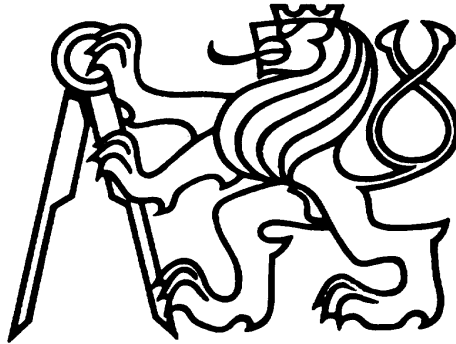


Czech Technical University in Prague

Faculty of Electrical Engineering, Department of Cybernetics



Bachelor Project

Visual design of Evolutionary Algorithms

Author

Gustav Šourek

souregus@fel.cvut.cz

supervisor

Ing. Petr Pošík Ph.D.

Prague, May 2010

BACHELOR PROJECT ASSIGNMENT

Student: Gustav Šourek
Study programme: Software Engineering and Management
Specialisation: Intelligent Systems
Title of Bachelor Project: Evolutionary Algorithms in MATLAB/Simulink Environment

Guidelines:

1. Learn MATLAB/Simulink environment and the possibilities it offers for visual design and experimentation with evolutionary algorithms.
2. Create a Simulink blockset containing individual functional blocks of evolutionary algorithms.
3. Test the functionality of the whole system on several selected problems.
4. Discuss the whole system from the viewpoint of useability and educational value.

Bibliography/Sources: Will be provided by the supervisor.

Bachelor Project Supervisor: Ing. Petr Pošík, Ph.D.

Valid until: the end of the winter semester of academic year 2010/2011


prof. Ing. Vladimír Mařík, CSc.
Head of Department




doc. Ing. Boris Šimák, CSc.
Head

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Gustav Šourek
Studijní program: Softwarové technologie a management
Obor: Inteligentní systémy
Název tématu: Evoluční algoritmy v prostředí MATLAB/Simulink

Pokyny pro vypracování:


1. Seznamte se s prostředím MATLAB/Simulink a s možnostmi, které nabízí pro vizuální návrh a experimentování s evolučními algoritmy.
2. Vytvořte sadu bloků pro Simulink, které budou realizovat jednotlivé stavební díly evolučních algoritmů.
3. Ověřte funkčnost celého systému na několika vybraných úlohách.
4. Zhodnoťte vytvořený systém z hlediska použitelnosti a názornosti.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí bakalářské práce: Ing. Petr Pošík, Ph.D.

Platnost zadání: do konce zimního semestru 2010/2011




prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry


doc. Ing. Boris Šimák, CSc.
děkan

Declaration

I hereby declare that this bachelor project is my own work and that I used only the sources (literature, project, software) stated in references.

In Prague, 5/25/2010

..... 

signature

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v příloženém seznamu.

V Praze, dne 25. 5. 2010

..... 

podpis

Abstract

Visual diagrams and flowcharts are utilities often used in education of iterative algorithms, yet their scope could reach furthermore. The goal of this paper is to explore possibilities of visual programming, concretely Matlab-Simulink language, for the implementation of a functional interactive scheme representing evolutionary algorithms and similar iterative optimizers. Description of the technology, development process and final preview of the product will be provided.

Index Terms

evolutionary algorithms, visual design, Matlab, Simulink

Abstrakt

V učitelské praxi se pro znázornění funkce iterativních algoritmů často používají různá schémata a vývojové diagramy, nicméně jejich použití může být daleko širší. Cílem této práce je prozkoumat možnosti vizuálního programování, konkrétně jazyka Matlab-Simulink, pro implementaci funkčního interaktivního schématu reprezentujícího evoluční algoritmy a podobné iterativní optimalizátory. Popsány budou vybrané technologie a vývoj, na konci bude prezentován výsledek implementace.

Klíčová slova

evoluční algoritmy, vizuální design, Matlab, Simulink

Contents

1	Introduction.....	9
1.1	Evolutionary algorithms.....	9
1.2	Guidepost.....	10
2	Visual programming.....	11
2.1	VPL's criteria for EA implementation.....	12
3	Visual design of evolutionary algorithms.....	13
3.1	Specification of demands.....	13
3.2	Standard EA blocks definition.....	13
3.3	Non-standard EA blocks.....	15
3.4	Structure of blockset.....	15
3.5	Specification of blocks.....	16
4	Simulink & Matlab.....	17
4.1	Evolutionary Algorithm as a dynamic system.....	18
5	Developing the Toolbox.....	19
5.1	Extending functionality of Simulink.....	20
5.2	Overview of the Embedded Matlab Subset.....	21
5.2.1	Embedded Matlab technology.....	21
5.2.2	Working with Variables.....	22
5.2.3	Working with Matrix Indexing Operations.....	22
5.3	Workarounds with Embedded Matlab.....	22
5.3.1	Variable sized data.....	22
5.3.2	Persistent variables.....	23
5.3.3	Extrinsic Functions.....	23
5.4	Creating custom blocks.....	24
5.5	Subsystem inputs and outputs intermezzo.....	25
5.6	Masking blocks.....	26
5.7	Creating a library.....	27
6	Library preview.....	28
7	Testing the blockset	31
7.1	Bridge-building structural optimization example.....	32
7.2	The algorithm.....	33
7.3	Construction examples.....	34
8	Summary.....	35
8.1	Usability report.....	35
8.2	Future work.....	35
8.3	Conclusions.....	36
9	References.....	37
10	Attachments.....	38
10.1	Specification of selected VPLs.....	39
10.2	Source list of selected Visual Programming Languages.....	40
10.3	VPL's descriptions appendix.....	41
10.3.1	EICASLAB.....	42
10.3.2	Executable UML.....	43
10.3.3	Labview.....	44
10.3.4	Game maker.....	45
10.3.5	MVPL.....	46
10.3.6	Openwire.....	47
10.3.7	Prograph.....	48
10.3.8	Simulink.....	49
10.3.9	VisSim.....	50

1 Introduction

In educational praxis, flowcharts, showing the operations that take place inside of an algorithm, are often used to demonstrate functionality of iterative optimizers. Although this has proved to be a very powerful tool for teaching, taking natural human way of perceiving and organizing information into account, its practical use ends right with the dummy scheme at the conceptual introduction to the problem.

The subject of this work, rising up from this limitation, is to make it possible to hold with the scheme as much as possible and profit from the educational value it provides; so that users could use a library to create a scheme not only for drawing out the algorithm, but also to set up the configuration, parameters and finally to launch it on and experiment with the optimizer, while still operating on the scheme.

1.1 Evolutionary algorithms

Evolutionary algorithms [4] (EA) are nature inspired technologies used in modern informatics as methods for robust optimization. Its basic idea comes from the evolution of species, where the units of higher quality are given higher probability to survive and reproduce. Using appropriate representation of genome, these methods can solve very complex problems from various fields of application.

Candidate solutions to the optimization problem play the role of individuals in a population and fitness function determines the environment within which the solutions operate; and as in biological evolution, these candidates are put to mechanisms like selection, reproduction, mutation and replacement. Evolution itself is then performed by repeated application of these operators. There are many architectures of evolutionary algorithms, probably the most basic and widely known has the following form:

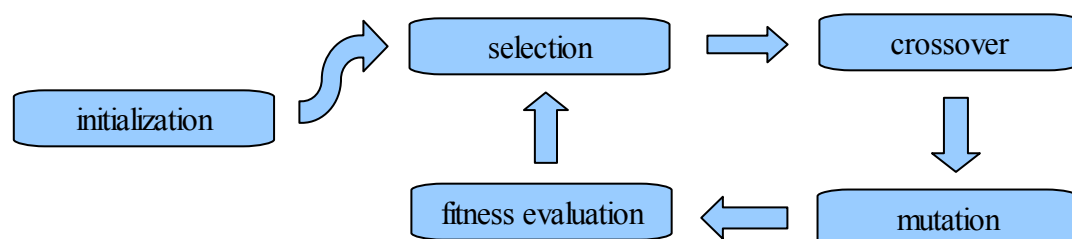


Fig. 1 - basic evolutionary algorithm scheme

As introduced above, this is example that most of students will typically face when learning about evolutionary algorithms. The structure may differ depending on the source, however the purpose remains. In the rest of the paper, attention will be paid on how to take this scheme and bring it to life, using the concepts from visual programming, and how to deliver all the benefits it can provide.

1.2 Guidepost

The techniques of creating functional schemes arise under the visual programming paradigm and several technologies are provided today to handle these tasks. You can find a brief introduction to the fields of visual programming in the next chapter (2) and you can see the overview of its technologies attached at the end of the paper (10). By comparison of these technologies, Matlab-Simulink language (4) has been chosen as the most suitable for the implementation of the evolutionary algorithm library, on which the information is provided in the specification of demands chapter (3). Description of the development process is then covered in the chapter on developing the toolbox (5). At the end of the paper there is a preview of the library (6), examples (7) and final review (8).

2 Visual programming

Expressing thoughts in a visual way has always been an important part of communication. As one of the most common senses, people got used to perceive and express problems by picturing and drawing them out. Visual representation of an issue, such as models and graphs, can often be very helpful for the listener to imagine the structure of the problem and relations between its substructures. During recent years this visual paradigm has also made its way to those types of computer-based communication, that were originally machine-application focused, to try to bring it closer to human understanding and make it more objective and user friendly.

The needs of sharing and understanding problems give rise to visual programming languages (**VPL**) which are becoming popular in several engineering and programming related fields such as designing, modeling, data processing and many others. VPL is any programming language that lets users create programs by manipulating program elements graphically rather than by specifying them textually. VPL allows programming with visual expressions, spatial arrangements of text and graphic symbols used either as elements of syntax or secondary notation. Many VPLs are based on the idea of "boxes and arrows", where boxes or other screen objects are treated as entities, connected by arrow lines which represent relations.[1]

VPLs may be further classified, according to the type and extent of visual expression used, into icon-based languages, form-based languages, and diagram languages. Visual programming environments provide graphical or iconic elements which can be manipulated by users in an interactive way according to some specific spatial grammar for program construction. [1]

Current developments try to integrate the visual programming approach with dataflow programming languages to either have immediate access to the program state resulting in online debugging or automatic program generation and documentation (i.e. visual paradigm). Dataflow languages also allow automatic parallelization, which is likely to become one of the greatest programming challenges of the future.

There are dozens (maybe hundreds) of VPL based on many different technologies, existing textual languages and dedicated to different fields of application e.g. multimedia, software modeling, physical system modeling, managing operating systems etc. Overview of VPL technologies figuring at these fields can be found in appendix (10).

As the nature of visual programming has now been introduced, we should take a look on the features it can provide us for our task – the implementation of a library that will provide users with functionality to draw out, set up and perform the evolutionary algorithm routine.

2.1 VPL's criteria for EA implementation

Although the Evolutionary Algorithm in its basic form is quite a simple structure as for the implementation in standard programming languages, for our purposes it would be good to explicitly mention several factors and dependencies that the language should meet to satisfy an easygoing development and use of an EA based application. Rankings of selected languages, derived from these criteria, can be found in attachment (10).

- a) Data types & structures:
 - It is desired that the language supports all the standard data types, e.g. integer and real, to represent the structure of the problem.
 - The data array structure or any related is most required to represent the character of the genetic information. For several purposes it would be also nice to have some strong and effective background of working with arrays.
- b) Numeric & statistic operations and related support
 - This is needed to perform a number of tasks from the initialization to the final interpretation. It would be really nice to have it already included in the environment in some effective and scalable form.
 - It's an essential when we try to look objectively on the results of the evolutionary algorithm run.
- c) High level programming
 - As a basic construct of programming it should support declaration of own functions to effectively reuse the code during the run.
 - It would be also nice to have a support of some stronger abstraction such as classes, objects and interfaces to represent the unit in population, its structure, belongings and possibilities.
 - On the other hand purely abstract languages are not desired for the need of an easy and quick implementation to a real problem/system.
- d) Visual side
 - As a specific requirement, the final use of the application should be able to go in a visual way, which affects demands to the language also.
 - A flowchart representing the algorithm should be transparent, divisible and easily reconfigurable, thus visual object paradigm is welcome.
 - It would be nice to be able to control the run and data flow of the program completely from the visual environment
 - It should be easy to read, divided into logical segments, and user friendly
- e) Scalability and flexibility
 - The implementation should offer several possibilities to choose and configure the algorithm, thus some support to read user's visual input would be nice.
 - Some powerful background or joint with some commonly used interface or C-like support would be appreciated.
 - For the prospective use, no close-set or contracted software is eligible.

3 Visual design of evolutionary algorithms

The visual interpretation of an issue has proved to be advantageous in pedagogical praxis. Besides the definition of the evolutionary algorithms and other iterative optimization techniques, visual schemes are often used, showing which operations are performed within an iteration of the algorithm. The motivation of this project is to gain a possibility to “draw” these techniques, set up the parameters and launch them on.

3.1 Specification of demands

It should be easy to create a simple optimization loop from the blocks provided by the blockset. At first we will focus on the EA in a standard form:

Algorithm	Standard evolutionary Algorithm
	<pre> begin X⁽⁰⁾ = Initialize() f⁽⁰⁾ = Evaluate(X⁽⁰⁾) g = 0 while not TerminationCondition() do X_{par} = Select(X^(g), f^(g)) X_{off} = Crossover(X_{par}) X_{off} = Mutate(X_{off}) f_{off} = Evaluate(X_{off}) [X^(g+1); f^(g+1)] = Replace(X^(g), X_{off}, f^(g), f_{off}) g = g + 1 end end </pre>

Fig. 2 - Evolutionary algorithm

The system should be as simple as possible and it should be also pretty quick to create well-arranged basic iterative optimizers.

3.2 Standard EA blocks definition

The blocks could be divided into several groups, independently on the representation they work with. Particular blocks then differ in the structure of inputs and outputs. In general, they can produce two kinds of outputs:

1. Population of units (further signed as X)
2. Evaluated population of units (further signed as a double (X; f))

From the dataflow programming point of view, functions which differ in data input or output should also be treated as blocks of different kind. This is a perspective that is necessary to consider when we want to keep up with standards for the visual programming.

The following table shows the block types, their inputs / outputs, functionality and differences between the blocks from this point of view:

Type of block	In	Function
	Out	
Initialization	~ Population \mathbf{X}	To initialize population of N units, usually randomly, also an option for other types of initialization should be provided.
Fitness function	Population \mathbf{X} Evaluated pop. ($\mathbf{X}; f$)	To evaluate a quality of units. Option to define user functions must be provided here, and also few benchmark fitness functions should be included.
Selection	Evaluated pop. ($\mathbf{X}; f$) Population \mathbf{X}	To simulate higher probability to survive and reproduce for units of higher quality. It should be possible to produce more units than the input size by copying the good ones.
Recombination (crossover/mutation)	Population \mathbf{X} Population \mathbf{X}	To create offspring from the input population. Various operators can be used to change the genome.
Replacement strategy	eval. old pop. ($\mathbf{X}; f$) eval. offspring ($\mathbf{X}; f$) eval. new pop. (\mathbf{X})	To simulate the “stronger survives” principle. In difference with Selection it does not copy units and filter them only. The size of output is thus always less than the sum of both inputs sizes.

Table 1 – specification of block types

Now we can observe the following facts:

- State of the algorithm is defined by the current evaluated population. It can be thus obtained from fitness or replacement-strategy block.
- Fitness function proceeds with an unevaluated population, thus its inputs can come from initialization, crossover or mutation blocks.
- Selection requires evaluated population as an input.
- Recombination operators work with unevaluated population. The way they are ordered in the schema is unlimited; they could be also used right on the initial population output.
- Replacement strategy requires two inputs, evaluated population both. The presumption is that the first will be the old state of algorithm and the second will be an output from a fitness function evaluating offspring.

3.3 Non-standard EA blocks

There are several other configurations of EA. Some of them can be constructed using standard blocks, for instance:

- Generational model of EA
 - Units live just for the time of one generation. None of them survives to the further one. This can be simply realized by not using the replacement strategy block.
- Evolutionary strategy
 - Using various generation or steady-state models. It uses “Selection” at the end of evolutionary period, which could be rather classified as a replacement strategy and so implemented within the frame of standard blocks.

However, some architectures of EA (or other iterative optimization algorithms) cannot be implemented in the scope of the standard blockset. For example algorithms from the Estimation of Distribution [5] (EDA) group use probabilistic model learning and sampling to produce new generation instead of using recombination operators. For this type of EA there will be a need to implement new blocks, representing the model learning and sampling and to find the right representation for them.

Parameter adaptation is also a possibility that needs to be implemented outside the scope of the standard blocks. For example we would like to dynamically tune the recombination ratio, based on the current population state. In this case, new recombination blocks having the ratio as an input instead of parameter and a general function block to determine the value of parameter will be needed.

Another example of optimizer is Particle Swarm Optimization [6] (PSO). This algorithm has no selection and uses generational model, thus no unit goes to the next generation unchanged. That was already stated; however it also works with unique recombination operators using information about actual best found positions of particles and global best position. This information forms a model with memory that is a state of the algorithm.

3.4 Structure of blockset

Hierarchical structure of the blockset should:

- make it easy to find the often used blocks
- offer a guideline where to find the non-standard ones

Every operator could have a separate directory that could be further divided according e.g. to used representation. The folder of non-standard algorithm can have the same structure as the root folder and it should contain only those operators, which implementation differs from default.

3.5 Specification of blocks

According to the block type *table 1* above, the following blocks should be implemented:

Block type	Specific block	Parameters
Initialization	Binary	Dimension (length of chromosome) Count (size o population) Probability of generating zero
	Real uniform	Dimension (length of chromosome) Count (size o population) Box constraints
	Real Gauss	Dimension (length of chromosome) Count (size o population) Vector of means (for each dimension) Vector of standard deviations(for each dim)
Selection	Tournament Selection	Tournament size Output size
	Roulette-wheel selection	Output size
	Truncation Selection	Portion of winners Output size
Crossover	Single point crossover	Probability of crossover (default 1)
	Double point crossover	Probability of crossover (default 1)
	Uniform crossover	Probability of crossover (default 1) Probability of taking gen from first parent
	Real arithmetic crossover	Probability of crossover (default 1) Distance parameter
Mutation	Binary bit flip mutation	Probability of mutation (default 1/dim)
	Real Gauss mutation	Probability of mutation (default 1/dim)
Replacement strategy	Truncation replacement	None
	Tournament replacement	Tournament size

Table 2 – specification of blocks

4 Simulink & Matlab

Simulink is an environment for multi-domain simulation and Model-Based Design for dynamic and embedded systems. It provides an interactive graphical environment and a customizable set of block libraries that let you design, simulate, implement, and test a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing [2] extended to use of evolutionary algorithms as well, using the Matlab interface to define your own functions and extend its capabilities. Let us see some of Simulink's features: [2]

- Extensive and expandable libraries of predefined blocks
- Interactive graphical editor for assembling and managing intuitive block diagrams
- Ability to manage complex designs by segmenting models into hierarchies of design components
- Model Explorer to navigate, create, configure, and search all signals, parameters, properties, and generated code associated with your model
- Application programming interfaces (APIs) that let you connect with other simulation programs and incorporate hand-written code
- Embedded Matlab Function blocks for bringing Matlab algorithms into Simulink and embedded system implementations
- Simulation modes (Normal, Accelerator, and Rapid Accelerator) for running simulations interpretively or at compiled C-code speeds using fixed- or variable-step solvers
- Graphical debugger and profiler to examine simulation results and then diagnose performance and unexpected behavior in your design
- Full access to Matlab for analyzing and visualizing results, customizing the modeling environment, and defining signal, parameter, and test data
- Model analysis and diagnostics tools to ensure model consistency and identify modeling errors

There have been several reasons for choosing Simulink as the final environment for our implementation. As you can see from the rating table (10.1) attached, it comes a winner as for the number of total points, but for a very doubtful and subjective character of these ratings, any of the leading group of languages could be also selected. The reason to give Simulink those extra points to raise it from the others, for example LabView that is pretty similar to Simulink, is the Matlab background base behind it, which is an independent and very powerful programming environment extending its functionality rapidly.

When you incorporate Matlab code in Simulink, you can call Matlab functions for data analysis and visualization. Additionally, Simulink lets you use Embedded Matlab code to design embedded algorithms that can then be deployed through code generation with the rest of your model [2].

Considering all these features you can see that Simulink meets all the requirements we have drafted and last but not least Matlab-Simulink is widely used at author's home faculty - CTU FEE as well as in the academic sphere all over the world, which meets the last requirement of effective use and application.

4.1 Evolutionary Algorithm as a dynamic system

As stated, Simulink is fully capable of our implementation of EA, however we have to consider a specific point of view that it brings to the problem, which is that Simulink is an environment based to design and model dynamic systems. This principally means that to implement the EA effectively, we have to think about it that way.

Considering evolutionary algorithm a dynamic system should not cause any confusion; when we look at the principle of each evolutionary algorithm, it does basically few things - selection, crossover and mutation of the current population which is afterward replaced with the new one; that is what every feedback dynamic system does - creating new states of a system by handling the old ones (through the state matrix for the linear systems). This consideration gives us an idea of the EA-look as a nonlinear dynamic system of the first order and by these terms also its possible view in Simulink.

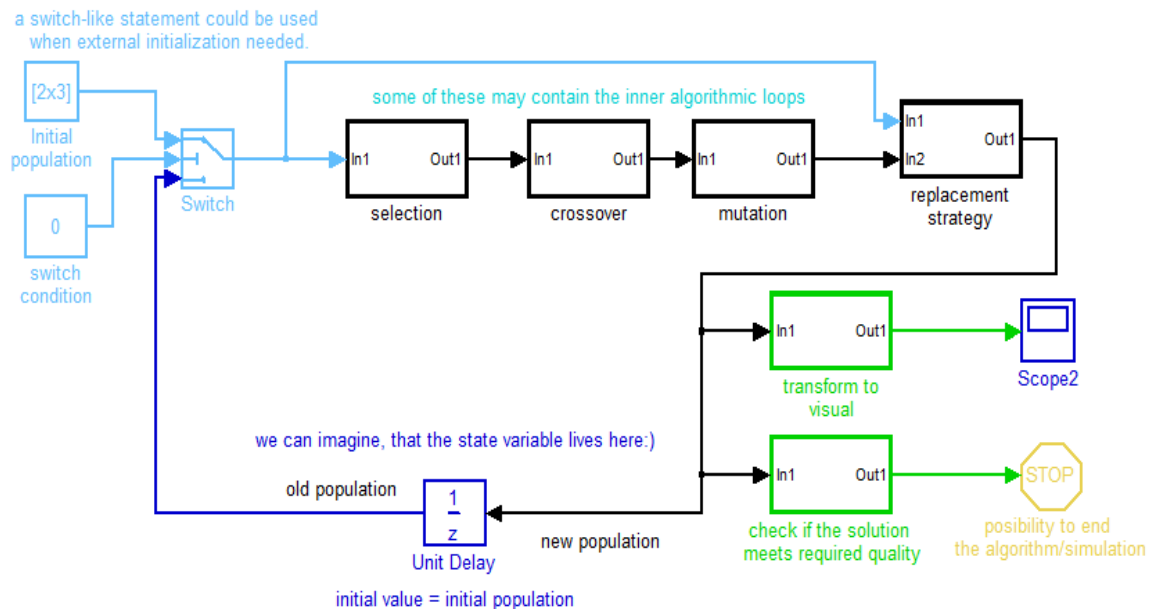


Fig. 3 – a void draft of the EA design in Simulink

Please notice the unit delay block at the bottom of the loop, telling us that we are working with the first order dynamic system. As you can also see in the design of EA, we are about to use a bunch of self-defined functions in connected blocks (crossover, mutation, selection) to handle EA specific operations. The following pages will provide information on how to develop such a block-set and bring this visual-designed EA system to life.

This has been the starting point of the blockset development. You will be able to see how it evolved during the implementation and how it affected the design. At the end, the final look of this functional EA loop will be provided for comparison.

5 Developing the Toolbox

The process of developing a custom library followed given specifications, which shaped its design. The concept of the block-set has risen from the draft design above, showing how the basic iteration loop should look like. Giving information on its development is to describe architecture of the block-set and primarily to provide user a guideline on how to extend its functionality further in a way to satisfy custom needs.

As we can see at the very beginning, the primary goal is to develop functional blocks, representing the operations running upon the population. The idea will be then to take EA from the perspective of functional programming and thus taking units within population as a whole matrix, not as single objects, which also provides us with some features, when taking Matlab matrix operations into account. As Simulink is nowadays able to handle matrix signals, there should be no problem about that.

From this point we will consider the current population matrix a state of the algorithm and the border between last and current state will be the “unit delay” block; like in a dynamic system. The purpose of this block inside the system is crucial. By dividing the iteration loop, we tell Simulink not to try to solve the system as an “algebraic loop”, which purpose is to calculate the balance of the inputs and outputs and to find mathematical equilibrium of the system, but instead we want it to really iterate the system with the explicitly given functionality. This block will also stand a gate to the loop for the initial population and will be covered by subsystem like others, not to confuse the user.

Having the functional blocks correctly set up and connected in the iteration loop, Simulink engine will handle the rest of calculations and put the system to work.


Generally, the paper will not concern about the involved evolutionary algorithm operations as various types of initialization, selection, crossover, mutation, replacement and all other non-standard EA operations required in the specifications, as they are widely known, their implementation is standard, and it is not a subject of this work.

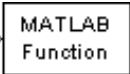
Concrete implementation solutions and code will also not be discussed as it generally comes under standard Matlab routines and user can freely check it out in the attached library, where also documentation on each block is provided in the block mask and in the code. Attention will be paid only to specific issues that come with certain types of blocks and solutions requiring some workaround from standard Matlab.

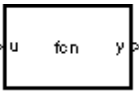
The following chapter will describe the process of developing in Simulink; from creating own functions and solving associated issues to incorporating the functions into blocks and covering them with a user interface mask and finally to group the blocks into library in the desired structure.

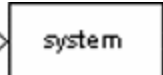
5.1 Extending functionality of Simulink

Now that we have design and the environment, we are about to build the real working model of the EA system consisting of self-defined blocks. There are several ways to bring custom functionality into Simulink. Either you built it up from already existing blocks default to Simulink, which is a nice way to quickly solve the easy ones, or, if this doesn't meet your requirements for expressivity, you put some code into the system. To put some code into the system, you can basically consider one of these ways:

 **Fcn** block applies the specified mathematical expression to its input. The expression differs from a Matlab expression in that the expression cannot perform matrix computations and does not support the colon operator (:). Its expressivity is very similar to a standard calculator that handles logical operators.

 **Matlab Fcn** block applies the specified expression, as the previous one, or additionally a Matlab function to the input. This block is also able to handle matrix and complex input signals in a way like `atan2(u(1),u(2))` or `u(1)^u(2)`; on the other hand it is slower than the Fcn block, because it calls the Matlab parser during each step.

 **Embedded Matlab Function** allows you to fully describe functionality in a code, including all algorithmic routines and matrix operations as in a standard custom Matlab function; however this block works with only a subset of the Matlab language, for more information about its restrictions please follow the next Embedded Matlab subset overview.

 **S-functions** or system-functions provide a powerful mechanism for extending the capabilities of the Simulink environment. An S-function is a computer language description of a Simulink block that is compiled using external tools. This block basically gives also a full expressivity stated in a code and moreover it provides tools to describe/override standard behavior of a block.

The Fcn and Matlab fcn block don't meet the required expressivity, they are just too simple. Embedded Matlab keeps most of its expressivity, is easily integrated with the rest of Matlab/Simulink environment and is the most efficient way to deliver highly optimized code, which can get useful alongside intensive EA fitness computations.

S-functions deliver required expressivity and they could be used as well; on the other hand they require additional settings and even though their capabilities are very complex, their reach out of the scope of Simulink, which makes them more difficult to integrate with the rest and thus break the demand of scalable and easy to use system.

5.2 Overview of the Embedded Matlab Subset

Using Embedded Matlab extends possibilities of Simulink to scheme out algorithms including calculations with N-dimensional arrays, matrix operations, work with complex numbers, fixed point arithmetic and many other functions from the Matlab language. The algorithm is afterwards compiled to C language using Real time Workshop, which is another feature of Matlab. This is more effective than programming right in C, because Matlab and Simulink offer already implemented and very complex functionality. The algorithms stated in Embedded Matlab and figuring in a Simulink model are also easier to read and debug.

5.2.1 Embedded Matlab technology

Embedded Matlab is a subset of the Matlab language that supports efficient code generation for deployment in embedded systems and acceleration of fixed-point algorithms. Embedded Matlab technology can generate efficient embeddable code thanks to a powerful inference engine. The inference engine analyzes your Matlab code and determines the size, class, and complexity of every expression. The information derived by the inference engine allows Real-Time Workshop to produce code that is efficient and tailored to your specific application, rather than produce generic code that can handle every possible set of Matlab inputs [2].

On the other hand, to generate this efficient code there are some things you have to give over. The Embedded Matlab subset does not support the following Matlab features: [2]

- Cell arrays*
- Command/function duality
- Matrix deletion
- Nested functions*
- Objects*
- Sparse matrices*
- `try/catch` statements
- Recursion

**signed features could be found useful in the implementation of EA*

Also the functions used by Embedded Matlab differ in implementation a bit. Those included in subset have the same name, arguments, and functionality as in Matlab; however, Embedded Matlab functions come with limitations to allow generation of efficient code.

In the following pages I will try to point out the main difficulties with embedded Matlab and differences from standard, which crossed my way and that a typical Matlab user will have to face when coming to embedded. At the end of each section I will briefly demonstrate where and how it affected the EA blockset implementation.

5.2.2 Working with Variables

In the Matlab language, variables change their properties dynamically so you can use the same variable to hold a value of any class, size, or complexity. In Embedded Matlab functions, you must assign variables explicitly to have a specific class, size, and complexity before using them in operations or returning them as outputs. This restriction helps Embedded Matlab generate efficient code for embedded languages, like C and VHDL that typically have a similar restriction. [2]

5.2.3 Working with Matrix Indexing Operations

The Embedded Matlab subset matrix indexing operations are limited and every matrix must be initialized explicitly before indexed. The thing is that Embedded Matlab never dynamically allocates memory so, for the size of the expressions that change as the program executes, use `for` loops instead. This issue however can be overcome by variable size data feature mentioned below.

- *Example 1 - variable matrix 'mutants' must be initialized to have specific precision and size before entering the loop.*

```
mutants=zeros(size(ingenomes));
```

```
%add a weighted difference of two vectors to the third one
for i=1:count
    mutants(i,:)= child(1,:) + F*(child(2,)-child(3,:));
end
```



5.3 Workarounds with Embedded Matlab

Finally, there will be few modifications or workarounds to the embedded Matlab coding style, which I've found crucial during the implementation of EA blocks. Solutions I've found for the issues, I've been dealing with, I grouped to following:

5.3.1 Variable sized data

It's a feature from the new Matlab 2009b and I consider it the biggest contribution along with matrix signals routing within the Simulink interface. Not only it allows you to change the size of own expressions, but it allows you to use the indexing functions as 'find' in its full expressivity, which is a substitute for logical indexing missing in embedded Matlab. Without this feature, 'find' issues error when standardly called.

Variable-size data is data whose size may change at run time. By contrast, fixed-size data is data whose size is known and locked at compile time, and therefore cannot change at run time. You can define variable-size arrays and matrices as inputs, outputs, and local data in Embedded Matlab Function blocks. However, the block must be able to determine the upper bounds of variable-size data at compile time [2].

- **Example 2** – The variable data size feature is not declared in code here as it can be also setup in the block configuration. Here, the 'find' uses it inside to allocate vector of variable size (up to 'cm' size), depending on the condition satisfaction.

```
sel = zeros(sizes,1);

%find the right pocket for every throw
for i=1:sizes
    sel(i)=find(cm>f(i),1,'first');
end
```



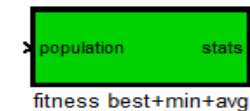
5.3.2 Persistent variables

This feature can be found very useful for the “blocks with memory” as various statistics, algorithm state updates and any block that keeps some information between iterations to the further generation. Persistent variables are local to the function in which they are declared, but they retain their values in memory between function calls.

- **Example 3** – persistent variable is used in a fitness statistics function to count and keep the global minimum.

```
persistent mini

%set up mini in a first iteration
if isempty(mini)
    mini = min(fitness);
end
```



5.3.3 Extrinsic Functions

This is an ultimate feature, again available from the new Matlab 2009b. This is the last one to use when no other solution is coming your way, or when you really need to use some outsourcing, like in user-defined fitness evaluation.

An extrinsic function is a function on the Matlab path that Embedded Matlab dispatches to Matlab software for execution. Embedded Matlab does not generate code for extrinsic functions, thus no embedded Matlab coding style restrictions are needed to follow. On the other hand, if these are included in a model, it cannot be compiled. There are two ways to declare a function to be extrinsic; either you dispatch the function using the directive:

- `eml.extrinsic('function');`

Or you call the extrinsic function using a function call:

- `y = feval('function', parameters);`

However, you have to be aware of the right extrinsic function calling to avoid errors. One of the typical is that every extrinsic function, whatever output you declare it, returns an mxArray also called a Matlab array. This one cannot be used within embedded Matlab and thus the compiler throws an error in this case:

!function output 'y' cannot be of Matlab type !

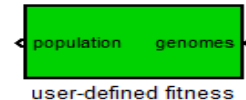
To avoid this issue, all you have to do is to let Embedded Matlab convert mxArray back to the type of the declared variable assigned to it. However, if the data in the mxArray is not consistent with the type of the variable, Embedded Matlab generates an error.

- `y = zeros(N,D); % y is a matrix of (N,D) size of double precision`
- `y = feval('function', parameters); %convert the output to match it`

Extrinsic function should be used wisely for including them in a model bounds the system with Matlab as it is send to it for execution. Therefore, you can run the simulation only on platforms where you install Matlab software.

- **Example 4** – extrinsic user defined fitness function, set by name in parameter of block, is called here using 'feval' function call.

```
%initialize fitness to right size
fitness = zeros(size(ingenomes,1),1);
%call user defined fitness from param gg
fitness=feval(char(gg),ingenomes);
```



5.4 Creating custom blocks

Now that we have explored Embedded Matlab functions we would like to package them into custom blocks and add them to the library. This is a simple routine that we manage by a *subsystem* block. A subsystem is a keystone for creating custom blocks, custom libraries and sub-libraries and it provides functionality like masking (5.6) to deliver customized final look of blocks.

When you have your custom functionality defined and prepared either in a form of embedded function or modeled by another Simulink blocks, you can simply cover them with subsystem by:

1. Copy the Subsystem block from the Ports & Subsystems library into your model.
2. Open the Subsystem block by double-clicking it.
3. In the empty Subsystem window, create/copy the subsystem. Use Inport blocks to represent input from outside the subsystem and Outport blocks to represent external output.[2]

5.5 Subsystem inputs and outputs intermezzo

We use subsystem blocks to systematically divide and group functionality to provide user with maximal readability and transparency. The same way you can deal with signal busses and handle them using *bus creator* - to bundle a group of parallel lines carrying different signals to one and *bus selector* - when you want to extract them back. For instance when we want to handle population, consisting of genomes and fitness:

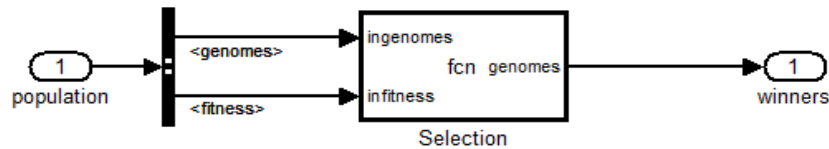


Fig. 4 – subsystem (*Selection*) expanding structured input signal



Fig. 5 – subsystem (*Fitness function*) creating structured output signal

Using bus blocks, default to Simulink, is more scalable than using structured signals right within the functions that need to be registered in Simulink workspace. By contrast to these, no other actions are required for bus creator/selector and user can freely expand or cover the structured signal wherever needed. All you need to be aware of is to keep the name conventions of the signals.

As Simulink is able to handle signals from bus blocks, e.g. propagate their name and determine their size, using them in a library complicates changing the library blocks and increases the likelihood of errors. It is recommended so, to be careful of the name conventions and structure of signals transferred between blocks, when bus blocks are covered from a user by subsystem as in our case. However once you build up the model, Simulink signalize nicely the use of structured signals by using the bold lines for them, no matter the level of system view, to keep you aware of their use.

Once you have a Simulink subsystem that models the desired behavior, you can convert it into a custom block by:

1. Masking the block to hide the block's contents and provide a custom block dialog.
2. Placing the block in a library to prohibit modifications and allow for easily updating copies of the block.

5.6 Masking blocks

After creating a block, providing desired functionality, we will setup the interface that shows up to the user. This is necessary for the users parametric input to the function inside as well as it offers various visual and other tunings to make blocks more user-friendly. For the complex capabilities of masking, please see masking in Simulink[2] as I will highlight just very few of it. To create a mask, right-click on your subsystem and choose *mask a subsystem*; then the mask editor comes up to provide you with:

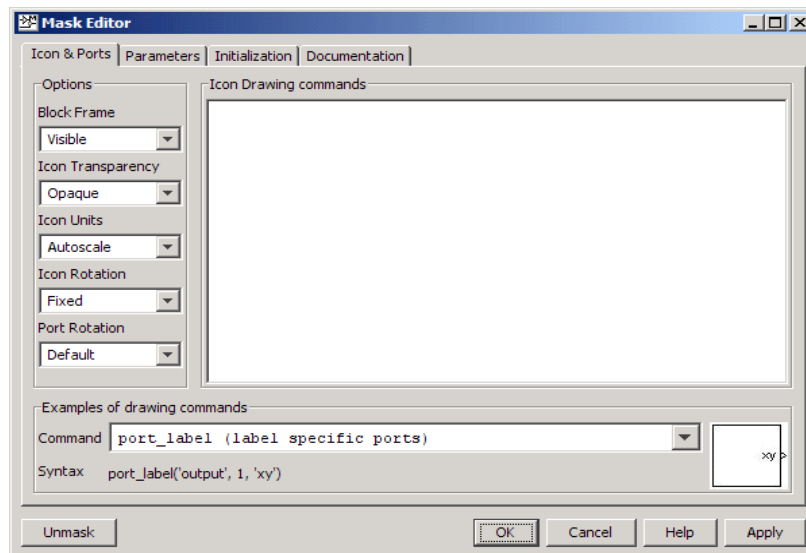


Fig. 6 - Simulink block mask editor window

Icon and ports - You can set up the look of the block by plot drawing commands and change the labels of the block input and output ports. You can also go further and set up a bitmap image mask or define a shape of the block; however these setting can slow down the upload of a library in the browser.

Parameters - You are able to let user define values of input parameters through the mask of a block, containing the function to use them. Just declare the name of the variable to be used inside and the name to be appeared to user. You can also check a box to make parameters “tunable”, which means to let user alter them during the simulation. This can be very useful for example to tune mutation on the run, but be aware that this behavior can somewhere cause variable size data problems as described in Embedded Matlab part.

Initialization - This is a place to write commands for the block to be progressed on the startup. I found this useful just for the preparation of parameters and interaction with them, e.g. for processing a string parameter in user defined external functions.

Documentation - Describe functionality of the block in plain text that shows up to a user in the library browser as well as help on the block.

5.7 Creating a library

After having all the blocks in their final look, we group them to a library. Creating a new library in Simulink consists of two steps:

- Create a new library as a model containing all the elements you wish to use in the library in desired structure.
 - At this step it is good to regroup the blocks using void subsystems to the structure given by specifications of the library. All the subsystems' names and masks shall be visible in the library browser.
- Add the library to the library browser. This one is a bit more complicated. In the directory including your model file, you have to include a `slblocks.m` file. That is a specific `.m` file configuring the settings and behavior of the library e.g. names, appearance, reactions and so on. The approach you use to create the `slblocks.m` file depends on the requirements for describing the library:
 - If just a minimal `slblocks.m` file meets your needs, then a sample file is available on mathworks site
 - If you want to describe the library more fully, consider copying an existing `slblocks.m` file to use as a template, editing the copy to describe your library.

For our purposes we need to alter the `slblocks.m`, change some settings and most importantly to register the library permanently. That can be done manually by adding the library directory to the Matlab path, or automatically by including following lines at the end:

```
➤ % Add the toolbox to the path
  addpath(genpath(fullfile(Matlabroot, '/toolbox/EA')))
  savepath
  % End of slblocks
```

Move the folder with the library and `slblock.m` files to the place, or change the path accordingly, but it is good to have them all together in the Matlab root toolboxes.

When everything is set, you restart Simulink and should be able to see the toolbox in your library browser. You are still free to alter the library in almost any way including using parts of other toolboxes, changing the masks, creating subfolders, labels etc., just right-click on the library in the browser and “open the library” as a standard model.

** please see the library folder on the CD attached to this paper. You might add it to your library browser by adding a path to it as described above. Then you will be able to see and check out the library and models shown bellow for yourself.*

6 Library preview

As the process of development of the blockset is now clear, we can have a look on the EA library. When you open your library browser, you should be able to see the 'EA' item in there (after setting the path to it – see attached files and previous chapter).

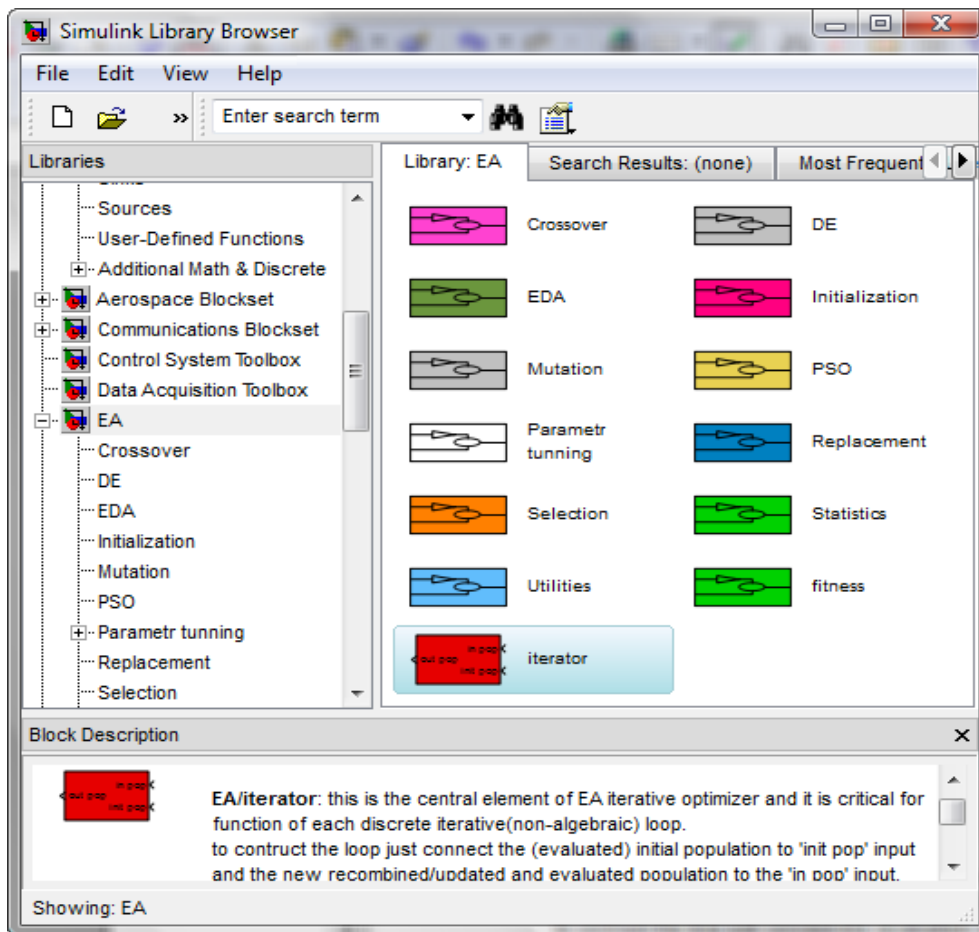


Fig. 7 – EA in Simulink library browser

As for the look, you can see there are colors used to distinguish operator types according to the folder tree and these colors are inherited by particular blocks; the crucial iterator block is red and green color is used for all the fitness related blocks.

You can now use it in the same way as any other library in your browser and even combine with any other signal-compatible library blocks. The structure of the library is hierarchically divided as demanded in specifications; on the top level the keystone of each iterative system stands – the red 'iterator' block. Together with it, there are directories for all the EA operators and some additional statistics tools, which are either further divided or include the particular blocks.

So to create an EA optimization system model, you just open a plain new one and drag & drop desired blocks from the library into your model to connect them. We will look on example of how a basic EA system using the blockset looks like.

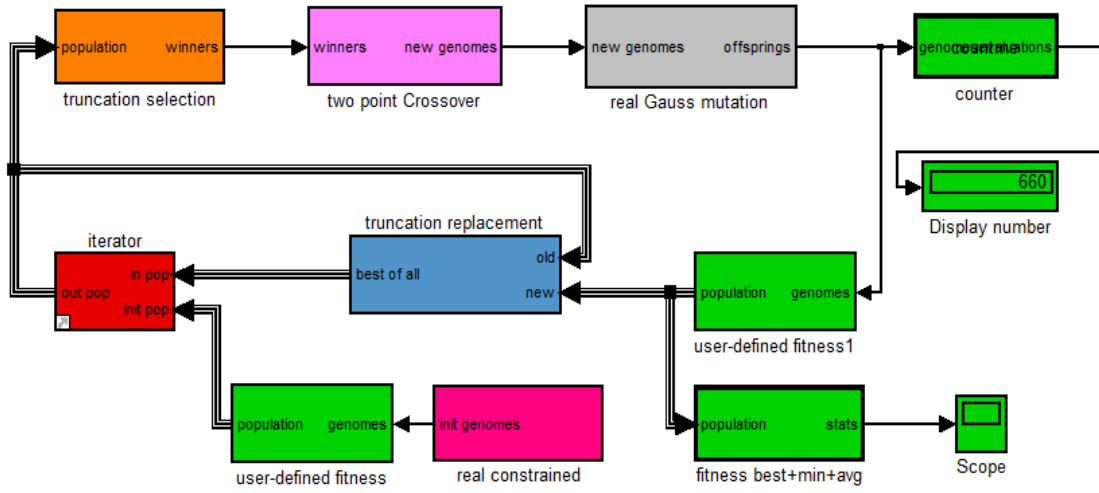


Fig. 8 – EA model

Please notice the structure and compare it with the conceptual design on *fig. 1*, drafted by demands on the system before the real implementation of the blockset. You can see the structure has been kept but for the need of well-arranged system, few features have risen. It's the initialization of population connected right within the 'iterator' block which stands for the 'unit delay' block; that makes it more transparent to manage the population signal. Also additional control of the loop was added, not to bother the user, but provide tools to monitor the optimization and option to affect it manually.

Also some non-standard architectures of EA have been implemented. Let us have a short look on how to build them up too:

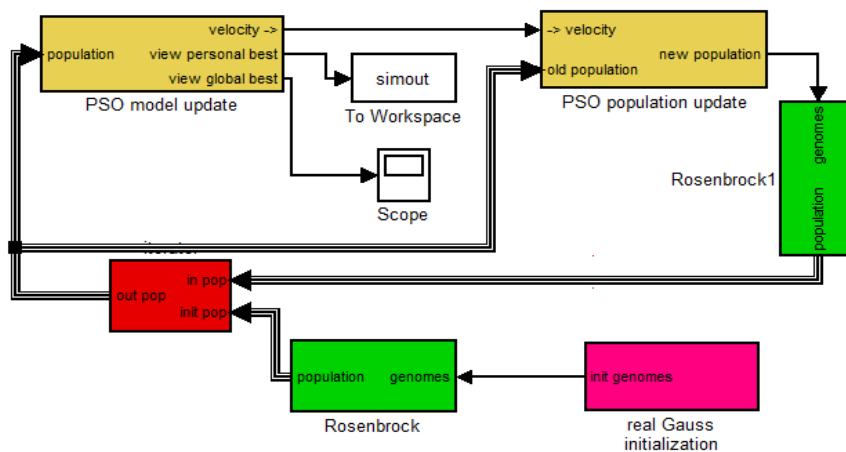


Fig. 9 – PSO model

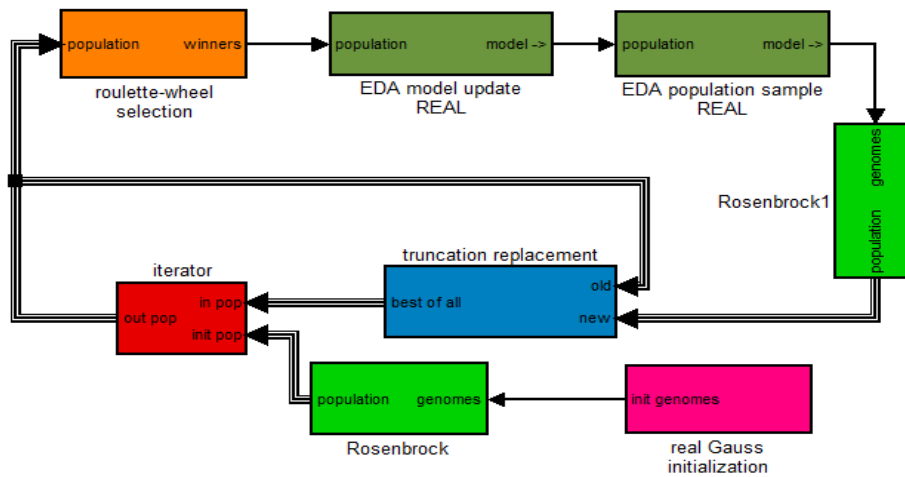


Fig. 10 – EDA model

Before you finally run the system, check all the blocks for their parameters. You should make them consistent in the whole system, especially from the population size point of view. This is the likeliest source of issues that the user will be dealing with. Let us concisely look at few masks to see parameters to set up for the standard EA system loop:

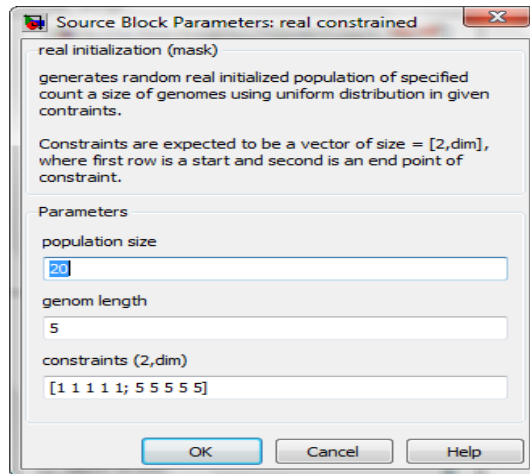


Fig. 11 - initialization mask

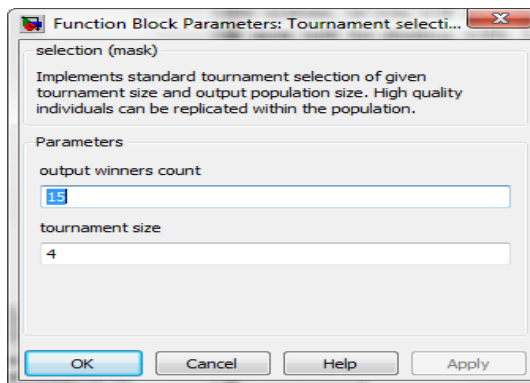


Fig. 12 - selection mask

7 Testing the blockset

Now according to the models above, the time has come to show some outputs. At first, using just the library itself, you can find several benchmark fitness functions in there. Just build up the model as described and choose one to put in. To monitor the optimization run and characteristics you can inspect the actual best, global best and average fitness in the output from fitness best+min+avg block and watch number of fitness evaluations in the output of counter block. To view these values you can use the scope or display block as shown above. You can also save their progression to a variable in workspace or to a file, using 'to Workspace' or 'to File' block, for further processing. We will look at few examples of what the scope gives:

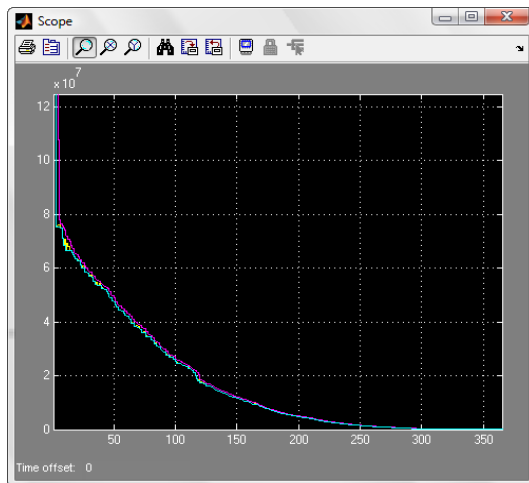


Fig. 13 - EA on Rosenbrock fcn

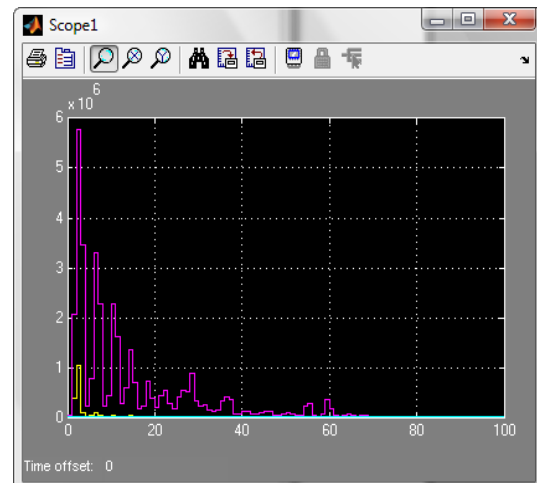


Fig. 14 - PSO on Rosenbrock fcn

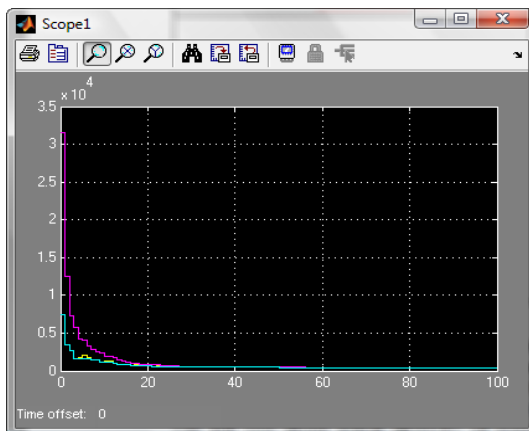


Fig. 15 - EDA on on Rosenbrock fcn

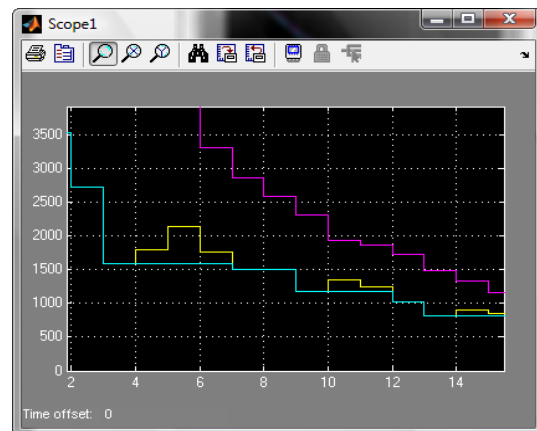


Fig. 16 - Characteristics closeup

If you look closer, you can see discrete steps progression of blue, yellow and violet line standing for global best, actual best and average fitness. If you use some kind of elitism strategy (e.g. Truncation), global and actual best bend with each other. Thus blue line sets the lower bound for yellow line which sets it for the violet one. Distances of these bounds can tell you about actual diversity of population and you can use it to step into the simulation and tune recombination rate.

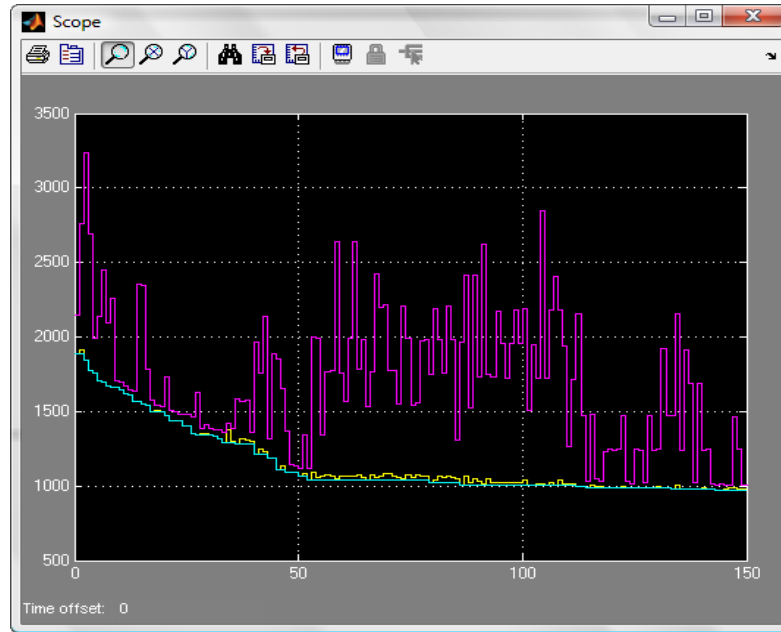


Fig. 17 - external manual tuning of mutation

As you can see here, average fitness progression is not as smooth as before and is rather kind of fluctuating instead; this is caused by external manual tuning of recombination. That can become handy when average fitness is too low, which is mostly caused by low diversity of population and progression can get stuck at local optima. You can then increase the mutation and crossover rate or distance parameters to try to get out of it.

7.1 Bridge-building structural optimization example

The goal of this example was to create and solve very complex problem using the blockset connected with the Matlab interface and so to demonstrate potential of joining these two technologies together.

The subject of this example is optimization algorithm trying to find the best possible construction of a railway bridge consisting of the least possible number of girders with given maximum tension limits. The structure is working upon a given triangle grid represented by 2D graph. The issue was given this simplified form to quicken the calculations by avoiding non-linear equations describing the bridge stability.

The optimization of structures or so-called multi-disciplinary design optimization is quite a popular field of application of evolutionary algorithms; however, by using the linear form of the issue we can incorporate also a linear programming optimization routine to save a lot of work from EA and make it more effective; and this is where Matlab environment features become beneficial, for we will use its implemented *linprog* function, that could be hardly achieved in pure Simulink.

7.2 The algorithm

This is just a basic introduction of a problem, for the closer description and related work please follow the link [3].

The representation of genome is a real N-dimensional vector, where N is a number of girders/variables respectively. The values inside the vector set the bounds for a maximal stress inflicting each girder. This is the structure upon which the EA operates and in this form it is also handed over to the linear programming routine. The *linprog* then searches within these bounds for a solution of a huge linear equation system describing the bridge physical stability.

The basic principle of the system is the equilibrium of every joint figuring inside the construction. The vector of tension force inflicted by the girder is of the same counter-wise oriented value on both endpoints for each particular girder. The external load pressure of the bridge is involved on the right side of equations as a constant.

In the initialization the borders are set to be relaxed in order to make it able to find at least some solution in the equation system. When some feasible units are found, the evolution may begin; for the unfeasible ones (those which genomes cause the bridge to fall) the fitness is penalized.

The shape of the fitness function, minimizing the use of girders, forces the tension values to go down around zero, which means the girder is off. Bellow is an example of how the fitness function looks like for each of girders (notice the non-convex shape which makes the issue unable to be solved just by using linear programming):

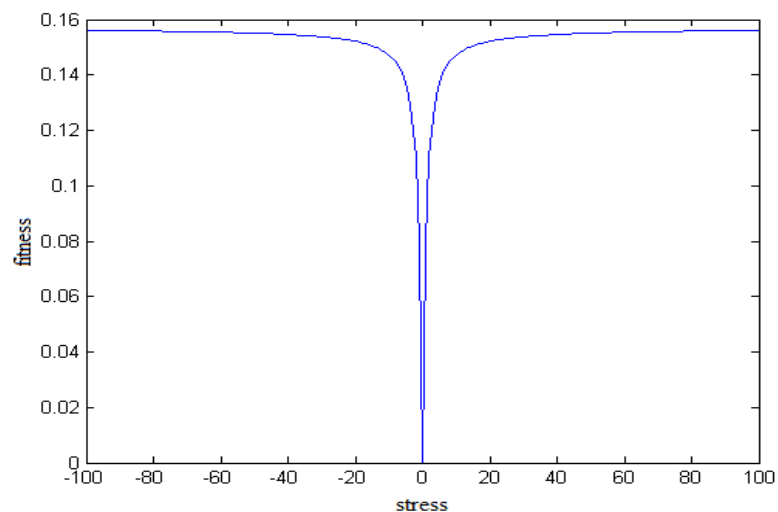


Fig. 18 - fitness function used in this example

7.3 Construction examples

Here is an example of what can be achieved by evolutionary optimization to reduce the number of girders, used material and costs of the construction.

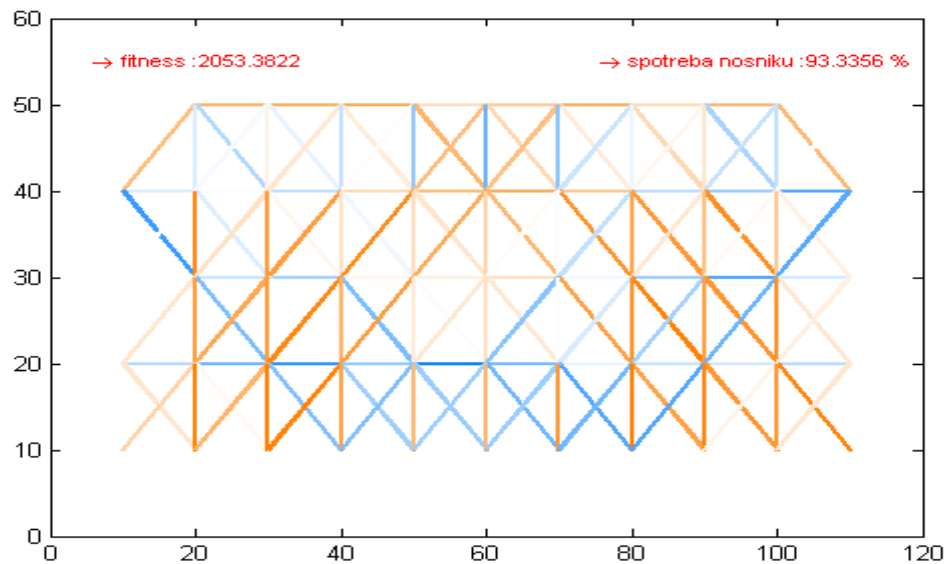


Fig. 19 - initial state of bridge construction

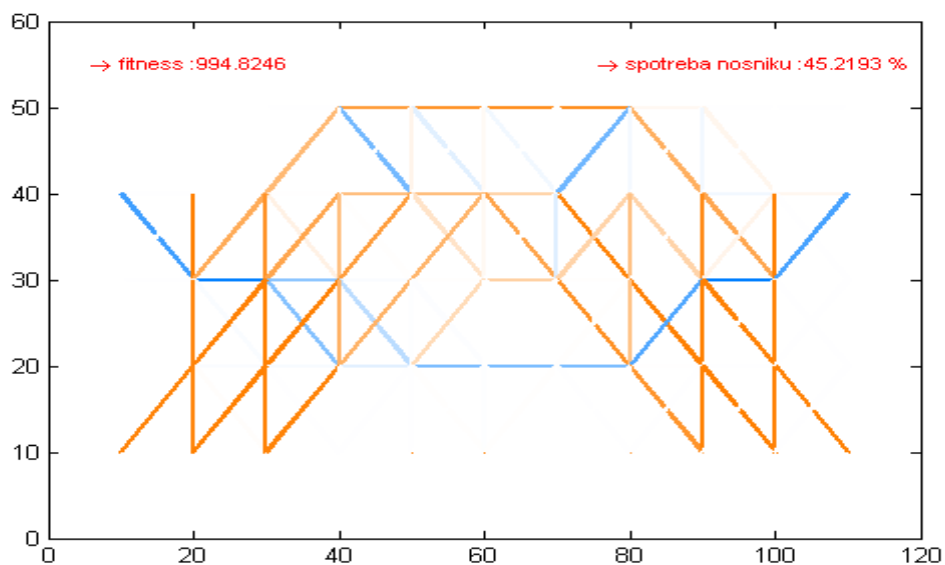


Fig. 20 - optimized bridge construction

*a road itself, which is not a part of the weighting support design of the bridge, is supposed to go on the second level from above (at the level of $y=40$).

8 Summary

8.1 Usability report

The process time to construct a new model of EA in a custom structure of operators used is taking just a few minutes at maximum. For instance the model of EA in the basic form took me less than 2 minutes to build up including the time to set up the parameters. Speaking nothing of the transparency and user-friendliness this is surely much more efficient way than programming it from scratch, or even copy fragments of a code trying to paste them together.

The system model is also easy to debug, when the whole optimization algorithm is separated from the instance of problem that is either covered in fitness function block or extrinsically called. Thus when the optimization problem structure is stated, it can stay completely and safely untouched, while the user is tuning the optimization and experimenting with EA. Simulink also uses a very transparent inference system to warn you about the issues in your code and reference you directly to the weak points of your model, even inside the embedded functions.

Experimenting with EA just couldn't be easier; you choose the operators you consider right and see how they fare. You run the whole optimization in a "click to play" way and watch the progress online. If the fitness is not having a good run or the number of evaluations is growing too much, you break the simulation and change the operators by just drag & drop manipulation. If you are satisfied with operators but would like to boost it up even more, you can tune the parameters of mutation and crossover and see how it affects diversity of population at the same time.

The educational value of the working model is clear and was stated on the very beginning; people just like schemes. The possibility to see all the functional parts of the algorithm on the same plane at the same time gives much better understanding of the problem. Relations between these parts, represented by the lines and arrows, make it so much more transparent than some hardly findable references hidden in the code.

8.2 Future work

The library was oriented to user friendliness as a quick and easy practice to construct an optimization algorithm model; however users might want to go behind and use it for advanced experimentation and testing of algorithm performance. This may include a need to repeatedly launch the model and cumulate the statistics results, which may be further processed in various ways. To perform any non-standard operations upon the population, user can use prepared block 'general pop function' to call self-defined function from Matlab, however when intending to proceed to the advanced level of testing, users have to reach outside the scope of blockset library.

For the advanced control of the algorithm model as a whole, which means to handle not just the population but to control the flow of optimization model itself, users are enabled to use Simulink connection with Matlab. Controlling Simulink models from Matlab is a very complex option; quick and easy - any action you could possibly make in Simulink model can also be achieved from Matlab environment - from building, connecting and setting parameters of blocks to loading, launching and stopping the model. Thus any work routine performed upon the model can be treated as a Matlab code in a script with equal options and functionality. For the set of functions realizing these routines (Simulink functions) see Simulink reference again [2].

Advanced users familiar with Simulink could use as well the conditional, triggered and enabled subsystems [2] combined with iterative blocks to simulate the behavior of restarting and modified control flow. As this behavior is a bit of advance, it requires additional attention, for example from the data size point of view. Models using these techniques then restrain from the original idea of transparency and easy to use systems, thus such external control of algorithm flow was not included in the library. However this could be a subject of future library enrichment.

8.3 Conclusions

The work was oriented for implementation of the library that you can see attached, this paper was only to describe its development and use.

On the beginning it was not clear that this task will be possible to implement, but the technology of Matlab Simulink, selected for the implementation, proved to be a good choice as it provided all the functionality necessary. The Simulink library set has been extended and final models constructed from the EA blockset meet all the requirements stated at the beginning.

Educational value and usability of these models have been proved in the structural optimization example as the library was successfully used and presented in an independent seminary work and saved a lot of effort when it came to optimization by evolutionary algorithms.

Although there have been some obstacles and restrictions with Simulink visual programming technology, it is now clear that the visual paradigm idea that raised at premeditation of this work was appropriate. Simulating algorithms and treating them as models is beneficial as it gives user better control over the algorithm, even during its run.

The library is ready to provide evolutionary algorithms functionality for either educational or basic practical use. As Simulink library is quite a scalable format, it can be easily extended by another custom optimizer or other routine, which is possible to be transformed to a block structure as described above.

9 References

- [1] Visual programming languages on Wikipedia, free encyclopedia:
http://en.wikipedia.org/wiki/Visual_programming_language
- [2] Documentation on Simulink technology on Mathworks, vendor sites:
<http://www.mathworks.com/access/helpdesk/help/toolbox/simulink>
- [3] Sourek, G., *Bridge-building structural optimization*, Y33OIS seminary work, available in Czech language on the CD attached.
- [4] Goldberg, David E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison- Wesley, 1989.
- [5] Larranaga, Pedro and Lozano, Jose A., *Estimation of Distribution Algorithms*, Kluwer Academic Publishers, 2002.
- [6] Kennedy, James and Eberhart, Russel C. *Particle swarm optimization*, IEEE Service Center, 1995.

10 Attachments

10.1 Specification of selected VPLs

Let me introduce some of the VPLs to outline the similarities and differences in character of each. I will try to mark out in common their capabilities and advantages to our task of visual programming of the evolutionary algorithms and its application. Few selected in a table below are introduced behind the link with a short general description and print-screen which is in my opinion a quick and nice way to make a picture of it.

The following table shows the qualities of each language according to our purpose. The criteria mentioned (2.1) have been grouped to logical fields, to make it easy to rate the individual aspects of selected VPLs. The total value should say something about the final usability of the VPL to implement, use and control the Evolutionary Algorithm.

Application -/ attribute	popular & visual	scalable & flexible	Math & support	Types & structures	smart & usable	Total
Simulink	8	8	10	9	9	44
Labview	8	8	10	8	8	42
OpenWire	7	8	7	8	9	39
EicasLab	7	7	8	7	7	36
MVPL	8	8	4	7	8	35
VisSim	6	7	7	7	8	35
Fx engine	5	8	5	8	8	34
SCADE	6	6	9	6	6	33
Limnor	6	6	6	7	7	32
Prograph	6	7	5	7	6	31
GameMaker	8	5	3	5	8	29
OpenDX	7	5	7	5	5	29
MSTworkshop	6	6	5	5	6	28
XUML	9	7	2	3	4	25
CODE	6	7	3	2	4	22
Quest 3D	7	4	3	3	3	20
Virtools	7	3	4	3	3	20
Authorware	6	3	3	4	4	20
Ladder logic	3	4	5	4	4	20
Functionblock	2	3	6	5	3	19
OpenMusic	6	5	2	1	4	18
Alice	7	3	2	2	3	17
ThingLab	6	4	2	2	3	17
DRAKON	3	3	6	3	2	17
Appware	3	5	2	4	3	17
Baltik	6	2	3	2	2	15

Table 3 – rankings of VPLs

**The rankings above do not tell anything about general quality of software. They are highly subjective and for specific use only.*

10.2 Source list of selected Visual Programming Languages

The information used for ranking the selected VPLs have been gathered from the vendor sites and Wikipedia [1]. For the quotations and further description of VPLs please check the links assigned in the following table:

Alice	http://www.alice.org/
Appware	http://en.wikipedia.org/wiki/AppWare
Authorware	http://www.authorware.com/
Baltik	http://en.wikipedia.org/wiki/Baltie
CODE	http://www.cs.utexas.edu/users/code/
DRAKON	http://sage.com.ua/en.shtml?e6l0
EicasLab	http://ids.fzi.de/acoduasis/
Functionblock	http://en.wikipedia.org/wiki/Function_block_diagram
Fx engine	http://www.smprocess.com/
GameMaker	http://www.yoyogames.com/gamemaker/
Labview	http://www.ni.com/labview/
Ladder logic	http://en.wikipedia.org/wiki/Ladder_logic
Limnor	http://www.limnor.com/
MSTworkshop	http://en.wikipedia.org/wiki/MST_Workshop
MVPL	http://msdn.microsoft.com/en-us/robotics/aa731536.aspx
OpenDX	http://www.opendx.org/
OpenMusic	http://recherche.ircam.fr/equipes/repmus/OpenMusic/
OpenWire	http://www.mitov.com/html/openwire.html
Prograph	http://en.wikipedia.org/wiki/Prograph
Quest 3D	http://quest3d.com/
SCADE	http://en.wikipedia.org/wiki/SCADE
Simulink	http://www.mathworks.com/products/simulink/
ThingLab	http://en.wikipedia.org/wiki/ThingLab
Virtools	http://a2.media.3ds.com/products/3dvia/3dvia-virtools/
VisSim	http://www.vissim.com/
XUML	http://www.executableumlbook.com/

Table 4 – information sources of VPLs

10.3 VPL's descriptions appendix

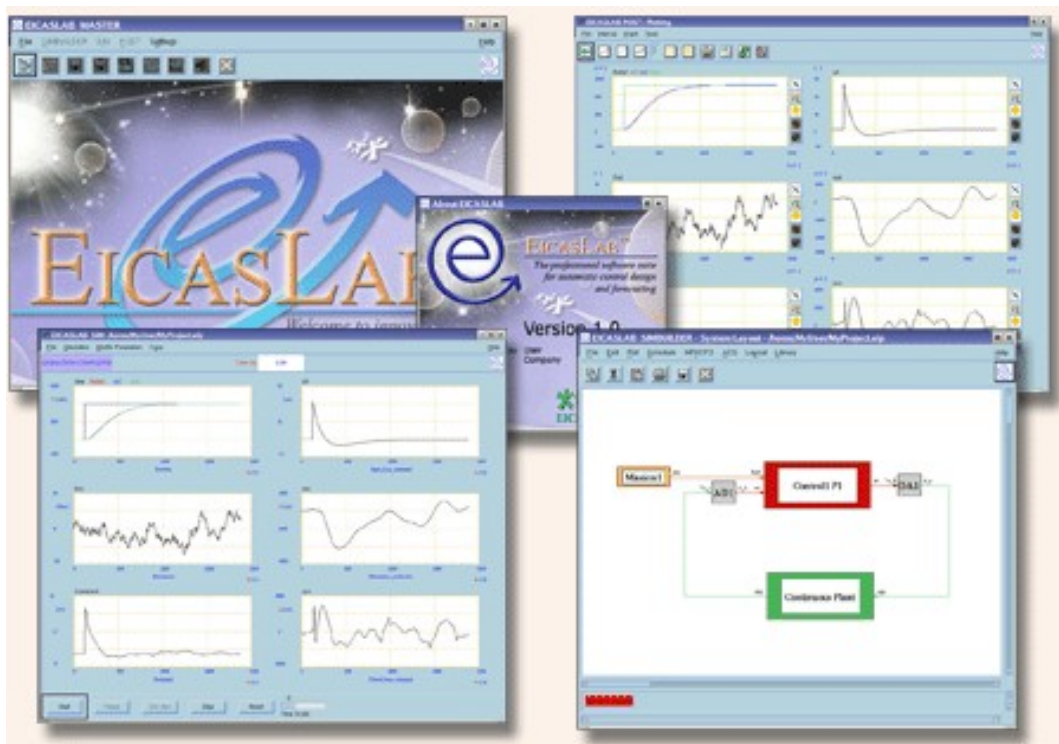
10.3.1 EICASLAB

The professional software suite for automatic control design and forecasting – represents an innovative approach to the design of automatic controls.

EICASLAB has been conceived and developed as a highly professional software suite supporting designers of automatic control system architectures.

EICASLAB supports the automation of industrial processes through powerful tools for modelling plants, designing and testing embedded control systems.

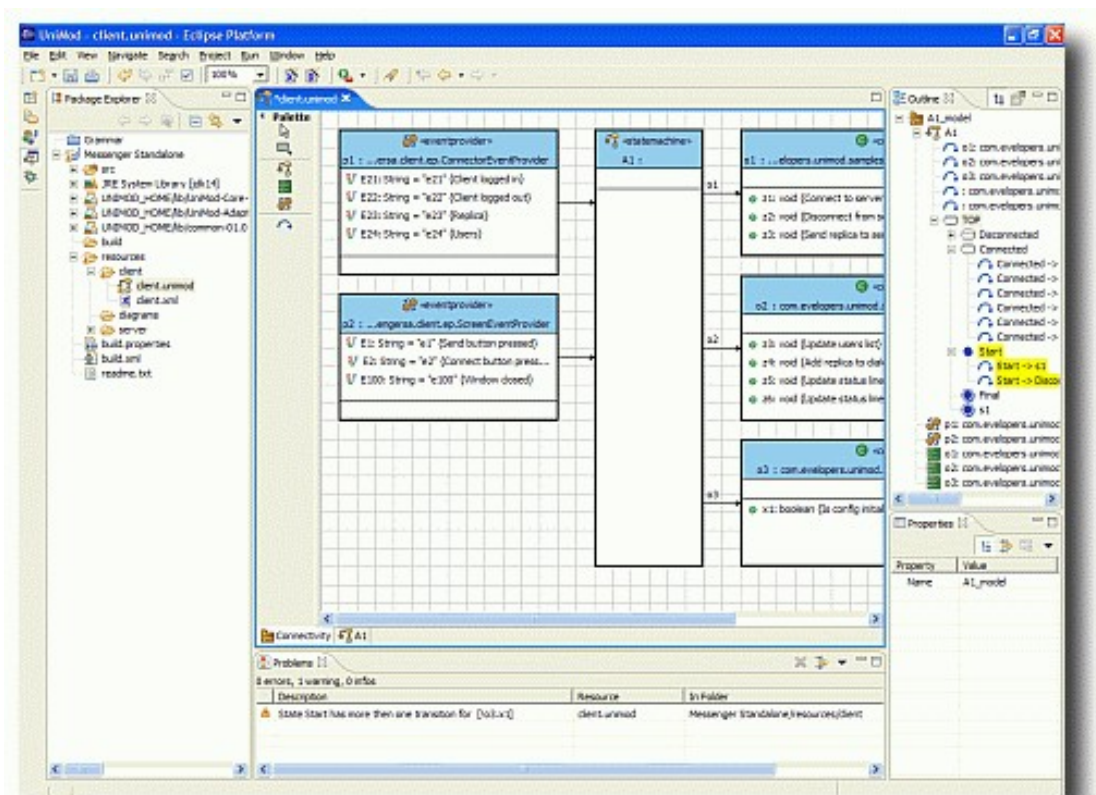
EICASLAB assists all phases of the design process of the control strategy: from system concept to finalised control structure.



10.3.2 Executable UML

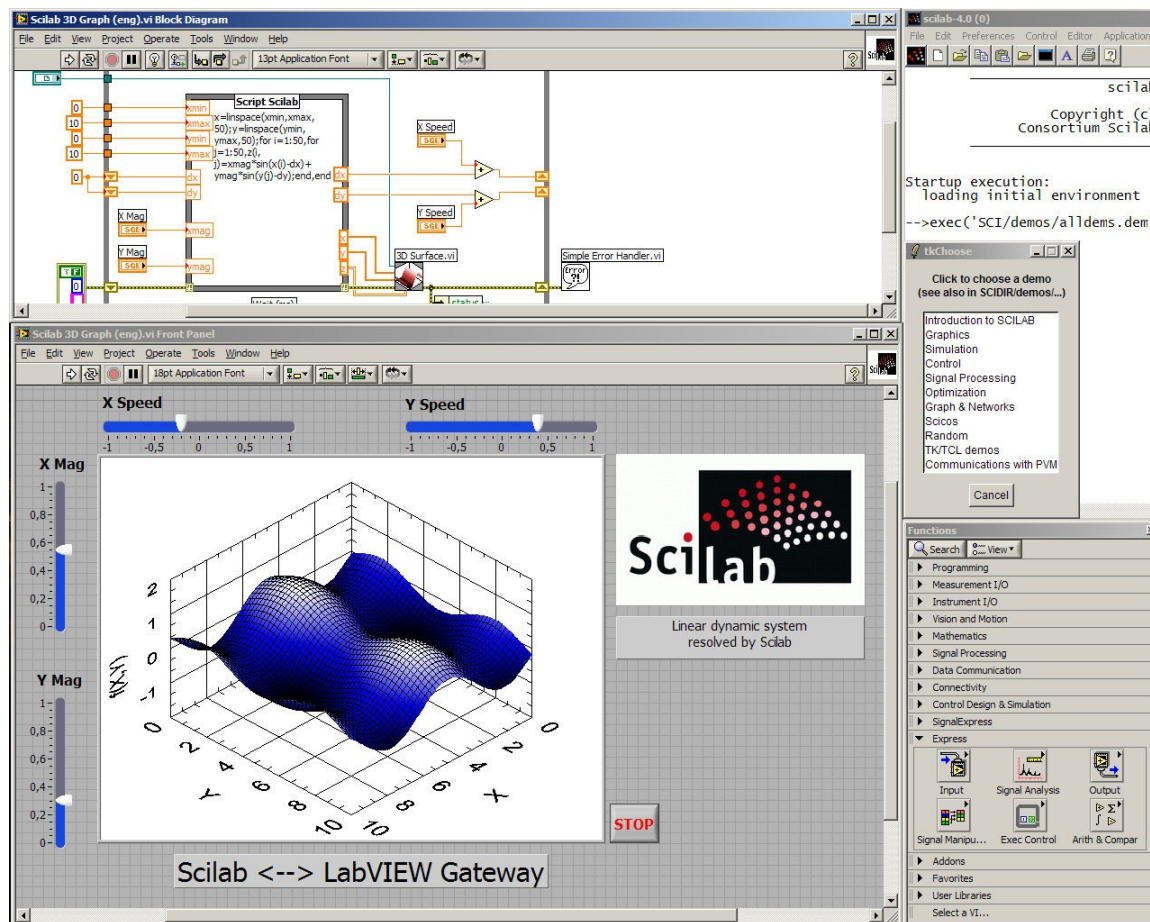
Executable UML is used to model the domains in a system. Each domain is defined at the level of abstraction of its subject matter independent of implementation concerns. The resulting system view is composed of a set of models represented by at least the following:

- The domain chart provides a view of the domains in the system, and the dependencies between the domains.
- The class diagram defines the classes and class associations for a domain.
- The statechart diagram defines the states, events, and state transitions for a class or class instance.
- The action language defines the actions or operations that perform processing on model elements.



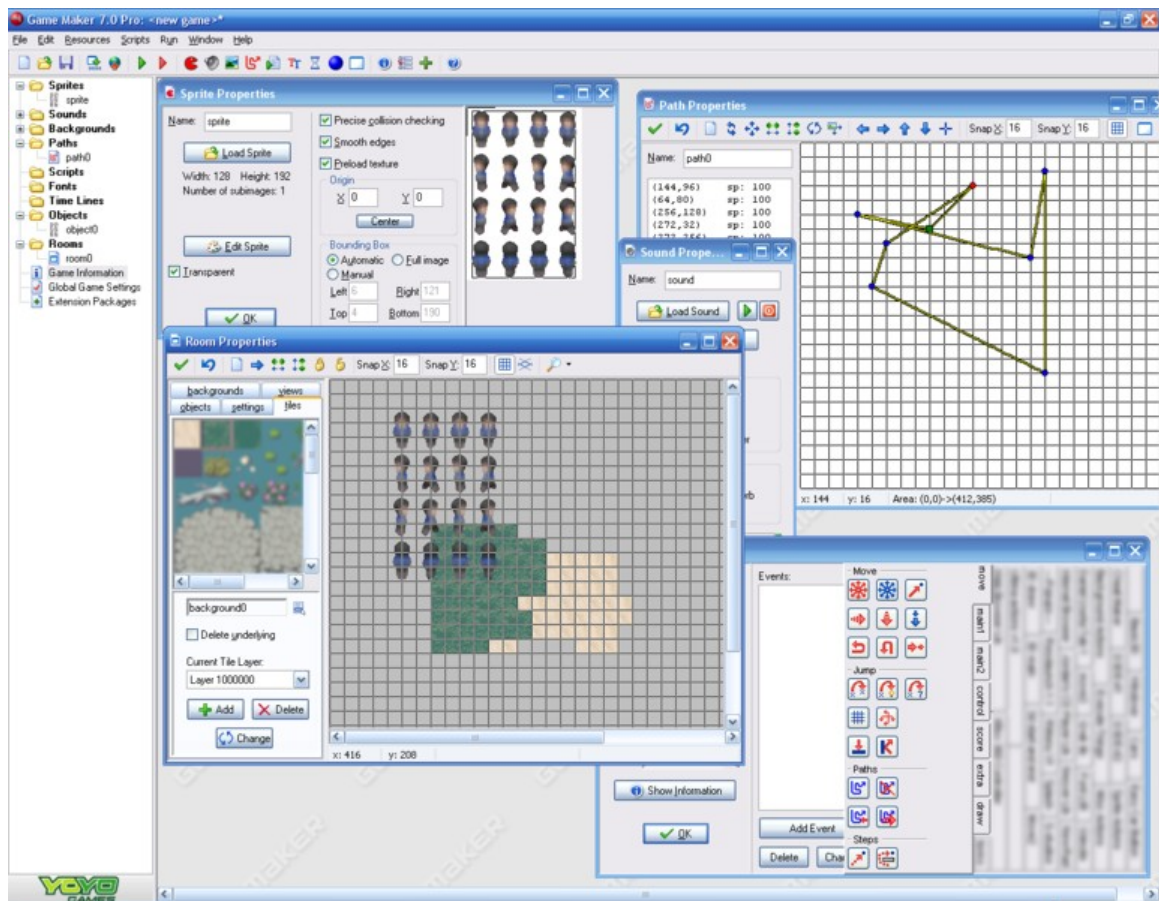
10.3.3 Labview

Labview is a graphical programming environment used by millions of engineers and scientists to develop sophisticated measurement, test, and control systems using intuitive graphical icons and wires that resemble a flowchart. LabVIEW offers unrivaled integration with thousands of hardware devices and provides hundreds of built-in libraries for advanced analysis and data visualization. The LabVIEW platform is scalable across multiple targets and operating systems, and since its introduction in 1986 has become an industry leader.



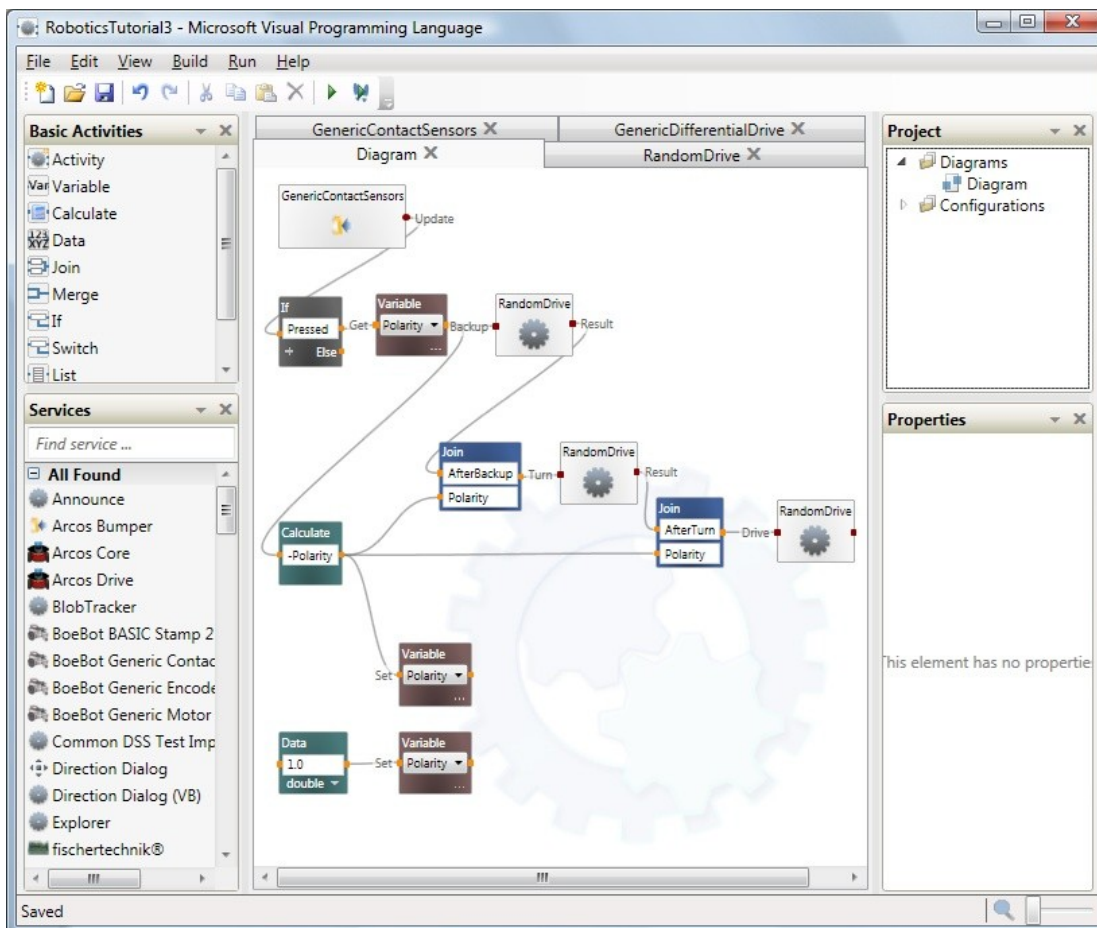
10.3.4 Game maker

This program is designed to allow its users to easily develop computer games without having to learn a complex programming language such as C++ or Java, while at the same time teaching the user basic syntax and OOP. For experienced users, Game Maker contains a built-in scripting programming language called the Game Maker Language (GML), allowing the user to further customize their game and expand features. Games can be distributed under any license subject to the terms of Game Maker's EULA, in non-editable executable .exe files or as .gmk (Version 7.x), .gm6 (Version 6.x), .gmd (Version 5.x and 4.x), and .gmf (Version 3) source files. Users of Game Maker are allowed to distribute and even sell their creations as long as they comply with the terms of the Game Maker EULA, which prohibits a number of illicit programs such as those which involve unauthorized use of copyrighted material or those which are unlawful in general.



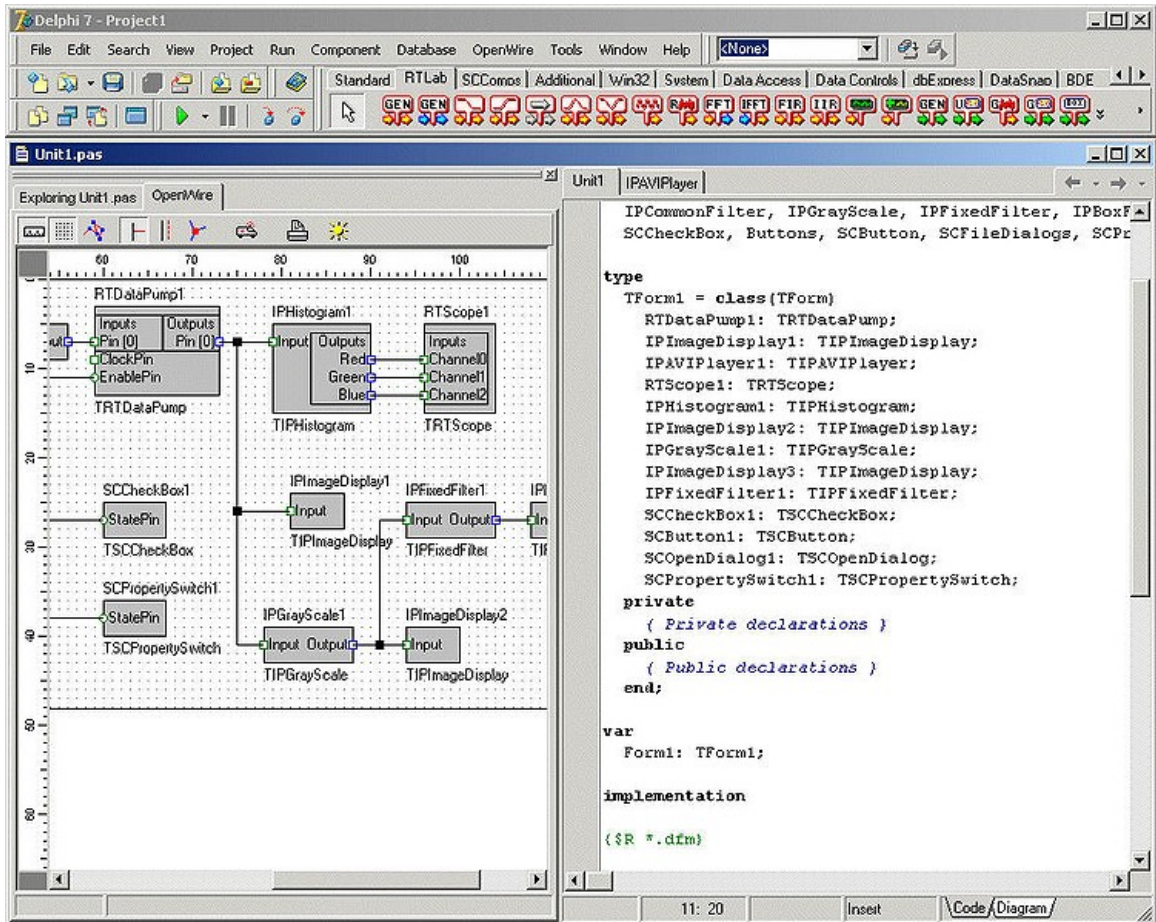
10.3.5 MVPL

Microsoft Visual Programming Language, or MVPL, is a visual programming and dataflow programming language developed by Microsoft for the Microsoft Robotics Studio. The Microsoft Visual Programming Language is distinguished from other Microsoft programming languages such as Visual Basic and C#, as it is the only Microsoft language that is a true visual programming language. Microsoft has utilized the term "Visual" in its previous programming products to reflect that a large degree of development in these languages can be performed by "dragging and dropping" in a traditional wysiwyg fashion.



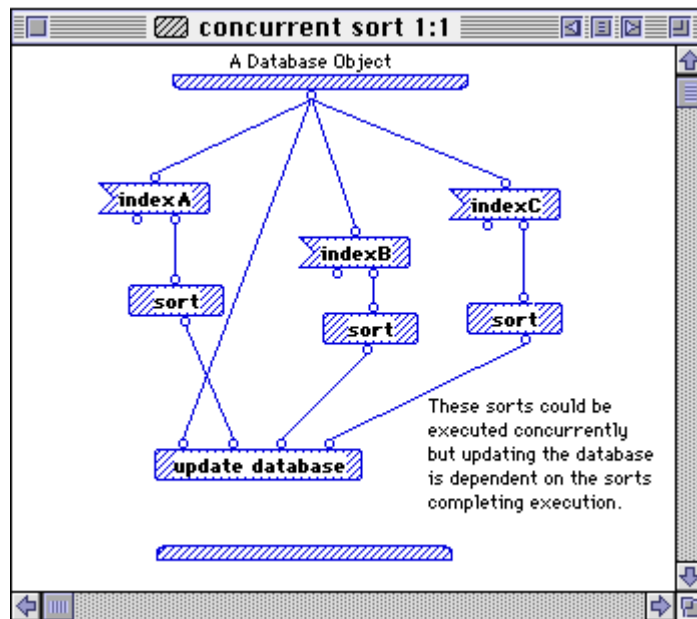
10.3.6 Openwire

OpenWire is an open source Dataflow programming VCL library that extends the functionality of Delphi and Lazarus by providing pin type component properties. The properties can be connected to each other. The connections can be used to deliver data or state information between the pins, simulating the functionality of LabVIEW.



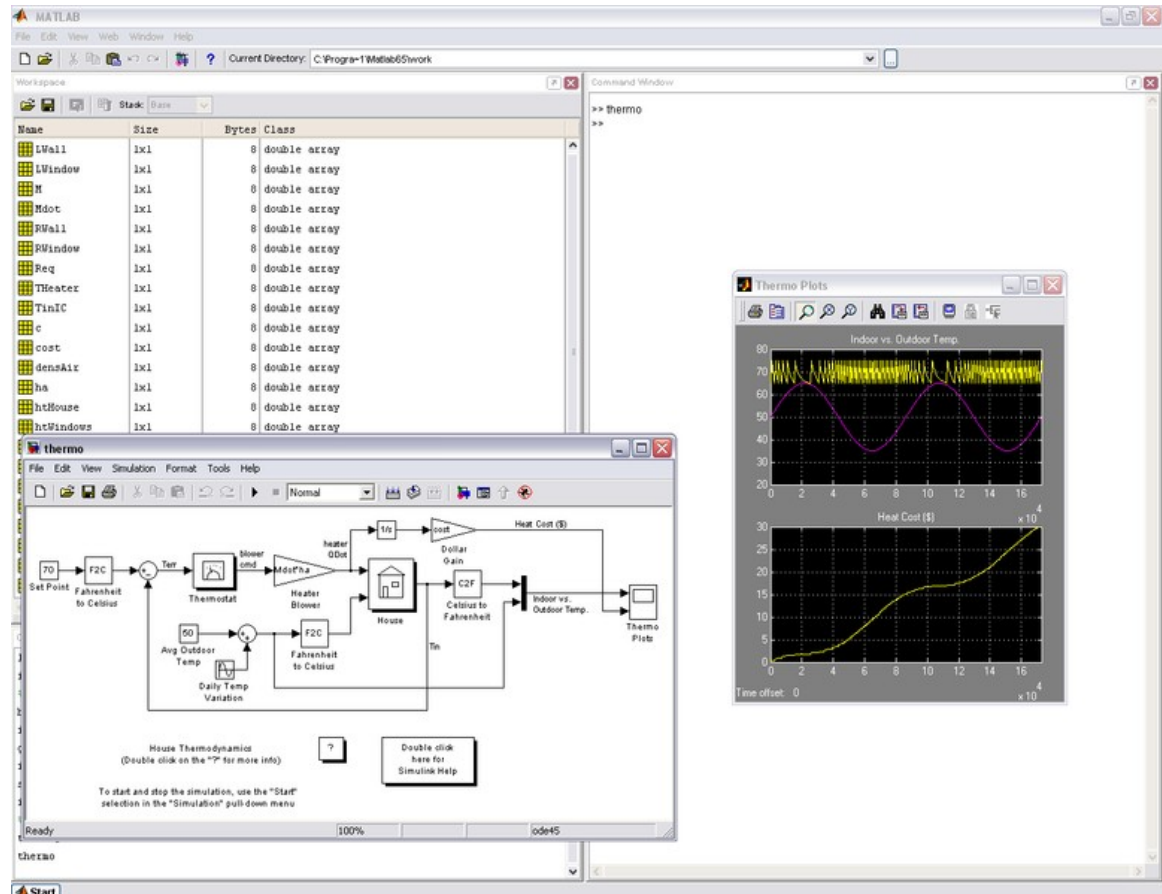
10.3.7 Prograph

Prograph is a visual, object-oriented, dataflow, multiparadigm programming language that uses iconic symbols to represent actions to be taken on data. Commercial Prograph software development environments such as Prograph Classic and Prograph CPX were available for the Apple Macintosh and Windows platforms for many years but were eventually withdrawn from the market in the late 1990s. Support for the Prograph language on Mac OS X has recently reappeared with the release of the Marten software development environment.



10.3.8 Simulink

Simulink, developed by The MathWorks, is a commercial tool for modeling, simulating and analyzing multidomain dynamic systems. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries. It offers tight integration with the rest of the Matlab environment and can either drive Matlab or be scripted from it. Simulink is widely used in control theory and digital signal processing for multidomain simulation and design.



10.3.9 VisSim

VisSim is a visual block diagram language for simulation of dynamical systems and Model-based embedded system development. It is developed by Visual Solutions of Westford, Massachusetts. VisSim is widely used in control system design and digital signal processing for multidomain simulation and design. It includes blocks for arithmetic, Boolean, and transcendental functions, as well as digital filters, transfer functions, numerical integration and interactive plotting. The free VisSim Viewer lets anyone run VisSim diagrams. Coupled with VisSim/C-Code, an add-on product, VisSim performs code generation for real-time implementation of embedded systems. It can target small 16-bit fixed point systems like the Texas Instruments MSP430 (using only 340 bytes flash and 64 bytes of RAM for a small closed loop PWM actuated system) as well as larger 32-bit floating point processors like Texas Instruments C6713. This technique of simulating system performance off-line, and then generating code automatically from the simulated diagram to run on the embedded system is known as "model based development". Model based development is becoming widely adopted for production systems because it shortens development cycles. VisSim Version 7 has introduced 3D plotting and interactive 3D VRML animation.

