.

TFM en Master in Robotics, Graphics and Computer Vision

Integration through genetic programming on heterogeneous systems

Autor

Enrique Bauzá Mingueza

Directores

Adolfo Muñoz Orbañanos, Darío Suárez Gracia

Escuela de Ingeniería y Arquitectura
2022

Abstract

Nowadays, numerous applications in various scientific fields require the integration of mathematical functions that, due to some of their characteristics, do not have an analytical expression for their antiderivative. These definite integrals are usually solved by numerical integration methods, which provide an approximation of the numerical value of the integral in the integration range. With this type of solutions, a higher precision of the approximation entails a longer computation time, being necessary a trade-off between both aspects.

In this work we present a genetic programming algorithm which provides mathematical expressions that approximate the antiderivative of analytically non-integrable functions. Heterogeneous devices, GPU and multicore CPU, have also been used in the development of the system to accelerate the parts suitable for it. The advantage of obtaining these approximate antiderivatives is the reduction of the computation time necessary to calculate the definite integral of the functions of interest, reducing it to simply evaluating the expression at the beginning and the end of the integration range.

The experiments performed and the results obtained demonstrate the system's ability to approximate several types of functions, including two-dimensional ones, and in particular its ability to approximate the antiderivatives of analytically non-integrable functions. In addition, the acceleration provided by the use of heterogeneous devices is also demonstrated.

Table of Contents

1 Introduction **4**

1.1 Objective 6

1.2 Structure 6

2 Fundamentals **8**

2.1 Evolutionary algorithms 8

2.2 Symbolic computation 10

2.3 Computing systems 12

2.3.1 Heterogeneous hardware 12

2.3.2 Parallel and heterogeneous programming 13

3 Methodology **16**

3.1 Programming languages and libraries 16

3.1.1 Python implementation 16

3.1.2 DPC++ implementation 16

3.2 Machine and Compiler Descriptions 17

3.3 Experiments Metrics 18

3.3.1 Time Measurement Methodology 18

3.4 Code repository 18

4 System Description **19**

4.1 System Overview 19

4.2 Individual Representation: Tree structure 19

4.3 Initialization 20

4.4 Evaluation 21

4.5 Selection 22

4.6 Reproduction 23

4.6.1 Crossover 24

4.6.2 Mutations 25

4.7 Replacement 28

4.8 Finalization . . . 28
4.9 Genetic parameters . . . 28

5 Random number generation in GPU 30
5.1 Implementation on GPU . . . 30
5.2 Implementation testing . . . 32
5.2.1 Generation time comparison . . . 32
5.2.2 Randomness tests . . . 32

6 Evaluation 34
6.1 Validation experiments . . . 34
6.1.1 Function approximation . . . 34
6.1.2 Genetic integration . . . 36
6.1.3 Comparison between Sequential and Parallel Versions . . . 38
6.1.4 Integration of no-integrable functions . . . 40
6.2 Single Scattering . . . 40
6.2.1 Loss function . . . 44
6.2.2 Function and variable sets . . . 44
6.2.3 Parallelization . . . 44
6.2.4 Experiments and results . . . 45
6.3 Image compression . . . 46
6.3.1 Loss function . . . 46
6.3.2 Function and variable sets . . . 50
6.3.3 Parallelization . . . 51
6.3.4 Experiments and results . . . 51

7 Conclusions 58

Bibliography 61

List of figures 63

List of tables 65

Chapter 1

Introduction

Nowadays countless applications in all scientific fields require the integration of varied functions. In many cases, it is not possible to analytically integrate a function by finding its antiderivative: the integrand might be more complex than an elementary function or a combination of them, or, simply, such symbolic antiderivative might not exist for such integrand. In this cases, the only alternative to analytic integration is numerical integration, that yield an *approximation* of the integral. There are several algorithms for approximating a numerical value for a definite integral [1], such as the Trapezoidal rule, Rectangle rule or Monte Carlo integration. Numerical integration gives us an approximate solution to the definite integral, and the error of this approximation is reduced as the number of evaluations of the integrand (samples) increases. Inevitably, this lead to a trade-off between integration error and computation time which is one of the main issues in numerical integration.

One specific field where it is necessary to solve integrals and where numerical integration is commonly used is rendering. Today, most rendering systems use techniques based on the integration of physical equations of light propagation and, due to the nature of this equations, this is solved by numerical integration. In particular, one widespread method in this field is Monte Carlo integration which, basically, consists of randomly sampling the integrand along the integration range. As it is another case of numerical integration, the quality of the rendered images (inverse to the integration error) increases with the number of samples. In some cases, like computer generated films, high quality images are required, which means hours of rendering for a unique frame or image. This issue has been addressed through the use of neural networks [2], however, like in many cases where they are used, the network parameters and architecture are highly dependant on the specific scene. In the rendering case, it would be tremendously interesting, in terms of computational time, to have an algebraic expression for the antiderivative of those light transport equations (or, if not possible, an approximation of it). In this way we could just evaluate that expression twice (beginning and end of the integration range), significantly reducing the time required to compute the definite integrals and, ultimately, to render images with increased accuracy.

One way to obtain those approximated algebraic expressions for the antiderivative can be regression and, although there are multiple and diverse methods for mathematical regression, in this work we focused on regression through evolutionary algorithms [3, 4]. Evolutionary algorithms (EA) solve optimization problems, inspired by the natural evolution process where the individual who better adapts to its environment survives and reproduces. In the case of EA, this *adaptation* is modelled as a mathematical fitness to an objective solution. EA are iterative algorithms where a set of potential solutions, called individuals, conform the population, and, even though there are different subfields of EA, all of them have common steps in each iteration (or generation):

– First, each individual is **evaluated** according to a given **fitness** criteria

– Then the best individuals are **selected** as parents for the next generation

– The third step is **reproduction**, where the genetic information from the selected parents is passed to the next generation by **mutation** (changes on one parent) or **crossover** (combination of two different parents)

– Finally, the new individuals (or some of them) **replace** part of the old ones to generate the new population, according to their fitness.

Due to the computationally expensive workload of optimization problems and the fast development of heterogeneous hardware in the last decade, it is common to apply heterogeneous computing, where more than one kind of processor are used, to these optimization tasks. Evolutionary Algorithms are not an exception, we are working with many independent solutions at the same time, so they present a high degree of data parallelism [5, 6]. Because of this workload behaviour, they are suitable for implementation on heterogeneous devices, where we can, for example, accelerate the population evaluation or divide the original population in several smaller subpopulations and work with them in parallel, making use of several computing units.

In this work we present a genetic programming regression system using heterogeneous devices 4. For this system it has been necessary to create a computer algebra library since we are representing the individuals of the evolutionary algorithm as algebraic expressions. This computer algebra library includes the representation of the individuals and some operations such as symbolic differentiation. As we are working with algebraic expressions, the genetic operators (mutations and crossover) are algebraic modifications or combinations of those expressions. The evaluation of the population is performed by evaluating each individual at a set of points of the given range and comparing those values with the evaluation of the objective function at the same points. All the genetic operators and the initial population generation are random processes so we can say that this algorithm needs a large amount of random numbers and, in this system, this random number generation has been implemented on a GPU (5), taking advantage of the execution throughput of that device to accelerate that process. Finally this system has been applied to two specific problems of the computer graphics field: rendering (6.2) and image compression (6.3). For the rendering application the purpose is to obtain an algebraic expression for the antiderivative of analytically non-integrable light transport equations (in particular a lower-dimensional case of single scattering). To obtain this integrals our objective function is the single scattering equation and what we compare to that is the derivative of the population individuals, so the population is directly conformed by a set of integrals which are the possible solution. And, for the image compression application, we look for a bidimensional function which relates the pixel value to the image coordinates in pixels, so, in this case, the sample points are the pixel coordinates and the pixel values are our "objective function".

1.1 Objective

The main objective of this work is to develop a system capable of obtaining a mathematical expression that serves as an integral for analytically non-integrable functions. The main component of this system is a genetic programming algorithm, which must be designed and implemented. In addition, a symbolic algebra library and a representation for the individuals of the genetic programming algorithm must be designed and developed. Heterogeneous systems will also be used to speed up two parts of the system: the evaluation of the solutions at each iteration of the algorithm and the generation of random numbers. Finally, the developed system will be evaluated through several validation experiments and its application to two specific problems: a particular problem of integration in rendering and image compression.

1.2 Structure

From here on, this document is structured in different chapters as follows:

- **Chapter 2: Fundamentals**, in this chapter we expose the basic knowledge of the fields in which we have been working in the project which are: Evolutionary Algorithms, Computer Algebra and Heterogeneous Systems.

- **Chapter 3: Methodology**, here we describe the tools used in the development of this project to ease the understanding and reproducibility of the results.

- **Chapter 4: System Description**, in this chapter we give an overview of the developed system, explain the individual representation for the evolutionary algorithm and give a detailed description of each step in the algorithm.

- **Chapter 5: Random number generation in GPU**, here we explain how we have leveraged a GPU to accelerate the random number generation in the system.

- **Chapter 6: Evaluation**: Here we present the applications and results of the systems in this work. There are several validation experiments, integration of a low-dimensional version of the single scattering equation and image compression.

- **Chapter 7: Conclusions**: In this chapter we comment on the conclusions of the work carried out, evaluate the results and propose future lines of work.

Chapter 2

Fundamentals

In the following chapter we will present the main fields in which we have worked during the development of this genetic integration system. First we will go trough Evolutionary Algorithms 2.1, the bunch of optimization algorithms which we have used for this work. Later we introduce Symbolic Computation (or Computer Algebra) 2.2, we have used this for mathematical expression manipulation. And, finally, we present Heterogeneous Computation 2.3, a set of hardware and software that can be leveraged to accelerate certain applications.

2.1 Evolutionary algorithms

Mathematical optimization consists of finding the global maxima or minima of a function, f, by choosing the best available inputs from a set of alternatives, A. Usually A is delimited by a set of constraints which defines the search space for the optimization problem. Given a function: $f : A$, and depending on what problem is being solved (minimization or maximization), the optimization can be stated as follows:

$$\text{Find: } \mathbf{x_0} \in A \text{ such that } f(\mathbf{x_0}) \leq f(\mathbf{x}) \text{ for all } \mathbf{x} \in A \ (\textit{minimization}) \tag{2.1}$$

$$\text{Find: } \mathbf{x_0} \in A \text{ such that } f(\mathbf{x_0}) \geq f(\mathbf{x}) \text{ for all } \mathbf{x} \in A \ (\textit{maximization}) \tag{2.2}$$

The function f may take different names depending on the problem, for minimization it is often called lost or cost function, for maximization is called utility or fitness function and in some physical related fields it is called energy function.

An **evolutionary algorithm (EA)** [3, 4] is an heuristic optimization algorithm inspired by natural processes of selection and evolution. In these natural processes the better are the individuals the more likely they are to survive and propagate their genetic material. Roughly, an evolutionary algorithm is a iterative algorithm where a set of possible solutions, called population, is randomly modified until one of the solutions is good enough. EA is a broad field with several different subsets of algorithms, but all of them share the same base of simulating the evolution of an initial population. In 2.1 we can see an overview of the algorithm and its different stages:

- **Initialization:** In this stage the first generation is created, which can be done randomly or based on some prior knowledge about the problem. This step is critical to the algorithm performance, a properly generated first population will lead to a faster convergence. The number of individuals in this first generation and the complexity of these ones are key design aspects that will determine the suitability of the first generation.

- **Selection:** The first step of every iteration is the selection of the individuals that are going to reproduce and be the parents for the next generation. This selection is based on some cost or fitness criteria, so in this stage it is also necessary to evaluate the population. There are several selection strategies to be implemented such as roulette wheel selection, where the probability of a individual to be selected is proportional to its score, or tournament selection, where the parent is the best individual from a subset of candidates. In this stage it could be interesting to give a little probability to be chosen to some individuals which are not the best so that the search space exploration is encouraged. The selection strategy must achieve a trade-off between search space exploration and convergence speed.

- **Reproduction:** In this stage the genetic information from the selected parents is passed to the next generation. There are two genetic operators to perform the reproduction, mutation and crossover, and its implementation is strongly dependant on the problem to solve and the representation of the individuals. The mutation is the operation where a part of the genetic information of one individual is altered. And the crossover consists of the combination of genetic information from two different parents to create a new individual. For example, in case the individuals are represented by bit strings, the crossover operation would consist of the combination of two substrings from two different parents and a suitable mutation would be to change one or several bits of the string.

- **Replacement:** And in this final stage the new individuals, or some of them, substitutes part of the old ones the generate a new population.

- **Ending condition:** The algorithm needs an ending condition. This condition can be based on a cost or fitness criteria (the score is equal or better than a desired score) or on a computation time criteria, usually there is a maximum number of iterations.

One key aspect of EA related to this work is the high degree of data parallelism that these algorithms present [5, 6]. In the evaluation step the operations to be performed on each solution of the population are completely independent from the others, so there it is a good point to improve the algorithm performance.

2.2 Symbolic computation

Symbolic computation or computer algebra is a computer science area which studies algorithms and software for mathematical expression manipulation. In contrast to numerical computation, which is based on approximate floating point numbers, symbolic computation is focused on exact computation of mathematical expressions containing variables which have no given value. Other advantage of symbolic computation is the ability to easily manipulate a mathematical expression before numerically evaluating it, which leads to several advantages when applied to different problems. A computer algebra system must include a set of routines to perform usual operations such us simplification, factorization, indefinite integration, differentiation by chain rule, ...

In symbolic computation, one efficient and widespread way of representing mathematical expression is in form of tree-like structures, like the one in figure 2.2. The expression trees are composed by two different types of elements: nodes and leaves. Nodes represent mathematical operators that can be unary (i.e. exponential function) or n-ary (i.e. product), and this will determine the number of operands for each node. The other type of elements, leaves, represents constants or variables which acts as the operands of the nodes. In image 2.3 we can see an example of chain rule differentiation where a new expression tree is generated by the differentiation of the existing one. These differentiation is performed recursively from the root to the leaves. Each intermediate node is treated as a root of a smaller sub-tree.

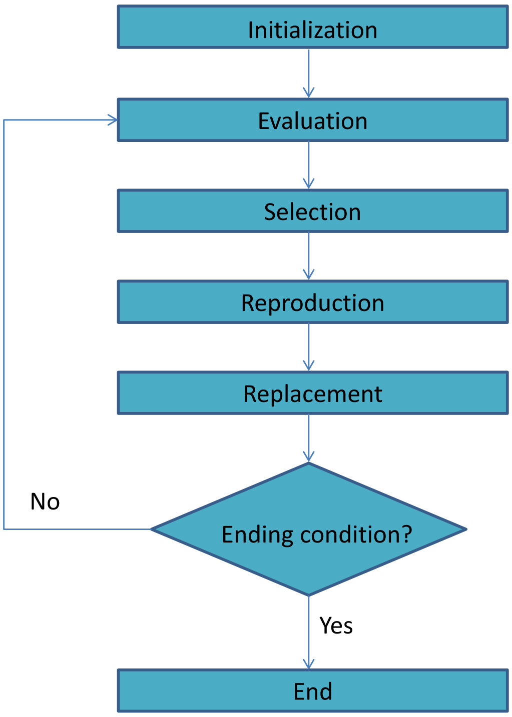


Figure 2.1: Evolutionary algorithms common steps.

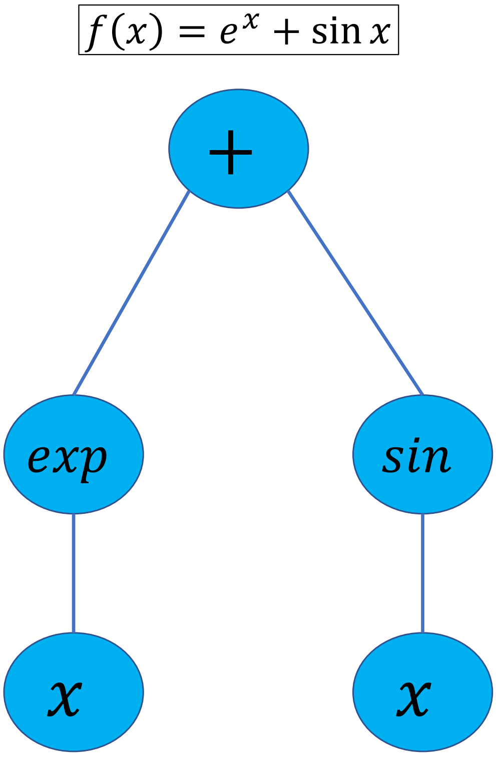


Figure 2.2: Mathematical expression represented in a tree srtucture.

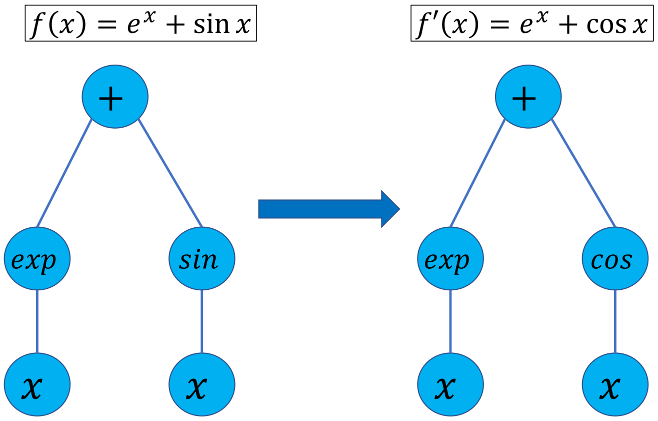


Figure 2.3: Mathematical expression differentiation in form of tree structure. $f(x) = exp(x) + sin(x)$; $f'(x) = exp(x) + cos(x)$

Genetic programming [7] is a subset of Evolutionary Algorithms very related to symbolic computation. The main difference between this subset of Evolutionary Algorithms and others is the representation of the solutions. In Genetic Programming the individuals are not fix-length and they are traditionally represented as tree structures, which made these algorithms really suitable for computer algebra systems. The genetic operators are also adapted to the tree structure. In this case, the crossover is accomplished by combining both tree structures, as we can see in figure 4.2. For the mutation, the genetic information can be modified by adding or deleting nodes, which will increase or decrease the complexity of the solutions respectively, or replacing the existing ones by randomly generated nodes. Other factors which significantly affect the algorithm performance are the initial population characteristics (number and complexity of individuals), the population size and the percentage of individuals which are affected by genetic operators and replacement between generations. Also, when solving an optimization problem, the node and leaf sets, which determine what functions and variables will conform the trees, are highly influential on the algorithm performance, in case those sets are not fit enough for the problem conditions, the exploration of the search space will be less efficient leading to more time to reach the final solution or, even, the inability to reach it.

2.3 Computing systems

2.3.1 Heterogeneous hardware

Until early 2000s, microprocessors based on a single Central Processing Unit (CPU) achieved a rapid performance improvement and cost reduction in computer applications by increasing the amount of transistor that fit in the microprocessor. However, energy consumption and heat dissipation issues have limited the increment of the clock frequency and the level of productive activities that can be performed in each clock period within a single CPU. Since then, the trend has been to switch to models where multiple processing units, or processor cores, are used in each chip to increase the processing power.

One type of these new models is the multicore CPU, which seeks to maintain the execution speed of traditional sequential programs while moving into multiple cores. A CPU is designed to optimize sequential code performance. It makes use of sophisticated control logic to allow instructions from a single thread to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution. Also, this model is provided with large cache memories to reduce the data and instructions access latencies inherent to complex applications.

Other multiprocessor model is the Graphics Processor Unit (GPU), in contrast with CPUs, GPUs are focused on the execution throughput of parallel applications. The design philosophy of GPUs is to optimize for the execution throughput of massive numbers of threads. This design saves chip area and power by allowing pipelined memory channels and arithmetic operations to have long latency. The reduced area and power of the memory access hardware and arithmetic units allows to have more of them on a chip and thus increase the total execution throughput. Small cache memories are provided to help control the bandwidth requirements of these applications so that multiple threads that access the same memory data do not need to all go to the DRAM. This design style is commonly referred to as throughput-oriented design since it maximizes the total execution throughput of a large number of threads in exchange for longer execution time for the individual threads.

And a third commonly used multiprocessor model are Field-Programmable Gate Arrays (FPGAs). An FPGA is a programmable hardware that can be reconfigured after manufacturing. It contains programmable logic gates, programmable interconnects, configurable memory modules and DSPs (dedicated multipliers). We can connect these elements to implement any arbitrary circuit. Therefore, many accelerators can be implemented on FPGA to do different computations. FPGAs are already used in many fields such as signal processing, high-performance computing, machine learning, etc. Hardware description languages, such as Verilog HDL, are used to program FPGAs and, usually, the accelerator design time is very large.

2.3.2 Parallel and heterogeneous programming

Today's applications presents several different workload behaviours, which goes from data intensive (e.g., image processing, simulation and modelling, data mining, etc) to control intensive (e.g., searching, sorting, parsing, etc). Each of these workload classes typically executes most efficiently on a specific type of hardware architecture. No single architecture is best for running all classes of workloads, and most applications presents a mix of the workload characteristics. For instance, control-intensive applications tend to run faster on CPUs, whereas data-intensive applications tend to run fast on vector architectures, where the same operation is applied to multiple data items at the same time. With so much heterogeneity, developing efficient applications for such a wide range of different hardware represents a great challenge for software developers. As hardware becomes more specialized, performance and energy efficiency improves at the cost of programability. Answering this necessity for more understandable, portable and flexible code for heterogeneous programming, in the last years, they have appeared new programming models which hide the complexities of the hardware, such as CUDA, from NVIDIA, OneAPI, form Intel or OpenCL, all of them supporting joint CPU-GPU execution of an application.

In parallel programming a initial problem is decomposed into smaller tasks or processes that can be executed simultaneously using the multiple available compute resources creating several threads. Based on how these threads communicate and synchronise we can distinguish between two main parallel programming models: Message passing model and Shared memory model. In the first one, each processor has its own local memory, so we have multiple memory spaces and need a communication network to connect the inter-processor memory. On the other hand, in the Shared memory model, all processors can access the same memory and same global address space. In this case the more processors we have the higher is the processor-memory bandwidth.

When working with Shared memory model, the one which concerns us in this work, we can establish two types of parallelism depending on the problem we are working on: data parallelism and task parallelism. We have data parallelism in a problem when there are no dependencies between the data, so it is possible to run the same task on different data elements simultaneously. For example, adding a constant to each element of a vector presents data parallelism. And we have task parallelism when it is possible to run several functions on the same data, for example, we can compute at the same time the mean and the minimum of a vector. The area of Evolutionary Algorithms is specially suitable for parallelization as many of the operations that are performed on the individuals, such as evaluation or mutation, presents data parallelism given that each individual is independent from the rest.

The potential application speed-up is given by the Amdahl's Law (2.3), and it depends on the fraction of parallel code and the number of cores running that code.

$$S = \frac{1}{\frac{P}{C} + s} \tag{2.3}$$

Where S is the potential speed-up, C the number of cores, P the parallel code fraction and s the sequential code fraction.

Chapter 3

Methodology

In this chapter we present the programming languages, software, hardware and metrics that we have used for the development of the system and the realization of the experiments.

3.1 Programming languages and libraries

This project, as described in Chapter 4, has been implemented in two programming languages: Python and Data Parallel C++ (DPC++)/SYCL. The first helped the prototyping and build the first proof of concept while the second provided the required support to build a faster heterogeneous solution.

3.1.1 Python implementation

This is a first implementation whose aim is the validation of the genetic algorithm design decisions (genetic operators, population characteristics, parameters for random processes, etc). For the symbolic computation, this implementation relies on the Sympy python library for symbolic mathematics[1].

3.1.2 DPC++ implementation

DPC++ [8] is single source, meaning that device and host code can be included in the same source file. A DPC++ compiler generates code for both the host and device. Any C++ compiler can compile programs that only use the host subset of DPC++. DPC++ programs are written in ISO C++ and use the Khronos* SYCL*[2] parallel programming model to distribute computation across processing elements in a device. DPC++ extends SYCL with features for performance and productivity.

[1] `https://www.sympy.org/en/index.html`

[2] `https://www.khronos.org/sycl/`

Once validated, the system has been implemented in DPC++ for its optimization on heterogeneous devices.

One key aspect is the design and implementation of an small symbolic computation engine that replaces SymPy. Internally, the engine manages the algebraic expressions with pointer-based tree structures and dynamically allocated nodes. We also have implemented the genetic operators (mutation and crossover) and the population management functionalities (initialization, replacement, ...) that are used in both applications, as well as the analytic differentiation of the algebraic expressions. This library also provides support for saving, loading and visualization of the algebraic expression.

The implementation of DPC++ from Intel named oneAPI provides many libraries to accelerate tasks. In particular, this implementation uses the Threading Building Blocks library (oneTBB) to parallelize the code on the CPU and the oneDPL library to generate random numbers on a GPU.

Since the image compression application required support for working with images, we have used the CImg [3] library.

3.2 Machine and Compiler Descriptions

– **Hardware:** 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, 8 virtual cores and 8GB RAM.

– **Compiler:** As we are working on heterogeneous devices with DPC++ we need a suitable compiler. Compiler version and compilation flags for this machine are shown in table 3.1.

Compiler	Version	Flags
Intel(R) oneAPI DPC++/C++ Compiler	2022.0.0.20211123	-O3 -fexceptions -std=c++17

Tabla 3.1: Compiler version and flags

Regarding the compilation flags, *-std=c++17* has been included in order to use the *std::clamp* function in some steps of the individual evaluation. And we use *-fexceptions* for handling possible C++ exceptions.

[3] `https://cimg.eu/`

3.3 Experiments Metrics

The main metric of interest for this master thesis is execution time. Quantitative results of execution time can help to evaluate whether genetic algorithms can efficiently support computer graphics applications. In the aforementioned machines, other metrics such as energy consumption tend to correlate with execution time and could be suitable for follow-up works.

3.3.1 Time Measurement Methodology

To ensure the accuracy of the results, each experiment run is composed by five application runs performed in the same machine and with the same random seed for each experiment repetition.

3.4 Code repository

Code from both mentioned implementations in section 3.1 are available in the following GitHub repository:

`https://github.com/EnriqueBM/TFM_EnriqueBauza`

Chapter 4

System Description

In the present chapter we describe the system we have developed. We explain the individual representation we have chosen for the genetic programming algorithm and we go through each one of the steps of the algorithm in detail.

4.1 System Overview

In this work, we have developed a computer algebra system for mathematical regression based on genetic programming. For a given n-dimensional target function and a sample dataset, the system output is another function that approaches the target one with additional properties; e.g., derivable, easy to compute, ...

To develop this system we first need a specific individual (or chromosome) representation, this is described in 4.2. On top of this representation, we have to implement the set of operations described in the following sections such as mutation and crossover (4.7), selection (4.5) or evaluation (4.4). Also we have to define and implement population initialization (4.3), population replacement (4.7) and finalization condition (4.8), as well as the genetic parameters for the algorithm (4.9).

The core of this system is the evolutionary algorithm exposed in this chapter. The algorithm iterates over an initial population of algebraic expressions until one expression satisfies a convergence criteria or the maximum number of generations is reached. At this point, the resulting functions mimics the target function and can replace it in the destination applications.

4.2 Individual Representation: Tree structure

For the individual representation we have chosen a tree structure, described in 2.2. The set of functions used for the nodes of the trees are the following:

1. *sin*, *exp* and *step* as unary functions.

2. *product* and *addition* as n-ary functions.

From the above, the specific functions used in each application are problem dependant and they are enumerated in the next chapter (6.2 and 6.3) when we explain each problem in detail. Each n-ary node can have up to four children nodes and, as n-ary nodes are sums and products, which have commutative and associative properties, any child can be again of the same type, yielding a n-ary expression with no prior bounds. Finally, as leaves, we have random constants or the correspondent variables of each problem.

This tree structure has been implemented making use of pointers and dynamic memory management in DPC++ and with the SymPy libary in Python.

1. **Python implementation:** In this case we have used an existing computer algebra library called Sympy[1]. This library has its own tree-structure representation for the mathematical expressions, as well as a wide set of mathematical functions from which we can select the ones required for the problem.

2. **DPC++ implementation:** In this case we have designed a pointer-based tree structure representation from scratch in DPC++ making use of *struct* and *union* data types for the dynamic allocated tree nodes. As well as all the computer algebra functions needed to manipulate that structure.

4.3 Initialization

The initialization step generates the initial population. There are two main aspects of the initial population that will affect the algorithm performance: the population size and the initial individuals complexity.

The initial population is generated as shown in algorithms 1 and 2. For a given number of individuals and a maximum depth, we first randomly select the depth for each individual between one and the maximum depth and then generate the individual, randomly as well.

```
Algorithm 1: Generate pop(maximum depth, n_inds)
Input: Maximum depth, number of individuals
Output: Initial population
for $i < n_inds$ do
    depth = random between 1 and maximum_depth;
    individual[i] = Generate tree(depth);
```

[1] https://www.sympy.org/en/index.html

Individual generation: Algorithm 2 describes the generation of the trees for each individual. For a given depth, the function generates recursively, from the root to the leaves, a random tree. To accomplish this, the children nodes are generated as subtrees with a smaller depth. The algorithm first check if the given depth is one, in that case it just generates a leaf which can be a variable, from the variable set, or a random constant. If depth is greater than one, it randomly selects a unary or n-ary function, from the function set, and then generates the arguments (several if it is n-ary or just one for unary functions) by calling recursively the same function with a decreased-by-one depth. This process is illustrated in figure 4.1.

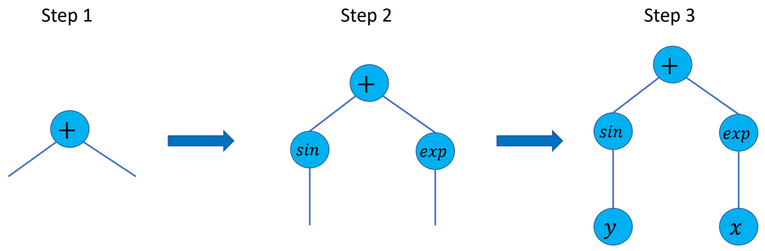


Figure 4.1: Individual initialization. For a given depth of 3 the tree nodes and leafs are randomly and recursively generated from top to bottom.

```
Algorithm 2: Generate tree($d$)
Input: Depth: $d$
Output: Random tree
if $d$=1 then
    Generate random leaf;
else
    Randomly select unary or n-ary function;
    if function is n-ary then
        for $i < n\_args$ do
            argument[i] = Generate tree($d-1$);
    else if function is unary then
        argument = Generate tree($d-1$);
```

4.4 Evaluation

This stage add the score to each individual. This score computation depends on the specific optimization problem solved because each problem needs its own cost or fitness function. But, as we are working with mathematical expressions and trying to approximate an objective function, we will need to evaluate the individuals and the objective function (which can also be represented in a tree structure) at a given set of points in order to compare them. In the simplest case, we can compute the score as e.g. the mean absolute error between the objective and the individual for the given set of points. Obviously, there are other possible and more complex ways to compute the score, for example, adding more terms to the score function so that it takes into account other problem characteristics, as we can see in 6.3 and 6.2 when we explain each solved problem in more detail. Anyway, the mathematical expression evaluation is a common feature, so what is inevitably needed, is a function to evaluate each expression at a point which we can then call iteratively over all the point set. The function to evaluate an individual at a point is recursive, evaluating from root to leaves. For each evaluation, it first checks what type of tree element is being evaluated. For leaves we can have constants, in this case it will return the correspondent real value, or variables, where it will return the correspondent value of the point component. And for nodes, it will first evaluate the arguments and then operate the node functions on the argument values.

4.5 Selection

The selection stage consists of determining which individuals will be the parents for the next generation. The are several suitable selection strategies, and many of them are focused on selecting the best individuals, making the probability to be chosen proportional to the score of each individual. The evident advantage of these type of strategies is a faster convergence as we are always selecting the best individuals of each generation, but, on the other hand, there is a drawback on the exploration of the search space which could lead to get stuck on a local minima.

In this stage we select a total of $n = \mathit{population\_size}/2$ individuals to be the parents in the reproduction stage and this selection is performed with replacement.

Tournament selection algorithm: In this system the implemented selection strategy is a **tournament selection**, in which a significant smaller random subset of individuals "compete" to be selected, given more probability to be chosen to the best individuals within the mentioned subset. With this strategy, although the best individuals have higher probability to be chosen, not-best individuals have more chance to be chosen than in other strategies. As the subset for the tournament is random, a worse scored individual can enter the tournament and, once in the tournament, it has a smaller, but significant, probability to be chosen. As explained before, in this way we are encouraging the search space exploration.

The tournament selection is implemented as shown in algorithm 3. First, three random individuals are selected and then a random real number between zero and one is generated. We also have three selection rates, $(r_1 > r_2 > r_3)$ and $r_1 + r_2 + r_3 = 1$, which determines the probabilities of each individual to be selected. The best individual will be selected with a probability equal to r_1 and so on. The higher is r_3 the more diverse will be our population, and therefore, we are encouraging the search space exploration in exchange for a slower convergence.

```
Algorithm 3: Tournament selection(population, scores, selection rates)
  Input: Current population, their scores and selection rates $(r_1, r_2, r_3)$
  Output: Selected parent
  Randomly chose $ind_1, ind_2$ and $ind_3$;
  Randomly generate $r$ between 0 and 1;
  if $r < r_1$ then
      return best individual;
  else if $r_1 < r < r_2$ then
      return second best individual;
  else if $r_2 < r < r_3$ then
      return third best individual;
```

4.6 Reproduction

In this stage the selected parents (4.5) are combined or mutated to generate the offspring. For the combination of the genetic information of two parents we have the crossover operator and for mutation we have three different types. First we select the individual to which we are applying the operator and then, we apply the correspondent operator depending on its probability to be applied $(r_{cross} > r_{m1} > r_{m2} = r_{m3})$. This process is shown in 4, for each of the parents we perform one of the genetic operators and generate a new individual for the next generation. For crossover it is necessary to select one more parent that will act as donnor. Each of the genetic operators is explained in the following sections.

```
Algorithm 4: Reproduction(Parents, reproduction rates)
  Input: Parents and reproduction rates ($r_{cross}, r_{m1}, r_{m2}, r_{m3}$)
  Output: Offsprings
  for $i <$ number of parents do
      Randomly generate $r$ between 0 and 1;
      if $r < r_{cross}$ then
          Select a donnor from population with tournament selection;
          offspring[i] = Crossover(parent, donnor);
      else if $r < r_{cross} + r_{m1}$ then
          offspring[i] = Mutation1(parent);
      else if $r < r_{cross} + r_{m1} + r_{m2}$ then
          offspring[i] = Mutation2(parent);
      else
          offspring[i] = Mutation3(parent);
```

4.6.1 Crossover

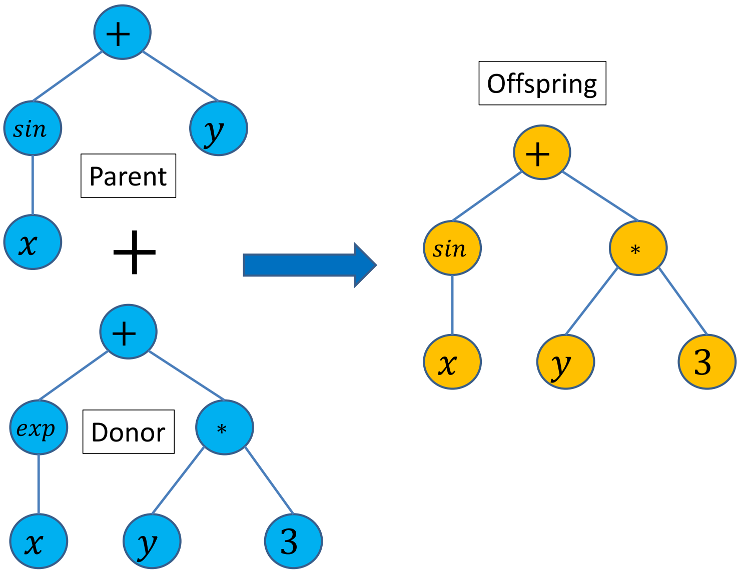


Figure 4.2: Crossover example.

As we are working with tree structures this operator consists of mixing the trees. First, we need to select one more random individual to act as a "donnor" in the operation. Then, we randomly select a random subtree from the parent and replace it by a random subtree from the donnor. This is shown in algorithm 5, where the subtree is selected by choosing a random node of the parent tree and and a random depth from that node.

```
Algorithm 5: Crossover(parent, donnor)
    Input: Parent and donnor
    Output: Offspring
1   Select random subtree from parent;
2   Select random subtree from donnor;
3   Substitutes parent subtree by donnor subtree;
```

4.6.2 Mutations

In this operation we modify genetic information of a certain individual to obtain a new one. Since the individuals are represented by trees, this modification consists of changing the tree elements in some way. In this system we have implemented three different type of mutations which are explained below.

Mutation 1

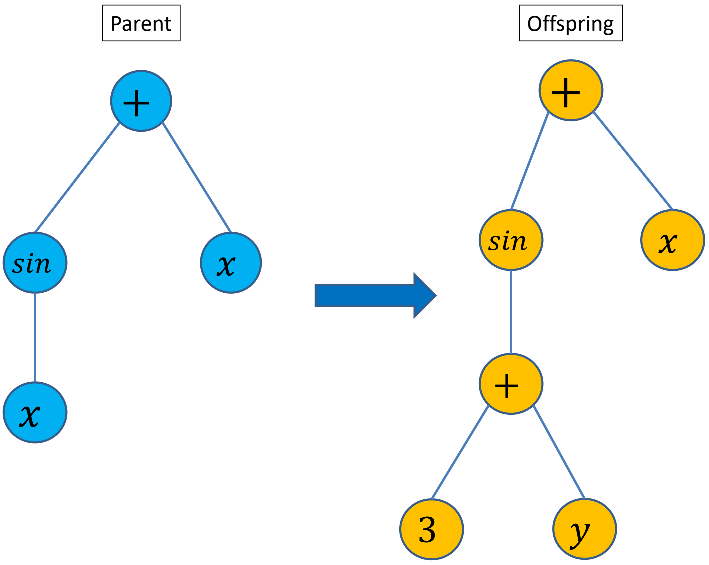


Figure 4.3: Mutation 1. A leaf is replaced by a random subtree

This mutation (figure 4.3) consists of replacing a leaf by a randomly generated subtree. In this way we are adding more genetic content to the individual and increasing its complexity. This mutation is implemented as shown in algorithm 6, this is a recursive function which traverse the tree, starting from the root, searching for a random leaf to replace. First, it checks whether the next element is a node or a leaf (for a n-ary node it will also choose randomly which child to check). If it is a leaf it will replace it by a random subtree, on the other hand, if it is a node, it will call the function recursively at that node until it finds a leaf. The random subtree generation is the same as in algorithm 2.

```
Algorithm 6: Mutation1(Node)
Input: Parent
Output: Offspring
if node is unary then
    if argument is leaf then
        Replace leaf by random subtree;
    else
        Mutation1(argument);
else if node is n-ary then
    Select random argument;
    if argument is leaf then
        Replace leaf by random subtree;
    else
        Mutation1(argument);
```

Mutation 2

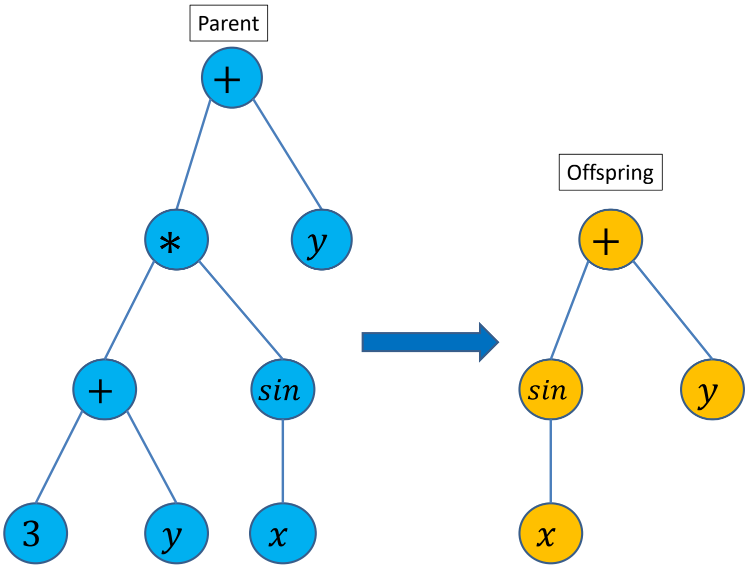


Figure 4.4: Mutation 2. Some genetic content is deleted (the left part of the tree is simplified).

This mutation (figure 4.4) consists of deleting some genetic content from the individual decreasing its complexity. This is accomplished by removing a leaf or a random subtree from the individual. The idea behind this implementation is that, sometimes, through crossover and the first type of mutation, an individual can increase its complexity without a significant improvement in its score, this can lead to a unnecessarily complex population which will make the system computationally more expensive.

Also, this mutation is a way to ”restart” some individuals (as it can, occasionally, delete most of the individual information or, even, all the information), which can be very useful in advanced populations where most of the individuals are genetically similar. This homogeneous populations are very likely to get stuck in a local minimum and this mutation is a way to increase the population heterogeneity and encourage the search space exploration.

This mutation is shown in algorithm 7, here the probability to delete the subtree from the current node is proportional to the depth of this node, so that the most drastic changes are less frequent. This is a recursive function which is called on the node argument if that node has not been selected for the deletion.

```
Algorithm 7: Mutation2(node, depth)
  Input: Node, depth (initially = 1)
  Output: Offspring
  generate $r$ between 0 and 1;
  if $r < mutation\_prob$ then
      delete subtree from current node;
  else
      if node is unary then
          Mutation2(argument, depth + 1);
      else if node is n-ary then
          select random argument;
          Mutation2(argument, depth + 1);
```

Mutation 3

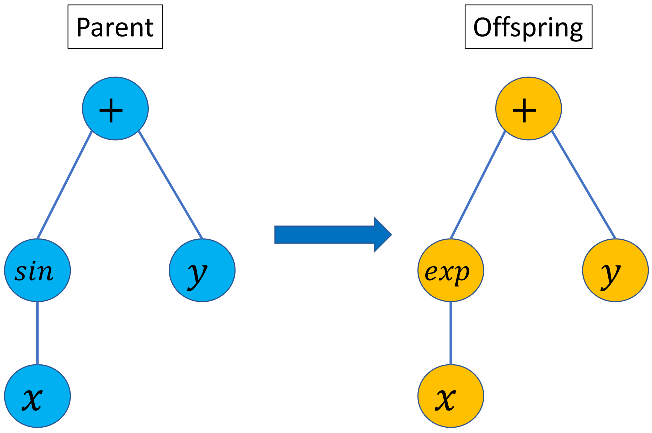


Figure 4.5: Mutation 3.Tree element substitution (*sin* node replaced by *exp* node).

In this mutation (figure 4.5) the change in the individual is performed by randomly substituting the tree elements by other of the same type, nodes by other nodes and leaves by other leaves. This mutation neither increases nor decreases the complexity and the genetic content of the individual, it just modifies it. The implementation is shown in algorithm 8.

Algorithm 8: `Mutation3`(*parent*)

```
Input: Parent
Output: Offspring
Select random tree element;
if element is node then
    replace by other random node;
else if element is leaf then
    replace by random leaf;
```

4.7 Replacement

In this stage a part of the current population is replaced by the new generated individuals. These replacement is based on the score of the individuals in such a way that the worst half population is replaced by the new individuals. By keeping the best half population from one generation to the next we are ensuring that the best individual for the new generation is, at least, as good as the best individual from the previous one.

4.8 Finalization

For the finalization, we can apply, even at the same time, two possible criteria: computation time, implemented in form of maximum number of generations, and best individual score. The specific values for these parameters are, obviously, dependant on the specific problem where the algorithm is applied and they must be tuned for each case. For example, in a low dimensional problem we can establish fewer maximum generations as the optimal solution is more likely to be found faster.

4.9 Genetic parameters

The genetic parameters of the algorithm are adjusted to the following values for all tested applications:

- **Population size:** There are 1000 individuals for each population. Half of them are replaced from one generation to the next.

- **Tournament selection rates:** For the tournament selection, the probabilities to be chosen are 80%, 15% and 5% for the first, second and third candidates respectively.

- **Mutation and crossover rates:** For each selected candidate the probabilities in the reproduction stage are: 50% of performing crossover, 20% for mutation 1 and 15% for mutation 2 and 3.

Chapter 5

Random number generation in GPU

As we have seen in chapter 4, this genetic programming system, and all evolutionary algorithms in general, has a high randomness, which leads to the generation of a large amount of random numbers. In particular, we are using random numbers in:

- the population initialization (section 4.3), as the depth of the individual is random and each node of an individual is also randomly generated;
- in the selection step (section 4.5) when we randomly select the individuals for the tournament and then also in the tournament to select the final parent;
- in the reproduction(section 4.7) step when we decide which genetic operator to apply and, specially, in those operators, both mutations and crossover, when we randomly traverse the individual trees, generate random subtrees for adding complexity, select random subtrees from parents in crossover, ...

So, for instance, in our case, for a population of 1000 individuals we have thousands of random number generations just in one iteration of the algorithm. Furthermore, as we are working with an iterative algorithm this random number generations are proportional to the experiment duration, which for a experiment of about 100 iterations, like the ones we expose in Chapter 6, may lead to millions of random number generations.

In this chapter, we expose how we have leveraged heterogeneous hardware, a GPU in this case, to accelerate this random number generation, and the results of this acceleration in terms of computation time and randomness quality.

5.1 Implementation on GPU

For this implementation we have used DPC++ (3.1.2) and Intel oneAPI toolkit. Specifically the oneDPL[1] library offers several random number engines.

[1] `https://docs.oneapi.io/versions/latest/onedpl/random.html`

The idea behind this random number generation is to avoid using the C++ standard libraries for randomness and substitute them by this customized generation, taking advantage of heterogeneous hardware such as GPU as some performance gains could be expected from generating the random values on the GPU versus the default CPU generation.

To speed-up the use of random values, the proposed implementation fills some large vectors with random numbers in the GPU, and, later, the CPU only reads them.

So the GPU with the oneDPL library fill multiple components of a vector at the same time at the beginning of the execution. In this way, the generation of each random number during runtime is just an access to the correspondent memory address where the number (*int* or *float*) is stored, which is faster than a call to a C++ random number generation. When all the numbers from a vector have been used, the access index is reset, and the vector is accessed from the beginning again. Although this reusage of numbers may compromise randomness at first glance, it is not really a problem if the vectors are large enough as we shall see later on.

The task of filling large vectors with random numbers is especially suitable for its implementation on GPU or other accelerator because this present a high degree of data parallelism as all the vector components are independent from each other and can be generated at the same time.

Regarding the vectors, we need different ones because, specially for integers, we need random numbers in different ranges depending on what they are going to be used for. The vectors and correspondent ranges used for the random number generation in the system are shown in table 5.1. For the real number generation, the generated numbers can be scaled for the specific purpose we need it; e.g., constants in a certain real range.

Function	Data type	Range	Size
Real number generation for constants and rates used for random decisions.	Float	[0.0, 1.0]	100000
Integer number generation to determine the type of function (unary, n-ary), the type of n-ary function (add, product) or the type of leaf (variable, constant). Mostly used in random trees generation.	Integer	[0, 1]	10000
Integer number generationto determine the type of unary function (sin, exp, step)	Integer	[0, 2]	10000
Integer number to determine which argument select from a n-ary function. Used in random tree traversal in genetic operators.	Integer	[0, 3]	10000

Tabla 5.1: Different vectors used for random number generation.

5.2 Implementation testing

We have compared this implementation to the random number generation using *rand()* function from the standar c library. For this comparison, we measure the time of the random number generation for real and integer numbers and we also compare the quality of the generated numbers with some tests for randomness [9].

5.2.1 Generation time comparison

This sub-section compares the generation time for both generator, standard and custom. The experiment consists of generating 30000 random numbers of each type with both generators, we made 5 repetitions of the experiment and compute the mean time. Results are shown in table 5.2.

Generator	Data type	Time [ns]
Stdlib generator	Float	399933.2
	Integer	369544.8
Custom generator	Float	50362.4
	Integer	37810

Tabla 5.2: Generation time comparison for *int* and *float* data types. Mean time of 5 repetitions of the experiment, 30000 random generations per experiment.

In table 5.2, we can see how the *float* generation is around 8 times faster with the custom generator, and the *int* generation is almost 10 times faster with the custom generator.

5.2.2 Randomness tests

In this sub-section, we perform two common empirical tests on the random distributions generated with both generators. These 2 empirical tests are the **Serial test** and the **Run test**.

The **Serial test** consists of verifying that in a sequence of random numbers sufficiently large, the different pairs of possible numbers (two consecutive numbers in the sequence), $(Y_j; Y_{j+1}) = (q; r)$, appear randomly with the corresponding probability. For this, a random sequence of numbers is generated, and the number of times each of the tuples, $(Y_j; Y_{j+1}) = (q; r)$ occurs for $0 \leq j < n$, appears is counted. In this way, we have d^2 categories and knowing that theoretically the probability of appearance of each category is the same, $p_r = \frac{1}{d^2}$, a X^2 test is applied on these d^2 categories.

And the **Run test** consists of measuring the length of monotone subsequences in a sequence of random numbers. A monotone sequence is a sequence that has elements that are either all increasing or all decreasing. The idea behind this test is to check the correlation between the appearance of long and short subsequences inside a long enough sequence of random numbers.

To perform these tests we have generated four sequences of random numbers of 30000 numbers each one. In the case of the custom generator we are using vectors of 10000 numbers each one, in this way, taking 30000 numbers, we are forcing to "reuse" the numbers from the vectors which, a priori, should be worse for randomness properties. We have 2 sequences of real numbers generated with both generators and other 2 sequences of integers numbers. Then we have performed both mentioned test over the integer sequences and the Run test over the float integer given that the Serial test is not suitable for this sequence as we do no have a finite numbers of categories. Results are shown in table 5.3.

Generator	Sequence	Test	Result
Custom generator	Float	Run	Passed
	Integer	Serial	Passed
		Run	Passed
Stdlib generator	Float	Run	Passed
	Integer	Serial	Not passed
		Run	Passed

Tabla 5.3: Results of the tests performed over the sequences of numbers generated with both generators.

So, as these results show, the custom generator has better random proprieties as it passes the serial test. This, in addition with results in table 5.2, make this custom generator suitable for the random numbers generation in the system development.

Chapter 6

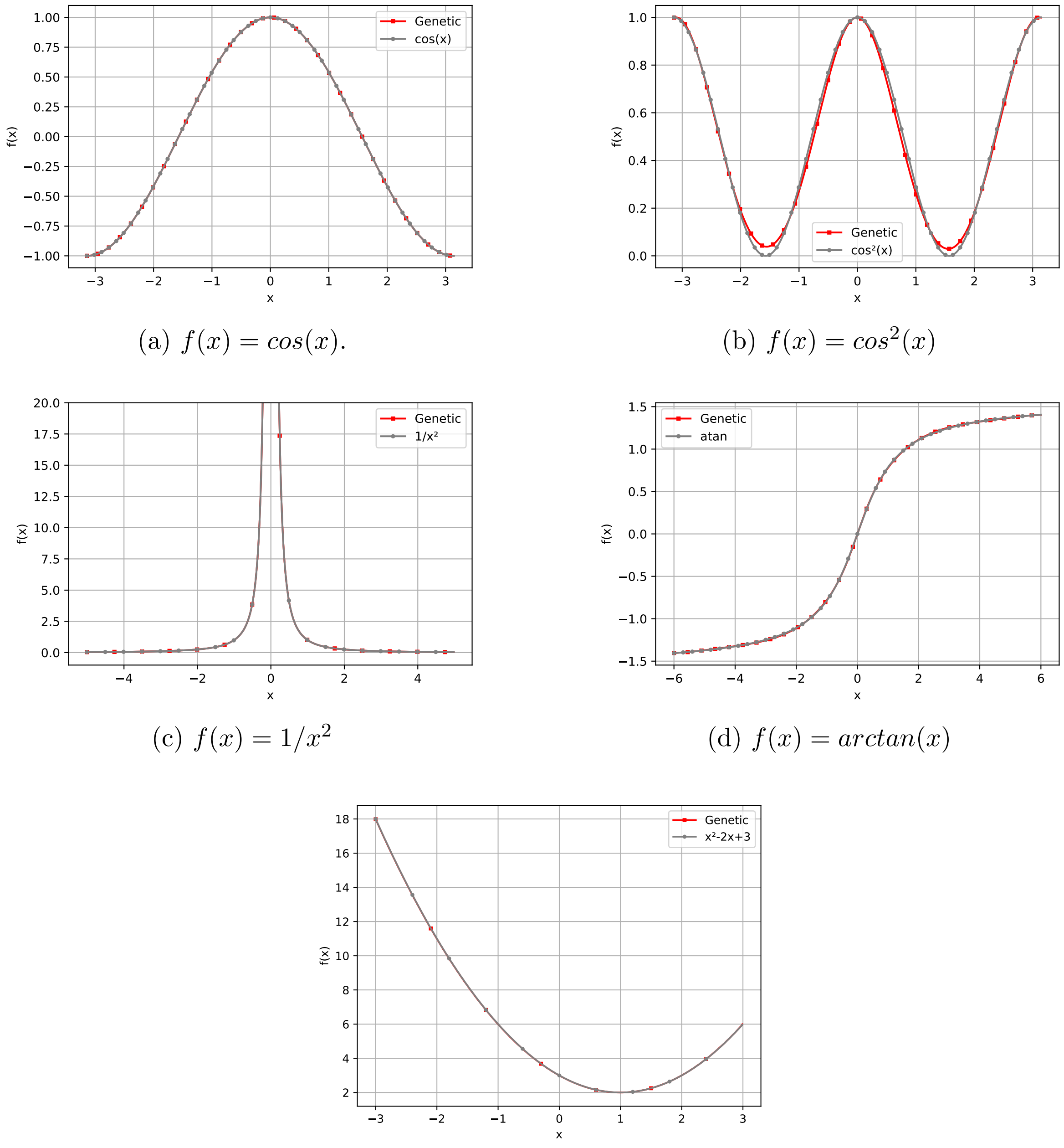
Evaluation

To test our approach, we have applied the solution to two relevant problems in the Computer Graphics domain: single scattering and image compression. Each problem requires its own function and variable sets, and, a different loss function to compute the individual scores. Previously, to validate the system, we have performed some experiments for less complex and controlled cases that are described in Section 6.1.

6.1 Validation experiments

6.1.1 Function approximation

We have first tested the system on function regression. The idea is to validate the system ability to accurately approach different functions which are not in the system function set or a combination of those. This capability is essential for solving more complex problems. The results are shown in results in figure 6.1, where we can see plotted both the objective function and the system solution; and table 6.1, where we have several experiment statistics for the DPC++ implementation as well as the resultant expression for each experiment. In these first validation experiments we have used Mean Squared Error between the objective function and the genetic output as loss function for the individual score computation.



(a) $f(x) = cos(x)$.

(b) $f(x) = cos^2(x)$

(c) $f(x) = 1/x^2$

(d) $f(x) = arctan(x)$

(e) $f(x) = x^2 - x + 3$

Figure 6.1: Function approximation results. In each subfigure we have the correspondent objective function and the system output.

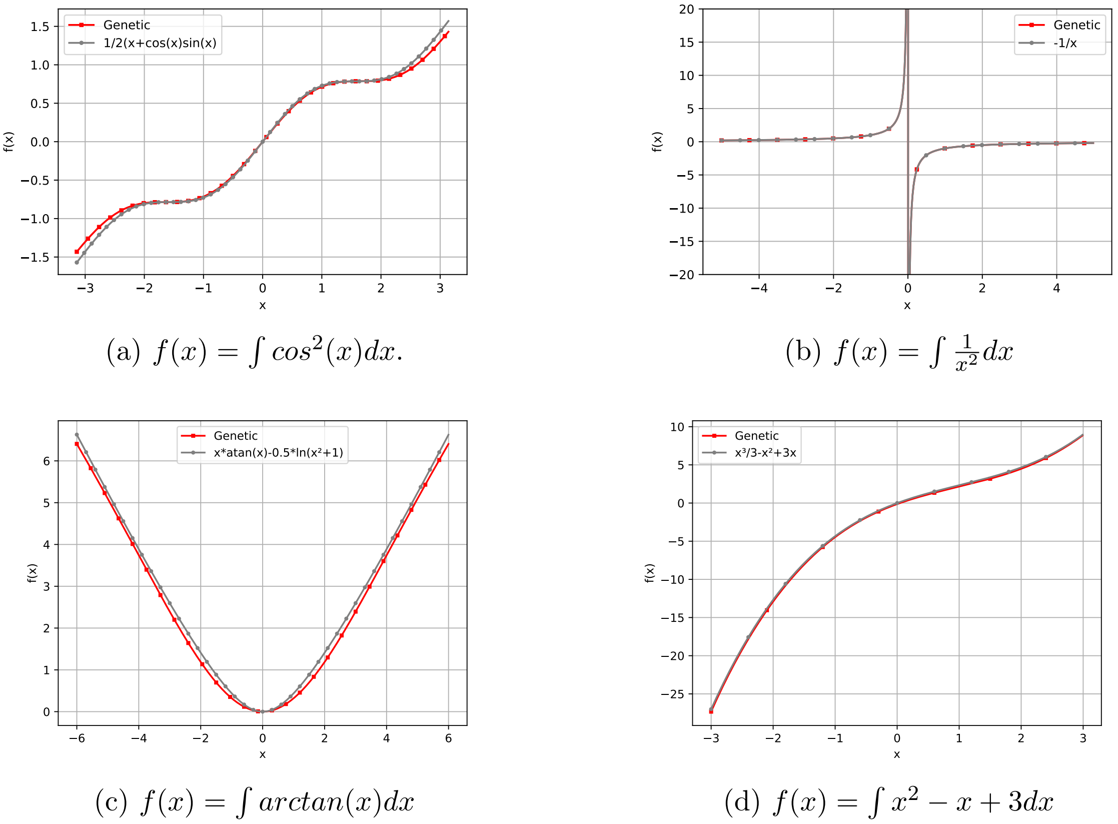
f(x)	Total time [ms]	Generations	Samples	Algebraic expression
cos(x)	6333	88	5000	$sin(x+1.572)$
cos^2(x)	37803	200	5000	$e^{-\frac{1.15sin^2(x)}{1.1e^{-1.15sin^4(x)}}}$
1/x^2	224	4	2000	$1/x^2$
atan(x)	40330	180	8000	too complex
x^2-2x+3	6850	132	2000	$x^2-1.993x+2.984$

Tabla 6.1: Function approximation experiments statistics and system output for each function.

These results show that the approximations are accurate in all cases with very few error. The obtained expressions are very similar to the objective functions in the cases where the objective function terms are in the system function set.

6.1.2 Genetic integration

Since the main purpose of the system is to integrate analytically non-integrable functions we need to test the ability of the system to approximate the antideriative of a certain function. In these first validation experiments we are not yet working with analytically non-integrable functions, instead, we are first approximating the antiderivate of integrable functions and comparing it with the analytical integral. For the experiments in this section, to compute the score of a certain individual, we first derive it and then evaluate its derivative in the given points. In this way, by fitting the individuals derivative to the objective function, we are directly obtaining an algebraic expression which approximates the objective integral in the given range. So, we have used Mean Squared Error between the objective function and the derivative of the individual as loss function for the individual score computation. We have tested the algorithm with several functions, and we can see the results in figure 6.2, were we plot both the integral of the objective function and the system output in the given range; and table 6.2, where we can see several experiment statistics as well as the resultant genetic antiderivatives. The experiments have been also performed with the DPC++ implementation of the system.



(a) $f(x) = \int cos^2(x)dx.$

(b) $f(x) = \int \frac{1}{x^2}dx$

(c) $f(x) = \int arctan(x)dx$

(d) $f(x) = \int x^2 - x + 3dx$

Figure 6.2: Genetic integration. In each subfigure we have the correspondent objective function and the system output.

$\int f(x)dx$	Time [ms]	Gens	Samps	Algebraic expression
$\frac{(x+cos(x)sin(x))}{2}$	442156	2100	5000	$x * e^{sin(1.25sin(-\frac{x^2}{3.83}))}$
-1/x	80567	50	100000	$-0.999993/x$
$x * atan(x) - \frac{ln(x^2+1)}{2}$	45215	200	8000	$-1.615x * sin(sin(sin(sin(-0.204x))))$
$x^3/3 - x^2 + 3x$	10765	200	2000	$x^3/2.9968 - 1.9998x^2 + 3.0012x$

Tabla 6.2: Genetic integration experiments statistics and system output for each objective function.

As we can see, the integrals are accurately approximated with very few error in all cases. The experiments time depends on the number of samples, which directly increases the number of evaluations, and on the complexity of the function to approximate. More complex functions means larger algebraic expressions as individuals of the population which are slower to evaluate. For the functions whose terms are in the system function set the obtained expression is similar to the analytical integral.

6.1.3 Comparison between Sequential and Parallel Versions

When profiling the sequential version of these validation experiments, we noticed that the expression evaluation at the sample points takes more than 60% of the computation time in the sequential version. So, taking this into account, we have accelerated that part of the program. We have used an eight-core CPU. For this parallelization we have two options: evaluate several of the 1000 individuals at the same time (figure 6.4) or evaluate one individual at several sample points at the same time (figure 6.3). In figures 6.5 and 6.6 we have the results of the comparison between sequential version and both parallel versions approximating the objective function $f(x) = cos^2(x)$.

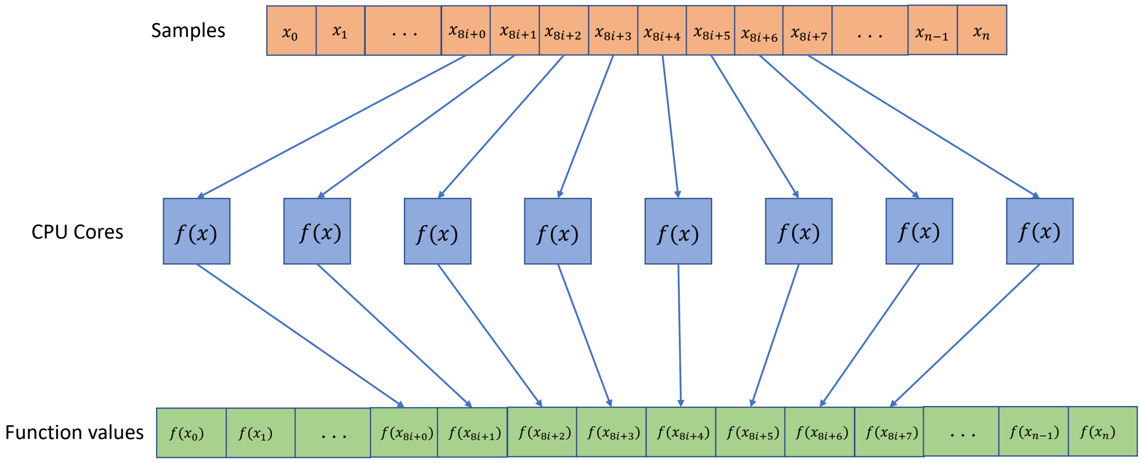


Figure 6.3: Evaluation of several samples in parallel. In this case we are parallelizing the computation of the individual value at each sample point, so each of the 8 CPU cores computes the individual function value, $f(x)$, on a different sample point, x_i.

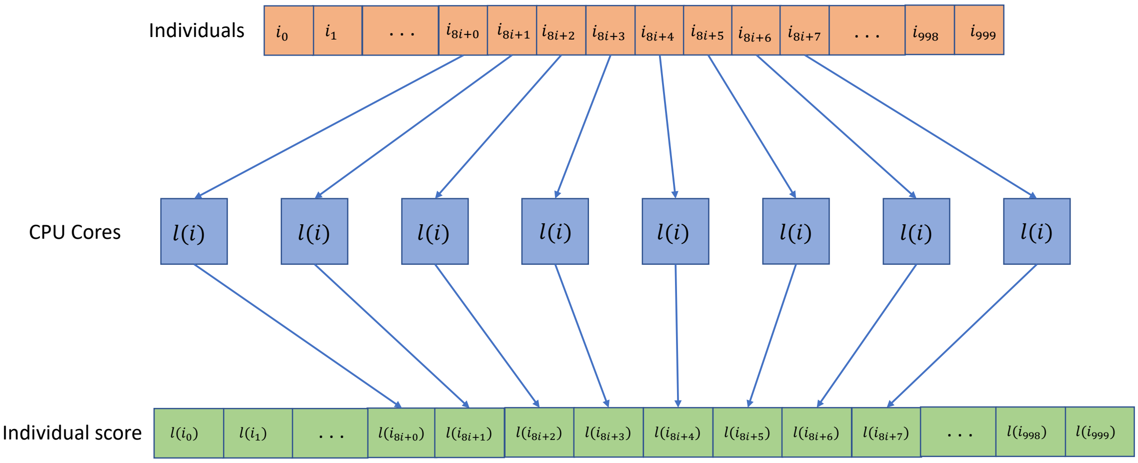
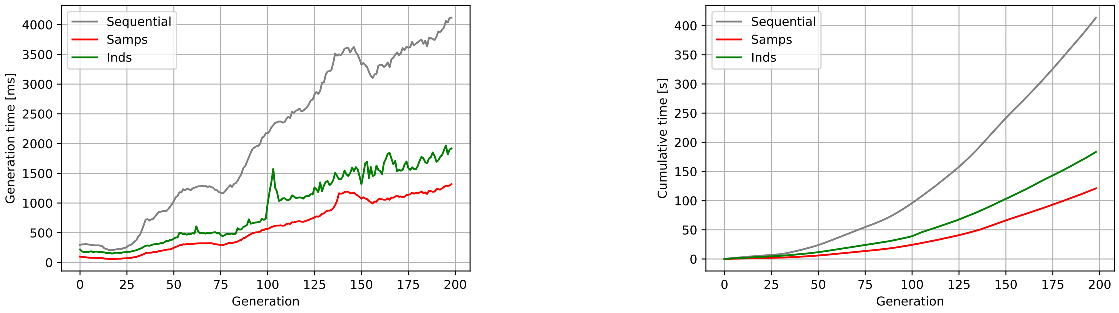


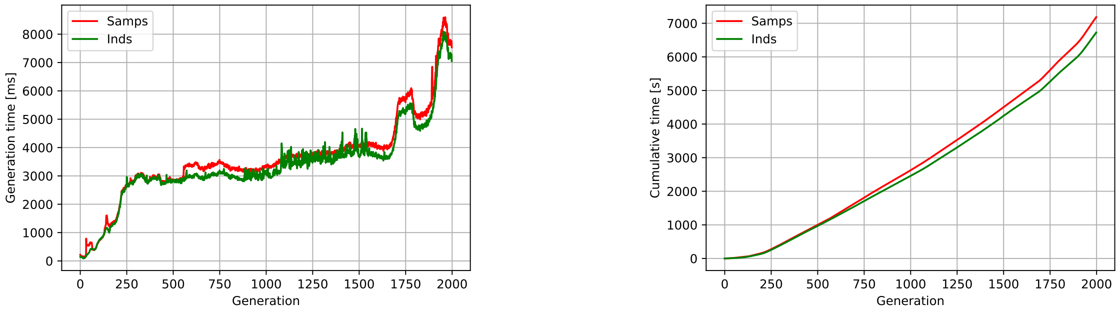
Figure 6.4: Evaluation of several individual scores in parallel. In this case we are parallelizing the computation of the 1000 individuals scores, so each of the 8 CPU cores computes the loss function, $l(i)$, on a different individual, i_i.



(a) Time per Gen.

(b) Cumulative Time per Gen.

Figure 6.5: Execution time comparison between sequential and parallel implementations.



(a) Time per Gen.

(b) Cumulative Time per Gen.

Figure 6.6: Execution time comparison between both parallel versions.

In the figures above, *Samps* corresponds to the parallel implementation where we evaluate an individual at several sample points at the same time; *Inds* corresponds to the parallelization of the 1000 individuals score computation; and *Sequential* to the sequential implementation. Figure 6.5 shows that both parallel versions are significantly faster than the sequential one, achieving a x3.3 acceleration for the individual parallelization and x2.4 for the samples parallelization. And in 6.6 we noticed that, for a longer experiment, both parallel versions are similar. For this specific objective function evaluating several individuals in parallel is a bit faster than evaluating several sample points in parallel for the same individual, but this results also depends on the complexity of the objective function and the number of variables.

6.1.4 Integration of no-integrable functions

In these experiments, we have obtained an algebraic expression which approximates the integral of two different functions without analytical integral. To measure the accuracy of the results we have compared the definite integral in a given range with the Monte Carlo integral in the same range with a high enough number of samples. In table 6.3 we can see the results.

$f(x)$	Time [ms]	Generations	Integral error	$\int f(x)dx$
$\sqrt{x^3-1}$	313193	82	0.7%	$(0.342x-1.435)(0.216sen(x)+0.1x^2-$ $-1.913)+\frac{1.327x^3}{x+1.98}-1.058)$
$ln(ln(x))$	282633	100	1.15%	$e^{\frac{e^{-2.79x^2}(\frac{1}{x^{2,43}}+x-2.43ln(x))}{x+.021}}$

Tabla 6.3: No-integrable functions results

In both cases, the error between the Monte Carlo integral and the "genetic" integral is around 1% (0.7% for the first function and 1.15% for the second one) and could be even lower with more generations and computation time. This can be, definitely, considered as an accurate approximation of the antiderivative in the experiment range.

6.2 Single Scattering

In light transport, scattering is a physical phenomenon which describes how light is differentiably affected by a medium while propagating through it. This kind of media, that can affect the light transport, is called participating media. In this application, we are targeting the integration of single scattering (a single bounce in the medium) for the particular case of a point light source in a homogeneous participating medium.

Light transport in participating media is modelled by the Radiative Transfer Equation [10]:

$$\frac{\partial L(\mathbf{x},\omega)}{\partial t} = \sigma_a(\mathbf{x})L_e(\mathbf{x},\omega) - \sigma_t(\mathbf{x})L(\mathbf{x},\omega) + \sigma_s(\mathbf{x})\int_\Omega p(\mathbf{x},\omega',\omega)L(\mathbf{x},\omega')\,d\omega' \tag{6.1}$$

where $\frac{\partial L(\omega_o)}{\partial t}$ represents the differential variation of radiance along the path of light t, $\mathbf{x}$ is the differential point at which the interaction occurs, ω represents the direction followed by light and ω' represents the direction of other light paths that reach the differential point $\mathbf{x}$ (Ω is the domain of integration, a sphere). The rest of the symbols represent the properties of each medium:

- σ_a is the absorption coefficient, energy that is differentially absorbed by the medium at every distance unit.

- σ_s is the scattering coefficient, energy differentially scattered by particles in the medium at every differential step.

- $\sigma_t = \sigma_a + \sigma_s$ is the extinction coefficient, energy that is either absorbed or out-scattered.

- $p(\mathbf{x},\omega',\omega)$ is the phase function, that defines the angular distribution of light scattering.

- L_e is the medium's emission.

In order to render participating media, we need to solve Equation 6.1 and obtain the radiance $L(\mathbf{x},\omega)$. Equation 6.1 is a linear ordinary differential equation, and therefore has an analytical integral solution:

$$L(\mathbf{x_0},\omega) = T_r(\mathbf{x_0},\mathbf{x_t})L(\mathbf{x_t},\omega) + \int_0^t T_r(\mathbf{x_0},\mathbf{x_s})\sigma_a(\mathbf{x_s})L_e(\mathbf{x_s},\omega)ds + \int_0^t T_r(\mathbf{x_0},\mathbf{x_s})\sigma_s(\mathbf{x_s})L_i(\mathbf{x_s},\omega)ds \tag{6.2}$$

where $\mathbf{x_0}$ is the origin point of the ray, $\mathbf{x_s} = \mathbf{x_0} + s\mathbf{d}$ is a differential point of interaction at the position parametrized by s ($\mathbf{d}$ is the light propagation direction) and t is the distance from which the light that enters the medium comes from. $T_r(\mathbf{x_0},\mathbf{x_s})$ is named *transmittance* and accounts for all the light that has traversed the medium between $\mathbf{x_0}$ and $\mathbf{x_s}$ without getting extinguished due to the medium's properties. $L_i(\mathbf{x_s},\omega)$

represents the in-scattered radiance (energy coming from different light paths) at point $\mathbf{x_s}$. They are defined as follows:

$$T_r(\mathbf{x_0}, \mathbf{x_t}) = e^{\int_0^t -\sigma_t(\mathbf{x_s}) ds} \tag{6.3}$$

$$L_i(\mathbf{x_s}, \omega) = \int_\Omega p(\mathbf{x_s}, \omega', \omega) L(\mathbf{x_s}, \omega')\, d\omega' \tag{6.4}$$

For the particular case of homogeneous media, in which the coefficients do not vary along the volume, transmittance, as defined in Equation 6.3, can be computed analytically:

$$T_r(\mathbf{x_0}, \mathbf{x_s}) = \frac{e^{-\sigma_t ||x_s - x_0||}}{\sigma_t} \tag{6.5}$$

Furthermore, we assume an isotropic (constant) phase function:

$$p(\mathbf{x_s}, \omega', \omega) = \frac{1}{4\pi} \tag{6.6}$$

and, last, besides an isotropic phase function, we assume single scattering from a point light source with emitting intensity I_l and position $\mathbf{x_l}$, which is a delta function in space, hence the integral for the in-scattering, as defined in Equation 6.4, is transformed into

$$L_i(\mathbf{x_s}, \omega) = \frac{I_l e^{-\sigma_t ||x_l - x_s||}}{\sigma_t 4\pi ||x_l - x_s||^2} \tag{6.7}$$

which accounts for the inverse square of the point light and the transmittance between the point light's position $\mathbf{x_l}$ and the differential point in the medium $\mathbf{x_s}$.

If we only focus on the in scattering term, which is the most challenging part, and we incorporate Equation 6.7 into the scattering term of the integral version of the Radiative Tranfer Equation (Equation 6.2), we get:

$$L(\mathbf{x_0}, \omega) = \int_0^t e^{-\sigma_t s} \frac{\sigma_s}{\sigma_t} \frac{I_l e^{-\sigma_t ||x_l - x_s||}}{4\pi ||x_l - x_s||^2} ds \tag{6.8}$$

$$= \frac{I_l \sigma_s}{4\pi\sigma_t} \int_0^t \frac{e^{-\sigma_t (s + ||x_l - x_s||)}}{||x_l - x_s||^2} \tag{6.9}$$

In order to minimize the number of variables for the genetic optimization, we are going to reparametrize the light's position with respect to the ray, into two parameters: light position along the ray (s_l) and distance between light and ray (d_l) that can be calculated as follows:

$$s_l = (\mathbf{x_l} - \mathbf{x_0}) \cdot \mathbf{d} \tag{6.10}$$

$$d_l = \sqrt{||\mathbf{x_l} - \mathbf{x_0}||^2 - s_l^2} \tag{6.11}$$

where $\mathbf{d}$, as discussed earlier, is the propagation direction that fulfils $\mathbf{x_t} = \mathbf{x_0} + t\mathbf{d}$.

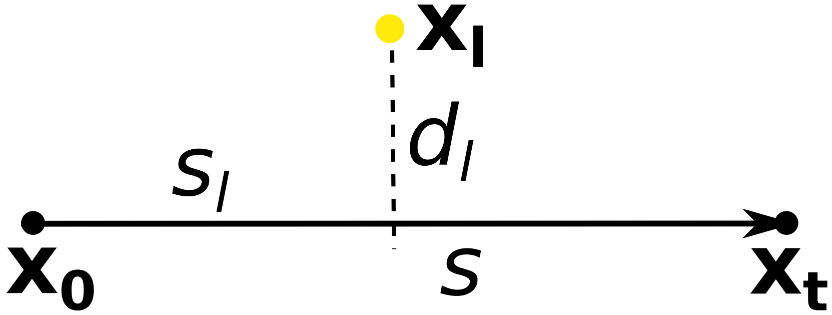


Figure 6.7: Illustration of setup for single scattering, including starting and ending positions ($\mathbf{x_0}$ and $\mathbf{x_t}$), light position ($\mathbf{x_l}$) and variables that define the light position with respect to the ray s_l and distance from the light source to the ray d_l.

With this parametrization, it is possible to obtain the distance between any point $\mathbf{x_s}$ along the trajectory of the ray as

$$||\mathbf{x_l} - \mathbf{x_s}|| = \sqrt{(s - s_l)^2 + d_l^2} \tag{6.12}$$

Therefore, the equation to solve, after this change of variable, is

$$L(\mathbf{x_0}, \omega) = \frac{I_l \sigma_s}{4\pi\sigma_t} \int_0^t \frac{e^{-\sigma_t\left(s+\sqrt{(s-s_l)^2+d_l^2}\right)}}{(s - s_l)^2 + d_l^2} \tag{6.13}$$

Traditionally, this integral would be solved through a numerical integration method, which very often would be either Monte Carlo integration (random samples along the trajectory of the ray) or a multistep rectangle rule (commonly called ray marching [11]).

In this work, our aim is to solve this problem by directly obtaining an algebraic function, $g(s, \sigma_t, d_l, s_l)$, that approximates the antiderivative of the integrand in Equation 6.13 in such a way that:

$$\frac{\partial g(s, \sigma_t, d_l, s_l)}{\partial s} = \frac{e^{-\sigma_t\left(s+\sqrt{(s-s_l)^2+d_l^2}\right)}}{(s - s_l)^2 + d_l^2} \tag{6.14}$$

If, using our system, we are able to obtain the function $g(s, \sigma_t, d_l, s_l)$ that satisfies Equation 6.14, then we will obtain a practical antiderivative that we would only need to, in the absence of objects that cast shadows, evaluate at the two limits of the integration range in Equation 6.13.

Dimensionality reduction: For this problem we were originally exploring a four dimensional space which, by itself, is already a really challenging task. Furthermore, in our system, the configuration space of the solutions which conform the population is, theoretically, infinite since our trees have not a maximum size and the genetic operators enable linear and non-linear combinations of the basic functions like function composition. Taking this into account, we have reduced the original dimensions of equation 6.14. But, even with this modification, the new single scattering function is also analytically non-integrable, so obtaining an approximate integral for this equation is still an interesting application of the system.

This reduction consists of passing from 4 variables to 2, fixing the value of the extinction coefficient, $\sigma_t = 0.1$, and the light position along the ray, $s_l = 0.1$. So the problem we are finally going to solve is the following:

$$\frac{\partial g(s, d_l)}{\partial s} = \frac{e^{-0.1\left(s+\sqrt{(s-0.1)^2+d_l^2}\right)}}{(s-0.1)^2 + d_l^2} \tag{6.15}$$

6.2.1 Loss function

For this problem we have defined the following loss function:

$$loss = \frac{\sum_{p_o}^{p_n} \left| \frac{\partial g(p_i)}{\partial s} - \frac{\partial L_o(p_i)}{\partial s} \right|}{n} \tag{6.16}$$

where $p_i(s, d_l)$ are each of the points in which we are evaluating both expressions, and n is the total number of samples, so this loss is, basically, the mean absolute error between the derivative of the individual and equation 6.15.

6.2.2 Function and variable sets

Variable set

This set must contain the amount of variables necessary to explore all the dimensions of the search space. In this problem we have the two variables, s, d_l, of equation 6.15.

Function set

In this problem the basic functions must be analytically differentiable since it is necessary to compute the individual derivative by chain rule. The function set is composed by the following functions:

1. *sin* and *exp* as unary functions,

2. *addition* and *product* as n-ary functions.

6.2.3 Parallelization

Similar to what we saw in the validation experiments 6.1.3, when profiling the sequential version of the program for this application, we noticed that most of the time was used in the evaluation of the derivative of the genetic output function at a sample point, and, in this application, we have 200000 sample points. So, under these conditions, we have parallelized the derivative evaluation at each sample point.

6.2.4 Experiments and results

Versions comparison

In this experiment we can see the comparison in computation time between the two different implementations of the system: sequential and parallel. The experiment have been performed under the same conditions: 100 generations as ending condition, same random seed and same machine (machine in 3.2).

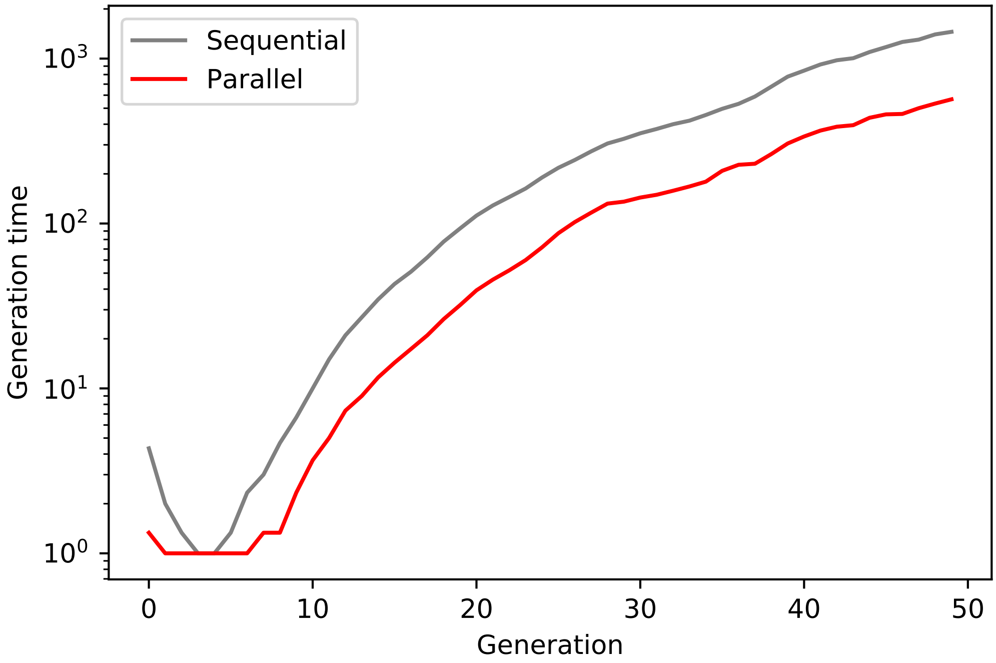


Figure 6.8: Time per generation in logarithmic scale

Figure 6.8 shows the time per generation in logarithmic scale. For the sequential implementation the execution time is around a half magnitude order greater than for parallel version. This make sense given that we have used a eight-core machine for the experiments, so, following the Amdahl Law (equation 2.3), the maximum speed-up would be $x8$ and, in this case as, obviously, not all the code is parallelized, we are obtaining around $x5$ acceleration.

Resultant images

In figures 6.9, 6.10 and 6.11 we can see the resultant images renderized with the obtained genetic integral and their comparison to images renderized with Monte Carlo integration. These scenes consist of a point light source in two different (corner and middle) positions in a homogeneous participating media and an orthographic camera without shadows. As we only need 2 function evaluations with this genetic integral to compute the light contribution of a certain light ray, in figures 6.9 and 6.10 we compare these "genetic" images with the renderized images obtained with Monte Carlo integration with 2 samples per pixel and we use as ground-truth two images renderized with Monte Carlo integration with 1024 samples per pixel.

As we have an analytical expression for the integral, although we are changing the point light position, we do not have to recompute the integral approximation, we can generate both images from the same genetic output just changing light position parameters, d_l, when computing the definite integral. Time and convergence statistics of this experiments are shown in figure 6.13.

As we can see in figures 6.9 and 6.10 the resultant images are quite similar to the ground-truth images and, contrary to the 2 samples Monte Carlo image, they have no noise. In figure 6.11 we can see how the Monte Carlo integration based images improve as the number of samples increase, but even for 256 samples the genetic image is still less noisy. In figure 6.12 we can see how the most difficult part of the image to approximate is the area near the light source, this is because of the divergence in equation 6.15 due to the square distance in the denominator.

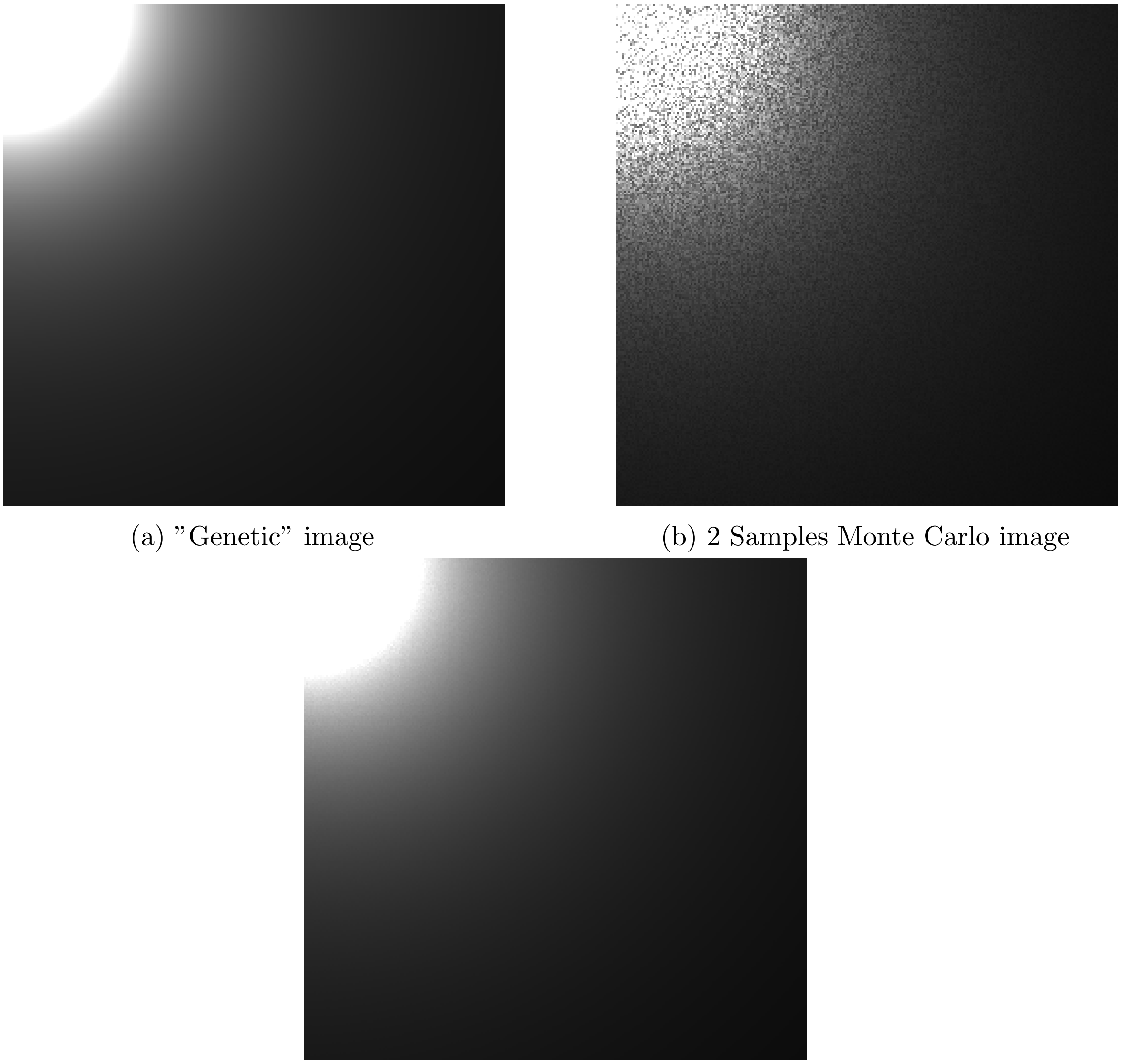
6.3 Image compression

The system has been applied to image compression. The idea behind this application is to represent the image, $I(x, y)$, in form of a bi-dimensional algebraic expression, $g(x, y)$, where the two variables (x, y) represent the pixel coordinates, height and width, as shown in 6.17. This is done by taking the pixel values, $I(x_p, y_p)$, as input data points and adjusting a bidimensional function to those points.

$$g(x_p, y_p) = I(x_p, y_p) \ \forall \ x_p, y_p \tag{6.17}$$

6.3.1 Loss function

For this problem we have defined the following loss function:



(a) "Genetic" image

(b) 2 Samples Monte Carlo image

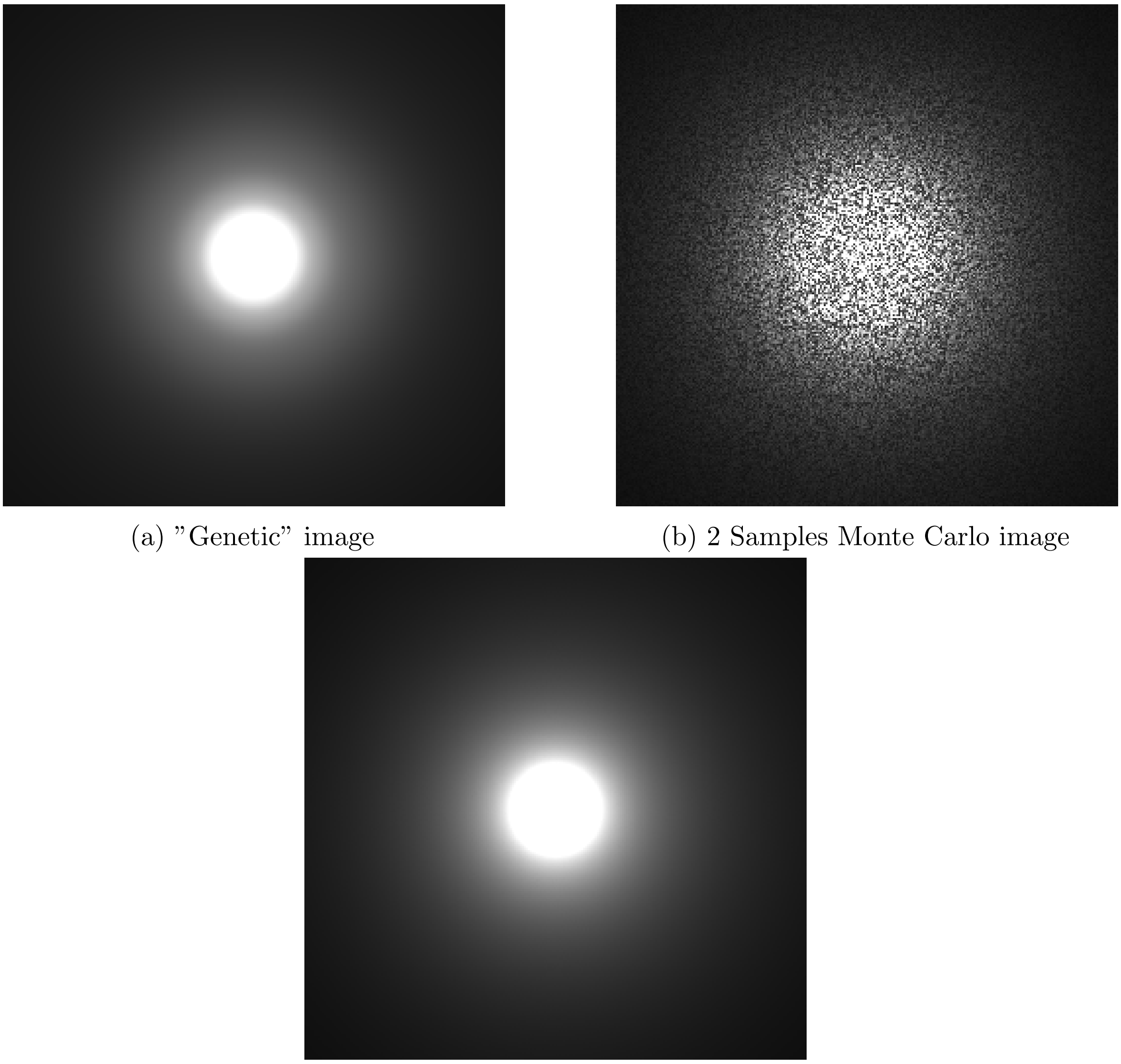
(c) Ground truth image

Figure 6.9: Comparison between images for a scene with a point light source in the corner of the scene and a participating media.

$$loss = \frac{\sum_{x_p=0}^{x} \sum_{y_p=0}^{y} |I_g(x_p, y_p) - I(x_p, y_p)|}{n_{pixels} * 255} + KS(g(x, y), I(x, y)) \tag{6.18}$$

Where:

- The first term is basically the **mean absolute error** between the pixel, (x_p, y_p), values of the objective image $I(x, y)$ and the image generated by the system output, $I_g(x, y)$, evaluated in the same pixel coordinates. As the second term is in the range from 0 to 1, and we want to give them the same weight in the loss computation, this first loss term is normalized, dividing it by 255.
- The second term is the value obtained by performing the **Kolmogrov-Smirnov**



(a) "Genetic" image

(b) 2 Samples Monte Carlo image

(c) Ground truth image

Figure 6.10: Comparison between images for a scene with a point light source in the middle of the scene and a participating media.

test [12]. This is a non-parametric test of the equality of continuous or discontinuous, one-dimensional probability distributions that can be used to compare a set of samples with a reference probability distribution, or to compare two sets of samples. In this particular case it is used to compare the image generated by the system output, $I_g(x, y)$, and the objective image, taking as sets of samples the pixels values from the two images. To compare these two sets, as we are working with discrete values from 0 to 256, we have computed the histograms for both images, representing the number of pixels for each pixel value, and that is what we are comparing with the KS test. The histogram comparison with Kolmogrov-Smirnov test is computed as follows:

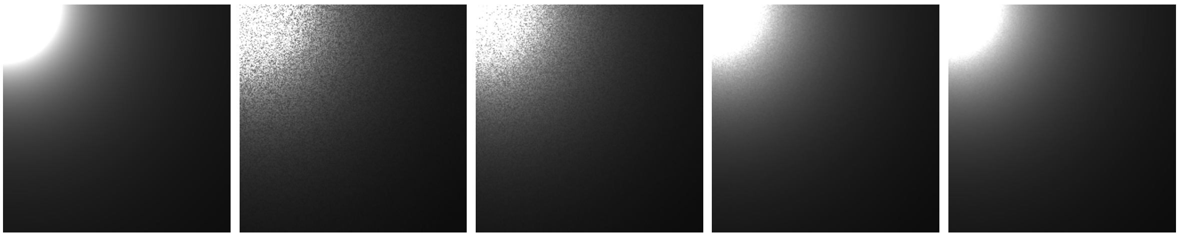


Figure 6.11: From left to right: "Genetic" image and 2, 4, 64 and 256 Monte Carlo images for the first scene.

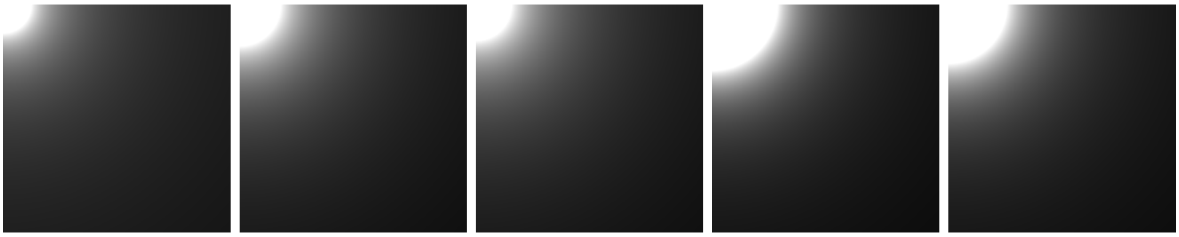
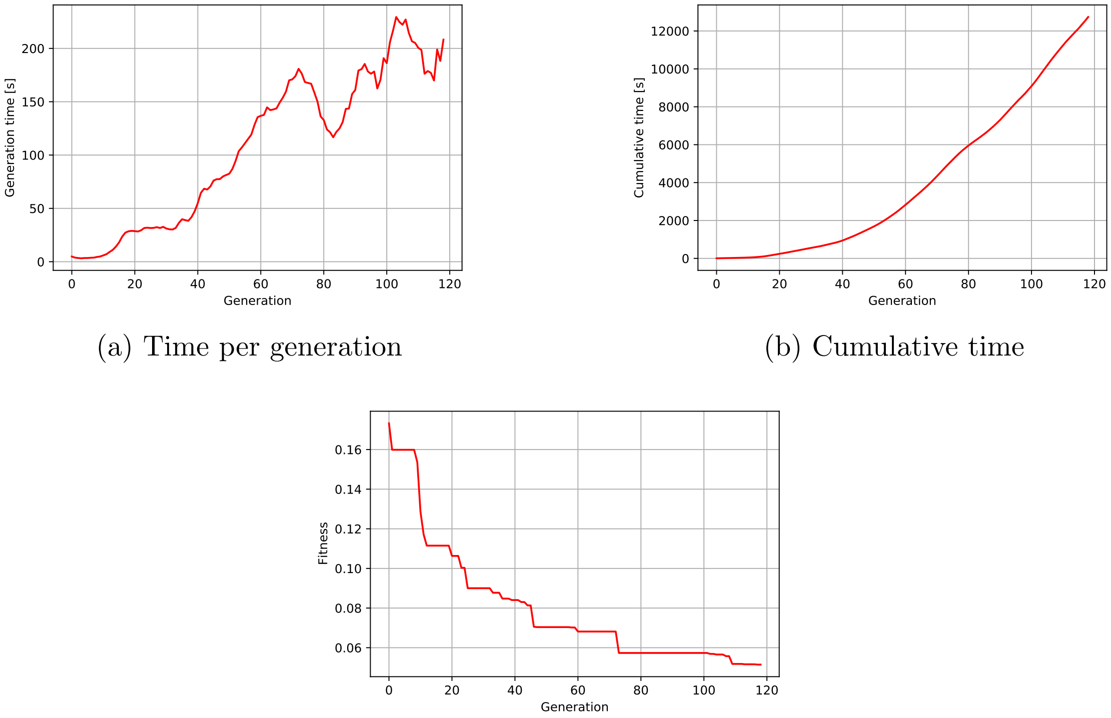


Figure 6.12: From left to right: Renderized images with the genetic integral obtained in generation 0, 30, 60, 90 and 120.

$$KS = max\,|CmH_g(x) - CmH_o(x)| \tag{6.19}$$

where $CmH_g(x)$ and $CmH_o(x)$ are the cumulative histograms for the generated and the objective images respectively and x is the pixel value from 0 to 256. So what we are trying to minimize here is the differences between the cumulative histograms.

In the first versions we tried to solve this problem without the Kolmogrov-Smirnov test term, using just pixel to pixel error measurements such as mean absolute error or mean square error. But, in this first experiments, the system output used to be homogeneous images whose pixel values were equal or very similar to the average pixel value, when using mean absolute error, or to the most repeated pixel value (black or white), when using quadratic error metrics. This was due to the system getting stuck in a local minimum because the exploration strategy was only based in pixel to pixel error. So, adding the Kolmogrov-Smirnov test term, we are improving the search space exploration since we are forcing the output to have the same pixel distribution as the objective image, and, in this way, the aim of the mean absolute error term is to "place" the pixels in the correct image coordinates.



(a) Time per generation

(b) Cumulative time

(c) Fitness per generation

Figure 6.13: Convergence and time statistics for the single scattering experiment.

6.3.2 Function and variable sets

Variable set

This set must contain the amount of variables necessary to explore all the dimensions of the search space. In this particular problem we are working with a black and white image where each pixel value is a function of its image coordinates, $p = I(x_p, y_p)$, so the variable set in composed by two variables, x and y, which represents the image coordinates.

Function set

For this problem the function set is very similar to the previous. The step function is specially useful in this case due to the common nature of images where we can find different shapes and elements. The step function can model the edges of the different shapes of the image. So the function set is composed by the following functions:

1. $sin(x)$, $exp(x)$, $ln(x)$ and $step(x)$ as unary functions,

2. *addition* and *product* as nary functions.

6.3.3 Parallelization

As in the previous application, when profiling the program execution for this application, we noticed that most of the time was used in the evaluation of each individual at a sample point. So, there are two main aspects of the system which are the most interesting to optimize through parallelization:

- **Individual loss computation:** This operation, for each individual, is completely independent from the rest, so this part of the algorithm presents a high degree of parallelism.

- **Pixel evaluation:** For each individual we are computing each pixel value, this computation is also independent between pixels. For a 400x400 image (like the ones we are using in this application), we have 160000 pixel value computations to run simultaneously.

6.3.4 Experiments and results

First image compression

In 6.14 and 6.15 we can see the results for bicolor image. In 6.16 we can see the time and convergence stats for the 110 generations of the experiment.

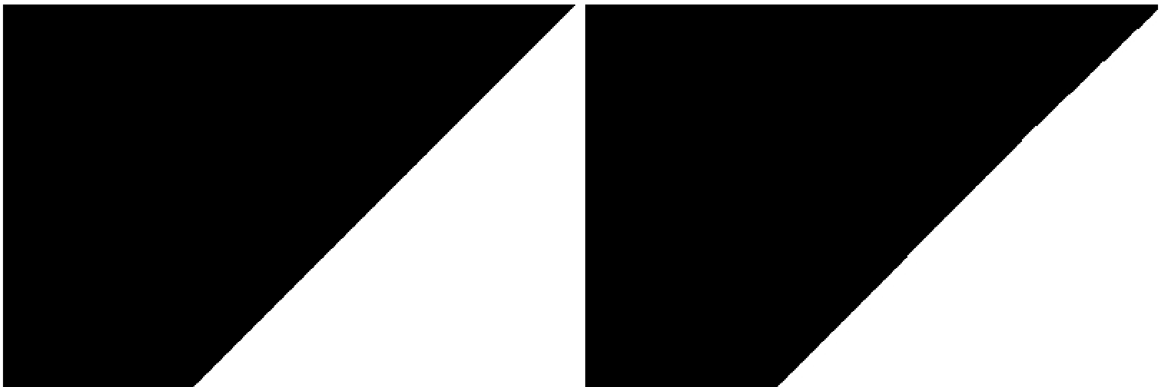


Figure 6.14: Right: objective image. Left: Genetic output image

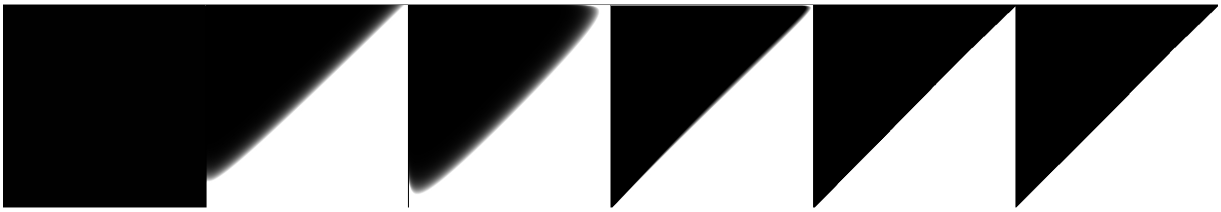


Figure 6.15: Results after 0, 22, 44, 66, 88 and 110 generations respectively

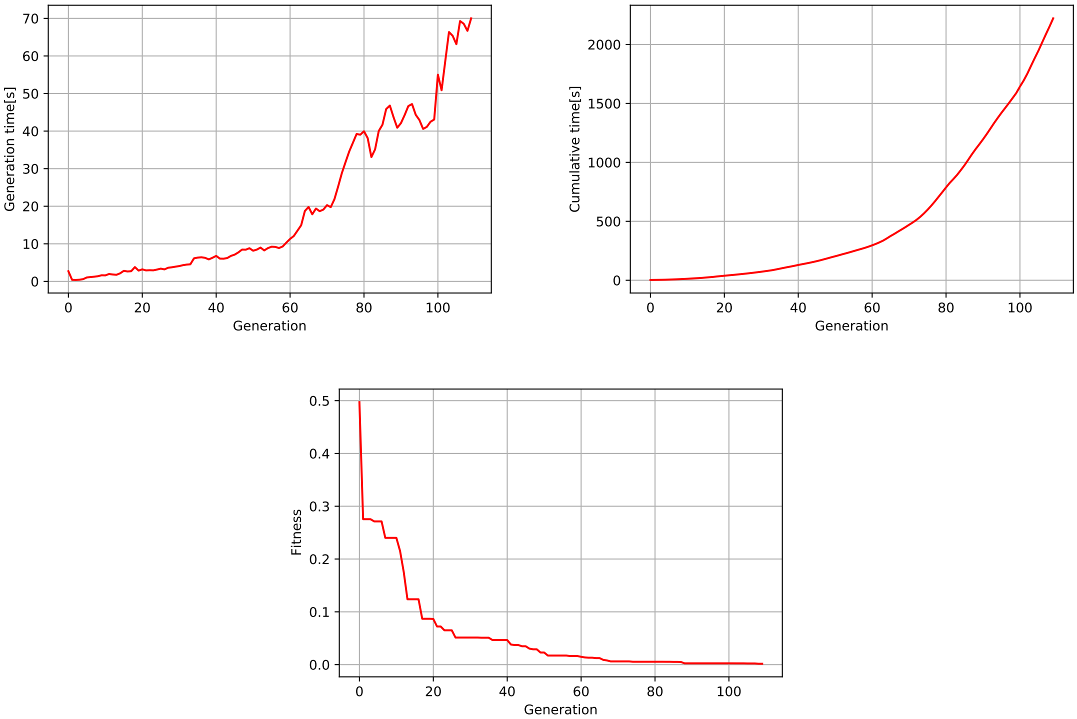


Figure 6.16: Time and convergence statistics for the first image compression experiment. Top left: time per generation; Top right: cumulative time; Bottom: fitness per generation.

In figures 6.14, 6.15 and 6.16 we can see how the objective image is correctly approximated even for less than 110 generations. The total time of the experiment is 2221 seconds.

The size of the original image is 400x400 and its memory size is 6.9 kilobytes. The memory size of the text file containing the genetic solution is 2.3 kilobytes.

Second image compression

In 6.17 and 6.18 we can see the results for bidireccional gradient image. In 6.19 we can see the time and convergence stats for the 110 generations of the experiment.

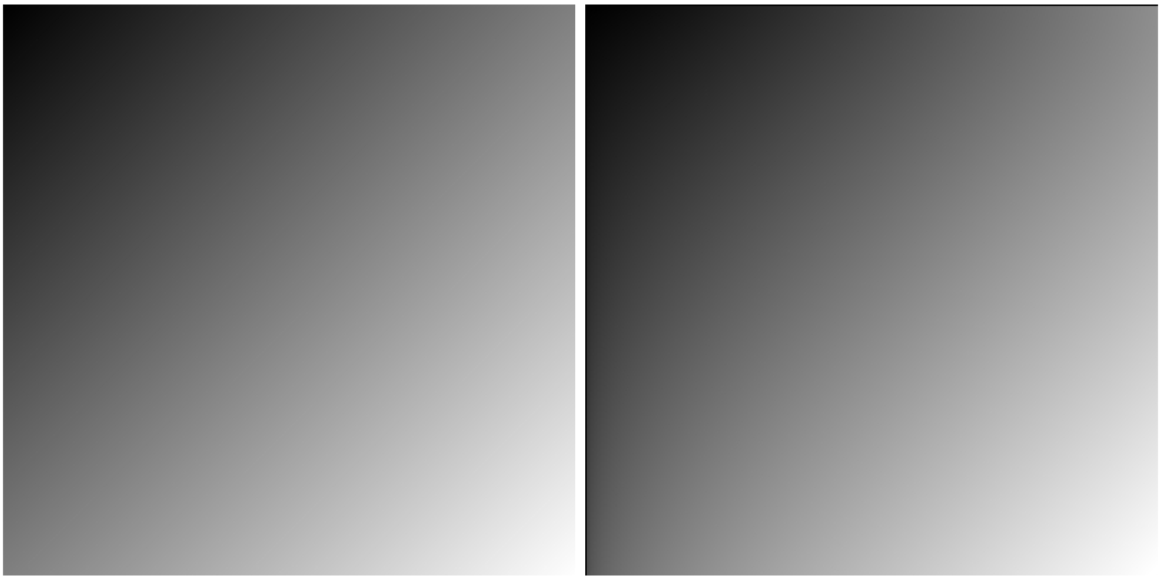


Figure 6.17: Left: objective image. Right: Genetic output image.

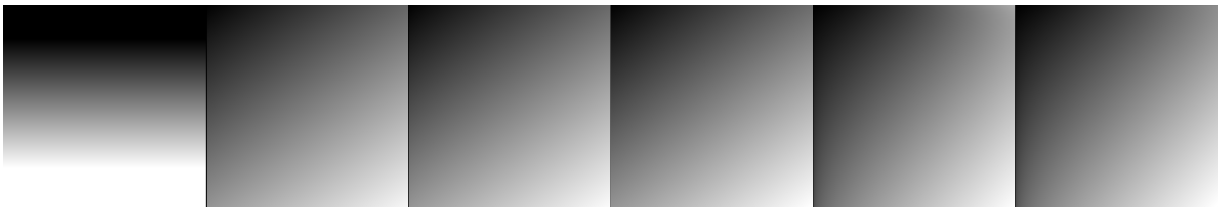


Figure 6.18: Results after 0, 22, 44, 66, 88 and 110 generations respectively

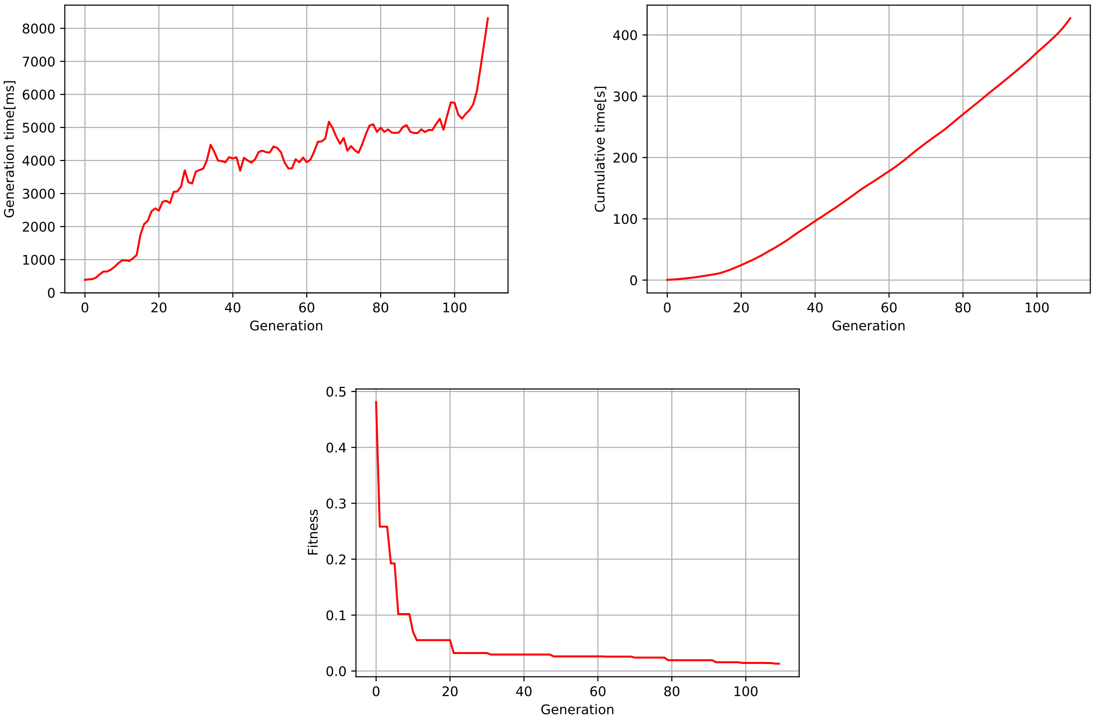


Figure 6.19: Time and convergence statistics for the second image compression experiment. Top left: time per generation; Top right: cumulative time; Bottom: fitness per generation.

In this case the experiment has been run for 110 generations but the objective image is accurately approximated earlier. The total time of the experiment is 425 seconds.

The size of the original image is 400x400 and its memory size is 30.2 kilobytes. The memory size of the text file containing the genetic solution is 451 bytes.

Third image compression

In figures 6.20 and 6.21 we can see the results for a black circle image. In figure 6.22 we can see the time and convergence stats for the 150 generations of the experiment.

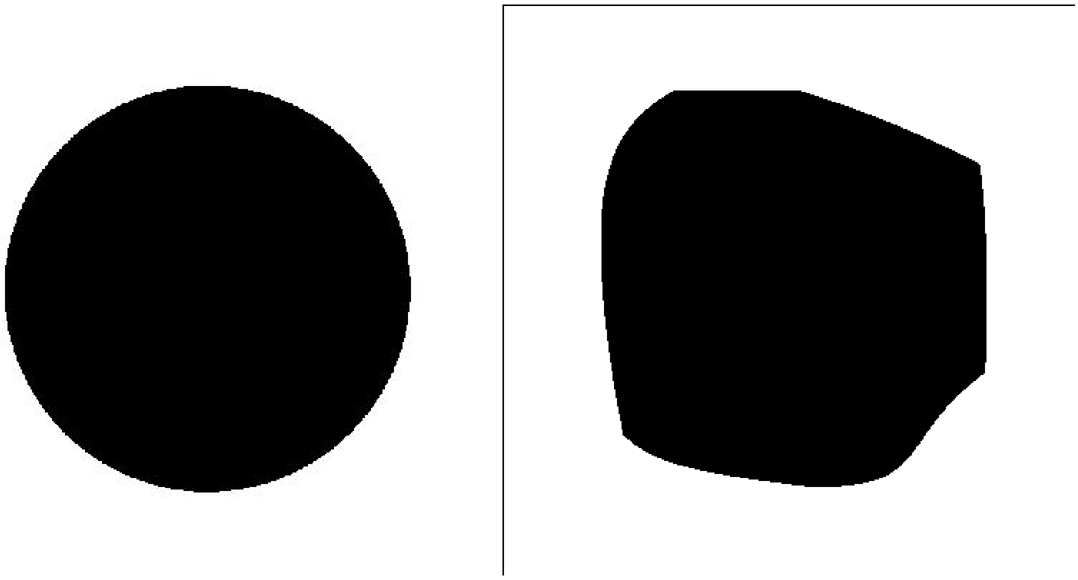


Figure 6.20: Left: objective image. Right: Genetic output image.

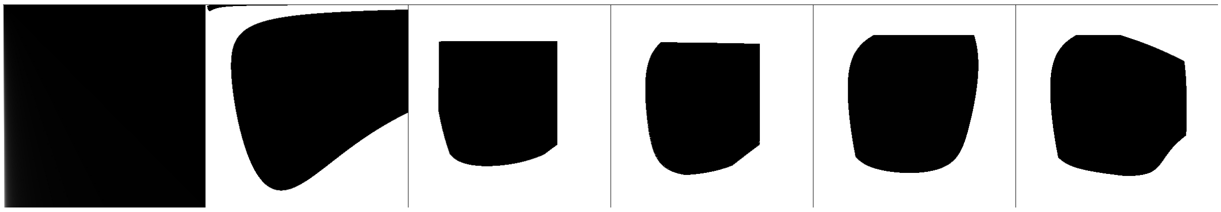


Figure 6.21: Results after 0, 30, 60, 90, 120 and 150 generations respectively

The results for this specific experiment show how the genetic solution is continuously getting better, but, in this case, the approximation is not completely accurate for the 150 generations. However, the result would be better for a longer experiment with more generations.

The size of the original image is 400x400 and its memory size is 14.7 kilobytes. The memory size of the text file containing the genetic solution is 6.2 kylobytes.

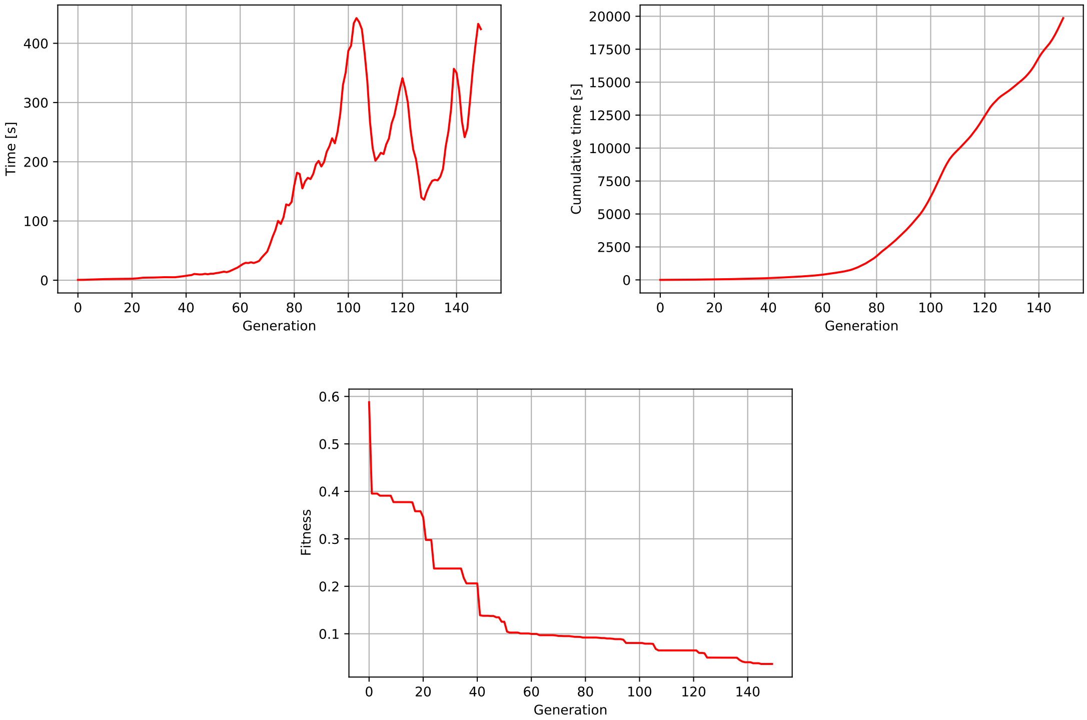


Figure 6.22: Time and convergence statistics for the third image compression experiment. Top left: time per generation; Top right: cumulative time; Bottom: fitness per generation.

Serial versus Parallel Comparison

In this experiment we can see the comparison in computation time between the three different version of the system: sequential, parallelized by pixel evaluation and parallelized by individual loss computation. The experiment have been performed under the same conditions: 100 generations as ending condition for the black circle experiment, same random seed and same machine (machine in 3.2).

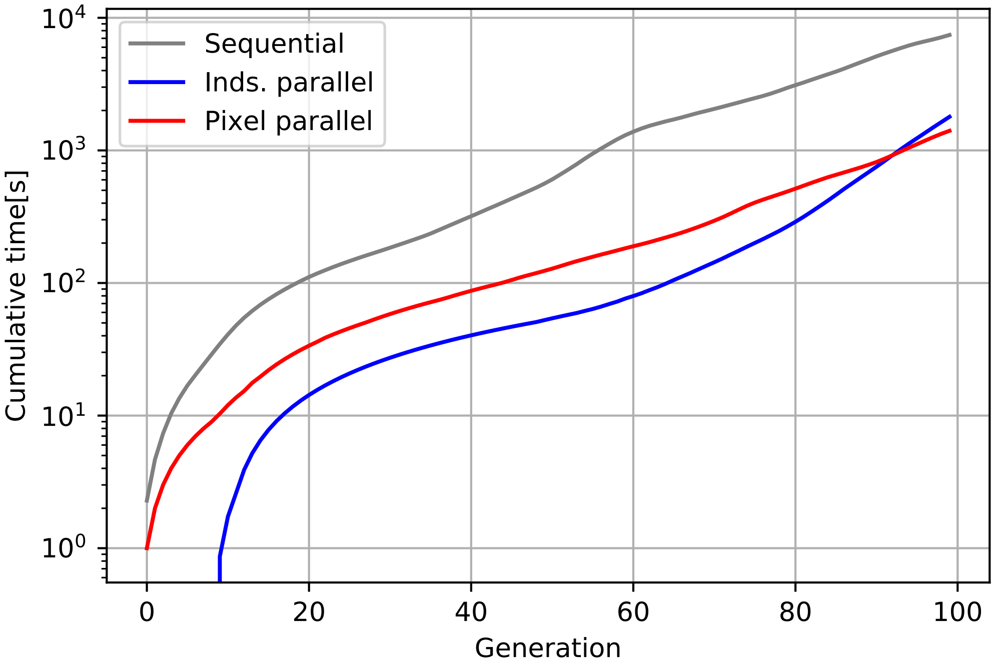


Figure 6.23: Cumulative time per generation in logarithmic scale

In figure 6.23 we have the cumulative time, in logarithmic scale, per generation for each version. The execution time for both parallel version are in the same magnitude order but the sequential execution time is significantly higher, about 6 times slower. As in the previous experiment, this make sense taking into account that we are using an eight-core CPU.

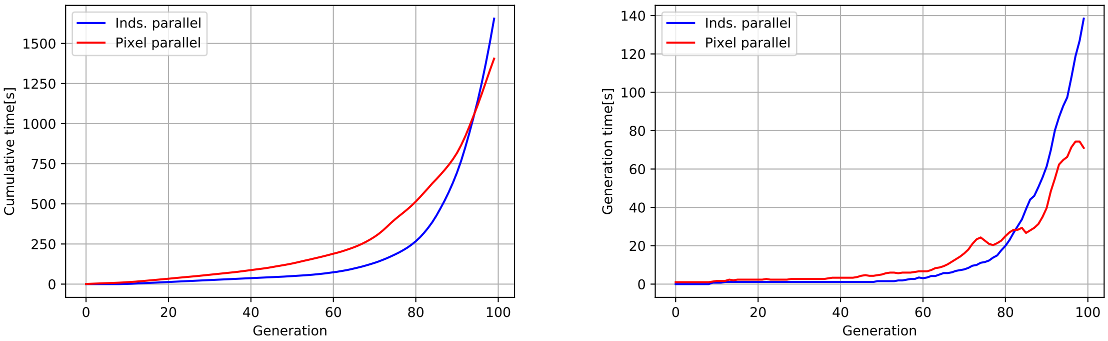


Figure 6.24: Execution time for both parallel versions. Cumulative (left) and time per generation (right)

Figure 6.24 shows the execution time comparison for both parallel version, by pixel, and by individuals. In these results we can see that for the first generations the individual parallelization is faster, but, when the loss decreases and the individual complexity increases the pixel evaluation parallelization is faster. So for longer and more complex applications it would be better to implement the parallelization by evaluating several pixel values of one individual at the same time. This observation suggest than an adaptive parallelism policy able to switch between pixel and individual parallelization could provide some extra benefit.

Chapter 7

Conclusions

In this work, we have designed a genetic programming algorithm for function regression whose main purpose is the integration of analytically non-integrable functions. The algorithm consists of randomly modifying a population of algebraic expressions and selecting the best after each iteration. The chosen modifications, called genetic operators, and the basic functions, which conform the algebraic expressions, will determine the search space exploration and, ultimately, the goodness of the function regression. We have also accelerated the heaviest part of the algorithm, the population evaluation which means around 60% of the computation time, on a multi-core CPU, obtaining a considerable reduction on the overall computation time, about half an order of magnitude. We have also significantly accelerated the random number generation, about 9 times faster, on a GPU without any drawback on the generated numbers randomness as we have demonstrated in Chapter 5. However, this significant acceleration in generation does not translate into an equally significant effect in total execution time because the random number generation does not have the same computational weight as others parts such as evaluation.

After performing some validation experiments, we have applied this genetic programming system to a specific rendering problem, single scattering, and to image compression.

– **Validation experiments:** In these validation experiments, we have confirmed that the system is able to accurately approximate one dimensional functions and their integrals, even when these functions are analytically non-integrable. Also, in these experiments, we see how, for the objective functions which are composed by the basic functions that are contained in the function set, the resultant algebraic expression is very similar to the objective one, so we could say that the search space exploration is satisfactory, which means that the genetic operators are well chosen and implemented.

- **Single scattering:** In this application, we have seen how the resultant image is almost equal to the ground-truth one and much less noisy than the 2-samples Monte Carlo image and even more-samples images. The most important aspect of these results is the fact that, as we have an algebraic expression for the integral, in absence of shadows, we only need two function evaluations to compute the definite integral, which makes much faster the rendering process compared to the rendering based on Monte Carlo integration which is one of the main problems of current renders.

- **Image compression:** In this case we have seen how the system output, a text file which can be also used as system input to generate the image, is smaller, in terms of memory size, than the original image for all the cases, so we can say that the image is effectively compressed. Also, as we are representing the image as a bidimensional function whose independent variables are the pixels coordinates, once we have obtained this image function, we can resize the image without losing quality. The resultant image for the third experiment (section 6.3.4) is not equal to the original for the given experiment length (150 generations), but we have seen how the loss in monotonically decreasing. So, for a long enough experiment the resultant image will eventually be equal to the objective.

In the previous section 6, when comparing the sequential and the parallel versions for the different applications, we can see that we have achieved a significant acceleration in the population evaluation, which is the computationally most expensive part of the algorithm. This speed-up makes it possible to increase the number of generations for the experiments, which leads to a better approximation, without implying a longer execution time. This significant acceleration is achieved with a multi-core CPU, but given the nature of the problem (thousands of individuals and thousands of samples) and its high degree of data parallelism, it could be very interesting to implement the system evaluation on other heterogeneous devices with many more computing units like GPUs of FPGAs.

Limitations and future work: following on from the last thought, probably the most interesting future work would be the implementation of the function evaluation on other accelerators like GPU or FPGA. For example, with this enhancement, it would have been possible to increase the number of generation of the third image compression obtaining a better result. The complexity of this task lies in the fact that the current tree structures are implemented using C++ pointers which makes their evaluation far from trivial in other heterogeneous devices. This implementation will require a modification of the current individual representation.

Bibliography

[1] Philip J Davis and Philip Rabinowitz. *Methods of numerical integration.* Courier Corporation, 2007.

[2] David B Lindell, Julien NP Martel, and Gordon Wetzstein. Autoint: Automatic integration for fast neural volume rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14556–14565, 2021.

[3] Seyedali Mirjalili. Genetic algorithm. In *Evolutionary algorithms and neural networks*, pages 43–55. Springer, 2019.

[4] Manoj Kumar, Dr Husain, Naveen Upreti, Deepti Gupta, et al. Genetic algorithm: Review and application. *Available at SSRN 3529843*, 2010.

[5] Erick Cantú-Paz. A survey of parallel genetic algorithms. *CALCULATEURS PARALLELES*, 10, 1998.

[6] Tomohiro Harada and Enrique Alba. Parallel genetic algorithms: A useful survey. 53(4), aug 2020.

[7] Ajith Abraham, Nadia Nedjah, and Luiza de Macedo Mourelle. Evolutionary computation: from genetic algorithms to genetic programming. In *Genetic Systems Programming*, 2006.

[8] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. *Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL.* Springer Nature, 2021.

[9] Dan Biebighauser. Testing random number generators, 2000.

[10] S. Chandrasekhar. *Radiative Transfer.* Dover Publications, Inc., 1960.

[11] Adolfo Muñoz. Higher order ray marching. In *Computer graphics forum*, volume 33, pages 167–176. Wiley Online Library, 2014.

[12] Vance W Berger and YanYan Zhou. Kolmogorov–smirnov test: Overview. *Wiley statsref: Statistics reference online*, 2014.

[13] Sympy library. `https://www.sympy.org/en/index.html`.

[14] Eva Cerezo, Frederic Perez, Xavier Pueyo, Francisco Serón, and François Sillion. A survey on participating media rendering techniques. *The Visual Computer*, 21, 06 2005.

[15] Wenshi Wu, Beibei Wang, and Ling-Qi Yan. A survey on rendering homogeneous participating media. *Computational Visual Media*, 8(2):177–198, 2022.

[16] Ivan De Falco, Antonio Della Cioppa, and Ernesto Tarantino. Mutation-based genetic algorithm: performance evaluation. *Applied Soft Computing*, 1(4):285–299, 2002.

List of figures

2.1 Evolutionary algorithms common steps. 11

2.2 Mathematical expression represented in a tree srtucture. 11

2.3 Mathematical expression differentiation in form of tree structure. $f(x) = exp(x) + sin(x)$; $f'(x) = exp(x) + cos(x)$ 12

4.1 Individual initialization. For a given depth of 3 the tree nodes and leafs are randomly and recursively generated from top to bottom. 21

4.2 Crossover example. 24

4.3 Mutation 1. A leaf is replaced by a random subtree 25

4.4 Mutation 2. Some genetic content is deleted (the left part of the tree is simplified). 26

4.5 Mutation 3.Tree element substitution (*sin* node replaced by *exp* node). 27

6.1 Function approximation results. In each subfigure we have the correspondent objective function and the system output. 35

6.2 Genetic integration. In each subfigure we have the correspondent objective function and the system output. 37

6.3 Evaluation of several samples in parallel. In this case we are parallelizing the computation of the individual value at each sample point, so each of the 8 CPU cores computes the individual function value, $f(x)$, on a different sample point, x_i. 38

6.4 Evaluation of several individual scores in parallel. In this case we are parallelizing the computation of the 1000 individuals scores, so each of the 8 CPU cores computes the loss function, $l(i)$, on a different individual, i_i. 39

6.5 Execution time comparison between sequential and parallel implementations. 39

6.6 Execution time comparison between both parallel versions. 39

6.7 Illustration of setup for single scattering, including starting and ending positions ($\mathbf{x_0}$ and $\mathbf{x_t}$), light position ($\mathbf{x_l}$) and variables that define the light position with respect to the ray s_l and distance from the light source to the ray d_l. 43
6.8 Time per generation in logarithmic scale 45
6.9 Comparison between images for a scene with a point light source in the corner of the scene and a participating media. 47
6.10 Comparison between images for a scene with a point light source in the middle of the scene and a participating media. 48
6.11 From left to right: "Genetic" image and 2, 4, 64 and 256 Monte Carlo images for the first scene. 49
6.12 From left to right: Renderized images with the genetic integral obtained in generation 0, 30, 60, 90 and 120. 49
6.13 Convergence and time statistics for the single scattering experiment. . . 50
6.14 Right: objective image. Left: Genetic output image 51
6.15 Results after 0, 22, 44, 66, 88 and 110 generations respectively 52
6.16 Time and convergence statistics for the first image compression experiment. Top left: time per generation; Top right: cumulative time; Bottom: fitness per generation. 52
6.17 Left: objective image. Right: Genetic output image. 53
6.18 Results after 0, 22, 44, 66, 88 and 110 generations respectively 53
6.19 Time and convergence statistics for the second image compression experiment. Top left: time per generation; Top right: cumulative time; Bottom: fitness per generation. 53
6.20 Left: objective image. Right: Genetic output image. 54
6.21 Results after 0, 30, 60, 90, 120 and 150 generations respectively 54
6.22 Time and convergence statistics for the third image compression experiment. Top left: time per generation; Top right: cumulative time; Bottom: fitness per generation. 55
6.23 Cumulative time per generation in logarithmic scale 56
6.24 Execution time for both parallel versions. Cumulative (left) and time per generation (right) 56

List of tables

3.1 Compiler version and flags 17

5.1 Different vectors used for random number generation. 31

5.2 Generation time comparison for *int* and *float* data types. Mean time of 5 repetitions of the experiment, 30000 random generations per experiment. 32

5.3 Results of the tests performed over the sequences of numbers generated with both generators. 33

6.1 Function approximation experiments statistics and system output for each function. 36

6.2 Genetic integration experiments statistics and system output for each objective function. 37

6.3 No-integrable functions results 40