, is used to optimize both the structure and parameters of the mapping [164, 166]. Genetic algorithms and genetic programming are both based on the propagation of generations of individuals based on selection through fitness. The individuals that comprise a generation are initially populated randomly and each individual is evaluated and their performance assessed based on the evaluated cost function. An individual in a genetic algorithm corresponds to a set of parameter values in a parameterized model to be optimized; this parameterization is shown in Fig. 2.2. In genetic programming, the individual corresponds to both the structure of the control law and the specific parameters, as discussed in the next section.

After the initial generation is populated with individuals, each is evaluated and assigned a fitness based on their performance on the cost function metric. Individuals with a lower cost solution have a higher fitness and are more likely to advance to the next generation. There are a set of rules, or genetic operations, that determine how successful individuals advance to the next generation:

Elitism: a handful of the most fit individuals advance directly to the next generation. Elitism guarantees that the top individuals from each generation do not degrade in the absence of noise.

Replication: individuals advance directly to the next generation with a probability related to fitness; also called asexual reproduction in genetic programming.

Crossover: two individuals are selected based on their fitness and random sections of their parameters are exchanged. These two individuals advance with the exchanged information.

Mutation: individuals advance with random portions of their parameter representation replaced with random new values.

Mutation serves to explore the search space, providing access to global minima, while crossover serves to exploit successful structures and optimize locally. Successful individuals from each generation advance to the next generation through these four genetic operations. New individuals may be added in each generation for variety. This is depicted schematically for the genetic algorithm in Fig. 2.3. Generations are evolved until the performance converges to a desired stopping criterion.

Evolutionary algorithms are not guaranteed to converge to global minima. However, they have been successful in many diverse applications. It is possible to improve the performance of evolutionary algorithms by tuning the number of individuals in a generation, the number of generations, and the relative probability of each genetic operation. In the context of control, genetic algorithms are used to tune the parameters of a control law with a predefined structure. For example, GA may be used to tune the gains of a proportional-integral-derivative (PID) control law [270].

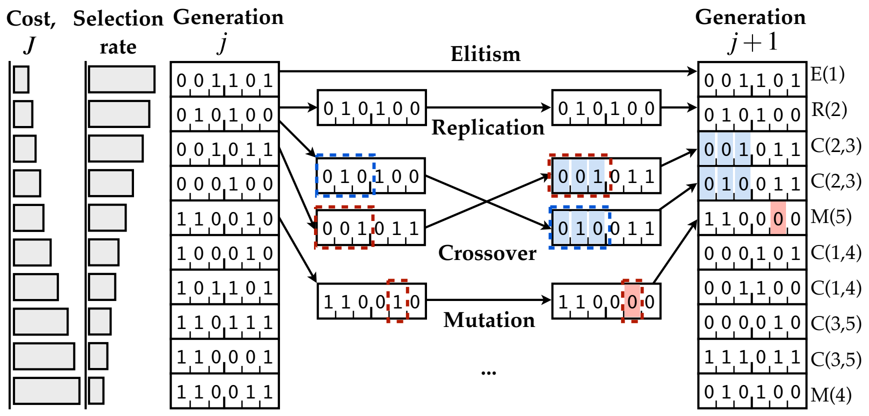


Fig. 2.3 Genetic operations to advance one generation of parameters to the next in a genetic algorithm. The probability of an individual from generation j being selected for generation $j+1$ is related inversely to the cost function associated with that individual. The genetic operations are elitism, replication, crossover, and mutation. Modified from Brunton and Noack, *Applied Mechanics Reviews*, 2015 [43].

2.1.3 Genetic programming

Genetic programming (GP) [166, 164] is an evolutionary algorithm that optimizes both the structure and parameters of an input–output map. In the next section, GP will be used to iteratively learn and refine control laws, which may be viewed as nonlinear mappings from the outputs of a dynamical system (sensors) to the inputs of the system (actuators) to minimize a given cost function associated with the control task.

The input–output mapping discovered by genetic programming is represented as a recursive function tree, as shown in Fig. 2.4 for the case of a control law $b = \mathbf{K}(\mathbf{s})$. In this representation, the root of the tree is the output variable, each branching point is a mathematical operation, such as $+, -, \times, /$, and each branch may contain additional functions. The leaves of the tree are the inputs and constants. In the case of MLC the inputs are sensor measurements and the root is the actuation signal.

Genetic programming uses the same evolutionary operations to advance individuals across generations that are used in genetic algorithms. The operations of replication, crossover, and mutation are depicted schematically in Fig. 2.5 for genetic programming. As in other evolutionary algorithms, the selection probability of each genetic operation is chosen to optimize the balance between exploration of new structures and exploitation of successful structures.

In the sections and chapters that follow, we will explore the use of genetic programming for closed-loop feedback control. In particular, we will show that using genetic programming for machine learning control results in robust turbulence control in extremely nonlinear systems where traditional control methodologies typi-

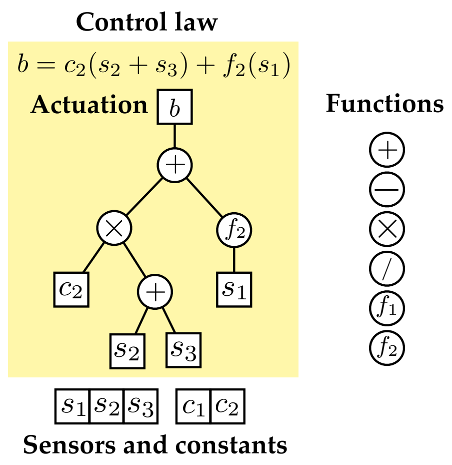


Fig. 2.4 Individual function tree representation used in genetic programming. Modified from Brunton and Noack, *Applied Mechanics Reviews*, 2015 [43].

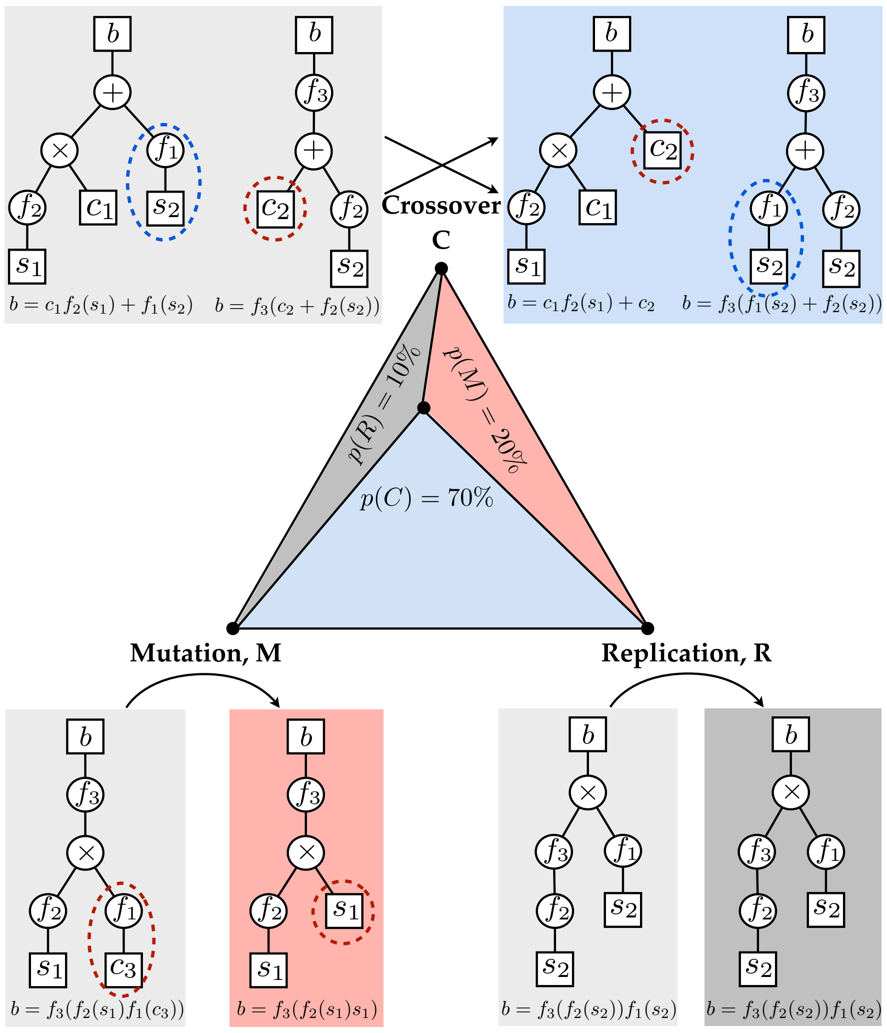

Crossover
C
$p(R) = 10\%$
$p(M) = 20\%$
$p(C) = 70\%$
Mutation, M
Replication, R
$b = c_1 f_2(s_1) + f_1(s_2)$
$b = f_3(c_2 + f_2(s_2))$
$b = c_1 f_2(s_1) + c_2$
$b = f_3(f_1(s_2) + f_2(s_2))$
$b = f_3(f_2(s_1) f_1(c_3))$
$b = f_3(f_2(s_1) s_1)$
$b = f_3(f_2(s_2)) f_1(s_2)$
$b = f_3(f_2(s_2)) f_1(s_2)$

Fig. 2.5 Genetic operations to advance one generation of function trees to the next generation in a genetic program. The operations that advance individuals from one generation to the next are crossover, mutation, and replication. In crossover, random branches from two individual expressions are exchanged. In mutation, a branch is randomly selected and replaced with another randomly generated branch. In replication, the individual is copied directly to the next generation. Modified from Brunton and Noack, *Applied Mechanics Reviews*, 2015 [43].

cally fail. We will also generalize the inputs to include time-delay coordinates on the sensor measurements and generalize the function operations to include filter functions to emulate dynamic state estimation.

2.1.4 Additional machine learning methods

There is a vast and growing literature on machine learning. This section only provides a brief overview of some of the most promising methods that may be used in machine learning control efforts. More extensive treatments may be found in a number of excellent texts [92, 30, 194, 168]. In addition, there is a good overview of the top ten methods in data mining [279].

The presentation on genetic programming above provides a high-level overview of the method, which may be directly implemented for machine learning control. In reality, there is an entire field of research expanding and developing these methods. Genetic programming has been used to discover governing equations and dynamical systems directly from measurement data [32, 239, 219]. There are variants on genetic programming using the elastic net [185], which result in fast function identification. It is also possible to use sparse regression [265, 146] to identify nonlinear dynamical systems from data [44]; this method is related to the recent use of compressed sensing for dynamical system identification [274].

Another promising field of research involves artificial neural networks (ANNs). ANNs are designed to mimic the abstraction capabilities and adaptability found in animal brains. A number of individual computational units, or *neurons*, are connected in a graph structure, which is then optimized to fit mappings from inputs to outputs. It is possible to train the network with stimulus by modifying connections strengths according to either supervised or unsupervised reinforcement learning. There are many approaches to modify network weights, although gradient search algorithms are quite common [61, 129]. Neural networks have been used in numerous applications, including to model and control turbulent fluids [171, 193, 189, 95]. Network-theoretic tools have been applied more generally in fluid modeling recently [154, 195]. ANNs have also been trained to perform principal components analysis (PCA), also known as proper orthogonal decomposition (POD) [203], as well as nonlinear extensions of PCA [204, 157].

Neural networks have proven quite adaptable and may be trained to approximate most input–output functions to arbitrary precision with enough layers and enough training. However, these models are prone to overfitting and require significant amounts of training data. Recently, neural networks have seen a resurgence in research and development with the associated field of deep learning [69, 78, 132]. These algorithms have shown unparalleled performance on challenging tasks, such as image classification, leveraging large data sets collected by corporations such as Google, etc. This is a promising area of research for any data-rich field, such as turbulence modeling and control, which generates tremendous amounts of data.

There are many other important machine learning algorithms. Support vector machines (SVMs) [252, 241, 258] are widely used because of their accuracy, simple geometric interpretation, and favorable scaling to systems with high-dimensional input spaces. Decision trees [221] are also frequently used for classification in machine learning; these classifications are based on a tree-like set of decisions, providing simple and interpretable models. Multiple decision tree models may be combined, or *bagged*, resulting in a random forest model [33]. Ensemble methods in machine learning, including *bagging* and *boosting*, have been shown to have significantly higher classification accuracy than of an individual classifier [83, 105, 236].

Many of the foundational methods in big data analysis [131] have been applied largely to static data problems in artificial intelligence and machine vision. There is a significant opportunity to leverage these techniques for the modeling and control of dynamical systems. These methods may be used for clustering and categorical decisions, dimensionality reduction and feature extraction, nonlinear regression, and occlusion inference and outlier rejection. Machine learning is having transformative impact across the scientific and engineering disciplines. There is tremendous opportunity ahead to employ machine learning solutions to modern engineering control.

2.2 MLC with genetic programming

Now we use genetic programming (GP) as a search algorithm to find control laws in MLC. This section provides details about GP in a context that is specific to control.

2.2.1 Control problem

Before applying any machine learning, it is necessary to pose the control problem as a well-defined cost minimization. In particular, the performance of a given control law is judged based on the value of a cost function J, and the machine learning algorithms will serve to minimize this cost.

There are many ways to formulate a cost function in order to promote different control objectives. In fact, cost function optimization is the basis of most modern control approaches [93, 251], which will be discussed more in Chapter 3. Consider a simplified cost function that depends on the state $\mathbf{a}$ and the actuation $\mathbf{b}$:

$$J(\mathbf{a},\mathbf{b}). \tag{2.1}$$

We often assume that the effects of the state and actuation on the cost are separable:

$$J(\mathbf{a},\mathbf{b}) = J_a + \gamma J_b, \tag{2.2}$$

where J_a is a performance measure on the state of the system and J_b is a value associated with the cost of actuation. The *penalization parameter* γ provides an explicit tuning knob to give priority to either the actuation cost (γ large) or the state cost (γ small). More objectives can be added to the cost function J by including norms on the various transfer functions from inputs to outputs; these may promote good reference tracking, disturbance rejection, noise attenuation, robustness, etc., and a good treatment of these choices is provided in [251]. In addition, the complexity of the controller $\mathbf{K}$ may be penalized to avoid overfitting.

2.2.2 Parameterization of the control law

When using genetic programming for MLC, the control laws are represented as recursive expression trees (also known as *function trees*), as shown in Fig. 2.6. These control laws are the *individuals* that will populate a generation in genetic programming. Expression trees are usually built from a number of elementary functions which can take any number of arguments but return a single value; example function nodes are $+, -, \times, /, \sin, \tanh, \ldots$ The arguments of these functions may be leaves or subtrees. In MLC, the root of the tree is the actuation signal b and the leaves are components of the sensor vector $\mathbf{s}$ or constants.

The tree in Fig. 2.6 represents the function:

$$b(s_1, s_2) = \cos(s_1) + \tanh((s_1 \times s_2)/0.21) - 2.32, \tag{2.3}$$

where s_1 and s_2 are the time-varying sensor values. It is useful to represent the function tree as a LISP expression.

Several representations can be used to manipulate functions inside the genetic programming architecture (e.g. trees, linear programming). We choose to use a tree-like representation. Two main advantages of this representation are that expression trees are readily interpretable, and they are easily synthesized and manipulated computationally using a recursive language such as LISP or Scheme.

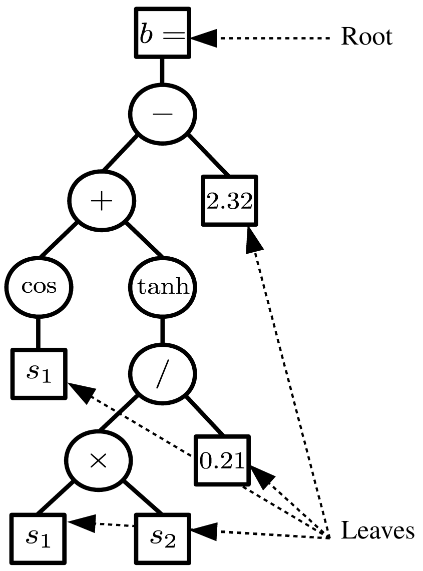


Fig. 2.6 An expression tree representing the controller function given by: $b = \mathbf{K}(\mathbf{s}) = \cos(s_1) + \tanh((s_1 \times s_2)/0.21) - 2.32$.

Equation (2.3) can be written as the LISP string in parenthesized Polish prefix notation:

$$(-\ (+(\cos\ s_1)\ (\tanh\ (/\ (\times\ s_1\ s_2)\ 0.21)))\ 2.32).$$

Though less readable than the expression tree, recursive algorithms can generate, derive, manipulate, and evaluate these expressions. The generation process of any individual control law starts at the root. Then a first element is chosen from the pool of admissible basis functions and operators. If a basis function or operator is chosen, new elements are added as arguments, and the process is iterated to include their arguments until all branches have leaves.

This process of individual generation is subject to limitations. A given tree-depth (the maximum distance between any leaf and the root) can be prescribed by preventing the last branch to generate a leaf before the aforementioned tree-depth is reached and by enforcing the termination of the branch in leaves when the tree-depth is reached. Similarly it is possible to ensure that each branch reaches the same given tree-depth which generates a full-density tree with the maximum number of operations. MLC can implement any of these distributions, from fully random trees to a given tree-depth distribution with a specified proportion of dense and less dense

Advanced material 2.1 Operation protection.

It is important to note that not all functions are defined for all real-valued arguments. For instance, the second argument of the division operator must not be zero and the argument of the logarithmic functions must be strictly positive. Thus, such functions must be protected. When the translation from LISP is achieved using the `OpenMLC` toolbox through the `readmylisp_to_formal_MLC.m` function, '(/ arg1 arg2)' is interpreted as 'my_div(arg1,arg2)', where `my_div.m` defines the function:

$$\begin{aligned}
\text{my\_div}(arg1,arg2) &= \frac{arg1}{arg2}, && \text{if } |arg2| > 10^{-3}\\
&= \frac{arg2}{|arg2|}\frac{arg1}{10^{-3}}, && \text{if } 0 < |arg2| < 10^{-3}\\
&= \frac{arg1}{10^{-3}}, && \text{if } |arg2| = 0.
\end{aligned}$$

Similarly, '(log arg1)' is interpreted as 'my_log(arg1)', where `my_log.m` defines the function:

$$\begin{aligned}
\text{my\_log}(arg1) &= \frac{arg1}{|arg1|}\log(|arg1|), && \text{if } |arg1| > 10^{-3}\\
&= \frac{arg1}{|arg1|}\log(10^{-3}), && \text{if } 0 < |arg1| < 10^{-3}\\
&= 0, && \text{if } |arg2| = 0.
\end{aligned}$$

In case the interpretation of the LISP expression is carried out by another function, for example on a real-time processing unit, these protections have to be implemented in order to avoid unexpected behaviors. These protections can be easily changed by editing both surrogate functions, or by defining another function to be called in the `parameters.opset` structure.

individuals. The first generation starts with a distribution of rather low tree-depth (2 to 8) and an equal distribution (1:1) of dense and less dense individuals in the default parameters of `OpenMLC`. This choice generally ensures enough diversity for the creation of subsequent generations. The initially low tree-depth takes into account that the genetic operations (see Sec. 2.2.7) have a tendency to make the trees grow. This phenomenon is known as bloating of the trees. To enforce more diversity in the population, any candidate individual is discarded if it already exists in the current population.

2.2.3 Genetic programming as a search algorithm

The flowchart for genetic programming is given in Fig. 2.7. An initial set (*generation*) of N_i control laws (*individuals*) is evaluated according to the cost function J. Next, successful individuals are selected to advance to the next generation and are evolved by genetic operations: elitism, replication, crossover and mutation. This procedure is repeated until a convergence or stopping criterion is met.

The implementation of genetic programming as a search algorithm for MLC is shown in Fig. 2.8. In this schematic, control laws are expression trees that take sensor outputs $\mathbf{s}$ of a dynamical system and synthesize actuation inputs $\mathbf{b}$. These controller individuals are optimized through the genetic programming algorithm.

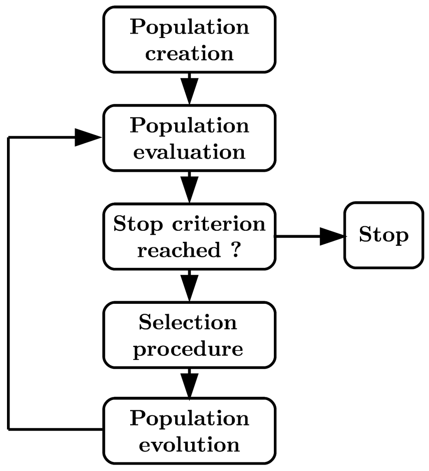


Fig. 2.7 Flowchart for the genetic programming algorithm.

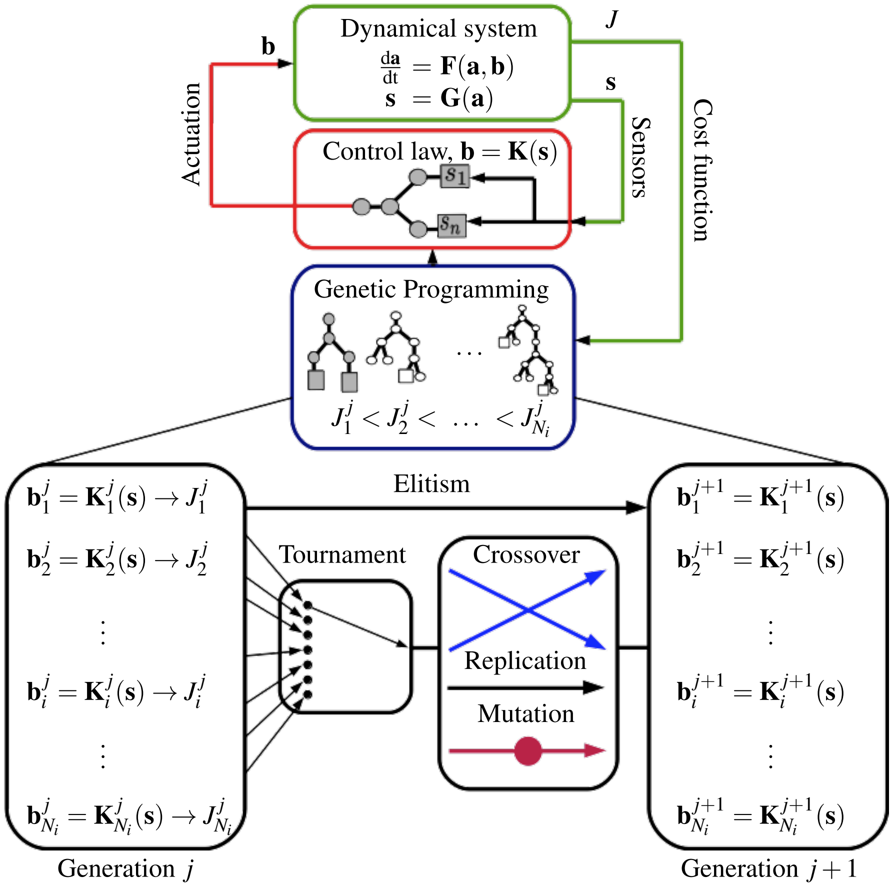

Dynamical system
$\frac{d\mathbf{a}}{dt} = \mathbf{F}(\mathbf{a},\mathbf{b})$
$\mathbf{s} = \mathbf{G}(\mathbf{a})$
J
$\mathbf{s}$
$\mathbf{b}$
Actuation
Sensors
Cost function
Control law, $\mathbf{b} = \mathbf{K}(\mathbf{s})$
s_1
s_n
Genetic Programming
$J_1^j < J_2^j < \dots < J_{N_i}^j$
$\mathbf{b}_1^j = \mathbf{K}_1^j(\mathbf{s}) \to J_1^j$
$\mathbf{b}_2^j = \mathbf{K}_2^j(\mathbf{s}) \to J_2^j$
$\mathbf{b}_i^j = \mathbf{K}_i^j(\mathbf{s}) \to J_i^j$
$\mathbf{b}_{N_i}^j = \mathbf{K}_{N_i}^j(\mathbf{s}) \to J_{N_i}^j$
Elitism
Tournament
Crossover
Replication
Mutation
$\mathbf{b}_1^{j+1} = \mathbf{K}_1^{j+1}(\mathbf{s})$
$\mathbf{b}_2^{j+1} = \mathbf{K}_2^{j+1}(\mathbf{s})$
$\mathbf{b}_i^{j+1} = \mathbf{K}_i^{j+1}(\mathbf{s})$
$\mathbf{b}_{N_i}^{j+1} = \mathbf{K}_{N_i}^{j+1}(\mathbf{s})$
Generation j
Generation $j+1$

Fig. 2.8 Model-free control design using GP for MLC. During the learning phase, each control law candidate is evaluated by the dynamical system or experimental plant. This process is iterated over many generations of individuals. At convergence, the best individual with the smallest cost function value (in grey in the GP box) is used for real-time control.

2.2.4 Initializing a generation

In genetic programming an entire population of individuals forms a generation, and these individuals compete to advance to future generations. A population contains N_i individuals that must be initialized. To form an expression tree, the initialization algorithm works from the root to the trees. The principle is to work from one seed (a marker which indicates where the tree is supposed to grow) or N_b seeds (if the actuation input **b** has multiple components), decide on a node (function/operation, or leaf), add as many new seeds as necessary (if this is an operation or function, include as many seeds as arguments) and recursively call the function that grows the tree until all new seeds have been generated. The process stops once all seeds have been replaced by leaves. Those leaves are chosen between randomly generated constants and one of the N_s sensors in $\mathbf{s}$. This algorithm can be configured so that some statistical properties of the population can be specified:

Tree-depth: If an operation or a leaf is selected through a probabilistic process, it is possible to prescribe the minimal and maximal tree-depth of the trees by forcing or forbidding the creation of leaves according to the tree-depth and/or the existence of other leaves at said depth.

Density: It is possible to prescribe a specific tree depth for all leaves. This is achieved by forbidding any leaf before a given depth, and forcing leaf selection at that depth. The resulting individuals possess the maximal number of operations possible (modulo the number of arguments of the selected operations) for the prescribed tree depth. Such an individual is referred to as a full (density) individual.

The initial population corresponds to the first exploration of the search space. As such, this population has a crucial impact on the following steps of the search process: future generations will converge around the local minima found in the initial generation. Therefore, it is important that this population contains as much diversity as possible. A first measure is to reject any duplicate individual in the population. Diversity is also enforced using a Gaussian distribution of the tree-depth (between 2 and 8 by default in `OpenMLC`), with half the individuals having full density.

Advanced material 2.2 Creation algorithm pseudo-code.

The function that generates expression trees as LISP expressions in `OpenMLC` is *generate_indiv_regressive.m*. As any expression tree manipulation function, the creation algorithm is auto-recursive. The principle is to work from a seed, decide on a node (function/operation, or leaf), add as many seeds as necessary (if this is an operation or function) and call the function back as many time as new seeds have been generated:

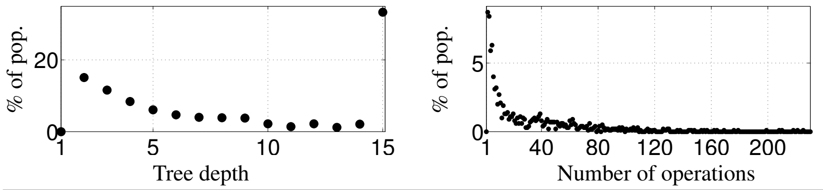
```
1 : new_LISP=grow_tree(old_LISP,parameters) % declaration
2 : find first seed in old_LISP, so that old_LISP=‘part1 seed part2’
3 : decide if the next node is an operation or a leaf
    if it is a leaf: replace the seed with either a constant, either a sensor
                     and return new_LISP=‘part1 leaf part2’
    if it is an operation: choose one randomly (called ‘op’) and replace seed so that:
    new_LISP=‘part1 (op seed) part2’ if ‘op’ takes one argument
    new_LISP=‘part1 (op seed seed) part2’ if ‘op’ takes two arguments
4 : recursively call back new_LISP=grow_tree(new_LISP,parameters) as many times as
    new seeds have been added.
5 : return new_LISP
```

An expression tree with N_b independent subtrees (corresponding to N_b actuation inputs) can be created using the tree growing function on ‘(root seed repeated N_b times)’.

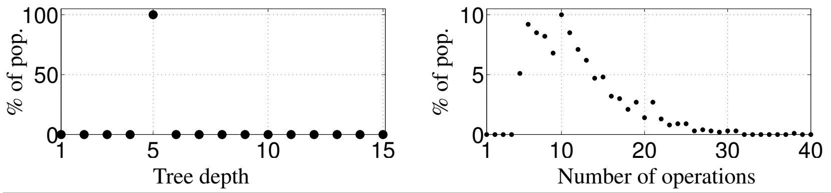
Advanced material 2.3 Effect of generation parameters in the first population diversity.

The tree-depth distribution is shown for each initialization method for the first generation (left). The histograms of the number of operations for each tree-depth is also shown (right). The average is marked by a bullet. These graphs are obtained with one instance of a population generation with default `OpenMLC` object properties except for the initialization method which is indicated in each graph. One indicator of diversity is the distribution of operations in the trees. If multiplying the different tree depth is a good factor to enforce diversity, the natural growth of the average tree-depth as new generations are evolved (a phenomenon known as *bloat*) indicates that it is better to keep low tree-depth in the first generation.

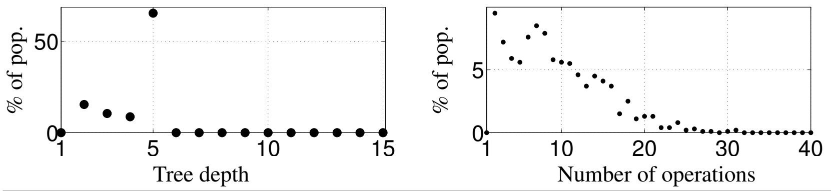
'random_maxdepth'



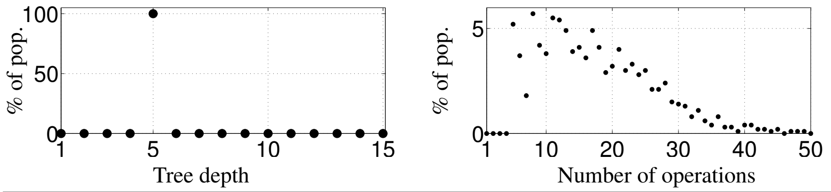
'fixed_maxdepthfirst'



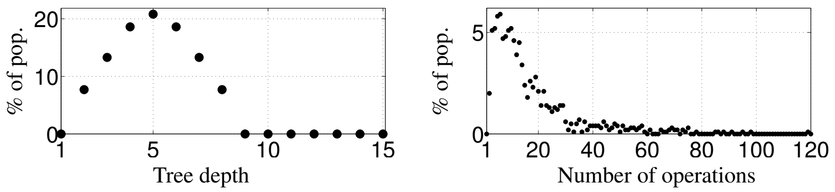
'random_maxdepthfirst'



'full_maxdepthfirst'



'mixed_ramped_gauss'



2.2.5 Evaluating a generation

After every new generation is formed, each individual must be evaluated based on their performance with respect to the regression problem. The value of the cost function J in Eq. (2.1) is computed for each individual. In MLC, this evaluation corresponds to the individual being used as the control law for the dynamical system under consideration. This evaluation results in a single value J, which is the result of Eq. (2.1) for the specific individual being evaluated.

A major complication encountered with experiments (or even some noisy numerical systems) is the fact that, contrary to deterministic systems, the re-evaluation of one individual does not necessarily return the same cost function value as a previous evaluation. A large error on the value, either due to a measurement error or an exceptional performance of a non-robust control law can lead to a non-representative grading of the control law. As explained throughout the book, the search space is primarily explored around the best-performing individuals. If an intrinsically low-performing individual gets a mistakenly good evaluation, this may be a significant setback. Thus, all individuals are evaluated even if they have already been evaluated in a previous generation, and the best individuals undergo a re-evaluation. By default in `OpenMLC`, the five best individuals are each re-evaluated five times, and their cost function is averaged. This procedure ensures that the best performing individuals are more carefully ranked so that the search process is not misdirected.

2.2.6 Selecting individuals for genetic operations

After the evaluation of each individual, the population evolution starts. In order to fill the next generation, genetic operations (see Sec. 2.2.7) are performed on selected individuals. The selection procedure is at the heart of any evolutionary algorithm as it determines the genetic content of the following generation. The selection process employed by default is a tournament. Each time an individual needs to be selected for a genetic operation, N_p unique individuals are randomly chosen from the previous generation to enter the one-round tournament. From this set of individuals, the one with the smallest cost function value is selected. As the population size N_i is fixed, N_i selection tournaments are run each time a new generation is created. The population is ranked by decreasing cost function value. If we discard uniqueness of the individuals and consider $N_i \gg N_p > 1$, the probability of individual i winning a tournament is $((N_i - i)/(N_i - 1))^{N_p - 1}$. On average, each individual will enter N_p tournaments. Each individual is sorted by their ranking, so that individual i has the i-th lowest cost function value; we define $x = i/N_i$ for each individual so that $x \in [0, 1]$), where $x = 0$ is the best individual and $x = 1$ is the worst individual. If an individual is ranked $i = x \times N_i$, then its genetic content will contribute on average roughly to $N_p \times (1 - x)^{N_p - 1}$ new individuals. A standard selection parameter sets $N_p = 7$ and ensures that only the first half of the ranked generation contributes consistently to the next generation. Lower-performing individuals can still contribute, but these are rare events. In this selection procedure, choosing N_p sets the harshness of the selection. On the other hand, this selection procedure does not take into account the global distribution of the cost function values, as in other selection processes such as a fitness proportional selection. If an individual performs much better than the rest of the population it will not have a better selection chance than an individual that barely performs better than the rest of the population. This choice ensures that the diversity in the population does not disappear too fast with an associated trade-off in convergence speed.

2.2.7 Selecting genetic operations

Four kinds of genetic operations are implemented: *elitism*, *replication*, *mutation* and *crossover*.

Elitism: The N_e best individuals of the evaluated population are copied directly to the next generation. This operation does not go through a selection procedure and ensures that the best control laws stay in the population. Once the elitism process is finished, $N_i - N_e$ individuals have to be generated through replication, mutation and crossover. The probability of each of these operations are P_r, P_m and P_c, respectively, with $P_r + P_m + P_c = 1$.

Replication: The selected individual is copied to the next generation.

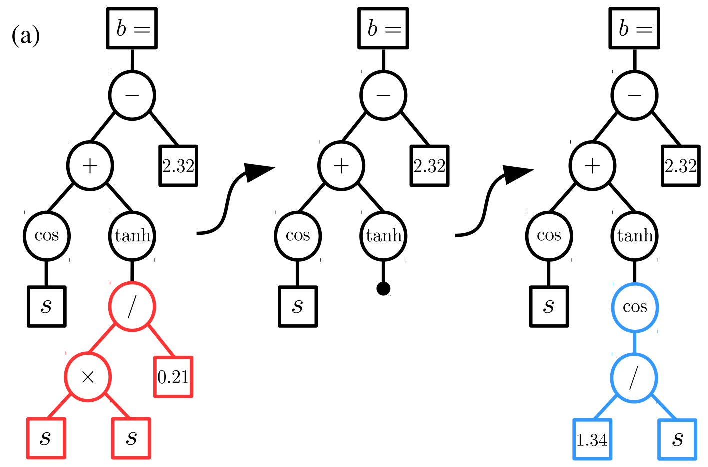
Mutation: There are four mutation operations: *Cut and grow* replaces an arbitrarily chosen subtree by a randomly generated new subtree. For that, the same procedure is used as in the creation of the first generation. *Shrink* replaces an entire randomly chosen subtree by a randomly chosen leaf (constant or sensor). *Hoist* replaces the tree by a randomly chosen subtree. If this tree corresponds to a MIMO control law with N_b actuation inputs, each control law has a $1/N_b$ chance to be mutated if the hoist mutation is chosen. *Reparametrization* sets a

Advanced material 2.4 Fitness proportional selection.

Fitness proportional selection is another popular process for selecting individuals for genetic operations. The inverse of an individual's cost $J_i = J(K_i)$ is a natural measure of its desirability. It goes to infinity as the optimal value zero is reached. The probability of the section of the ith individual is set proportional to this desirability

$$P_i = \frac{J_i^{-1}}{\sum_{j=1}^{N_i} J_j^{-1}}. \qquad (2.4)$$

If one individual performs much better than the rest of the population, it will be selected more often in the same proportion, thus encouraging optimization around the best performing individuals while still allowing sub-optimal individuals to be selected. However, diversity can disappear rapidly if an individual performs one or several orders of magnitude better than the rest of the population. This is an undesirable feature which we avoid by a selection based on the relative ordering.



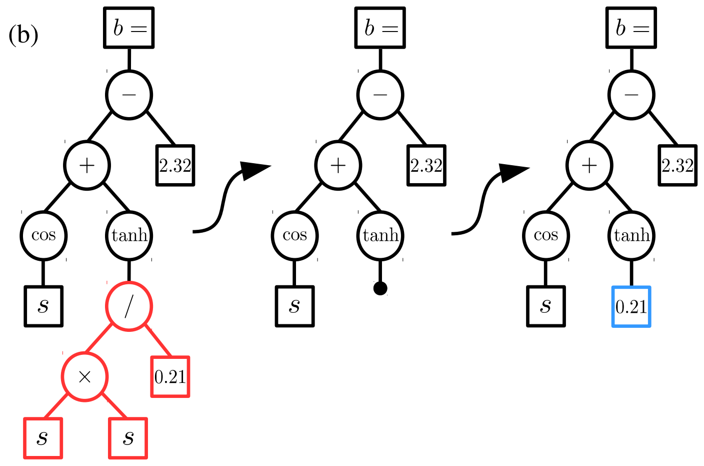
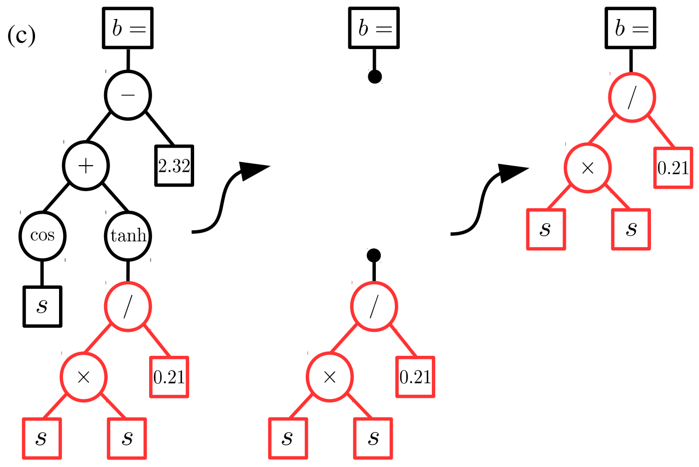


Fig. 2.9 Two of the four types of mutations implemented in MLC: a) cut and grow, b) shrink.

50% chance for each constant to be randomly replaced with a new value. All of these mutations are displayed in Figs. 2.9 and 2.10.

Crossover: uses two selected individuals and exchanges one randomly chosen subtree between them (see Fig. 2.11).



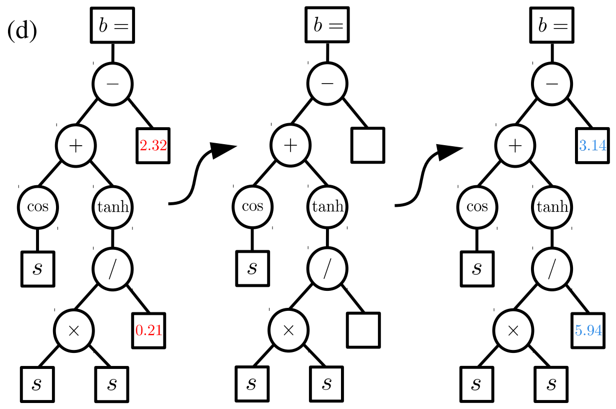


Fig. 2.10 Two of the four types of mutations implemeted in MLC: c) Hoist and d) re-parameterization.

Replication ensures some stability of the convergence process: it guarantees that a part of the population stays in the vicinity of explored local minima of the search space, keeping potentially useful mechanisms in the population and further exploiting them before they are discarded. Crossover and mutation are responsible for the exploitation and exploration of the search space, respectively. As we progress among the generations, the probability to cross similar individuals increases: the

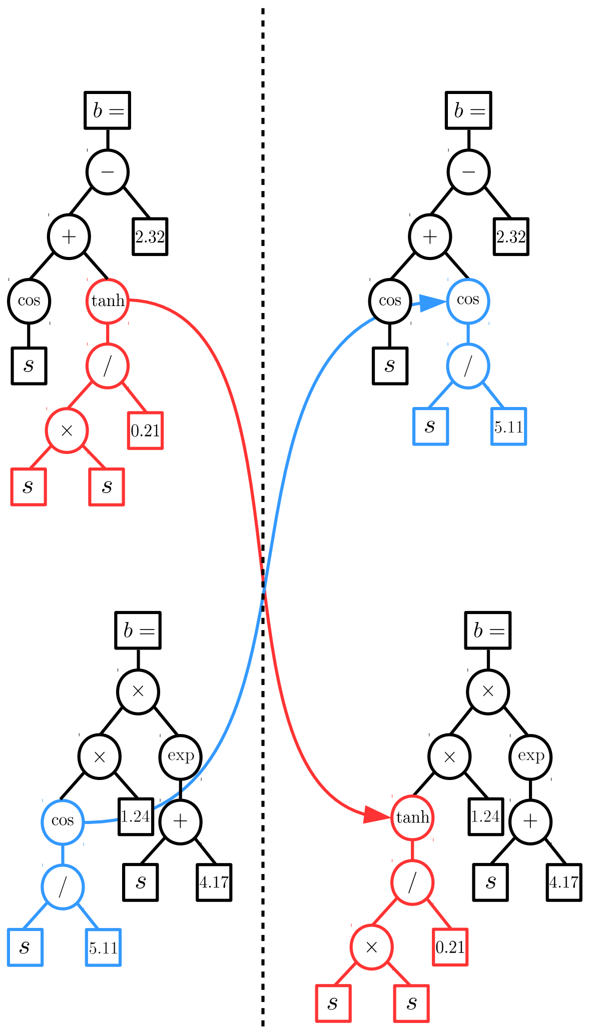


Fig. 2.11 Crossover example. The selected individuals of a considered generation (left) exchange subtrees to form a new pair in the next generation (right).

best individual will propagate its genetic content N_p times on average. If this successful genetic content allows the concerned individuals to stay in the first positions of the ranking, it will be replicated about $N_p^k \times P_c$ times after k generations. Then crossovers of individuals selected in the top of the ranking will soon cross similar individuals and explore the vicinity of the dominant genetic content. On the other hand, mutations introduce new genetic content in the population, hence allowing large-scale exploration of the search space. Fig. 2.12 illustrates how an evolution-

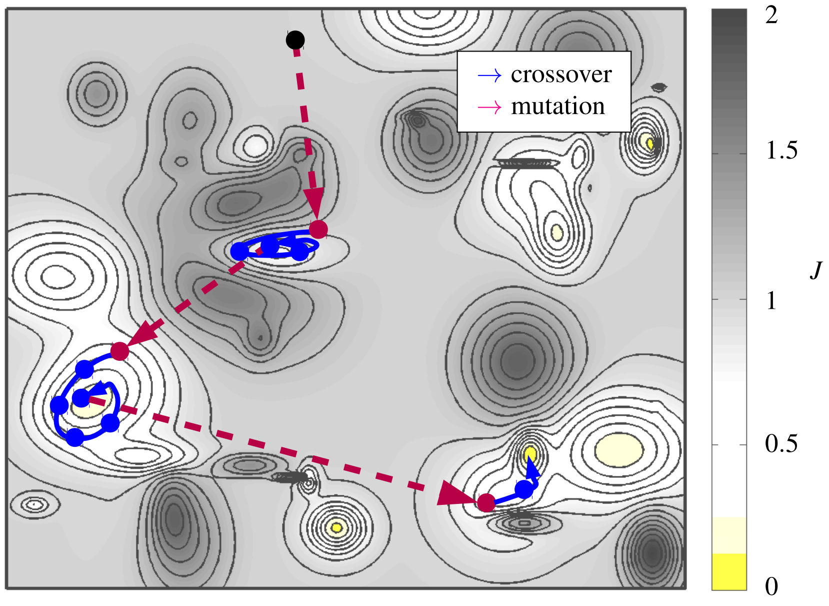


Fig. 2.12 Conceptual 2-dimensional representation of the search process of evolutionary algorithms. The level curves display the geometry of the cost function and its local minima and maxima. The arrows follow one branch of the lineage of one individual (displayed by a black point) that leads to the global minimum of the cost function. The exploitation of the local minima is achieved by crossovers while the large-scale exploration of the search space is achieved by the mutations.

ary algorithm explores the search space of a two-dimensional problem with local minima: the association of these operations enables exploitation around the local minima while still exploring the search space for better solutions.

2.2.8 Advancing generations and stopping criteria

There are no fool-proof general rules to choose optimal parameters for evolutionary algorithms. A common practice is to check the optimality of the solution offered by genetic programming by reproducing the process a number of times using different sets of parameters. This way, guiding statistics can be obtained. In the case of experimental turbulence control, one is more interested in finding an effective control law than in determining an optimal search algorithm. The main impact of modifying parameters is on the ratio between exploitation (i.e. convergence) and exploration of the search space. Monitoring the evolution of the evaluated populations is the best way to fine-tune the MLC process. Now, we discuss the role of the MLC parameters:

- Population size N_i: more individuals in the first generation will result in more exploration of the search space. On the other hand, a large initial population requires more evaluation time without any evolutionary convergence. Let us consider 1000 evaluations. If only the first generation is evaluated, then it is equivalent to a Monte Carlo process. Alternatively, one could devote 1000 evaluations to 20 generations with 50 individuals in each generation. This implies 20 iterative refinements of the best individuals through evolutionary operations.

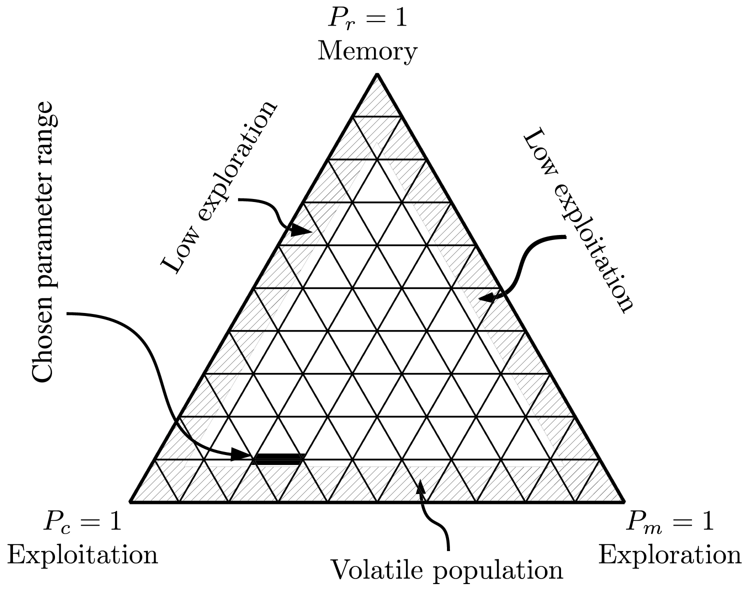


Fig. 2.13 Probability selection for the genetic operations. There is no globally optimal parameter selection. Depending on the problem, diversity or convergence needs to be modified. The range of parameters used in the present study is represented by the black area. Hatched areas represent the parametrical space where essential aspects of the evolutionary algorithm are endangered with volatile population translating in previous winning options to be forgotten once a better solution is found (not enough replications), low convergence (not enough crossovers) or low exploration (not enough mutations).

- Genetic operation probabilities (N_e/N_i, P_r, P_m, P_c): Elitism is encouraged as it ensures that the N_e best performing individuals will remain in the population. Replication, mutation and crossover probabilities parametrize the relative importance between exploration and exploitation (Fig. 2.13). A large mutation rate P_m implies large-scale exploration and thus population diversity. If all individuals in the population share a similar expression and have similar costs then the mutation rate should be increased. On the other hand, the crossover rate will impact the convergence speed. If the individuals are different and the cost function value histogram does not show a peak around the lowest value, then the convergence is not sufficiently rapid and the crossover probability should be increased. Finally, replication ensures that a given search space area will be explored during a certain number of generations. This parameter will at the same time ensure diversity and exploration of different areas.

- The harshness of the selection procedure also influences the diversity of the next generation. The number of individuals N_p that enter a tournament directly influences the number of individuals that will contribute to the next generation. Reducing N_p increases the diversity while increasing it will accelerate the convergence.
- The choice of the elementary functions is intimately linked to the regression problem. This choice should be determined in concordance with the choice of the sensors and actuators.
- The maximum number of generations is eventually determined by the available testing time. A stopping criterion can end the iterations prematurely, for instance if the optimal solution is reached ($J = 0$) or if the average and minimum of the cost function distribution converge.

GP experts usually recommend a high rate of crossover and a low rate of mutation with a large initial population, though specific values depend on the problem. These choices, clearly aimed to obtain a fast convergence, are possible when the evaluations can be parallelized. In experiments, however, the total time to evaluate a generation of individuals is critical and one cannot afford a large population. The convergence of the cost function evaluation is also affected by the measurement error. A good compromise in experiments is to deal with reduced populations (on the order of 50 to 500 individuals) associated with a high mutation rate (from 25 to 50%). It is simpler to keep these values constant during the course of each experiment, but further performance improvement can be achieved by adapting them with respect to the phase (exploration or exploitation) of the learning process.

Throughout the book many examples of applications of MLC to different control problems can be found, from numerical dynamical systems to experiments, and their specific implementation using `OpenMLC` is discussed. Chapter 7 is dedicated to providing the reader with best practices for the use of MLC.

2.3 Examples

As outlined above, Machine Learning Control (MLC) formulates control design as a regression problem: Find a law which minimizes a given cost function. Genetic Programming (GP) is a regression tool and key enabler of MLC. In the following, we illustrate genetic programming for a simple two-dimensional data fit (Sec. 2.3.1) and and a control problem (Sec. 2.3.2). We ease the replication of the results by employing `OpenMLC`, a freely available Matlab® toolbox designed for MLC (see the Appendix for instructions about how to download and install). `OpenMLC` has been used for most numerical and experimental control problems of this book.

2.3.1 Fitting a function through data points

In the first example, we search for a one-dimensional function passing as close as possible to given data points.

Problem formulation

Let us consider the 201 points in the plane:

$$s_i = i/10 \tag{2.5a}$$
$$b_i = \tanh(1.256\, s_i) + 1.2, \quad i = -100 \dots 100. \tag{2.5b}$$

The goal is to find a function $s \to b = K(s)$ that goes through these points as close as possible using GP. A canonical cost function reads

$$J = \frac{1}{201} \sum_{i=-100}^{100} \left(b_i - K(s_i)\right)^2. \tag{2.6}$$

We do not assume a particular structure of the function K, like an affine, constant-linear-quadratic or polynomial structure. This excludes the least-mean-square approach for parameter identification.

In the following, we describe how GP solves the regression problem

$$K(s) = \underset{K'(s)}{\operatorname{argmin}}\, J\left[K'(s)\right], \tag{2.7}$$

i.e. finding a function $b = K(s)$ which minimizes the cost function (2.6).

This example is implemented as the default toy problem for `OpenMLC` which will be used throughout the section.

Code 2.1 Creation of the default regression problem.

```
clear all
close all

mlc=MLC % Creates a MLC object with default values that
        % implements the simple regression problem.
```

Problem solution

We take typical parameters for GP. These are also the default parameters of `OpenMLC`.

Table 2.1 Parameters for MLC with genetic programming for example of fitting a function through data points.

Parameter	Value
N_i	1000
P_r	0.1
P_m	0.2
P_c	0.7
N_p	7
N_e	10
Node functions	$+,-,\times,/,\exp,\log,\tanh$

Our 'plant' is then written as follows:

Code 2.2 Evaluation function for the first regression problem (2.7).

```
function J=toy_problem(ind,parameters,i,fig)

%% Creation of the points to fit.
s=-10:0.1:10;
b=tanh(1.256*s)+1.2;

%% Initialisation of b_hat as a vector.
b_hat=b*0;

%% Evaluation.
% Evaluation is always encapsulated in try/catch.
% Structure to account for the unpredictible.

try
   % Translation from LISP.
idv_foraml=readmylisp_to_formal_MLC(ind,parameters);
idv_formal=strrep(m,'S0','s'); % Replace S0 leaf with variable
     s
   % Obtain the estimated s.
eval(['b_hat=' idv_formal ';'])
   % Obtain the cost function value.
J=sum((b-b_hat).^2)/length(b);
catch err
    % If something goes wrong, asign a bad value.
    J=parameters.badvalue;
    fprintf(err.message);
end
```

The code 2.2 comprises all the necessary steps needed in order to build the problem with corresponding cost function in `OpenMLC`. The evaluation function takes an individual

$$\hat{b} = \hat{K}(s) \tag{2.8}$$

as argument and returns a cost function value J. In addition, GP parameters enter as arguments.

One critical step is the translation of the individual from a LISP expression to something Matlab® can use as a function. This is realized through the function `readmylisp_to_formal_MLC` which recursively interprets each element of the tree. Generically all inputs are numbered $S0$, $S1$ to SN_s. Here, only one input is declared for the function that is to be learned. Thus only functions involving $S0$ are created. For the sake of readability, we replace $S0$ by s, so that the strings created are functions of s. Once the individual is interpreted, it can be used to approximate the data with the mapping (2.8) and compute the corresponding cost function.

MLC run and results

Here GP is launched for 50 generations using

```
mlc.go(50);
```

or

```
mlc.go(50,1);
```

to have graphical output. At the end of the 50th generation, typing

```
mlc
```

returns the best individual, its cost function value and other statistics:

```
After generation 50:
Best individual:
(+ (sin (+ (sin (sin (tanh (/ (/ S0 -2.511) (sin -8.741)))))
(sin (tanh (/ (/ S0 -2.629) (sin -8.741)))))) (log (log (+
(tanh (+ (sin (tanh (/ (sin (/ S0 -2.511)) (* -3.802 7.167))))
(tanh (tanh (+ (sin (+ (sin -8.741) (sin -8.741))) (sin 6.774))))))
(* -3.802 7.167)))))

Best J (average over 3 occurance(s)):   2.380030e-06
```

This implies that the returned function is much more complicated than the actual function and produces an average error of 2.38×10^{-6}. Note that we did not penalize complexity of the individuals.

Typing

```
mlc.show_best_indiv;
```

will additionally display the original and learned relationship between the dataset used for regression (Fig. 2.14).

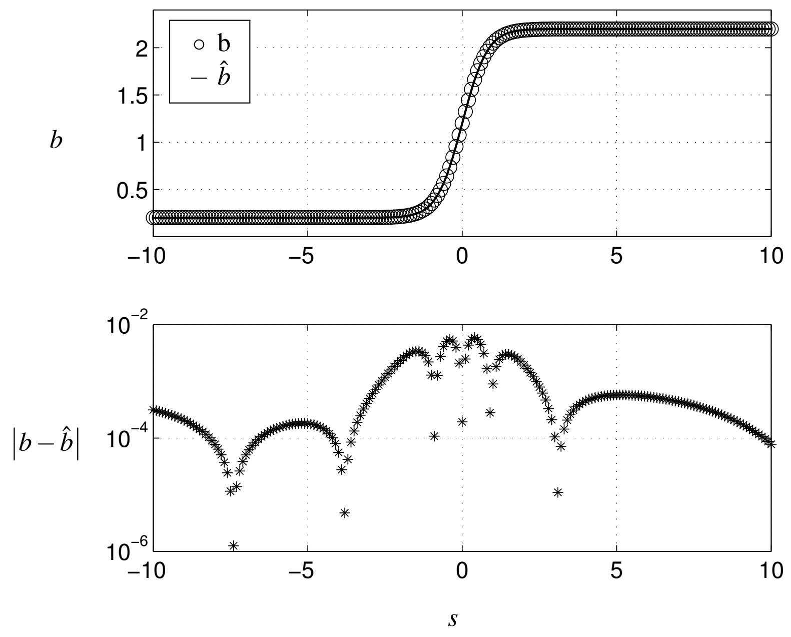


Fig. 2.14 Best individual of the regression problem (2.7) after 50 generations, obtained with '`mlc.show_best_indiv`' on the '`toy_problem`' regression problem.

2.3.2 MLC applied to control a dynamical system

The second study exemplifies MLC for control design of a noise-free, linear, one-dimensional ordinary differential equation, arguably the most simple example of Eq. (1.1).

Problem formulation

We investigate the ordinary differential equation

$$\frac{da}{dt} = a + b, \tag{2.9a}$$

$$s = a, \tag{2.9b}$$

$$b = K(s), \tag{2.9c}$$

with the initial condition

$$a(0) = 1. \tag{2.10}$$

The cost function to be minimized penalizes a deviation from desired state $a_0 = 0$ and actuation,

$$J = \frac{1}{T} \int_0^T \left[a^2 + \gamma b^2\right] dt. \tag{2.11}$$

Here, T is the evaluation time and γ is a penalization coefficient for the cost of actuation.

This problem is set-up in the `OpenMLC` script `unidim_DS_script.m` and the associated object can be instantiated by calling:

```
mlc=MLC('unidim_DS_script');
```

This command implements the new control regression problem

$$K(s) = \underset{K'(s)}{\operatorname{argmin}} J\left[K'(s)\right], \tag{2.12}$$

i.e. finding a control law which minimizes the cost function (2.11).

The search space for the control law contains compositions using operations taken from $(+, -, \times, \tanh)$. This includes arbitrary polynomials and sigmoidal functions. A full list of GP parameters used can be obtained by typing:

```
mlc.paramters
```

The most important ones are:

Table 2.2 Parameters for MLC with genetic programing for simple control design example.

Parameter	Value
N_i	50
P_r	0.1
P_m	0.4
P_c	0.5
N_p	7
N_e	1
Node functions	$+, -, \times, \tanh$

Problem implementation

The evaluation function for this simple dynamical system control problem is provided under the name `unidim_DS_evaluator.m`.

Code 2.3 Evaluation function for control regression problem (2.12).

```
function J=unidim_DS_evaluator(ind,mlc_parameters,i,fig)
%% Obtaining parameters from MLC object.
Tf=mlc_parameters.problem_variables.Tf;
objective=mlc_parameters.problem_variables.objective;
gamma=mlc_parameters.problem_variables.gamma;
Tevmax=mlc_parameters.problem_variables.Tevmax;

%% Interpret individual.
m=readmylisp_to_formal_MLC(ind);
m=strrep(m,'S0','y');
K=@(y)(y);
eval(['K=@(y)(' m ');']);
f=@(t,y)(y+K(y)+testt(toc,Tevmax));

%% Evaluation
try                          % Encapsulation in try/catch.
tic
[T,Y]=ode45(f,[0 Tf],1);   % Integration.
if T(end)==Tf                % Check if Tf is reached.
    b=Y*0+K(Y);              % Computes b.
    Jt=1/Tf*cumtrapz(T,(Y-objective).^2+gamma*b.^2); %
        Computes J.
    J=Jt(end);
else
    J=mlc_parameters.badvalue;   % Return high value if Tf is
        not reached.
end
catch err
   J=mlc_parameters.badvalue     % Return high value if ode45
       fails.
end

if nargin>3   % If a fourth argument is provided, plot the
    result
    subplot(3,1,1)
    plot(T,Y,'-k','linewidth',1.2)
    ylabel('$a$','interpreter','latex','fontsize',20)
    subplot(3,1,2)
    plot(T,b,'-k','linewidth',1.2)
    ylabel('$b$','interpreter','latex','fontsize',20)
    subplot(3,1,3)
    plot(T,Jt,'-k','linewidth',1.2)
    ylabel('$(a-a_0)^2+\gamma b^2$','interpreter','latex','
        fontsize',20)
    xlabel('$t$','interpreter','latex','fontsize',20)
end
```

First of all we retrieve the parameters which are stored in the structure

```
mlc.parameters.problem_variables
```

This structure is empty by default and can be used to specify variables that are specific to the problem. Here, the evaluation time T, the penalization coefficient γ, the desired state $a_0 = 0$ and the integration time T_{max} are available from the structure `problem_variables` and retrieved in the first lines of the evaluation function. This allows one to parametrize the problem, so that running the same problem with different values for the parameters can be implemented in a loop or scripted.

The individual is interpreted as a control law using the `OpenMLC` toolbox function `readmylisp_to_formal_MLC`. This transforms the LISP string to a Matlab® expression string if copied in the console. By default, sensors are named $S0, S1, \ldots, Sn_a$. As in the first example, we replace $S0$ by the sensor symbol s, with a `strrep` (string replacement) parsing. Finally, a symbolic function is created inside a call to `eval` that allows one to use the formal expression in the individual string to define the control function.

The dynamical system is then written as a actuated dynamics F defining the derivative of the state a, including the control law, and a `OpenMLC` toolbox function `testt(toc,Tevmax)` which returns an error when the time elapsed for this evaluation is higher than the time specified in `Tevmax`. This works with the placement of the `tic` function at the beginning of the evaluation function.

The integration of the controlled dynamical system is realized in a try/catch structure which allows error management. As `OpenMLC` will allow any combination of the elementary functions to be tested, it is likely that many systems will diverge or cause integration errors. The integration of the controlled dynamics becomes expensive with complex control laws. Here, a generation contains 1000 individuals to test. Hence, a `testt` function is provided for numerical problems, so that an error is automatically generated when `Tevmax` seconds have passed in the real world. When an error is generated, the program continues in the catch section, where a high value specified in `parameters.badvalue` is attributed to the individual and the evaluation is stopped.

If no error is generated, the J value is computed according to Eq. (2.11), stocked into the J variable which will be send back to the `OpenMLC` object.

This could be the end of the evaluation function, but `OpenMLC` allows the use of an optional fourth argument to force a graphic output. This can be used to observe one specific individual. For instance,

```
unidim_DS_evaluator(mlc.population(4).individuals{5},mlc.
    parameters,1,1)
```

will display the fourth individual of the fifth generation. If the problem is launched using:

```
mlc.go(15,1)
```

A graphical output will be generated for the best individual of each generation. The best individual is represented according to the section included in the `if nargin>4` structure (if the **n**umber of **arg**uments **in**puted is strictly larger than 3). Here, the state, the control and the cost function integration are plotted against time.

MLC run and results

We choose 15 generations to solve the control problem. MLC is launched using

```
mlc.go(15);
```

Typing:

```
mlc
```

returns the best individual and statistics such as its cost function value:

```
After generation 15:
Best individual:
(* (* 3.128 S0) (tanh (tanh -6.613)))
Best J (average over 7 occurence(s)):   2.437028e-01
```

Once again, typing:

```
mlc.show_best_indiv
```

will provide the graphs specified in the evaluation function for the best individual (Fig. 2.15). The time-dependent cost function

$$J(t) = \frac{1}{T}\int_0^t \left[a^2(\tau) + \gamma b^2(\tau)\right] d\tau \qquad (2.13)$$

quantifies the contribution of its integrand during time $[0,t]$ to Eq. (2.11), Note that $J(t)$ must be monotonously increasing from 0 to $J(T) = J$, as the normalization with $1/T$ is fixed and Eq. (2.13) does not define a sliding-window average, .

OpenMLC also contains methods to analyze the whole evolution process. For instance,

```
mlc.show_convergence
```

displays a succession of histograms of the cost function values for each generation (see Fig. 2.16). For the default values (Figs. 2.16a and b), 1000 bins are created with a logarithmic progression from the minimal to the maximal value of J within all generations (excluding values equal or above `mlc.parameters.badvalue`). Section 7.4.1 provides a detailed description. The colormap reflects the ordering of individuals by cost function, not by the quantitative values. Thus, the 2D-view (Fig. 2.16a and c) illustrates the J values of the individuals while a 3D-view (Fig. 2.16b and d) reveals also the population density associated with J-values. The detailed map is obtained by the command:

```
mlc.show_convergence(50,0,1,1,3)
```

Most evolutive processes of dynamical systems will feature similar behaviors and properties:

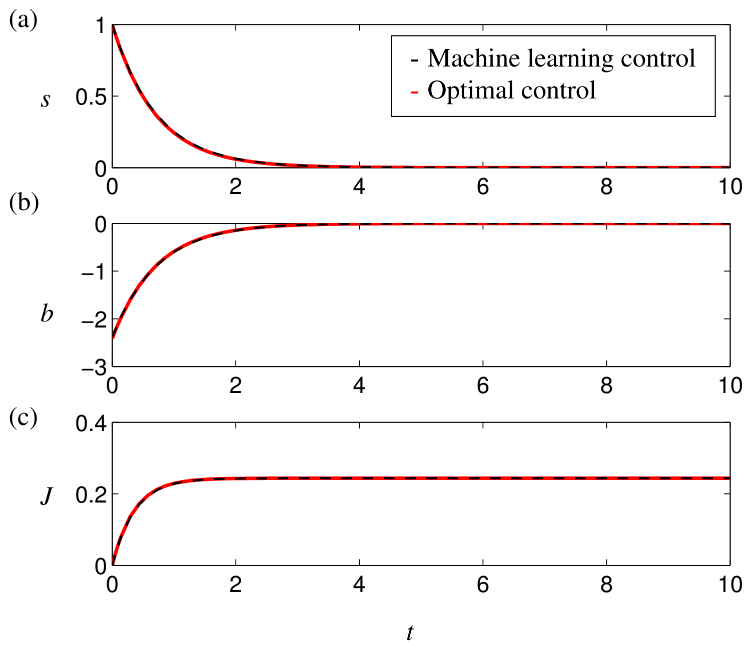


Fig. 2.15 Best individual example after 15 generations, obtained with 'mlc._show_best_indiv' on the 'unidim_DS_evaluator' control problem. The red continuous line shows the optimal control (see Chapter 3) for this control problem.

- The progression is achieved by successive jumps. Once a better individual is found, it is used many times for the evolution and rapidly an island around its J-value is formed. Then, a better individual is found and the population gradually shifts toward the better one.
- Some diversity is kept and some individuals and/or other islands are found in the graph for every generation.
- There is one streak which is present for all generations, though it is far from optimal: these are the many incarnations of the zero-individual. The zero can be created in many ways: subtraction of the same subtree, multiplication by zero, etc. Each generation can be expected to create such zero-individuals.
- Potentially other far-from-optimal islands will be present for other generations: it can be saturated individuals if you impose, for instance, too narrow bounds on the actuation command, or other problem specific phenomena.

Another way to check the convergence process is by typing:

```
mlc.genealogy(15,1)
```

which shows the ancestors of the best individual of the 15th generation (Fig. 2.17). Individuals are linked by parenthood and colors show the genetic operation that resulted in the next individual: yellow for elitism, red for mutation, blue for crossover

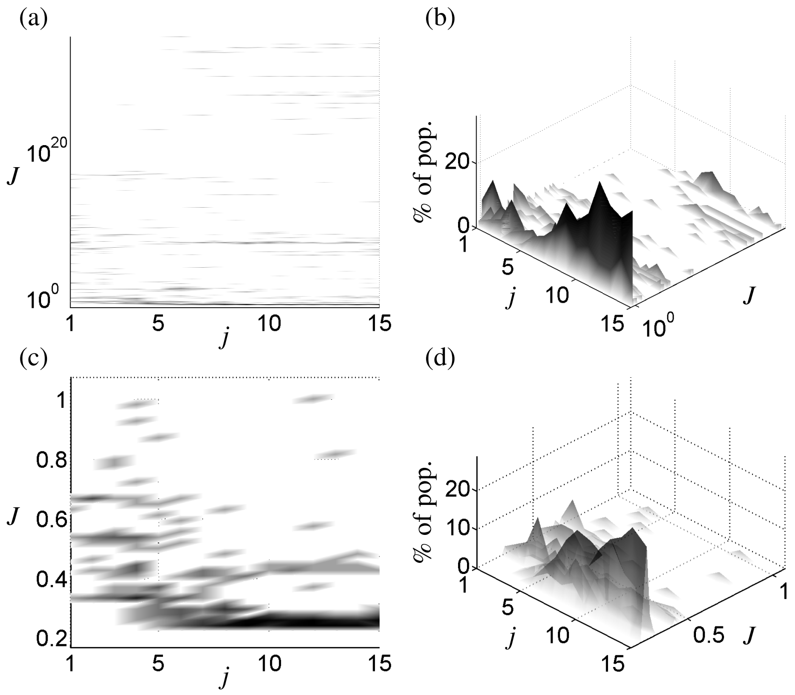


Fig. 2.16 Successive histograms of the population repartition in the cost-function value space. (a) and (c) (detail): Top view. Well performing individuals were obtained in the first generation and no significant progress has been achieved after the 4th generation. (b) and (d) (detail): 3D view. Progressively the population sees a large proportion around the best individuals (d) while it can be observed that some diversity is kept (b). Generated using `mlc.show_convergence`, parameters in text.

and green for replication. More importantly, the background shows which proportion of the population is in a given order of magnitude of the cost function.

`mlc.show_convergence` and `mlc.genealogy` both are customizable. Full options can be found by typing:

```
help MLC/show_convergence
help MLC/genealogy
```

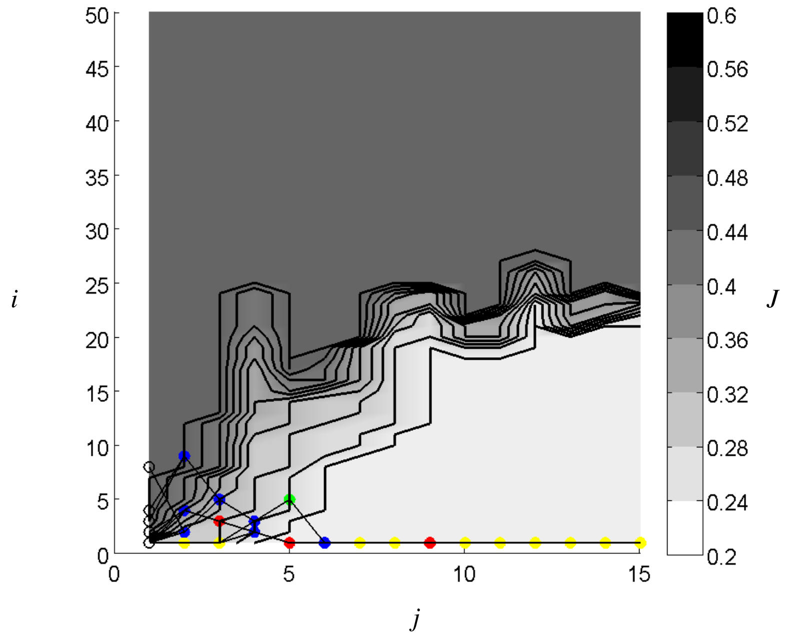


Fig. 2.17 Genealogy of the best individual. All individuals are ranked by performance. The best individual is linked to its parent(s). Color indicates the process: yellow is elitism, red is mutation, green is replication and blue is crossover. The background colormap indicates the J-values for the individuals of index i in generation j. It appears that the best individual after 15 generations has been found from the 10th generation, and that the global population is not evolving any more.

2.4 Exercises

Exercise 2–1: Transform the example from Sec. 2.3.1 in order to achieve a surface regression:

$$\begin{aligned} r_i &= i/10 \\ s_j &= j/10 \\ b_{i,j} &= \tanh(1.256\, r_i\, s_i) + 1.2 \sin(s_i), \quad i,j \in \{-100 \ldots 100\}. \end{aligned}$$

Exercise 2–2: Transform the example from Sec. 2.3.2 in order to learn $b = K(a)$ so that the following dynamical system is stabilized to a fixed point:

$$\begin{aligned} \frac{da}{dt} &= a\left[\frac{a}{10} - \frac{a^2}{10000}\right] + b \\ b &= K(a). \end{aligned}$$

Exercise 2–3: Stabilize the following Lorenz system to each of its three fixed points using MLC:

$$\begin{aligned} \frac{da_1}{dt} &= \sigma\,(a_2 - a_1) \\ \frac{da_2}{dt} &= a_1\,(\rho - a_3) - a_2 \\ \frac{da_3}{dt} &= a_1 a_2 - \beta a_3 + b \\ b &= K(a_1, a_2, a_3), \end{aligned}$$

with $\sigma = 10$, $\beta = 8/3$ and $\rho = 28$.

(a) Determine the three fixedpoints.
(b) Write the three cost function associated to the stabilization of each of the fixed points.
(c) Run MLC for each of the cases.

2.5 Suggested reading

Texts

(1) **Learning from Data**, by Y. S. Abu-Mostafa, M. Magndon-Ismail, H.-T. Lin, 2012 [2].

This book is an exquisite introduction into the principles of learning from data and is highly recommended as a first reading.

(2) **Pattern Classification**, by R. O. Duda, P. E. Hart, and D. G. Stork, 2000 [92].

This classic text provides a serious introduction to machine learning and classification from the probabilistic perspective.

(3) **Pattern Recognition and Machine Learning**, by C. Bishop, 2006 [30].

This text provides a complete overview of machine learning with a self-contained prior on probability theory. Bayes' theory is highlighted in this text.

(4) **Machine Learning: a Probabilistic Perspective**, by K. P. Murphy, 2012 [194].

This comprehensive text describes automated methods in machine learning that may be applied to increasingly big data. There are numerous examples using methods that are computationally available.

(5) **Genetic Programming: On the Programming of Computers by Means of Natural Selection**, by J. R. Koza, 1992 [164].

This seminal text provides a complete overview of the theory of genetic programming. This material has since become the gold standard evolutionary algorithm.

(6) **Genetic Programming: An Introduction**, by W. Banzhaf, P. Nordin, R. E. Keller, and R. D. Francone, 1998 [17].

This text provides an excellent introduction to genetic programming and its applications.

Seminal papers

(1) **Top 10 algorithms in data mining**, by X. Wu *et al.*, *Knowledge and Information Systems*, 2008 [279].

This paper provides an excellent description of ten of the most ubiquitous and powerful techniques in machine learning and data mining that are in use today.

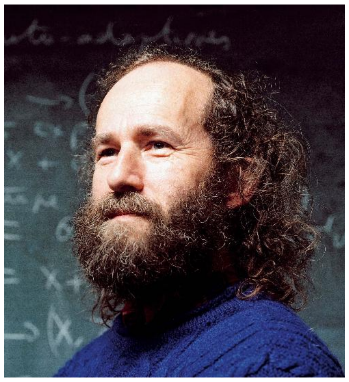
(2) **A tutorial on support vector regression**, by A. J. Smola and B. Schölkopf, *Statistics and Computing*, 2004 [252].

This paper describes the support vector regression, which has become one of the highest performing classifiers in machine learning.

(3) **Random forests**, by L. Breiman, *Machine learning*, 2001 [33].

This paper describes the natural generalization of decision trees: random forests. In this framework, an ensemble of decision trees is used to improve classification performance.

2.6 Interview with Professor Marc Schoenauer



Professor Marc Schoenauer is Principal Senior Researcher (Directeur de Recherche 1ére classe) at INRIA, the French National Institute for Research in Computer Science and Control. He graduated at Ecole Normale Supérieure in Paris, and obtained a PhD in Numerical Analysis at Université Paris 6 in 1980. From 1980 until Aug. 2001 he has been full time researcher with CNRS (the French National Research Center), working at CMAP (the Applied Maths Laboratory) at Ecole Polytechnique. He then joined INRIA, and later founded the TAO team at INRIA Saclay in September 2003 together with Michèle Sebag. Marc Schoenauer has been working in the field of Evolutionary Computation (EC) since the early 90s, more particularly at the interface between EC and Machine Learning (ML). He is author of more than 130 papers in journals and major conferences of these fields. He is or has been advisor of 30 PhD students. He has also been part-time Associate Professor at Ecole Polytechnique in the Applied Maths Department from 1990 to 2004.

Marc Schoenauer is chair of the Executive Board of SIGEVO, the ACM Special Interest Group for Evolutionary Computation. He was Senior Fellow and member of the Board of the late ISGEC (International Society of Genetic and Evolutionary Computation), that has become ACM-SIGEVO in 2005. He has served in the IEEE Technical Committee on Evolutionary Computation from 1995 to 1999, and is member of the PPSN Steering Committee. He was the founding president (1995-2002) of Evolution Artificielle, the French Society for Evolutionary Computation, and has been president of the French Association for Artificial Intelligence (2002-2004). Marc Schoenauer has been Editor in Chief of Evolutionary Computation Journal (2002-2009), is or has been Associate Editor of IEEE Transactions on Evolutionary Computation (1996-2004), Theoretical Computer Science - Theory of Natural Computing (TCS-C) (2001-2006), Genetic Programming and Evolvable Machines Journal (1999-now), and the Journal of Applied Soft Computing (2000-now), and is Acting Editor of Journal of Machine Learning Research (JMLR) since 2013. He serves or has served on the Program Committees of many major conferences in the fields of Evolutionary Computation and Machine Learning.

Authors: Dear Marc, you have pushed the frontiers of evolutionary algorithms by numerous enablers. You proposed one of the first mechanical engineering applications of genetic programming, namely identifying the behavioral law of materials. Many readers of our book are from fluid mechanics with little background in evolutionary algorithms. Could you describe the need for evolutionary algorithms and the particular role of genetic programming? How did you get attracted to genetic programming shortly after Koza's discovery in 1992?

Prof. Schoenauer: First of all, let me state that the work you refer to dates back 20 years now. Furthermore, it was a team work, in collaboration with Michèle Sebag on the algorithmic side, François Jouve on the numerical side, and Habibou Maitournam on the mechanical side.
To come back to your question, the Genetic Programming motto - write the program that writes the program - would be appealing to any programmer who wouldn't call it crazy. I got interested in Evolutionary Computation after hearing a talk by Hugo de Garis in 1992 [77], who evolved a controller for walking humanoid stick-legs. I realized the potentialities of evolution as a model for optimization for problems which more classical methods could not address. Remember that I was trained as an applied mathematician.
The identification of behavioral laws, like many inverse problems, was one of such problems. However, it should be made clear that applied mathematicians also make continuous progresses, solving more and more problems of that kind – though also unveiling new applications domains for evolutionary methods.

Authors: How would you compare genetic programming with other regression techniques for estimation, prediction and control, e.g. linear or linear-quadratic regression, neural networks, genetic algorithms, Monte-Carlo methods and the like.

Prof. Schoenauer: This is a tricky question, especially in these days of Deep Neural Networks triumphs in image and video processing, in games, etc.
One usual argument for genetic programming compared to other regression techniques is the understandability of the results. It is true that there are several examples of such understandable results, starting with Koza's work on digital circuit design [165] (and including our findings in behavioral laws). However, many other applications in genetic programming result in rather large trees, and their interpretation is more problematic. Some interesting work by Hod Lipson [180] of Tonda et al [110] demonstrate that it should be possible to preserve this advantage. Nevertheless, in most cases, the user has to choose how to handle the trade-off between precision and concision - and if precision is preferred, then genetic programming might not be the best choice (keeping in mind that as of today, all such claims are problem-dependent).

Authors: Which advice would you give a fluid dynamicist for shortening the learning time with machine learning control? Which enablers and show-stoppers do you see?

Prof. Schoenauer: My first advice would be to start with what they know (from fluid mechanics and applied maths), and clearly state the limits of the standard methods. From there on, Machine Learning or Evolutionary Computation might be the solution they are looking for. Then they should consider different approaches, at least genetic programming and neural network-based approaches (note that there are very nice results too that hybridize both, as mentioned). And overall, if choosing such non-traditional approach, they should consider all simplifying hypotheses that they have made then trying to solve the problem using standard methods, as these hypotheses might not be necessary any more when using ML/EC methods, opening wide news areas of alternative solutions. In any case, they should not look for a nail for their genetic programming hammer.

Authors: We thank you for this interview!