CGP4Matlab - A Cartesian Genetic Programming MATLAB Toolbox for Audio and Image Processing

Rolando Miragaia[1], Gustavo Reis[1], Francisco Fernandéz[2], Tiago Inácio, and Carlos Grilo[1]

[1] School of Technology and Management,
Computer Science and Communications Research Centre,
Polytechnic Institute of Leiria, Portugal
{firstname.lastname}@ipleiria.pt,
[2] University of Extremadura, Spain,
fcofdez@unex.es

Abstract. This paper presents and describes CGP4Matlab, a powerful toolbox that allows to run Cartesian Genetic Programming within MATLAB. This toolbox is particularly suited for signal processing and image processing problems. The implementation of CGP4Matlab, which can be freely downloaded, is described. Some encouraging results on the problem of pitch estimation of musical piano notes achieved using this toolbox are also presented. Pitch estimation of audio signals is a very hard problem with still no generic and robust solution found. Due to the highly flexibility of CGP4Matlab, we managed to apply a new cartesian genetic programming based approach to the problem of pitch estimation. The obtained results are comparable with the state of the art algorithms.

Keywords: cartesian genetic programming, cgp, matlab toolbox, audio processing, image processing, pitch estimation, automatic transcription of music

1 Introduction

Cartesian Genetic Programming (CGP) has already demonstrated its capabilities on synthesizing complex functions, extracting main features from images and performing image segmentation [1].

Although there are a number of public domain genetic algorithm and genetic programming toolboxes for MATLAB, there are no toolboxes for cartesian genetic programming. CGP4Matlab was developed as a contribution to the community, providing a free toolbox that can be used and extended by other researchers, allowing them to benefit from MATLAB's great mathematical potential on audio and image processing. Also, with this toolbox, researchers that already work with genetic programming in MATLAB are now able to try the cartesian version of genetic programming.

CGP4Matalb toolbox is generic and flexible enough to be applied in any kind of audio or image processing problems. It is completely free and available for download at `https://github.com/tiagoinacio/CGP4Matlab`. This toolbox has already been useful on solving some simple problems such as linear regression, and also on addressing the problem pitch estimation of piano music [2].

The next section describes the cartesian genetic programming process. Section 3 describes the CGP4Matlab architecture and its implementation. In Section 4, we describe a new approach to the picth estimation problem using our toolbox and show our experimental results. Finally, Section 5 presents our conclusions.

2 Cartesian Genetic Programming

Genetic Programming is a type of evolutionary algorithm based on Darwin's theory of evolution, where in each generation (iteration) exists a population of possible solutions (candidate solutions) to the problem, which are referred as individuals. During each iteration, all the individuals are evaluated by an evaluation function, often referred to as fitness function. After the evaluation, individuals are submitted to a process of selection, where the best are preferably chosen. Those individuals can be recombined and suffer mutations. The resulting individuals will constitute the next population in the new generation. **Cartesian Genetic Programming** (CGP) grew out of the work of Miller et al. [3], as a method of evolving digital circuits. However, the term "Cartesian Genetic Programming" appeared two years later in [4]. According to Miller [4], CGP is more efficient than standard GP methods in learning Boolean functions.

CGP is *Cartesian* because it encodes programs as a two-dimensional grid of nodes that are addressed in the *Cartesian* coordinate system (see Section 2.2). In its classic form, it uses a very simple integer based genetic representation of a program in the form of a directed graph instead of a tree. Graphs are very useful program representations, more general than trees.

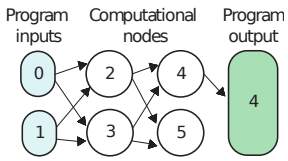

Fig. 1. Overall strucuture of a CGP program. Program inputs and computational nodes are numbered sequentially. The program outputs can link to any computational node or program input.

2.1 Programs

CGP programs have three major components: program inputs, computational nodes and program outputs. **Computational nodes** are structures organized

and composed by input connections and a function. The input connections of a node have their origin in any program input or other precedent nodes. The function is among the ones previously defined in a look-up table and it takes as arguments the values received through the node's inputs. The node itself is indexed by an integer value so that it can be referenced by other node input connections. The computational nodes, organized in a two-dimensional grid of nodes, are numbered sequentially and linked directly between them in a feed-forward manner (see Figure 1). A program can have several inputs, named **program inputs**. **Program outputs** are indexes that link to some nodes. For example, if the program's output is the number 4, the result of the program is the value computed by node 4's function (see Figure 1). Program inputs and nodes are referenced by sequential numbers. The idea is best explained with a simple example. In Figure 1 we can see that the program has two inputs, four nodes and one output.

2.2 Genotype

The genotype is the codification of a program as it is used and manipulated by the CGP algorithm. It describes what are the programs inputs, computational nodes, program outputs and how they are connected together. In general, it is a list of genes where each gene is an integer. As we have seen earlier, program inputs and nodes are referenced by their index. Since a node is a structure with input connections and a function, each node has multiple genes (see Figure 2). The genetic structure that encodes a node first references the function value and then the values of the node's connections sources. In Figure 2, the list of genes to encode the node are: 2 3 4.

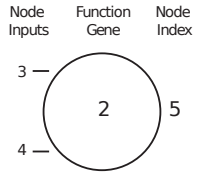

Fig. 2. Example of a node that has two connection genes: node 3 and node 4. It computes the function number 2 in the function-set. The node is referenced by the number 5.

Each node has a function gene which is an address in a look-up table of functions. Usually, all functions have as many inputs as the maximum function arity and unused connections are ignored. This introduces an additional redundancy into the genome. In the example of Figure 2, the node 5 has nodes 3 and 4 has inputs and it applies the function number 2 defined previously. If function 2 represents a *sum*, node 5 would compute the following:

$$y = c_1 + c_2, \tag{1}$$

where c_1 is the value coming through the first connection and c_2 is the value coming through the second connection. If $c_1 = 2$ and $c_2 = 1$, the value of node 5 would be $2 + 1 = 3$ (see Figure 3).

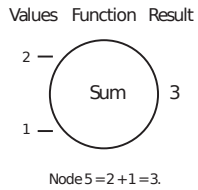

Fig. 3. The result of node 5 will be 2 + 1 = 3.

There are a few number of parameters that we need to define in order to encode a CGP program. The number of program inputs is given by ni and the number of program outputs is given by no. Given that nodes are organized in a tabular way, the number of columns is given by nc and the number of rows by nr. For example, the program in Figure 4 has the following attributes: $ni = 3$, $no = 1$, $nc = 3$, $nr = 1$ and the genotype is the following list of integers: 512 303 102 and 4. Knowing that $ni = 3$, the genotype encodes the first node at index 3, since the first three indexes represent the program inputs and the first index is 0. The first node in the genotype, node 3, computes function 5, and its connections are the program input 1 and program input 2. Node 4 computes function 3, and its connections are the program input 0 and the value of node 3. The output of that program is the value of node 4. We point that there are no program outputs nor nodes whose input connections reference node 5. This means that this node cannot influence the program output.

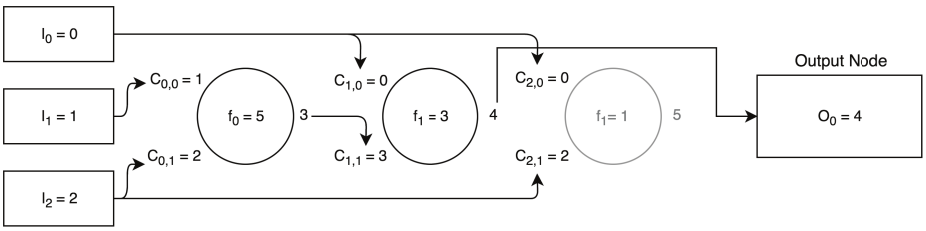

Fig. 4. CGP graph, where $ni = 3$ and $no = 1$. The grid has $nc = 3$ (columns) and $nr = 1$ (row).

Table 1, enumerates a few parameters of the program illustrated in the above figure.

Table 1. Parameters of the program illustrated in Figure 4

Parameter	Value
Number of Inputs (ni)	3
Number of Outputs (no)	1
Number of Rows (nr)	1
Number of Columns (nc)	3
Inputs (i_i)	0,1,2
Functions (f_i)	5,3,1
Outputs (O_i)	4
Genotype	512 303 102 4
Phenotype	512 303 4

There are some allelic constrains, that the genotype must respect. The alleles (values) of function genes f_i must take valid address values in the look-up table of primitive functions. Let nf represent the number of allowed functions. Then f_i must obey to the following range:

$$0 \leq f_i < nf. \tag{2}$$

There is another parameter called **levels-back** l, which determines how many previous columns of nodes may connect to a node in the current column. When $nr = 1$ and $l = nc$, any node can have input connections coming from any program input and any node on its left, which allows unrestricted connectivity. However, if $nr > 1$, nodes cannot connect to other nodes in the same column. Then, having a node in column j, and $j \geq l$, node connections, C_{ij}, must obey to the following range:

$$ni + (j - l)nr \leq C_{ij} \leq ni + j \times nr. \tag{3}$$

If $j < l$, then the following condition must be met:

$$0 \leq C_{ij} \leq ni + j \times nr. \tag{4}$$

Program output genes O_i can connect to any node or program input:

$$0 \leq O_i < ni + Ln, \tag{5}$$

where Ln is the number of nodes in the genotype, computed by the following:

$$Ln = nr \times nc. \tag{6}$$

This representation is very simple, flexible and convenient for many problems.

2.3 Genotype-Phenotype

One of the key characteristics of CGP is the genotype-phenotype mapping. The genotype is of fixed-length but the phenotype is not, due to the fact that the genotype can have inactive genes. Thus, they are redundant because they cannot influence the programs output. The corresponding genes are called **non-coding genes** or **inactive genes**. This means that we can have a phenotype different

from the genotype because non-coding genes are not expressed in the phenotype, that is, the program that will run in practice.

The output or outputs of the CGP are nodes that point to other nodes (connection genes) and so on. Decoding the program is recursive in nature and works from the program output genes first. To decode the program outputs, the active nodes should be identified. The process begins by looking at which nodes are directly connected to the output genes. Then, these nodes are examined to find out which nodes are directly linked to them. Since non-coding genes are not addressed, they present little computational overhead.

2.4 Algorithm

The evolutionary strategy widely used for CGP is a special case of the strategy $\mu + \lambda$ [5] where $\mu = 1$ (Algorithm 1). This means that, in this special case, the population size is always one. At each iteration (generation), λ new offspring are generated from the current one through mutation. Then, the best among the current individual and the offspring becomes the current individual in the next iteration. An offspring can become the current individual in the next iteration when it has the same fitness as the current individual and there is no other individual with a better fitness.

Algorithm 1 Algorithm $((1 + \lambda)\, EA)$

1: $t \leftarrow 0$;
2: Set current individual I_0 as the best of λ individuals created randomly;
3: **while** a stop condition is not fulfilled, **do**
4: **for** i = 1 to λ **do**
5: Create a copy x_i of current individual I_t;
6: Mutate each gene of x_i with probability p;
7: **end for**
8: Set new current individual I_{t+1} as the best of $I_t \cup \{x_1, \ldots, x_\lambda\}$;
9: $t \leftarrow t + 1$;
10: **end while**

3 Cartesian Genetic Programming Toolbox

For the first step of our research on applying cartesian genetic programming to sound processing, we decided to create a MATLAB Toolbox for audio processing. The idea was to have a highly flexible toolbox, configurable throughout parameters and function callbacks, so that, we could move and focus on the problem of Pitch Estimation by applying and configuring the same toolbox to our particular case.

The CGP4Maltab's architecture will be introduced throughout this section. Then, each component will be explained in detail.

3.1 Architecture

The CGP Toolbox is very simple to use and allows to quickly encode a problem. The structure of classic CGP is reproduced in the toolbox. One of the main goals was to have a generic toolbox that could help us to encode from smaller to bigger problems. With that in mind, a few design decisions were made that will be explained next. All the combinations of rows and columns are possible, considering that $nr > 0$ and $nc > 0$. The allelic constrains are generated dynamically, depending on the cartesian representation of the nodes. *Levels-back* was also taken into consideration. Additionally, the toolbox is prepared to use parameters in the genotype. There are no limits for the number of parameters. The fitness function is any function provided by the user. The toolbox is prepared to receive one or more program inputs of any types and values. The number of program outputs can be one or more, in order to address different problem requirements. The function-set is also provided by the user and the look-up table is automatically generated. Furthermore, there is a system of callbacks which is discussed later. The Evolutionary Algorithm (EA) used is the $1 + \lambda$, referred previously in Section 2.4. The goal is to have a toolbox as generic as possible, so a few parameters for the evolutionary process were chosen to be configurable.

The number of **offspring** (λ) is defined by the user. This is useful because there can be some problems that require a small number of offspring and others that require a bigger number of offspring. The **mutation rate** (p) is also configurable. This is the mutation probability for each gene. The maximum number of **runs**, mr, and maximum number of **generations**, mg, are also required parameters. Finally, the last parameter is the maximum or minimum **fitness**, f, for a solution to be considered valid, depending if we want to maximize or minimize the fitness function. The EA needs to know when a candidate solution can be considered as a valid solution for the problem, in order to stop the evolutionary process.

The toolbox is divided into several components (see Figure 5). Each one has its purpose and special role. The first one is the **CGP** component. It exposes all the functionality to encode an application built on top of the toolbox. This component communicates with the **EA** and **Structure** components. The Structure is just an helper, which stores the positions of the genes according to the type of gene (connection, function, program output and parameter). The EA component is responsible for initializing the runs in the evolutionary algorithm. It starts with a certain number of **Offspring**, created by the **Genotype** component which, in turn, is composed by the **Connection**, **Functions**, **Ouputs** and **Fitness** components. **Run** is connected to the **Generation** component, by executing it multiple times. In each generation, **Mutation** can occur, which will change the genotypes (using the Connection, Functions, Outputs and Fitness components). Figure 5 shows the overall structure of the toolbox's components. Each component will be addressed in detail next.

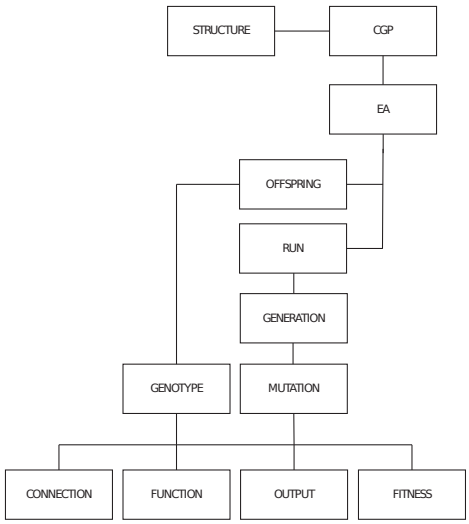

Fig. 5. Components that are part of the toolbox.

3.2 Classes

The toolbox was built using Object Oriented Programming methodology of MATLAB (version R2016a). All the *classes* that compose the toolbox will be introduced throughout this section. For some classes, a detailed explanation of the most relevant properties and methods is also presented.

CGP The *CGP* class provides access to an API that lists all the features needed to encode a program. It is the *core* of the toolbox and its primary component. The *CGP* class lets the user add the program inputs, provide the fitness function, add parameters and define the function-set. The constructor takes a configuration object. This object will contain all the configuration necessary for the CGP and for the EA.

For the CGP, the parameters are divided into: number of rows, number of columns, number of levels back and number of program outputs. Since some CGP approaches assume that the output node is the last node of the graph, this option was also taken into consideration. So, if we pass the value *last* to the *output_type*, the last node of the genotype will be considered the program output. This option only works when the number of program outputs is set to 1, otherwise it will be ignored. Having these parameters configurable, the user has total control of the grid layout of the generated program (genotype).

For the EA, the parameters are: maximum number of generations, maximum number of runs, number of offspring, mutation rate, the fitness threshold and the fitness operator. The fitness threshold is the limit for which a candidate solution is considered a valid solution to the problem. This allows the evolutionary process to stop or skip to the next run. In some kind of problems, the goal is to minimize

an error rate, where 0 would be the best value for the fitness. Also, there are other problems where the goal is to maximize the fitness function. The *fitness_solution* property covers that necessity. However, the operator to use in the comparison between fitness values also needs to be configurable, because the optimization of those values is different. The fitness operator, O, is the operator to use when comparing the new fitness candidate solution with the parent's fitness, and can take the following values: '>', '<', '>='and '<='. For example, consider the parent's fitness as f_0 and an offspring fitness as f_1: if O = '>', $f_0 = 0.5$ and $f_1 = 0.6$, then the offspring will replace the parent in the new generation; if O = '<', $f_0 = 0.5$ and $f_1 = 0.6$, the parent will remain as it has the best fitness. This operator is also used for checking if a solution is a valid solution for the given problem. Therefore, it is also used for comparison between a solution's fitness and the *fitness_solution* value, also configurable. Table 2 describes every possible field for the configuration.

Table 2. Configuration table with the fields that the structure should have, the type of value and the description of each one.

Key	**Type**	**Description**
rows	double	number of rows
columns	double	number of columns
levels_back	double	number of levels-back
outputs	double	number of outputs
output_type	string	set the program output as the last node (last, random)
runs	double	number of runs
generations	double	number of generations
offspring	double	number of offspring
mutation	double	probability of mutation
fitness_solution	double	fitness for a solution to be considered valid
fitness_operator	string	fitness operator (>,<,>= or <=)

At the time of instantiaton, the *CGP* class verifies if all the required settings were passed in the configuration object. This class also exposes the functionally of adding program inputs. Each problem requires a specific set of program input or inputs. Some may require one integer as input, others may require an array, or even a complex type of object. To address this abstraction, the input provided for the CGP toolbox is of type **struct** (structure). Each field in the structure is a program input. Therefore, the program inputs can be of any type: integers, strings, structs, arrays, matrix, etc. The number of fields present in the structure indicates the number of inputs that the toolbox needs to set in the genotype, which is dynamically set: there is no need to specify how many program inputs the programs will have.

The fitness function is passed by callback (function pointer) to the program.

The toolbox reads the function set from a specific directory provided by the user. This directory should have all the functions that could be used in the genotype. All the functions should receive as many inputs as the maximum function arity. This is a requirement for the program to work. Besides the maximum function arity, if the user added parameters to the genotype, these should also be

passed to each function. This method iterates through all the MATLAB files in the directory passed as argument, and it creates a function handle for each one.

Some specific signal processing functions might require special arguments like ranges or constants to be executed (e.g.: a low pass filter needs to know which percentage of the original signal will be attenuated). Those parameters might need to evolve through time, because their best values for the contribution to the solution of the problem is unknown beforehand. The genotype can encode those parameters and add them to the evolutionary process. Parameters should have integer or double values. Each parameter is encoded by a structure with a name, a callback function for the initialization of the parameter value, and another callback function for mutating the value. The initialization and mutation functions should return an integer or a double. The mutation function should also accept an argument, that is the value of the current parameter to mutate. When running the algorithm, there are a number of events from the evolutionary process that can be useful to handle, for running additional scripts or simply to add some kind of report. In order to have that range of possibilities, the user is able to pass optional callbacks, each of which, will fire at the following events: the configuration has been set, a fittest solution is achieved after a run, a fittest solution is achieved in a generation, a new solution is created, a new generation starts, a new run starts and a genotype is mutated. After adding all the program inputs, fitness function, parameters and callbacks, the configuration callback is fired, with a few useful parameters about the configuration of the program.

Structure There are several components that need to know how many genes are in the genotype, or if a specific gene is a function-gene or a connection gene. Instead of having to determine those properties multiple times and at different stages, this information is only computed once, in this class. The *Structure* class serves as an helper throughout the entire evolutionary process. The main goal is to classify each gene *a priori*, according to its type: connection, paremeter, program output or function. For example, if we have 3 genes per computational node and our genotype starts at number 1 (MATLAB does not accept zero-based vectors), we know in advance that gene 1 represents a function and genes 2 and 3 both correspond to connections. Since this class is responsible for defining the type of genes, it needs to know a few parameters, such as: the number of genes, the number of genes per node, the connection genes per node, the number of computational nodes and the number of parameters.

EA The *EA* class is responsible for starting the evolutionary process. It iterates for the maximum number of runs, defined in the configuration of the CGP, storing the fittest candidate solution of each one.

If the callback *Run Ended* is provided, it will be fired after each run, with a few parameters, such as the genes of the fittest solution and their fitness.

Run The *Run* class is responsible for initializing a run. First, it generates a few candidate solutions. Then, it will start the evolutionary loop over the genera-

tions. The class stores the best candidate solution, while evaluating if a solution for the problem was found.

The *Run* class contains two callback events. The *Fittest Solution* occurs when a candidate solution has better fitness than the previous stored solution. The *Generation Ended* occurs each time a new generation ends.

Generations The *Generation* class is responsible for initializing a new generation. It starts with the previous fittest candidate solution (parent), and generates a few mutated versions, according to the configuration provided. If the λ chosen in the configuration phase is 4, it will generate four mutated versions of the parent solution. All the new genotypes are evaluated, and the fittest solution is stored.

The *Generation* class contains two callback events: *New Solution In Generation* and *Fittest Solution Of Generation*. The first, occurs every time a new solution is generated. The last one, occurs each time a new solution is generated and has a better fitness than the parent.

Offspring The *Offspring* class is responsible for the initialization of a specific number of offspring, previously defined, at random, before iterating through the generations. It initializes randomly different genotypes which are then evaluated. The fittest solution is stored and used as the parent solution, for the generation loop initialization.

Genotype The *Genotype* class is responsible for the creation of a genotype, restricted to the configuration provided: number of columns, number of rows, number of program inputs, parameters, and so on. First, the function genes are added to the genotype. Then, the connection genes are randomly generated, as well as the parameters and program outputs. After the genotype is created, the active nodes are recursively found by analyzing the program outputs. For each output, the connection nodes are retrieved and stored in an array. For each of those, their connections are also saved in that array, and so on. This process stops when there are no more nodes to analyze. Lastly, the fitness of this new candidate solution is computed.

Connection The *Connection* class is responsible for generating a random and valid connection for a specific node. It receives the connection gene index as argument. The class first finds which node belongs the connection gene. This is done by subtracting the number of program inputs from the gene index and dividing that value by the number of genes per node. Then, it finds all the possible connections for that node. This is achieved by recursively iterating through the previous nodes, taking into account that nodes in the same row cannot be connected between each other, and also taking into account the number of levels-back. Lastly, it randomly pick one connection from the possible connections.

Functions The *Functions* class is responsible for randomly generating the function genes for the genotypes. It takes into account the number of functions present in the function-set, to be able to generate valid function genes. It can generate one function gene at a time or multiple function genes. This is is useful, because we find where all the function-genes are positionated in the genotype, and call this class once, which returns function genes to all those positions. If we have 10 nodes, we have to generate 10 function-genes in the genotype. If our function-set is composed by 5 functions, this class generates 10 random values between 1 and 5, each corresponding to a function-gene mapped to one of the functions in the function-set.

Output The *Output* class is responsible for generating a valid program output. Depending on the settings provided initially, this class can pick the last node to be the program output, or randomly pick any program input or computational node in the genotype.

Fitness The *Fitness* class is responsible for calling the fitness callback provided in the configuration phase. A few properties are passed to that callback, such as the genes in the genotype, active nodes, function-set, program inputs and others. It has a validation of the type returned by the function, which should return an integer or double value. The returned value, is stored and used as the fitness of that particular candidate solution.

Mutation The *Mutation* class receives a genotype and iterates over its genes. All the genes have the same mutation probability. For a gene being mutated, we first find what type of gene it is: connection, function, parameter or program output. If it is a program output, the *Output* class is used. If it is a connection gene, the *Connection* class is used. If it is a function gene, the *Functions* class is used. Recall that when we add parameters to the CGP, we must provide an initialization function and a mutation function. If it is a parameter gene, the mutation function provided is called.

After iterating all genes, the active nodes are found again, and the fitness is recalculated. If the *Genotype Mutated* callback is provided, it will be called, having as arguments the genes before the mutation, the genes after the mutation and the index of the mutated genes.

4 Using CGP approach to pitch estimation on piano notes

As mentioned before, the CG4Matlab toolbox has already proven to be useful during our first approach on addressing the pitch estimation problem [2]. However, to better demonstrate the capabilities of the developed toolbox, we decided to extend our previous work and propose a new approach. The problem of Pitch

estimation on sound signals, also known as F0 detection, is a very important task of Automatic Music Transcription.

Music transcription is a very difficult problem from both musical and computational points of view: although there has been much research devoted to it, it still remains an unsolved problem. Over the years, there has been a lot of research on Pitch Estimation [6,7,8,9,10]. However, to the best of our knowledge, there are no Cartesian Genetic Programming approaches for addressing this problem.

In our CGP approach to Pitch Estimation, we have multiple inputs and we have only one row of graph nodes, one output (the result of the corresponding classifier), and $levels\text{-}back = nc$. To perform pitch detection using CGP, we developed a system where some important decisions and tasks were made besides the CGP. We had to define what kind of inputs to use from the original piano audio signal, through a preprocessing task. We also had to develop a process to reach a binary output in order to perform our fitness function.

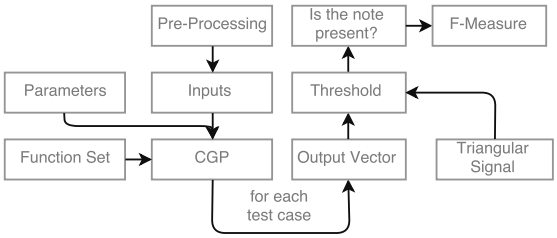

Fig. 6. System architecture.

The block diagram of our proposed system is much more than a simple CGP process and is depicted in Figure 6. Our goal is to train 61 classifiers, each one corresponding to one pitch or piano note: from C1 to C6. To train one classifier, we first start with a set of learning cases: a group of audio signals corresponding to the pitch that we want to identify and a group of audio signals without that pitch. Those audio signals are pre-processed in order to extract some important features that will be used as program inputs like, for example, the magnitude spectrum. The computational nodes in the genotype have two connection inputs, one function and two parameters. Each program is an evolved mathematical function, which is applied to each of the learning cases. The output of that function is compared to a triangular signal, where a threshold is applied, for binary classification. After the binary classification of all learning cases, the fitness function is applied.

The polyphonic audio signals of the piano notes were extracted from the MAPS database [11]. This is a huge data-set with multiple piano samples, chords and melodies in wave format.

4.1 Experiments and Results

Table 3 shows the values of the configurable parameters for our system. The evolutionary process consisted of 30 runs with 5000 generations each, using 50 positive and 50 negative cases for each musical note. The number of computational nodes is 100. The classifiers were evaluated using the F-measure.

Table 3. List of parameters used in the experiments.

Parameter	Value
Frame Size	4096
Fitness Threshold	0.5
Positive Test Cases	50
Negative Test Cases	50
Outputs	1
Rows	1
Columns	100
Levels Back	100
Offspring	4
Mutation Probability	5%
Runs	30
Generations	5000

After the training process, each classifier was tested with a different test set. Each test set consisted in 144 negative notes (48×3) and 5 positive notes, comprising a total of 149 piano sound samples. Table 4 shows our results. We made a more complete set of tests then the preliminary results, and we trained and tested 61 different classifiers. These results are very encouraging, since for almost all notes we achieved a classifier with F-Measure values greater than 70%.

Table 4. Test results for 61 classifiers

classifier	tp	tn	fp	fn	f-measure
24	5	138	6	0	0.63
25	5	127	17	0	0.37
26	5	127	17	0	0.37
27	5	127	17	0	0.37
28	5	127	17	0	0.37
29	5	124	20	0	0.33
30	4	122	22	1	0.26
31	5	108	36	0	0.22
32	5	132	12	0	0.46
33	4	138	6	1	0.53
34	5	111	33	0	0.23
35	5	139	5	0	0.66
36	5	140	4	0	0.71
37	5	121	23	0	0.30
38	5	140	4	0	0.71
39	5	113	31	0	0.24
40	4	130	14	1	0.35
41	4	138	6	1	0.53
42	4	124	20	1	0.28
43	4	138	6	1	0.53

classifier	tp	tn	fp	fn	f-measure
44	5	112	32	0	0.24
45	5	135	9	0	0.53
46	3	138	6	2	0.43
47	4	119	25	1	0.24
48	5	135	9	0	0.53
49	5	136	8	0	0.55
50	5	140	4	0	0.71
51	5	127	17	0	0.37
52	5	138	6	0	0.63
53	5	142	2	0	0.83
54	5	128	16	0	0.39
55	5	138	6	0	0.63
56	5	128	16	0	0.39
57	5	139	5	0	0.67
58	5	139	5	0	0.67
59	5	137	7	0	0.59
60	5	142	2	0	0.83
61	4	142	2	1	0.73
62	4	144	0	1	0.88
63	4	144	0	1	0.88

classifier	tp	tn	fp	fn	f-measure
64	5	138	6	0	0.63
65	5	141	3	0	0.77
66	5	139	5	0	0.67
67	5	141	3	0	0.77
68	5	141	3	0	0.77
69	5	141	3	0	0.77
70	5	142	2	0	0.83
71	5	142	2	0	0.83
72	5	142	2	0	0.83
73	5	144	3	0	0.77
74	5	146	1	0	0.91
75	5	142	5	0	0.66
76	5	142	5	0	0.66
77	5	146	1	0	0.91
78	5	143	4	0	0.71
79	5	144	3	0	0.77
80	5	143	4	0	0.71
81	5	147	0	0	1
82	5	147	0	0	1
83	5	146	1	0	0.91
84	5	145	2	0	0.83

The graph depicted in Figure 7 shows, besides F-measure, the error rate in percentage for the data-set test with 96ms frames. Our pitch estimator using

cartesian genetic programming reaches the mean error rate of 6%. When compared to the state of the art, these are very encouraging results. According to Emiya [12], the three main monophonic pitch estimators are: Parametric F0 estimator, the Nonparametric F0 estimator and the YIN estimator [13] and those estimators have mean error rates of 2.4%, 3.0% and 11.0% respectively. Our CGP approach to F0 estimation reaches the mean error rate of 6%.

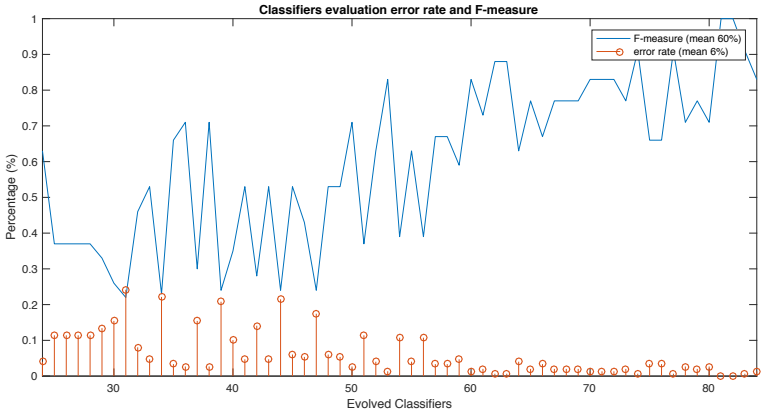

Fig. 7. Graph with 61 classifiers evaluation results in error rate and F-measure.

5 Conclusion

This paper presented the CGP4Matlab toolbox. This toolbox is generic and flexible enough to be applied to any kind of signal or image processing problems. Its internal architecture and modules were also presented and discussed.

A cartesian genetic programming strategy for addressing the pitch recognition of piano notes was also presented using our toolbox. The obtained results show the feasibility of our approach. Also, the results accomplished with the CGP technique are in line with the most popular algorithms for pitch recognition on piano notes.

Planning ahead, we aim to continue our research on addressing of polyphonic pitch estimation using cartesian genetic programming with our toolbox.

6 Acknowledgements

The authors would like to thank Spanish Ministry of Economy, Industry and Competitiveness and European Regional Development Fund (FEDER) under projects TIN2014-56494-C4-4-P (Ephemec) and TIN2017-85727-C4-4-P (DeepBio); Junta de Extremadura FEDER, projects GR15068, GRU10029 IB16035

Regional Government of Extremadura, Consejería of Economy and Infrastructure, FEDER.

References

1. Harding, S., Leitner, J., Schmidhuber, J.: Cartesian genetic programming for image processing. In: Genetic Programming Theory and Practice X, pp. 31–44. Springer (2013)
2. Inácio, T., Miragaia, R., Reis, G., Grilo, C., Fernandéz, F.: Cartesian genetic programming applied to pitch estimation of piano notes. In: Computational Intelligence (SSCI), 2016 IEEE Symposium Series on. pp. 1–7. IEEE (2016)
3. Miller, J., Thomson, P., Fogarty, T.: Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study 219 (1997), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.27.7671
4. Miller, J.F.: An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In: Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2. pp. 1135–1142. GECCO'99, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999), http://dl.acm.org/citation.cfm?id=2934046.2934074
5. Hansen, N., Arnold, D.V., Auger, A.: Evolution Strategies, pp. 871–898. Springer Berlin Heidelberg, Berlin, Heidelberg (2015), http://dx.doi.org/10.1007/978-3-662-43505-2_44
6. Yeh, C., Roebel, A., Rodet, X.: Multiple fundamental frequency estimation and polyphony inference of polyphonic music signals. Trans. Audio, Speech and Lang. Proc. 18(6), 1116–1126 (Aug 2010), http://dx.doi.org/10.1109/TASL.2009.2030006
7. Klapuri, A.P.: Multiple fundamental frequency estimation based on harmonicity and spectral smoothness. IEEE Transactions on Speech and Audio Processing 11(6), 804–816 (Nov 2003)
8. Reis, G., Fernandéz de Vega, F., Ferreira, A.: Audio analysis and synthesis-automatic transcription of polyphonic piano music using genetic algorithms, adaptive spectral envelope modeling, and dynamic noise level estimation. IEEE Transactions on Audio Speech and LanguageProcessing 20(8), 2313 (2012)
9. Marolt, M.: A connectionist approach to automatic transcription of polyphonic piano music. IEEE Transactions on Multimedia 6(3), 439–449 (June 2004)
10. Mueller, M., Wiering, F. (eds.): An efficient temporally-constrained probabilistic model for multiple-instrument music transcription. ISMIR, Malaga, Spain (October 2015)
11. Emiya, V., Bertin, N., David, B., Badeau, R.: Maps-a piano database for multipitch estimation and automatic transcription of music (2010)
12. Emiya, V., David, B., Badeau, R.: A parametric method for pitch estimation of piano tones. In: 2007 IEEE International Conference on Acoustics, Speech and Signal Processing-ICASSP'07. vol. 1, pp. I–249. IEEE (2007)
13. De Cheveigné, A., Kawahara, H.: Yin, a fundamental frequency estimator for speech and music. The Journal of the Acoustical Society of America 111(4), 1917–1930 (2002)