






Article

# Imperative Genetic Programming

Iztok Fajfar , Žiga Rojec , Árpád Bűrmen, Matevž Kunaver , Tadej Tuma , Sašo Tomažič and Janez Puhan 

Faculty of Electrical Engineering, University of Ljubljana, Tržaška 25, 1000 Ljubljana, Slovenia

\* Correspondence: iztok.fajfar@fe.uni-lj.si; Tel.: +386-1-476-8722

**Abstract:** Genetic programming (GP) has a long-standing tradition in the evolution of computer programs, predominantly utilizing tree and linear paradigms, each with distinct advantages and limitations. Despite the rapid growth of the GP field, there have been disproportionately few attempts to evolve ‘real’ Turing-like imperative programs (as contrasted with functional programming) from the ground up. Existing research focuses mainly on specific special cases where the structure of the solution is partly known. This paper explores the potential of integrating tree and linear GP paradigms to develop an encoding scheme that universally supports genetic operators without constraints and consistently generates syntactically correct Python programs from scratch. By blending the symmetrical structure of tree-based representations with the inherent asymmetry of linear sequences, we created a versatile environment for program evolution. Our approach was rigorously tested on 35 problems characterized by varying Halstead complexity metrics, to delineate the approach’s boundaries. While expected brute-force program solutions were observed, our method yielded more sophisticated strategies, such as optimizing a program by restricting the division trials to the values up to the square root of the number when counting its proper divisors. Despite the recent groundbreaking advancements in large language models, we assert that the GP field warrants continued research. GP embodies a fundamentally different computational paradigm, crucial for advancing our understanding of natural evolutionary processes.

**Keywords:** evolutionary algorithms; tree genetic programming; linear genetic programming; imperative programming



check for updates

**Citation:** Fajfar, I.; Rojec, Ž.; Bűrmen, Á.; Kunaver, M.; Tuma, T.; Tomažič, S.; Puhan, J. Imperative Genetic Programming. *Symmetry* **2024**, *16*, 1146. <https://doi.org/10.3390/sym16091146>

Academic Editor: Shi Cheng

Received: 31 July 2024

Revised: 27 August 2024

Accepted: 29 August 2024

Published: 3 September 2024



**Copyright:** © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

Genetic programming (GP) is a prominent sub-field of evolutionary algorithms (EAs), simulating Darwinian processes on a computer. The GP paradigm was established by John Koza in the early 1990s [1] and had been steadily growing until recently [2]. This trend appears to have reversed with the emergence of large language models (LLMs) [3]. However, we believe that GP will continue to be a significant study area, both as a complementary approach to LLMs and an independent research topic.

Despite the rapid growth of the GP field since the 1990s, there have been disproportionately few attempts to evolve ‘real’ Turing-like programs. Most research focuses on less complex logical or arithmetic expressions, without incorporating iteration or memory [4]. One reason is that the original GP concept is not Turing complete, a limitation addressed by [5] through the introduction of indexed memory. Another reason is that the original GP paradigm encodes a program as a tree, necessitating viewing a computer program as a sequential application of functions and operators to arguments (so-called functional programming). While not a limitation per se, this is not the most natural way to conceptualize computer programs. Shortly after the traditional tree representation of a computer program in GP, linear and graph representations emerged [6,7]. In contrast to the functional programming language expressions encoded by trees in traditional GP, linear genetic programming evolves sequences of instructions from an imperative programming language. The difference in program representation necessitates different genetic operators,

making both approaches even more distinct. Both approaches, tree and linear, have their respective advantages and disadvantages, and researchers and practitioners select one based on the specific requirements of their problem. There has been criticism that the GP concept has an intrinsic flaw in that it cannot produce real software effectively [2,4], primarily because computer code is not as robust as genetic code. It is extremely susceptible to even the smallest changes. This is one of the main reasons that the existing research focuses mainly on specific special cases where the structure of the solution is partly known [8,9]. The vast majority of studies are limited to symbolic regression and classification problems [2,10–22]. Other important domains where GP is being used include, but are not limited to, control systems [23,24], analog optimization [8,25–28], scheduling [29,30], and image processing [31,32]. To the best of our knowledge, no systematic research has been conducted to evolve general Turing-like (imperative) programs (as contrasted with functional programming) from scratch, with no a priori assumptions on the solution structure.

The contribution of this paper is twofold. First, we introduce a computer program representation that merges tree and linear representations, using trees for expressions and a linear representation for encoding the overall computer program. We use Python as a programming language for generated programs. Second, we systematically apply our approach to several well-known algorithms of varying complexities to identify the limits of the proposed method.

The structure of this paper is as follows. Sections 2 and 3 detail the encoding of programs and the methodology for generating a concrete program from this encoding. Section 4 describes the evolutionary algorithm employed, while Section 5 outlines the overall experimental setup. The final sections present and discuss the results.

## 2. The Proposed Program Genotype

### 2.1. The Basic Idea

A computer program is a linear sequence of instructions, generally asymmetrical. However, a program can contain nested conditional and loop statements, which imply an inherently symmetrical tree structure. Expressions also exhibit a tree structure. This symmetrical/asymmetrical duality of a computer program was the most important issue we had to address when devising the structure of our genotype. Another critical consideration was the possibility of a randomly created loop-controlling expression resulting in an infinite loop or a loop with an unreasonably high number of iterations. Limiting the number of loop iterations is crucial when composing and executing thousands of randomly generated programs.

We encoded the program itself as a linear set of statements. Whenever there is a control statement header, a certain number of the following lines form the statement's body. That number is stored with the header and is subject to evolutionary operations.

Most authors address the problem of non-halting programs or programs with excessive loop iterations by setting an upper limit on the number of executed instructions. We adopted a slightly different approach by limiting the number of iterations for any loop. This method prevents the potentially destructive effects of genetic operators, which could compromise the program by including parts of already functioning code in a loop. If the code is such that successive repetitions have no different effects than a single iteration, the program will continue to function correctly. Conversely, if the number of instructions is limited, parts of the code outside the loop may never execute.

We implemented the upper limit on loop iterations using a for loop combined with a break statement. For example, we encode a while loop with a control expression *expr* and an upper limit of *maxIter* iterations in the following manner:

```
for i ← 1 to maxIter do
  if not expr then break
end if
// Loop body comes here
```

**end for**

Here, the loop iterator  $i$  is a safeguard, while  $expr$  is the control expression of the equivalent while loop:

```
while  $expr$  do
  // Loop body comes here
end while
```

## 2.2. The Structure of the Genotype

We composed the genotype of a program with a fixed number of consecutive lines, thus eliminating bloat and simplifying genetic operators. To make the system even simpler and more robust, every program line has the same structure, containing the necessary information to be decoded into any possible line depending on its position in the program. That way, we are always able to build a syntactically correct program. The consequence of this universality is a large amount of redundant code stored in our genotype. The redundant pieces of code are not expressed in the phenotype (an actual program) and are usually referred to as introns. This redundancy seems like a downside, but it is also believed that introns reduce search space and speed up the convergence by dynamically hiding the genotype segments not needed for the ultimate solution [33].

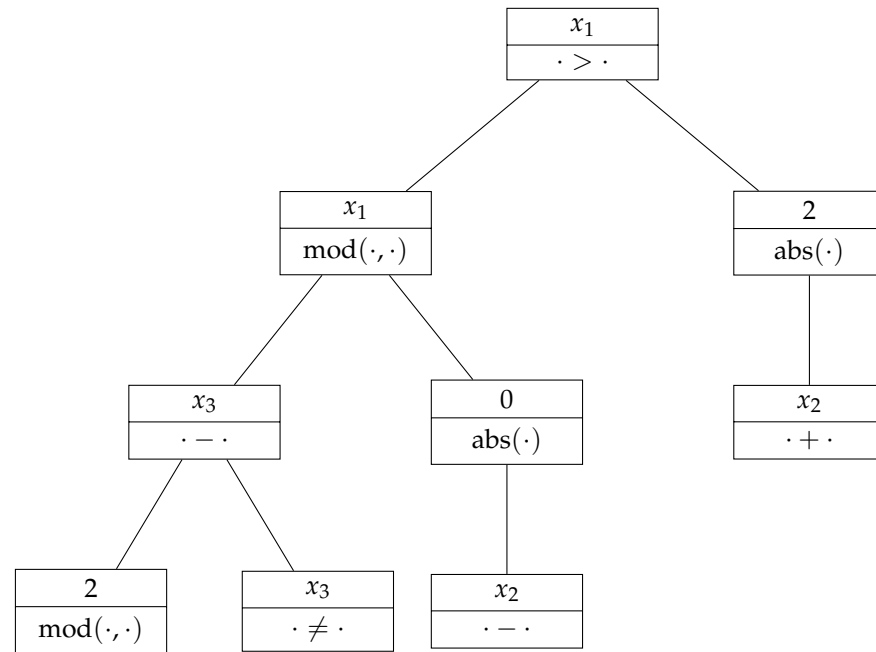
The structure of a single line of code is depicted in Figure 1. After the type of the line, which can be an assignment, a macro, a control statement's header, or simply the pass placeholder, the line also includes the parameter  $bodyLen$  (holding the control statement body length), the expression tree, the macro index, and the list of variables. The  $bodyLen$  parameter is only relevant when the line type is a control statement's header, and its value ranges from 2 to the maximum body length (see Table 1).

assignment/macro/for/if/else/pass
$bodyLen$
An expression tree
$macroIndex$
A list of variables ( $varList$ )

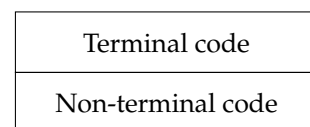
**Figure 1.** The structure of a single line of a program.

Figure 2 shows an example of an expression tree. Each node of the expression tree contains two code snippets: the first one is used when the node is terminal, while the second one is used when the node is non-terminal (see Figure 3). As seen in Figure 2, the root of the tree—a non-terminal node in our case—produces the greater-than operator, which will compare the expressions derived from the left and right subtrees. We derive the expression  $abs(x_2)$  from the right subtree because the first node is non-terminal while the second is terminal. In the same way, we use the mod,  $-$  (minus), and  $abs$  from the three non-terminal nodes of the left subtree, and 2,  $x_3$ , and  $x_2$  from the three terminal nodes. Thus, from this tree, we derive the expression  $mod(2 - x_3, abs(x_2)) > abs(x_2)$ .

It is probably worth mentioning at this point that we have limited ourselves to using only scalar variables. A serious consequence is that functions that can be added as prefabricated elements to our genetic material cannot return more than a single value. Note that although Python works exclusively with references, one still cannot pass a scalar variable by reference to obtain an output value from a function. This limitation does not allow us to use, for example, a function that swaps the values of two variables. For that reason, we added macros to our genotype. The parameter  $macroIndex$  is used to select a macro from the list of predefined macros.



**Figure 2.** An example of a tree representing the expression  $\text{mod}(2 - x_3, \text{abs}(x_2)) > \text{abs}(x_2)$ .



**Figure 3.** The structure of a single node of an expression tree.

Finally, the list of variables holds as many variables as there are placeholders to fill in the largest macro from the list. If the line type is an assignment, then the first from the variable list will be used as a left value in the assignment.

Notice that some data contained in program lines and expression trees may appear redundant in specific contexts. They are nevertheless retained to standardize crossover, mutation, and code extraction procedures. Moreover, these 'redundant' data might encapsulate hidden genetic material that was once beneficial and could prove useful again (see, e.g., [34]).

At the beginning of each evolutionary run, we need a population of randomly generated programs, which are constructed using several parameters summarized in Table 1 together with a short explanation of their meaning. The first parameter limits the number of variables used in the generated program. All the variables share the same name prefix with added numbers (e.g.,  $x_0, x_1, x_2, \dots$ ). Next comes a list of operators and function names. The operators must be selected among the standard Python operators. At the same time, the function names can be arbitrary as long as they are defined separately and their definitions added to the list of function definitions. Following the list of macro definitions and constants is a list of probabilities indicating the likelihood of each line type being selected during the initial random genotype creation. Those probabilities also guide the random line type selection during the mutation procedure.

The limitation of the depth of control statement nestings is also important. More than a single nested loop is hardly necessary. At the same time, it would be extremely time-consuming if we allowed it. On the other hand, it is important to allow deeper nesting of a conditional statement since there are a lot of cases in which such a statement comes in handy when nested in an already nested loop. Apart from loop nesting, the maximum number of iterations should also be limited, lest the programs could unreasonably slow down the evolution.

As expressions in our programs need not be too complex, we limited the expression tree depth. Initial depth is limited to two but grows later due to bloat. We employed two mechanisms to fight bloat. The first is a limit on the depth to which a tree is evaluated, and the second is a limit on the maximum allowed depth by trimming a tree after each crossover and mutation. The first of the two limits is lower, so some hidden genetic material usually stays in a tree. Note that, technically, it is not a problem to limit the evaluation depth since every node includes a code to be used in both cases—when the node acts as a terminal or as a non-terminal. The evaluation algorithm simply selects the proper one and ignores the other.

The last three parameters in Table 1 represent three more important limits on evaluation trees' densities, the overall program length, and the maximum control statement body length. Note that the actual program length can be shorter than the given length because of possible pass statements.

**Table 1.** The program parameters used in our experiments. Values in parentheses are default values used in our experiments. If there is no default value (N/A), the value must be explicitly provided for each experiment by the practitioner.

Number of variables (5)	How many variables will be used in the program
Operators (N/A)	List of operators and/or function names
Function definitions (N/A)	List of function definitions
Macro definitions (N/A)	List of macro definitions
Constants (N/A)	List of constants
Line type probabilities (assignment = 0.55, for = 0.10, if = 0.15, else = 0.10, pass = 0.10)	How probable it is, during a random program generation and mutation, for a certain line type to be selected
Macro selection probability (0.15)	If at least one macro is defined, this probability is added to the above list. All the probabilities are proportionally reduced so that they sum to 1
Maximum loop nesting (1)	The maximum allowed depth of loop nesting
Maximum if nesting (2)	The maximum allowed depth of conditional statement nesting
Maximum loop iterations (100)	The maximum allowed number of iterations of a single loop
Tree generation depth (2)	The maximum initial depth of an expression tree
Tree evaluation depth (3)	The maximum depth to which an expression tree is evaluated
Maximum tree depth (5)	The depth to which trees are trimmed after crossover and mutation
Tree density (0.7)	The density of an expression tree during generation and mutation
Program length (15)	The number of lines in the program
Maximum body length (5)	Maximum number of statements within a control statement body

### 3. Building a Python Program

Algorithm 1 illustrates the construction of actual Python code from the genotype introduced in Section 2. To thoroughly comprehend the algorithm, one must understand the role of indentation in Python code. In Python, the body of a control statement consists of indented lines. All indented lines belong to the statement's body. Conversely, the first line with the same indentation level as the control statement's header is no longer part of that control statement but it succeeds it.

**Algorithm 1** Algorithm for constructing a Python program from genotype.

---

```

1: procedure COMPOSEPYTHONPROGRAM
2:   indent  $\leftarrow$  0
3:   for each line in the program do
4:     if lineType = assignment then
5:       code  $\leftarrow$  code + varList[0] + "=" + expressionFromTree()
6:     else if lineType = for then
7:       if indent < maxLoopNest then
8:         code  $\leftarrow$  code + "for i in range(" + maxIter + ")"
9:         indent  $\leftarrow$  indent + 1
10:        code  $\leftarrow$  code + "if " + expressionFromTree() + ": break"
11:       end if
12:     else if lineType = if then
13:       if indent < maxIfNest then
14:         code  $\leftarrow$  code + "if " + expressionFromTree() + ":"
15:         indent  $\leftarrow$  indent + 1
16:       end if
17:     else if lineType = else then
18:       if inside an if or for block then
19:         if the block contains at least one line of code then
20:           indent  $\leftarrow$  indent - 1
21:           code  $\leftarrow$  code + "else:"
22:           indent  $\leftarrow$  indent + 1
23:         end if
24:       end if
25:     else if lineType = macro then
26:       selectedMacro  $\leftarrow$  macroList[macroIndex]
27:       Replace placeholders in selectedMacro with variables from varList
28:       code  $\leftarrow$  code + selectedMacro
29:     else if lineType = pass then
30:       code  $\leftarrow$  code + "pass"
31:     end if
32:     if number of statements in current block equals bodyLen then  $\triangleright$  The block has
33:       indent  $\leftarrow$  indent - 1  $\triangleright$  reached the maximum allowed length
34:     end if
35:   end for
36:   code  $\leftarrow$  code + "pass"  $\triangleright$  In case the last line of the program is if, for, or else.
37: end procedure

```

---

In Section 2, we saw that the genotype of an individual program consists of multiple sequential lines of code. Each line is divided into five distinct parts, with the first part specifying the line's type. This type determines how subsequent parts are utilized during code construction. Algorithm 1 operates through a loop that processes each line of the program genotype. Prior to this loop, the code indentation is initialized to zero (see Line 2). Within the loop, various line types are addressed using an if–else chain. The first type in this sequence, denoted as 'assignment' (see Line 4), builds an assignment statement. Here, *varList*[0] (i.e., the first variable in the list) is used as the variable name on the left side of the assignment operator, while the expression derived from the corresponding expression tree forms the right side. Note that both these elements—the variable list and the expression tree—are encapsulated within the line's genotype. Note that *code* is a string, with the + symbol serving as a concatenation operator. Moreover, every line of code from Algorithm 1 starts with the correct indentation and ends with a newline character—details omitted from the algorithm for clarity.

The next line type addressed in the algorithm is the for statement (see Line 6). It is important to note that this line is bypassed once the maximum permitted loop nesting depth is reached. Otherwise, the algorithm generates a two-line code segment. The initial



line forms a conventional Python loop executing *maxIter* times. Recall that *maxIter* sets the upper limit for the number of loop iterations. Following this, an indented if statement is introduced—indicating its inclusion within the loop body—which triggers an early loop exit when its expression evaluates as true. As explained in Section 2.1, this design effectively creates a while loop equipped with a safeguard against excessive iterations, crucial to prevent undue hindrance in the evolutionary process.

The if statement (Line 12) is managed similarly. It is skipped entirely when the maximum permitted nesting depth is attained. Following the addition of the control statement header, the indentation increases to ensure that the following lines fall within the statement body.

The else part, handled in Line 17, is incorporated only when nested within the body of an if or for statement that contains at least one line of code. Note that in Python, a for loop can also have an optional else segment, which is executed upon loop completion. However, it will not be executed if the loop is interrupted by a break statement. Recall, in our context, the for loop functions akin to a while loop with a capped number of iterations. Hence, the else segment only comes into play when this iteration limit is met. Such a design can prove beneficial during evolution, though it might be extraneous in final programs. When integrating the else keyword, the indent level is reduced beforehand and then increased afterward, effectively closing the current block and commencing a new one.

Line 25 processes a macro. It simply takes a macro from the list and replaces its placeholders, in order of appearance, with the variables from the variable list.

The final line type addressed is the pass statement (see Line 29), serving as a placeholder for potential subsequent code. After that (see Line 32), it is necessary to verify if the current block length is reached, and if so, conclude that block by decreasing the indentation. Upon completing the program, an additional pass statement is appended (see Line 36) to avoid an error if the program's last line is a control statement header.

#### 4. The Evolutionary Algorithm

The evolutionary algorithm is outlined in Algorithm 2. First, some initialization procedures are carried out in Lines 2 to 4. After some preliminary runs, we settled with the genetic parameters summarized in Table 2 that we used in all our experiments. We explain the meaning of each parameter later on in the context of the algorithm. The program parameters, however (see Line 3 of Algorithm 2), are initialized differently for each run, depending on the type of program we want to evolve. Recall that the used program parameters are summarized in Table 1 at the end of Section 2.

**Table 2.** The genetic parameters used in our experiments.

Population size	1000
Array of training data length	20
Selection	Linear ranking with elitism
Selection pressure	1.3
Elite size	10
Number of generations	2000
Mutation probability	0.5
Line crossover probability	0.3
Number of crossover lines	4
Toggle terminal probability	0.3
Mutation depth	2
Fitness calculation method	Least squares
Premature stopping criterion 1	Fitness does not change for 600 generations
Premature stopping criterion 2	Fitness drops to zero

Line 4 of Algorithm 2 prepares the training data. This is simply an array of input/output pairs of data that our program should produce. For instance, if we wanted to evolve a program that returns the largest of three input values, we would need the

array  $[[[2, 7, 12], 12], [[-4, 56, 21], 56], [[6, -15, 3], 6], \dots]$ . The array is generated randomly using some preset limit values. It turned out that without them, the algorithm might not work in limited cases. For example, in multiplication, it is necessary to include training data involving ones and zeroes in different combinations. It is also important that these critical values appear as first and second parameters. It happened that the algorithm trained on the data missing multiplication with zero as the second parameter worked for multiplications of the form  $0 \cdot x$  but not  $x \cdot 0$ . It also happened that in evolving the algorithm detecting primes, the training array contained only even non-primes. Naturally, the evolved algorithm erroneously detected even numbers instead of primes.

The last thing we need to do before entering the main program loop is generate random programs and evaluate their fitness values. As seen in Table 2, we use the least squares method to calculate the program fitness. To do that, we run the program and then calculate the sum of squared differences between actual and required outputs. The program with the smallest fitness is the best.

---

**Algorithm 2** The evolutionary algorithm.

---

```

1: procedure EVOLVEPROGRAM
2:   Initialize genetic parameters
3:   Initialize program parameters
4:   Calculate training data
5:   Generate a population of random programs
6:   Calculate the fitness of each program in the population
7:   for generation = 1 to Number of generations do
8:     Selection
9:     for Pair of programs in selection do                                     ▷ Parents
10:      if Random value in [0,1) <  $p_{\text{line crossover}}$  then
11:        Line crossover
12:      else
13:        Program crossover
14:      end if
15:    end for
16:    for program in selection do                                       ▷ Children
17:      if Random value in [0,1) <  $p_{\text{mutation}}$  then
18:        Mutate program
19:      end if
20:      Calculate fitness
21:    end for
22:    Replacement
23:    if at least one of the premature stopping criteria met then
24:      Stop evolution
25:    end if
26:  end for
27: end procedure

```

---

#### 4.1. Selection

In the main program loop, we first select programs to participate in genetic operations. Because the fitness landscape of a computer program's population is extremely rugged, we opted for rank selection to give less fit programs an equal chance to reproduce. We used linear ranking [35], where selection probability is linearly dependent on the rank position of the individual in the population. We calculate the probability  $p$  for rank position  $r_i$  as

$$p(r_i) = \frac{1}{n} \left( sp - (2sp - 2) \frac{i - 1}{n - 1} \right), \quad 1 \leq i \leq n, \quad 1 \leq sp \leq 2.$$

Here,  $n$  is the size of the population and  $sp$  is selection pressure. Notice that  $sp = 1$  gives equal probabilities for all the population members, which means there is no selection

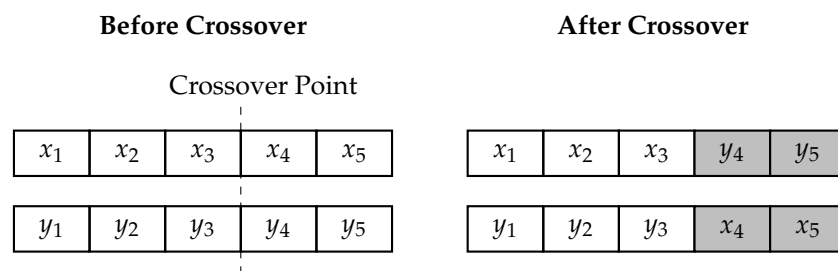


pressure. On the other hand, if  $sp = 2$ , selection pressure is very high. As seen in Table 2, we set selection pressure to 1.3, which—combined with the elite size of 10—produced the best results.

#### 4.2. Crossover

The next operation in the main program loop is crossover. We form random pairs of programs from the group selected for genetic operations and perform either a program or line crossover, depending on a preset probability  $p_{\text{line crossover}}$ .

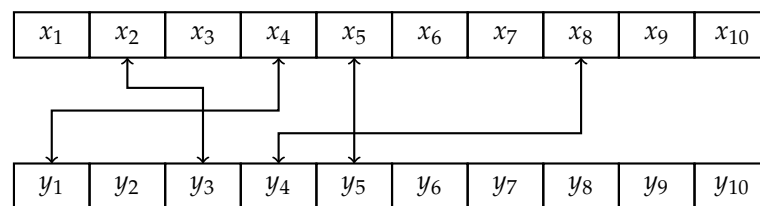
The crossover of the whole program is straightforward. We select a random cutting point (the same for both programs in question) and then swap cutoff parts of the programs as shown in Figure 4.



**Figure 4.** The crossover of two 5-line programs  $x$  and  $y$ . The crossover point lies between the 3rd and 4th lines. The operation swaps the lines  $x_4$  and  $x_5$  with  $y_4$  and  $y_5$ .

The line crossover is a little more elaborate. First, we randomly select a prescribed number of lines from each program. The number is one of the genetic parameters shown in Table 2. As the line structure is universal, there is no limit on which lines to select. Figure 5 shows an example of the line crossover of two programs. On each pair of lines, we perform one of the following crossover operations:

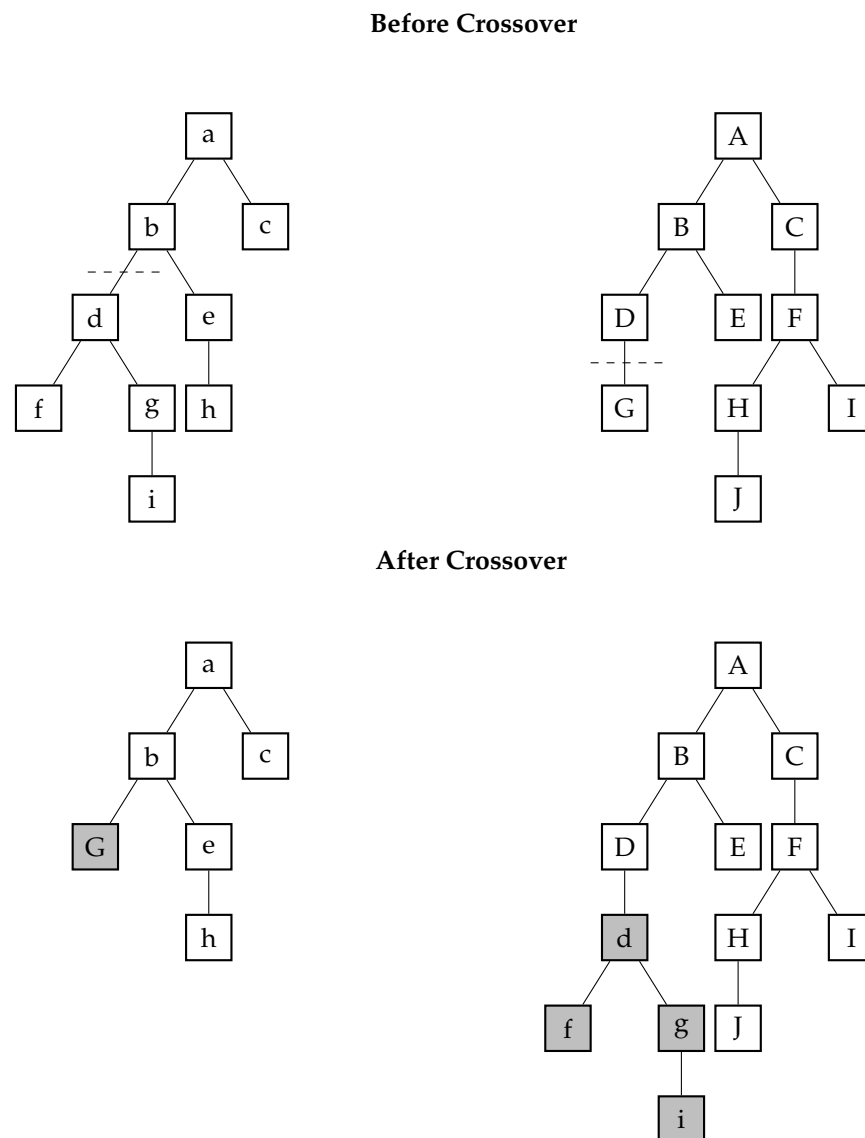
- Exchange the first part of the line (i.e., the type of statement).
- Exchange the second part of the line (i.e., the *bodyLen* parameter).
- Perform the expression tree crossover.
- Exchange the fourth part of the line (i.e., the *macroIndex* parameter).
- Perform the list of variables crossover.



**Figure 5.** An example of the line crossover of two 10-line programs  $x$  and  $y$ . Lines  $x_2$ ,  $x_4$ ,  $x_5$ , and  $x_8$  from the first program and lines  $y_1$ ,  $y_3$ ,  $y_4$ , and  $y_5$  from the second program were selected for the crossover.

The parts of the lines that hold a single value (i.e., the type of statement and the parameters *bodyLen* and *macroIndex*) are exchanged. When the crossover is performed on expression trees, the operation follows the standard tree crossover procedure where we cut a random subtree from each tree and swap the cutoff parts. Figure 6 shows the procedure. Note that, if both cutting points are above the roots of the trees, the expression trees are merely exchanged.

In order to limit bloat—as already mentioned in Section 2.2—the trees are trimmed after the crossover not to exceed the prescribed maximum depth (see Table 1).



**Figure 6.** In tree crossover, two random subtrees are selected and swapped.

The crossover on the list of variables is carried out the same way as it is on the whole program (see Figure 4). Like in expression tree crossover, the whole lists can also be exchanged if the crossover point appears before the first element of the list.

#### 4.3. Mutation

The children obtained as a result of the crossover are mutated with the probability  $p_{\text{mutation}}$  (see Line 16 in Algorithm 2). If the child is selected for mutation, one component of one line of its code is picked up randomly and mutated in the following manner:

- The line type is randomly replaced by one of the six possible choices.
- The parameter *bodyLen* is randomly replaced with the number between 2 and the maximum allowed number of statements within a control statement body.
- In the expression tree, a randomly selected node (not deeper than mutation depth—see Table 2) is mutated as described below.
- The parameter macro index is replaced by a randomly chosen index.
- In the list of variables, a randomly selected variable is replaced by another randomly picked up variable.

The mutation of a node in an expression tree consists of two operations. First, the terminal code is replaced by a randomly selected element from the list of variables and constants.

Second, the non-terminal code is replaced by a randomly selected element from the list of functions and operators. If the newly selected function or operator needs more arguments, additional random subtrees are generated to support them. Finally, we toggle the type of the node (i.e., terminal or non-terminal) with the toggle terminal probability (see Table 2).

#### 4.4. Replacement and Stopping

After the genetic operations have been completed, we replace the whole population except for the elite (see Table 2) with the obtained children. If the fitness of the best individual in the population drops to zero or the fitness does not change for 600 generations (see Table 2), the evolution stops. Otherwise, the evolution continues until the maximum number of generations has been reached.

### 5. Experiment Setup

Given that we aim to evolve our programs from the ground up, without using any a priori knowledge about the solution structure, we conjectured that the problem set for our experiment should consist mainly of basic, with some intermediate, programming problems. We constructed the problem set using assignments from the introductory programming course at our university, selecting 35 different problems to test our approach. The selection of problems themselves including the sets of operators and functions/macros used must be diverse enough to push our approach to its limits—both lower (no success) and upper (100% success rate). At the same time, most problems should fall within the intermediate range, with success rates in between.

Table 3 summarizes all the problems with the operators, functions, macros, and constants used to evolve the programs for their solutions. The number of arguments and return values for each problem is given in parentheses after the problem name. The functions `mod`, `idiv`, `mul`, and `bigMul` are safe modulo, integer division, and multiplication operators, respectively, with their definitions listed in Appendix A. The rest of the parameters were common to all the runs and are listed in Table 1.

**Table 3.** The problems used in our experiments.

Problem Name (No. of Input/Output Values)	Operators	Notes	Constants
absolute (1/1)		Absolute value of a number	0,1
absoluteDifference (2/1)	−, <	Absolute difference of two numbers	0,1
absoluteDifferencePlus (2/1)	−, <	No minus operator	0,1
absoluteDifferencePlusMacro (2/1)	+, <	Utilize macro <code>if b &gt; a: a, b = b, a</code>	0,1
absoluteDifferencePlusSorted (2/1)	+, <	First argument not less than second	0,1
collatz (1/1)	+, −, <, =, mul, mod, idiv	Length of Collatz sequence	0,1,2,3
collatzMacro (1/1)	+, <, =	Utilize macro <code>a = a // 2 if a % 2 == 0 else a * 3 + 1</code>	0,1
collatzStep (1/1)	+, <, =, mul, mod, idiv	The next number in Collatz sequence	0,1,2,3
countDigits (1/1)	+, −, <, =, idiv	Number of digits in a natural number	0,1,10
exactDivision (2/1)	+, −, <, =	Integer division without remainder	0,1
exactDivisionPlus (2/1)	+, <, =	No minus operator	0,1

Table 3. Cont.

Problem Name (No. of Input/Output Values)	Operators	Notes	Constants
exactDivisionTimes (2/1)		Utilize multiplication operator	
factorial (1/1)	+, <, =, mul	$n!$	0,1
fibonacci (1/1)	-, <, bigMul	$n$ -th number of Fibonacci sequence	0,1
fibonacciMacro (1/1)	+, <, =	Utilize macro a, b = b, a	0,1
gcd (2/1)	+, <, =	Greatest common divisor	0,1
gcdMacro (2/1)	-, <, =	Utilize macro a, b = b, a	0,1
gcdModulo (2/1)	<, mod	Utilize modulo operator	0
integerDivision (2/1)	-, <, =, ^, mod	Floor division, dismiss remainder	0,1
integerDivisionRem (2/2)	+, -, <	Floor division, quotient and remainder	0,1
lcm (2/1)	+, -, <	Least common multiple	0,1
lcmMacro1 (2/1)	+, -, <, =, idiv, mul	Utilize macro if a > b: a = a - b	0,1
lcmMacro2 (2/1)	+, -, <, =, idiv, mul	Utilize macro a = a - b	0,1
lcmMacro4 (2/1)	+, -, <, =, idiv, mul	Utilize macro a, b = b, a	0,1
max2 (2/1)	+, -, <, =, idiv, mul	Larger of two numbers	0,1
max3 (3/1)	<	Largest of three numbers	0,1
multiplication (2/1)	<	Product of two integers	0,1
multiplicationNonneg (2/1)	+, -, <, =	Nonnegative integers	0,1
prime (1/1)	+, -, <, =	Test primality of a number	0,1
primeMacro (1/1)	+, <, =, mod	Utilize macro if mod(a, b) == 0: c = 0	0,1,2
properDivisors (1/1)	+, <, =, mod	Count proper divisors	0,1,2
properDivisorsMacro (1/1)	+, -, <, =, mod	Utilize macro if mod(a, b) == 0: c = c + 1	0,1
remainder (2/1)	+, -, <, =, mod	Remainder of floor division	0,1
sort (2/2)	-, <, =	Sort two numbers	0,1
triangularNumber (1/1)	<	Sum of natural numbers from 1 to n	0,1
	+, <, =		0,1

For the most part, Table 3 is self-explanatory. There are, however, some points that need further explanation. The problems with the same names but different suffixes evolved under the same conditions, the only difference being the set of used operators, functions, and macros. For instance, *absoluteDifference* and *absoluteDifferencePlus* differ only in that the first uses the subtraction and the other the addition operator. Or, *fibonacci* and *fibonacciMacro* differ in that the second uses a macro that swaps the values of two variables (for the reader unfamiliar with Python, it might be useful to know that the code

a, b = b, a swaps the values of the variables a and b.). Whenever we use a macro as the building block for the solution, the used macro is listed in Python format in the Notes column. Notice that some macros could also be implemented as functions without affecting the results.

## 6. Results and Discussion

We performed 1000 evolution runs for each program from the previous section using 20 2.66 GHz Core i5 (four cores per CPU) machines, which took 3 weeks of computing time. Table 4 lists the percentages of successful evolution runs for each program, with the average number of needed generations (averaged over successful runs only) and estimated program complexity. We calculated the Halstead metrics for the hand-written programs, and we list in the table programming effort (E), program difficulty (D), and intelligence content (I). The calculated complexity measures generally agree with the evolution success rate. One notable exception is the gcdModulo (greatest common divisor) function using the modulo operator. The success rate is surprisingly high with extremely high values of the E and D measures and a relatively high I measure. When we looked deeper into the matter, we discovered an error by one of our researchers. Namely, his version was a brute force gcd algorithm trying divisions with all integers between 1 and the smaller of the two parameters, which resulted in unreasonably high complexity measures. The evolution came up with a much smarter version of the algorithm, which, after removing the statements with no effect and replacing the for and break statements with the while loop, looks like this:

```
def gcdModulo(x, y):
    while y > 0:
        tmp = x
        x = y
        y = tmp % y
    return x
```

Halstead complexity measures for the above function are  $E = 640$ ,  $D = 10.40$ , and  $I = 5.92$ , which better agrees with the evolution success rate for that function.

Table 4 also shows that using a macro for coding a part of the solution invariably increases the success rate. That was, of course, expected, because including an appropriate building block necessarily decreases the algorithm complexity. That way, we can successfully evolve more complex algorithms that would otherwise defy evolution. For example, we could not evolve the collatz function because of the high complexity. We were, however, able to evolve collatzStep function with high probability and then use this function to evolve collatzMacro. We observed a similar situation with the functions absoluteDifferencePlus, sort, and absoluteDifferencePlusMacro. Indeed, any macro utilized in our experiments is sufficiently straightforward that we could evolve it effectively as a function. The next step in such cases would be to automatize the definition of a helper function and its use in the same evolutionary process. The idea of *automatically defined functions* (ADFs) has already been introduced in [1] as a way of reusing code in genetic programming but has not received very much attention [36,37]. The solutions are limited to tree-formed GP and impose serious constraints on the genetic operators, as different branches are not allowed to directly exchange genetic material, leaving this question an important open research issue.

We carried out additional experiments using a different set of operators not shown in Table 4. In some cases, fewer operators increased the success rate significantly. For example, removing the less than operator from the Fibonacci function increased the success rate to 27.4%, and removing the minus operator from the countDigits function increased it to 26.7%. Both results are significant with  $p$ -value  $p = 0.01$ . Removing the equality, minus and conjunction operators, and the constant value 1 from the gcdModulo function did not change the success rate but dropped the average number of generations to 339, which was also a statistically significant result with  $p = 0.01$ . In some other cases, we also observed some improvement although not statistically significant. Those improvements could be

attributed to the smaller search space we created with fewer building blocks. Generally, the search space dimension increases exponentially with the number of building blocks [7]. It is difficult to say how strong the influence of certain superfluous operators is. Still, at least the operators that can be replaced by the ones already included in the set should be removed. For example, one does not need less than and greater than operators in the same set. Usually, even equality or inequality operators are extra in conjunction with the less-than operator. By the same token, we observed that often more learning samples give better results. One possible explanation is that more samples increase the resolution of the search space, making it easier to descend towards the minimum.

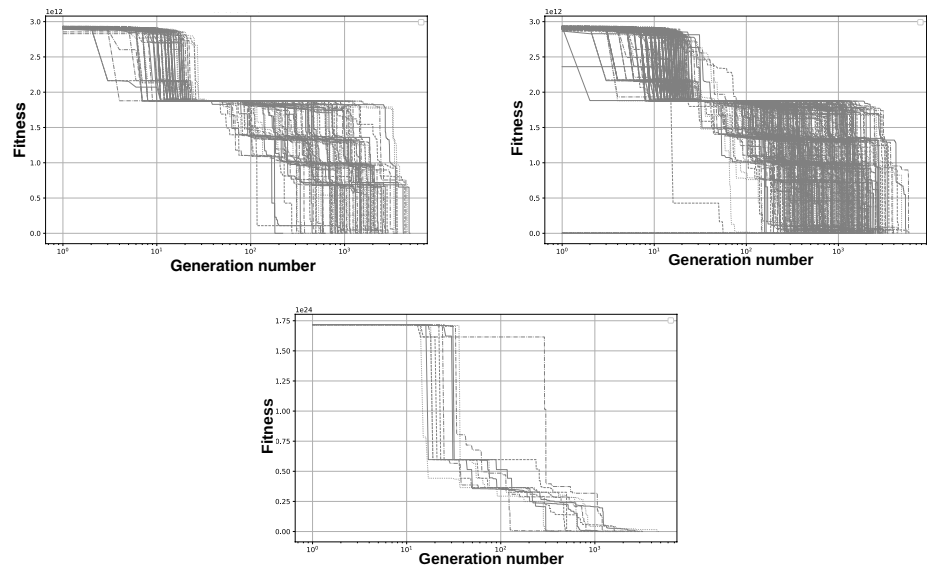
**Table 4.** Proportions of solved problems (1000 runs) with an average number of generations and Halsteas metrics. To calculate the average number of fitness evaluations, multiply the number of generations by the population size (1000). Since each fitness evaluation involves running a program on all the input/output pairs in the training data array, the actual number of program executions is 20 times larger, corresponding to the length of the training data array.

Program Name	Program Complexity			Success Rate (%)	Avg. Number of Generations
	E	D	I		
absolute	269	8.75	3.52	100.0	6
max2	301	8.17	4.51	100.0	11
absoluteDifference	403	9.33	4.62	100.0	23
sort	484	10.50	4.39	100.0	32
max3	596	10.00	5.96	99.7	65
gcdMacro				90.0	404
collatzStep	694	10.00	6.94	87.7	259
collatzMacro				71.3	515
gcdModulo	2337	19.50	6.15	61.6	619
absoluteDifferencePlusSorted	768	10.50	6.96	36.0	562
primeMacro				26.7	571
lcmMacro1				22.0	796
remainder	637	13.50	3.50	21.0	650
absoluteDifferencePlusMacro				20.6	619
countDigits	715	11.0	5.91	19.7	853
fibonacciMacro				17.1	1186
prime	1054	12.83	6.40	16.7	801
exactDivisionTimes	781	10.83	6.65	16.0	799
triangularNumber	684	10.80	5.86	9.8	747
integerDivision	873	11.67	6.42	8.4	692
exactDivision	693	10.00	6.93	6.4	806
fibonacci	868	9.71	9.20	6.3	1584
multiplicationNonneg	873	11.67	6.42	4.6	622
properDivisorsMacro				2.4	1441
integerDivisionRem	811	10.83	6.91	2.2	1098
lcmMacro2				2.2	1043
exactDivisionPlus	856	10.29	8.09	2.1	643
factorial	880	13.00	5.21	1.3	1947
gcd	2346	26.67	3.27	1.6	895
lcm	2166	20.00	5.42	0.9	714
absoluteDifferencePlus	1848	16.67	6.65	0.0	N/A
collatz	3164	24.29	5.37	0.0	N/A
multiplication	2667	20.58	6.29	0.0	N/A
properDivisors	1636	16.50	6.01	0.0	N/A

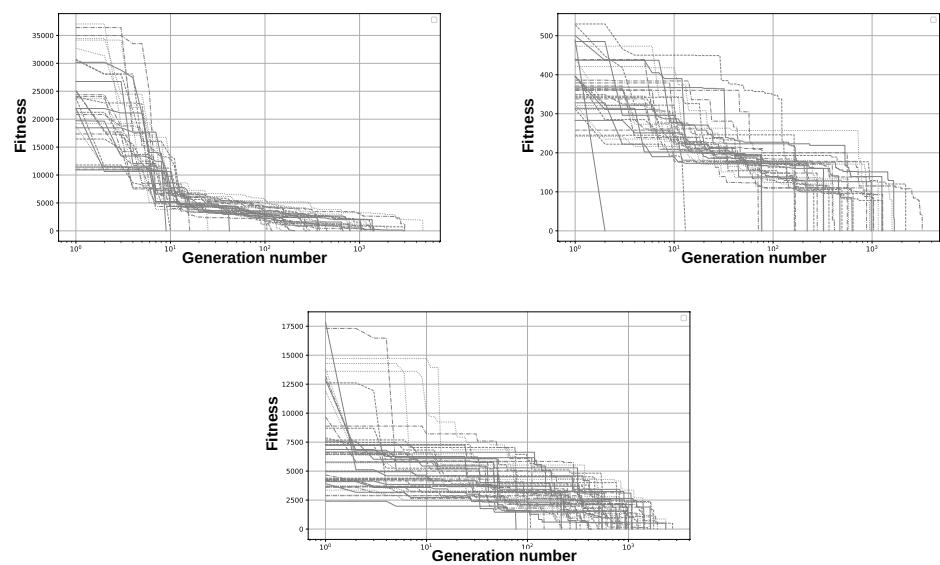
Another noteworthy observation is that some of the evolutions appear to have huge jumps in fitness value, while the majority exhibit a more or less steady drop. Figure 7 shows the convergence of Fibonacci and factorial functions in which significant drops can be observed. Interestingly, these drops happen in different runs at approximately the same levels as the graphs show (notice the suggested horizontal lines formed by the alignment



of graphs). On the other hand, functions in Figure 8 show more steady convergence, although here, one also observes lots of sudden drops that stem from the highly nonlinear nature of the search process caused by the destructive nature of genetic operators in linear GP [2,4]. The more severe fitness drops in Figure 7 are caused by a coarse fitness landscape. Namely, Fibonacci and factorial values are positioned far apart, while values obtained from, for instance, multiplication cover the space more evenly.



**Figure 7.** Convergence of functions fibonacci, FibonacciMacro, and factorial.



**Figure 8.** Convergence of functions multiplication, exactDivisionPlus, and gcd.

We already looked at a specific evolved program (for finding the greatest common divisor) at the beginning of this section where the evolution came up with quite a cunning solution. Of course, we discovered more similarly interesting solutions. The following one is the program that counts the number of proper divisors.

```

def properDivisorsMacro(n):
    count = 0
    div = 1
    x1 = 0
    x2 = 1
    x2 += 1
    while x2 <= n:
        if n % div == 0: count += 1
        if x1 % 2 == 0: div += 1
        x2 += div
        x1 += 1
    return count

```

We removed from the original program all the statements with no side effects, renamed the variables, and made some other minor changes so the program is more human-readable. We retained, however, the basic idea behind the solution, which is quite fascinating. Namely, if one wants to count all proper divisors of  $n$ , one does not need to try divisions by numbers greater than  $\sqrt{n}$ . And that is exactly what the above program is doing—instead of trying divisions with numbers greater than  $\sqrt{n}$ , it counts each division with numbers in the (left-closed and right-open) interval  $[2, \sqrt{n})$  twice. Of course, the program would not have to try each division twice but only count them. It is, nevertheless, an interesting solution.

It is easy to see in the above program that each division with numbers greater than one is carried out twice. But does the process stop at  $\sqrt{n}$ ? Notice that variable  $x2$ , responsible for halting the program, starts with 2 ( $= 1 + 1$ ). Then, we have

$$x2 = 2 \sum_{k=1}^{n-1} + n = n^2.$$

We made some more similar observations. For example, in a program that detects primes, the division was first tried by two, then only with odd numbers smaller than the number tested. Certainly, all these observations took some serious looking because evolved programs are generally quite obscure and often follow confusing logic with many redundant operations, but they are by no means wrong. One possible direct application of these programs would be software obfuscation. Let us conclude this section with an example of an unedited function returning the  $x0$ th number of a Fibonacci sequence:

```

def fibonacci(x0):
    x1=x2=x3=x4=0
    x4=x2==x1
    x2=x4+x1
    for i in range(100):
        if x0+x3<x0<x0: break
        x4=x4
        x3=x3==x4==x0+x2
    x3=x2==x3
    for i in range(100):
        if x3==x0: break
        x4=x4
        x1=x2
        x2=x4+x2
        x3=1+x3
        x4=x1
    pass
    return x4

```

## 7. Conclusions

In this paper, we studied the possibility of evolving imperative computer programs that are not purely expression-oriented but allow for the linear sequence of statements. Specifically, we evolved Python programs. The basis for our work is a special gene-encoding approach combining linear sequences of statements and trees encoding expressions. Because the program population is spawned randomly, it is important to prevent infinite loops. Therefore, we encoded the while statement as the for loop with a break. The iterator of the for loop serves as a safety net setting the upper limit on the number of iterations, while the break statement contains the actual loop condition. We tried in our experiments to evolve different simple programs for which we also calculated Halstead metrics. As expected, we had less success with programs with higher complexity measures, or at least the evolution lasted more generations. We observed that increasing the number of used operators or decreasing the number of training samples could hinder the evolution, often with statistical significance. Whenever we added a useful prefabricated building block (in the form of a macro) to genetic material, the evolution results were better. Specifically, the success rates for the next number of the Collatz sequence calculation and the sequence length counting using a macro were quite high. In contrast, the evolution of the sequence length counting alone was unsuccessful. That leads to the possibility of augmenting the fitness function to support automatically defined functions in the same evolution for more complex tasks, which we feel is an important open research question. Last, we observed some smart solutions evolved by our approach that went beyond a simple brute force approach. It is important to point out that those solutions appeared without being explicitly enforced by the fitness function. We believe that with additional fitness criteria that would favor more efficient solutions, we could obtain more optimal programs. There is also room for improvement in several other directions. For example, incorporating array processing would be an enormous step towards more useful programs.

In conclusion, we believe our paper is an important step that will hopefully motivate further research in this direction. Although the advent of large language models may make the GP approach seem outdated, it is important to emphasize that the pure evolutionary approach operates at a fundamentally different level of computation. While it is lower in abstraction than higher-level computational methods, it is by no means inferior in significance.

**Author Contributions:** Conceptualization, I.F. and Ž.R.; methodology, J.P.; software, Ž.R. and M.K.; validation, J.P., Á.B. and T.T.; formal analysis, Á.B.; investigation, M.K. and J.P.; resources, S.T.; data curation, I.F., M.K. and Ž.R.; writing—original draft preparation, I.F.; writing—review and editing, Ž.R.; visualization, I.F.; supervision, T.T. and Á.B.; project administration, T.T. and S.T.; funding acquisition, S.T. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was funded by the Slovenian Research and Innovation Agency (Javna agencija za znanstvenoraziskovalno in inovacijsko dejavnost Republike Slovenije) through the program P2-0246 (ICT4QoL—Information and Communications Technologies for Quality of Life).

**Data Availability Statement:** Data are contained within the article.

**Conflicts of Interest:** The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

## Appendix A. Functions Used in Evolution

This appendix lists all the functions that we used as building blocks for the programs to be evolved. The functions implement basic mathematical operators with built-in safety mechanisms that guard against undesired scenarios like division by zero or multiplication overflow.

```

def mod(x, y):          #Safe modulo
    if y == 0: return 1 #Prevent division by zero
    return x % y

def idiv(x, y):         #Safe floor division
    if y == 0: return 0 #Prevent division by zero
    return x // y

def mul(x, y):         #Limited multiplication
    if x * y > 10000: return 10000
    if x * y < -10000: return -10000
    else: return x * y

def bigMul(x, y):      #Limited multiplication
    if x * y > 1000000000000000: return 1000000000000000
    if x * y < -1000000000000000: return -1000000000000000
    else: return x * y

```

## References

1. Koza, J.R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*; MIT Press: Cambridge, MA, USA, 1992.
2. Yampolskiy, R.V. Why We Do Not Evolve Software? Analysis of Evolutionary Algorithms. *Evol. Bioinform.* **2018**, *14*, 1176934318815906. [\[CrossRef\]](#)
3. Romera-Paredes, B.; Barekatin, M.; Novikov, A.; Balog, M.; Kumar, M.P.; Dupont, E.; Ruiz, F.J.R.; Ellenberg, J.S.; Wang, P.; Fawzi, O.; et al. Mathematical discoveries from program search with large language models. *Nature* **2023**, *625*, 468–475. [\[CrossRef\]](#) [\[PubMed\]](#)
4. Woodward, J.; Bai, R. Why evolution is not a good paradigm for program induction: A critique of genetic programming. In Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation, Shanghai, China, 12–14 June 2009; pp. 593–600. [\[CrossRef\]](#)
5. Teller, A. Turing completeness in the language of genetic programming with indexed memory. In Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence, Orlando, FL, USA, 27–29 June 1994; Volume 1, pp. 136–141. [\[CrossRef\]](#)
6. Banzhaf, W.; Francone, F.D.; Keller, R.E.; Nordin, P. *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1998.
7. Brameier, M.F.; Banzhaf, W. *Linear Genetic Programming*; Springer: New York, NY, USA, 2007. [\[CrossRef\]](#)
8. Fajfar, I.; Puhan, J.; Bürmen, Á. Evolving a Nelder-Mead Algorithm for Optimization with Genetic Programming. *Evol. Comput.* **2016**, *25*, 351–373. [\[CrossRef\]](#) [\[PubMed\]](#)
9. Cramer, N.L. A representation for the adaptive generation of simple sequential programs. In Proceedings of the First International Conference on Genetic Algorithms and Their Applications, Pittsburgh, PA, USA, 24–26 July 1985; Psychology Press: London, UK, 1985; Volume 183, p. 187.
10. Augusto, D.A.; Barbosa, H.J.C. Symbolic regression via genetic programming. In Proceedings of the Sixth Brazilian Symposium on Neural Networks, Rio de Janeiro, Brazil, 25 November 2000; Volume 1, pp. 173–178. [\[CrossRef\]](#)
11. Icke, I.; Bongard, J.C. Improving genetic programming based symbolic regression using deterministic machine learning. In Proceedings of the 2013 IEEE Congress on Evolutionary Computation, Cancun, Mexico, 20–23 June 2013; pp. 1763–1770. [\[CrossRef\]](#)
12. Evans, B.; Al-Sahaf, H.; Xue, B.; Zhang, M. Evolutionary Deep Learning: A Genetic Programming Approach to Image Classification. In Proceedings of the 2018 IEEE Congress on Evolutionary Computation (CEC), Rio de Janeiro, Brazil, 8–13 July 2018; pp. 1–6. [\[CrossRef\]](#)
13. Bi, Y.; Xue, B.; Zhang, M. *Genetic Programming for Image Classification: An Automated Approach to Feature Learning*; Springer: Berlin/Heidelberg, Germany, 2021. [\[CrossRef\]](#)
14. Najaran, M.H.T. A genetic programming-based convolutional deep learning algorithm for identifying COVID-19 cases via X-ray images. *Artif. Intell. Med.* **2023**, *142*, 102571. [\[CrossRef\]](#)
15. Bakurov, I.; Castelli, M.; Scotto di Freca, A.; Vanneschi, L.; Fontanella, F. A novel binary classification approach based on geometric semantic genetic programming. *Swarm Evol. Comput.* **2021**, *69*, 101028. [\[CrossRef\]](#)
16. Espejo, P.G.; Ventura, S.; Herrera, F. A Survey on the Application of Genetic Programming to Classification. *IEEE Trans. Syst. Man Cybern. Part C Appl. Rev.* **2010**, *40*, 121–144. [\[CrossRef\]](#)
17. Dara, O.A.; Lopez-Guede, J.M.; Raheem, H.I.; Rahebi, J.; Zulueta, E.; Fernandez-Gamiz, U. Alzheimer’s Disease Diagnosis Using Machine Learning: A Survey. *Appl. Sci.* **2023**, *13*, 8298. [\[CrossRef\]](#)

18. Rovito, L.; Bonin, L.; Manzoni, L.; De Lorenzo, A. An Evolutionary Computation Approach for Twitter Bot Detection. *Appl. Sci.* **2022**, *12*, 5915. [[CrossRef](#)]
19. Muni, D.; Pal, N.; Das, J. Genetic programming for simultaneous feature selection and classifier design. *IEEE Trans. Syst. Man Cybern. Part B Cybern.* **2006**, *36*, 106–117. [[CrossRef](#)]
20. Oğuz, K.; Bor, A. Prediction of Local Scour around Bridge Piers Using Hierarchical Clustering and Adaptive Genetic Programming. *Appl. Artif. Intell.* **2022**, *36*, 2001734. [[CrossRef](#)]
21. Alturky, S.; Toma, G. A Metaheuristic Optimization Algorithm for Solving Higher-Order Boundary Value Problems. *Int. J. Appl. Metaheuristic Comput.* **2022**, *13*, 1–17. [[CrossRef](#)]
22. Sobania, D.; Schmitt, J.; Köstler, H.; Rothlauf, F. Genetic programming for iterative numerical methods. *Genet. Program. Evolvable Mach.* **2022**, *23*, 253–278. [[CrossRef](#)]
23. Brabc, M.; Żegklitz, J.; Grepl, R.; Babuska, R. Control of Magnetic Manipulator Using Reinforcement Learning Based on Incrementally Adapted Local Linear Models. *Complexity* **2021**, *2021*, 6617309. [[CrossRef](#)]
24. García, C.A.; Velasco, M.; Angulo, C.; Marti, P.; Camacho, A. Revisiting Classical Controller Design and Tuning with Genetic Programming. *Sensors* **2023**, *23*, 9731. [[CrossRef](#)] [[PubMed](#)]
25. Beşkirli, A.; Dağ, İ. An efficient tree seed inspired algorithm for parameter estimation of Photovoltaic models. *Energy Rep.* **2022**, *8*, 291–298. [[CrossRef](#)]
26. Beşkirli, A.; Dağ, İ. A new binary variant with transfer functions of Harris Hawks Optimization for binary wind turbine micrositeing. *Energy Rep.* **2020**, *6*, 668–673. [[CrossRef](#)]
27. Beşkirli, A.; Dağ, İ. Parameter extraction for photovoltaic models with tree seed algorithm. *Energy Rep.* **2023**, *9*, 174–185. [[CrossRef](#)]
28. Beskirli, A.; Özdemir, D.; Temurtas, H. A comparison of modified tree–seed algorithm for high-dimensional numerical functions. *Neural Comput. Appl.* **2020**, *32*, 6877–6911. [[CrossRef](#)]
29. Zhan, R.; Zhang, J.; Cui, Z.; Peng, J.; Li, D. An Automatic Heuristic Design Approach for Seru Scheduling Problem with Resource Conflicts. *Discret. Dyn. Nat. Soc.* **2021**, *2021*, 8166343. [[CrossRef](#)]
30. Xu, M.; Mei, Y.; Zhang, F.; Zhang, M. Genetic Programming and Reinforcement Learning on Learning Heuristics for Dynamic Scheduling: A Preliminary Comparison. *IEEE Comput. Intell. Mag.* **2024**, *19*, 18–33. [[CrossRef](#)]
31. Mahmood, M.T. Defocus Blur Segmentation Using Genetic Programming and Adaptive Threshold. *Comput. Mater. Contin.* **2022**, *70*, 4867–4882. [[CrossRef](#)]
32. Correia, J.; Rodriguez-Fernandez, N.; Vieira, L.; Romero, J.; Machado, P. Towards Automatic Image Enhancement with Genetic Programming and Machine Learning. *Appl. Sci.* **2022**, *12*, 2212. [[CrossRef](#)]
33. Wineberg, M.; Oppacher, F. The Benefits of Computing with Introns. In Proceedings of the First Annual Conference, Stanford, CA, USA, 28–31 July 1996; Koza, J.R., Goldberg, D.E., Fogel, D.B., Riolo, R.L., Eds.; Stanford University: Stanford, CA, USA, 1996; pp. 410–415. [[CrossRef](#)]
34. Abdelkhalik, O. Hidden Genes Genetic Optimization for Variable-Size Design Space Problems. *J. Optim. Theory Appl.* **2013**, *156*, 450–468. [[CrossRef](#)]
35. Baker, J.E. Adaptive Selection Methods for Genetic Algorithms. In Proceedings of the 1st International Conference on Genetic Algorithms, Pittsburgh, PA, USA, 24–26 July 1985; pp. 101–111.
36. Ferreira, C. Automatically Defined Functions in Problem Solving. In *Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 233–273. [[CrossRef](#)]
37. Ferreira, C. Automatically Defined Functions in Gene Expression Programming. In *Genetic Systems Programming: Theory and Experiences*; Nedjah, N., Abraham, A., Macedo Mourelle, L.D., Eds.; Springer: Berlin/Heidelberg, Germany, 2006; Volume 13, pp. 21–56. [[CrossRef](#)]

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.