Strongly Typed Cartesian Genetic Programming and its Applications

J. A. Forrester

A thesis submitted for the degree of

Doctor of Philosophy

at the

School of Computer Science and Electronic Engineering

University of Essex

February 2026

Abstract

Genetic Programming (GP) and its graph-based variant, Cartesian Genetic Programming (CGP), have proved highly effective machine-learning frameworks that can evolve human-readable programs that can match or exceed hand-crafted solutions across many tasks. Strongly Typed Cartesian Genetic Programming (ST–CGP) extends CGP by assigning explicit data-types to every input, output and operator, and allowing features to have varying arities. These type constraints prune infeasible regions of the search space while preserving CGP's directed-acyclic-graph representation and single-row genome, leading to faster, semantically correct evolution. Unlike standard CGP, ST–CGP also introduces two forms of crossover—full two-point recombination and "genetic rewiring"—providing a second source of variation that is rarely available in conventional CGP frameworks.

Because operators are typed, ST–CGP can be rapidly retargeted: numeric, boolean and higher-level domain primitives (e.g. OpenCV filters) can coexist in a single run, enabling one framework to span diverse problem domains. This versatility is illustrated in this thesis by three application areas. In computer vision, ST–CGP evolved segmentation, detection and classification pipelines that solved benchmark object-sorting problems and achieved convolutional-neural-network-level accuracy on a 27,000-image malaria-cell dataset with far smaller training sets and CPU-only resources. In agriculture, it classified field parcels into low, high and reference yield zones using laboratory soil measurements with competitive accuracy and markedly low variance relative to traditional models. Finally, it learned predictive models mapping five-minute VOC gas "fingerprints" from an electronic-nose sensor to multiple soil health indicators, delivering laboratory-grade predictions, an application which has now been adopted by UK agronomists in commercial practice.

Collectively, these results demonstrate that the combination of strong typing, an enriched operator palette and novel crossover elevates CGP to a general-purpose, interpretable evolutionary programming system capable of tackling data-rich tasks from medical imaging to environmental sensing within a single unified framework.

Acknowledgements

I would like to acknowledge a number of people who have been instrumental in helping me over the last several years. Life has a way of throwing curveballs relentlessly, which in my case has resulted in delay after delay to the completion of this thesis. Nevertheless, these people stood by me throughout it all, and for that, I cannot adequately express my thanks in words.

I am especially grateful to my wife (and proofreader!) Anya, who has been there to support me throughout. Whether I've needed a shoulder to cry on, someone to vent at, someone to motivate me to get the last part of a section or chapter written, Anya has been there. Thank you so much.

My mother, Andrea, and sister, Imogen, have also supported me immeasurably. Without their motivational messages and support, I quite possibly wouldn't have even started this PhD!

Dr Adrian Clark, whose patience is matched only by his ability to spot split infinitives, has supervised me since my undergraduate degree. His guidance has allowed me to develop my knowledge and take my research in my own direction while making sure I still stayed on the right path. I would also like to thank Dr Renato Amorim, who stepped in towards the end of my studies as my supervisor following Adrian's retirement.

I dedicate this thesis to Aurora and Xander, my daughter and son. You are my whole world, and I love you so much.

List of Publications

Forrester, J. A., and Clark, A. F. (2025), Evolving Effective Solutions Using Strongly-Typed Cartesian Genetic Programming, *Currently under review*

Contents

1 Introduction **1**

1.1 Motivation and Background 1

1.2 Strongly Typed Cartesian Genetic Programming 2

1.3 Applications 2

1.4 Thesis Outline 3

2 Literature Review **5**

2.1 Computer Vision 6

2.1.1 Shape 7

2.1.2 Colour 7

2.1.3 Texture 8

2.2 Genetic Programming & Program Synthesis 8

2.2.1 Tree-based GP 8

2.2.2 Graph-based Genetic Programming 11

2.2.3 LLMs: An alternative program synthesis methodology 13

2.3 Machine Learning approaches to Computer Vision 15

2.3.1 Genetic Programming Approaches 15

2.3.2 Deep Learning Approaches 17

2.3.3 Summary 18

3 Strongly Typed Cartesian Genetic Programming **20**

3.1 Architecture 21

3.1.1 Types 22
3.1.2 Functions 24
3.1.3 Genotype & Phenotype Representation 27
3.2 Evolutionary Mechanisms 33
3.2.1 Mutation 36
3.2.2 Evolution Strategy 43
3.2.3 Hyperparameters 45
3.3 Advanced Optimisation Techniques 46
3.3.1 Crossover 47
3.3.2 Multi-objective Optimisation 50
3.3.3 Subgrids 52
3.3.4 Efficiency Improvements 54
3.3.5 The Reference Implementation 55
3.4 Testing ST–CGP 59
3.4.1 Results 65

4 Application: Computer Vision 68

4.1 Adapting ST–CGP for Computer Vision 69
4.1.1 Making ST–CGP understand images 71
4.1.2 Painter 87
4.1.3 Proof-of-concept toy problem 87
4.2 A Real-world Experiment: Detecting Malaria 95
4.2.1 Dataset 95
4.2.2 Methodology 96
4.3 Results 98
4.4 Discussion 101
4.5 Conclusion 103

5 Soil Health Analysis: Yield 105

5.1 Background and Context 106

5.1.1 The Importance of Soil Health 106
5.1.2 Traditional Soil Health Measurement Techniques 108
5.2 Experimental setup 111
5.2.1 Yield as a Proxy for Soil Health 112
5.2.2 Data Collection 113
5.2.3 Soil Health Indicators 116
5.2.4 Initial Analysis 119
5.2.5 Pre-processing 122
5.2.6 Tests 124
5.3 Results and Discussion 126
5.4 Following Work 129

6 Soil Health Analysis: Time Series Sensor Data 130
6.1 The PES Soil Test 131
6.1.1 Anatomy of a PES Test 131
6.1.2 Data characteristics 131
6.2 Preprocessing 133
6.2.1 Data cleaning 135
6.2.2 Smoothing and noise removal 137
6.3 Final experiment 157
6.4 Results 158

7 Conclusion 164

List of Figures

3.1 An example of a program produced by ST–CGP. Whilst the program is simple, note the lack of state, and its functional nature. 26

3.2 A polymorphic implementation function with several node functions, as used in the reference implementation written in C#. 28

3.3 The process of Genotype-Phenotype conversion in ST–CGP. 34

3.4 Code generated from a tree. 35

3.5 Illustration of Full Crossover applied to two individuals. 49

4.1 A screenshot of the painter interface. 87

4.2 Sample images from both the multi-model and whole-image pipelines. . . 90

4.3 Segmentation masks produced by the segmentation model in the multi-model pipeline. 92

4.4 Segmentation programs generated by the segmentation stage of the multi-model pipeline. 93

4.5 Bounding boxes detected using the segmentation mask, applied to the actual image. 94

4.6 Objects classified by colour vs objects classified by shape. 95

4.7 Examples of uninfected and infected cells. 96

4.8 An example cell classifier generated by ST–CGP. 102

5.1 The data collection sheet used by agronomists during sample collection. . 115

5.2 A cluster chart showing Yield vs Microbial Biomass. 120

5.3 A cluster chart showing Yield vs Extractable Nitrate. 120
5.4 A cluster chart showing Yield vs Geosmin. 121

6.1 An example of a resistance curve with some areas of concern highlighted. 133
6.2 The derivative of the resistance curve, before data cleaning. 136
6.3 SMA. 139
6.4 EMA. 140
6.5 WMA. 141
6.6 Savitzky-Golay. 145
6.7 Frequency spectrum for the resistance data (without detrending.) 148
6.8 Cutoff frequency parabolas for various smoothing factor n. 151
6.9 Inverse FFT (without detrending.) 152
6.10 Detrended data. 153
6.11 Frequency spectrum for the resistance data (with detrending) 154
6.12 Inverse FFT (with detrending.) 155
6.13 Curves smoothed using the low-pass parabolic Fast Fourier Transform filter.156

List of Tables

3.2 The hyperparameters exposed by the reference implementation of ST–CGP. 58
3.3 The default hyperparameters chosen empirically through minimal ablation testing. 59
3.4 Genetic parameters used in tests. 61
3.5 The operators supplied for every even parity test. 63
3.6 Additional operators supplied for multi–type even parity tests. 64
3.7 Results from single type configuration. 65
3.8 Results from multiple type configuration. 66
3.9 Results from previous studies. 67

4.1 The image operators provided to ST–CGP for all computer vision problems. 75
4.2 The default hyperparameters. 91
4.3 Per-config performance over 100 independent runs (mean $\pm$ SD). 99
4.4 All-pairs MCC comparisons over 100 independent runs per config. Δ denotes the difference in mean MCC (B$-$A) across runs. 95% CIs for Δ are computed by bootstrap resampling over runs. Two-sided Welch t-tests are used for p-values, with Holm correction across all 6 pairwise comparisons. 'Significant?" indicates Holm-adjusted $p < 0.05$. 99

5.1 Correlation coefficient of each variable wrt Yield. 122
5.2 The hyperparameter search space for each machine learning algorithm. . . 127
5.3 Results of the machine learning techniques tested. 128

6.1 Performance of ST–CGP, Random Forest, and Neural Networks on various soil health indicators. Results are the average of k-fold cross validation, with $k = 5$. 160
6.2 Performance of ST–CGP, Random Forest, and Neural Networks on the test set. 162

Chapter 1

Introduction

1.1 Motivation and Background

Automatic program synthesis has long promised to relieve engineers of routine coding, yet widespread adoption has been hindered by two persistent difficulties. First, traditional tree-based Genetic Programming (GP), one of the most promising approaches, relies on variable-length tree representations that are prone to "**bloat**," generating ever-larger programs during training without commensurate gains in fitness. Second, neither canonical GP nor its graph-based variant, Cartesian Genetic Programming (CGP), possesses any notion of a **data type** by default. The absence of type information inflates the search space with syntactically invalid candidates and limits the range of problems that can be tackled in a single evolutionary run. Types have been introduced to GP in the form of Strongly Typed GP, and have yielded significant gains in performance. However, similar approaches have not been applied to CGP. The relevant background and literature is covered in more detail in Chapter 2.

The objectives of this thesis are twofold:

- **First**, design and implement a strongly typed variant of CGP.
- **Second**, determine whether the addition of strong typing to CGP yields benefits in the evolutionary process or not, through testing on a variety of problem types.

Strongly Typed Cartesian Genetic Programming (ST–CGP) addresses both issues by marrying CGP's fixed-length, acyclic genotype with a compile-time type system comparable to that found in modern statically typed languages. Each node advertises the types of its inputs and outputs; evolutionary operators are constrained so that every mutated individual is guaranteed to be type-safe before execution. In doing so, ST–CGP reduces wasted evaluations, enables heterogeneous data flows and, crucially, preserves CGP's advantageous neutrality—*up to 95% of nodes may remain dormant*, providing a rich reservoir of redundancy from which useful innovation can emerge.

1.2 Strongly Typed Cartesian Genetic Programming

Chapter 3 formalises the ST–CGP architecture. A library of polymorphic node functions supports arbitrary arity, while genotype–phenotype conversion yields **human-readable** functional programs. The evolutionary engine combines an "until-active" mutation scheme with optional crossover, multi-population migration and NSGA-II multi-objective optimisation. Default hyper-parameters, identified through extensive toy-problem studies, result in reliable performance across domains. Taken together, these design choices produce a machine learning algorithm that is expressive, computationally tractable and—importantly—capable of generating programs whose logic can be inspected, tested and trusted by domain experts.

1.3 Applications

To demonstrate generality, the thesis applies ST–CGP to three very different real-world settings:

- **Computer vision**: After validating the image operator library on synthetic shape, colour and texture tasks, ST–CGP was challenged with classifying 27,558 Giemsa-stained cell images as malaria-infected or healthy. With as few as ten training examples per class, the evolved classifiers achieved a mean accuracy of 0.966, surpassing the best convolutional-neural-network baseline reported for the same dataset and doing so with far smaller data and compute budgets. The resulting programs remain interpretable: every pixel transform, channel selection and threshold is explicit, enabling laboratory scientists to verify that the system is responding to biologically meaningful features rather than spurious artefacts.

- **Soil-health assessment via yield labels**: Working with agronomists, 405 samples from 45 UK arable fields were labelled low-, high- or reference-yield and paired with 15 chemical, physical and biological indicators. Against a suite of conventional learners, ST–CGP placed second only to a tuned random forest, delivering 62.5% accuracy and a Matthews correlation of 0.436 while exhibiting the smallest run-to-run variance—evidence that the typed search converges to stable, compact explanations of complex agro-ecological interactions.

- **Time series gas-sensor data**: The PES electronic-nose test records multi-channel resistance and capacitance curves as soil VOCs are released during a five-minute heat cycle. ST–CGP evolved feature-extracting programs that map these "VOC fingerprints" directly to the quantitative soil-health indicators established in the yield study. The resulting models match—or exceed—laboratory accuracy while reducing analysis time from days to minutes, enabling the technology's commercial deployment in routine farm management across the UK.

1.4 Thesis Outline

The remainder of the thesis is organised as follows. Chapter 2 reviews prior work in computer vision and GP, motivating the need for a typed evolutionary framework. Chapter 3 introduces ST–CGP in detail. Chapters 4 to 6 present the three application studies summarised above, each concluding with a critical discussion of strengths and limitations. Chapter 7 summarises the findings of this research; through these investigations the thesis establishes ST–CGP as a practical and versatile tool for evolvable program synthesis in domains where type diversity, interpretability and data efficiency are paramount.

Chapter 2

Literature Review

The software which forms the main focus of this thesis can appear complex to those unfamiliar with the techniques used. There are two major components: the machine learning component of the system uses a novel form of Cartesian Genetic Programming as its basis, and this operates using Computer Vision principles which range from simple, traditional image processing operators to more recent, complex and esoteric ones. This chapter serves to help the reader understand the fundamental concepts used, by briefly detailing the history and relevant research behind Genetic Programming and Computer Vision. For many readers it will serve as a gentle reminder, but should a reader be completely unfamiliar with these concepts and need more information, or simply wish to study them in more detail, direction is given to the essential pieces of literature.

This chapter is structured as follows. Section 2.1 gives a very brief overview of Computer Vision and the concepts pertinent to this research. Section 2.2 briefly details the history and development of both tree-based and Cartesian Genetic Programming, with a very brief comparison to cutting-edge LLM-based program synthesis. Finally, Section 2.3 summarises the literature which combines the use of Genetic Programming and Computer Vision.

2.1 Computer Vision

Computer Vision is a vast subject with many aspects, and it is beyond the scope of this section to attempt to explain it all. Instead, this section will give an overview of those aspects which are pertinent to this research.

The purpose of Computer Vision is to allow computers to interpret and in some sense understand images [1], similarly to how Machine Learning aims to give computers the ability to learn. Ordinarily, the computer vision pipeline will roughly consist of the following steps: pre-processing, feature extraction, and (usually) segmentation and classification.

Pre-processing traditionally referred to techniques such as denoising, normalisation, and colourspace transformation to name just a few. More recently, with the rise of (deep) convolutional neural networks, image augmentation and similar pre-processing steps have become more common. Image augmentation is a procedure whereby input images are duplicated and modified to provide more training data [2]. This has proven to be very effective at improving the training results of Convolutional Neural Networks [3], to the surprise of some researchers.

Feature extraction is perhaps the most challenging aspect of Computer Vision: to solve a particular problem it is necessary to extract the right features. It is these features which are used and interpreted by some sort of algorithm, whether handwritten or based on machine learning, to give a useful output. Feature extraction can consist of techniques such as edge and corner detection, gathering statistics such as mean and standard deviation of colour values, texture analysis, and so on [4]. Broadly speaking, extracted features fall into one of three categories: shape, colour, and texture [5, 6]. Some features may qualify as a combination of two of these categories, or indeed all three. Human vision has been demonstrated to work in a very similar way; the brain contains distinct areas intended to process and interpret each of these three characteristics [7]. These three categories are explored in a little more depth in the following subsections.

2.1.1 Shape

Shape is perhaps the most fundamental category of image feature: it describes what an object *actually looks like*. Perhaps the most well-known set of shape descriptors are Hu's moment invariants [8], which describe the shape of an object irrespective of the scale, rotation, or translation of the object. Another commonly used technique is the Scale-Invariant Feature Transform (SIFT) [9], which captures the gradient structure in an image patch around an interest point in such a way that is invariant to scale and rotation. More recently, contour-based approaches have gained popularity as a way of describing shape on the basis of detected edges in images [10, 11].

Shape features are used in a wide variety of applications from agricultural applications such as plant identification [12] and leaf classification [13], to medical applications including cancer biomarker detection [14, 15] and classification of X-Ray, MRI and CT scans[11], to image retrieval and target tracking systems [16].

2.1.2 Colour

As with shape features, colour-based features are used across a wide range of industries and disciplines. Colour is one of the foremost methods by which content-based image retrieval (CBIR) systems work [17]. Colour plays a big role in many agricultural systems: for example, fruit ripeness detection[18, 19, 20], and even yield prediction [21]. In retail, colour can be used to classify product images [22]. Colour is even used in self-driving cars, to detect traffic light status [23] and to detect traffic signs [24].

Colour is most often analysed using histograms, which allow the colour distribution of an image or region to be analysed [25]. An extension of this idea is the colour-coherence vector (CCV) which augments the histogram with a measure of the coherence of each colour: a colour is coherent if there are large regions of the image occupied by that colour or similar colours, while an incoherent colour indicates that the pixels corresponding to that colour are scattered in small disconnected clusters [26]. This allows the spatial relationship of colours to be analysed.

2.1.3 Texture

Texture describes the visual pattern or spatial arrangement of colours or intensity variations in an image or image region [27]. Perhaps the most widely used texture features are Haralick's Co-occurrence Features, also known as grey-level co-occurrence matrix (GLCM) features. This technique quantifies texture by analysing the spatial relationships between different grey-level intensities at a specified direction and orientation [28, 29]. This allows statistical measures to be computed which characterise the texture based on how often specific grey-level combinations occur.

Gabor filters are another common technique, operating in a similar way to GLCM features; they are a linear filter (adapted from a 1-dimensional filter first proposed in [30]) which is applied to an image at multiple scales and orientations [31]. This allows for texture and edge information to be extracted, resulting in responses that effectively capture local frequency and orientation information. This mimics the early stages of the human vision processing system [32].

Texture has been used in many applications. Medical applications include the detection of breast cancer tissue in stained cells [33] and pneumonia in x-ray images [34]. Agriculture makes another appearance: aerial images of crops can be classified using texture [35], as well as the classification of leaf disease on crops [36]. There is even some evidence that visual texture of lentils and rice grains can predict the physical texture after processing and cooking [37].

2.2 Genetic Programming & Program Synthesis

2.2.1 Tree-based GP

Genetic Programming (GP) is a type of Evolutionary Algorithm which operates on (and produces as its output) computer programs. While Turing theorised about the possibility of automatic programming, such a technique only appeared in the literature formally in 1981 [38]. This initial paper introduced the community to a biologically inspired paradigm in which a simulation of Darwinian evolution and natural selection are used to guide the machine learning process. Boolean expressions were represented as trees, and badly performing trees were “killed off” and replaced with mutated versions, or versions produced by “mating” two good rules. It was found that this technique produced better results than standard linear discriminant analysis, and it was concluded that further investigation was necessary.

GP next appeared in the literature half a decade later [39], wherein an attempt was made to produce a language that could represent programs whilst being easily manipulable by genetic operators. Two languages were used, which were adapted from Turing complete languages—“goto” operators were removed to ensure programs would always terminate. The languages were used with genetic operators to evolve programs with two numerical inputs and one numerical output. This paper also used multiple scoring criteria for the “evaluation score” of each program (how well it performed—known as the fitness score in practically all GP research). For instance, long programs were penalised, as were programs which took a long time to execute. This concept of a “multi-objective fitness” score has been the subject of much later research. This research used a population size of 50 programs.

While the previous two papers were the first formal publications based on GP, it became more prominent due to Koza—indeed, Koza is practically synonymous with GP for many researchers. Koza’s first introduced his version of GP in 1989 [40]: five problems were presented, of varying natures, and a GP system written in Lisp was used to attempt to evolve a solution. In contrast to the previous GP research, a comparatively large population size was used—between 300 and 500 individuals per population. Koza followed this research with four books [41, 42, 43, 44] which are considered by many to be the seminal work on the subject. The books present what tends to be thought of as a “typical GP framework” where programs are represented as trees of function and terminal nodes. A typical GP run in such a framework consists of a population of program trees (typically in the range of 300-1000 for a standard problem), with multiple generations, each of which sees a new population be generated through elitism, crossover, and mutation. A selection algorithm is used to determine which “parents” will be used to produce new individuals. The first book also saw the introduction of Automatically Defined Functions (ADFs) [41], where modular, reusable functions are evolved as part of a GP run and used within the generated programs. This modularity allows more powerfully expressive outputs.

In his books, Koza uses the term “human competitiveness” to describe how closely an algorithm or program can compete with the performance of a human for a particular task. Much of the literature focuses on achieving human competitiveness with GP, and in 2008 it was shown that human competitive results are common with GP. Indeed, many pieces of research have resulted in human-competitive solutions—with [45, 46, 47, 48] to name just a few.

The human competitive nature of GP has allowed it to be used as an “invention machine” [49]. This echoes earlier research which used GP to invent solutions to two problems which had been solved by humans in the past [50]. Human competitive solutions have even been used to generate patentable solutions to problems [51].

The majority of GP driven solutions build trees that operate on a single data type. In contrast, many programming languages in use today have many types. Many such languages are strongly typed. This means that variables and functions have well defined type constraints: these can be used to ensure syntactical correctness of programs. This is generally a positive thing for human-driven software development. In 1995, Montana [52] introduced Strongly Typed Genetic Programming (STGP). In STGP, every node, and every terminal, has a type. This allows trees to be constructed in a way that they are guaranteed to be syntactically correct. Montana demonstrated on two problems that enforcing type constraints resulted in more efficient problem solving. Other studies have since used and extended STGP [53, 54, 55].

While GP can produce encouraging results, it is not without its downfalls. “Bloat” is perhaps the most pernicious. Soon after the introduction of GP it was discovered that, left unchecked, program size started to increase with each generation, with no increase in fitness [41, 56, 57]. Indeed, in many cases, the best programs were actually those that were shortest [58].

The exact cause of bloat has been the source of much debate [59]: an early theory [60] proposed that the success of an individual is related to its ability to have offspring which function similarly to the parent—GP therefore evolves larger and larger individuals with similar functionality. This is known as the "replication accuracy" theory. Another theory [61] has suggested that it may be a characteristic inherent to variable length representations such as GP, especially when using a fixed evaluation function, because a long program can have many more possible representations of the same solution than a short program. Other research suggests that bloat serves to protect the non-redundant code, by making it much more likely that genetic operators will hit redundant code. Perhaps the most widely accepted theory is the "crossover bias theory [62]." This states that, because crossover frequently produces single-node or very small programs, and because very small programs are very unlikely to perform well on most tasks, these small programs will routinely be ignored by selection. As such, programs of an above average size have a "selective advantage" over programs of a lower than average size, and over time, this causes the mean program size to increase.

which states that crossover is biased towards selecting longer subtrees and therefore results in larger offspring over time [63].

There have been several tactics for dealing with bloat, from applying "parsimony pressure" [64] (where the fitness measure is decreased based on program length) to setting absolute limits on the depth and breadth of generated programs [65]. However, many of these techniques can negatively affect program performance [66]. Instead, some researchers have turned to alternative ways of representing programs, eliminating trees and tree-based genetic operators.

2.2.2 Graph-based Genetic Programming

Rather than representing GP individuals as trees, it is also possible to use a graph-based program representation. An early example is Poli's Parallel Distributed Genetic Programming (PDGP) [67, 68]. PDGP represents individuals as directed graphs in which nodes correspond to functions and terminals and edges represent the flow of intermediate values; these graphs may be arranged, for example, on a two-dimensional grid of nodes. PDGP enables *non-parameterised reuse* of subgraphs (i.e., reuse without explicit subroutines): a reused subgraph is evaluated once and its resulting value is shared wherever it is referenced, rather than being recomputed at each occurrence. PDGP is able to use a direct representation to produce neural networks, recurrent transition networks, and even finite state automata, and supports standard GP operators such as mutation and crossover, It has been applied to a variety of problems, including natural language recognition [69].

Other graph-based GP representations exist. Cartesian Genetic Programming (CGP) is a form of GP introduced in 1999 [70], which represents individuals as directed acyclic graphs. Each graph consists of a fixed amount of nodes, which in the original research (and indeed most subsequent implementations) are represented as strings of integers. Each node has two "connections" to other inputs. This representation has a number of consequences.

While early implementations of CGP were fully connected, meaning all nodes were "active" and there was no redundancy, representations quickly grew to include redundant nodes. The reader will recall that in tree-based GP, individuals with a high degree of redundancy (i.e. bloated individuals) are often worse than those without [58]. In contrast, CGP has been shown to *benefit* from a high degree of redundancy—with optimum results being produced when ˜95% of nodes are redundant [71].

CGP does not typically make use of a crossover-like genetic operator. The graph based representation means that mutation is the primary means by which individuals are modified between generations [72]. This, combined with the fixed length representation, means that CGP does not suffer from bloat [73]. This matches the suggestion in [61] that bloat may be inherent to variable length representations—CGP has a fixed number of nodes. It should be noted that although the number of nodes is typically fixed, the resulting program length is not (since the connections chosen define the eventual shape of the program).

Despite the benefits of strong typing in GP, there has been little research in the same area for CGP. Harding et al. took a step towards strongly typed CGP by introducing Multi-Type CGP (MT-CGP) in 2012 [74]. In MT-CGP functions can accept one of two types: a real number or a vector of real numbers. The operations performed by each function change depending on the type passed in. The results showed that giving MT-CGP access to more than one data type allowed evolution to quickly find competitive results. In a later experiment, Wilson [46] extended MT-CGP to also include a matrix type to enable CGP to process images without using direct pixel values as the inputs. These papers were a step in the right direction, but do not constitute strong typing in the same sense as STGP.

2.2.3 LLMs: An alternative program synthesis methodology

Large Language Models (LLMs) are a comparatively recent paradigm of machine learning, emerging from the combination of large-scale self-supervised learning and the transformer architecture [75]. Whilst earlier neural language models were already capable of modelling simple syntactic patterns, the transformer enabled substantially improved long-range modelling and, crucially, efficient scaling via parallel training. This scaling trend culminated in so-called "foundation models" that exhibit strong in-context learning behaviour, with perhaps the first widely-known model being GPT-3 [76]. Shortly thereafter, models trained or fine-tuned explicitly on code (and associated natural-language data such as documentation, comment strings and issue discussions) demonstrated that the same next-token prediction objective could be repurposed for program synthesis tasks, including generating executable functions from natural language specifications [77]. Subsequent systems combined large-scale sampling with selection and filtering pipelines to solve harder algorithmic tasks (e.g. competitive programming), indicating that performance is often obtained not from a single deterministic generation but from generating many candidates and selecting those that satisfy constraints [78], not unlike GP workflows.

In synthesis terms, LLMs typically construct programs by producing code tokens autoregressively conditioned on a “prompt”, a natural language description of the program to be produced. This implicitly makes use of the statistical regularities of human-written software. This process differs fundamentally from GP, which treats program synthesis as an explicit search/optimisation process guided by a user-defined fitness function. The LLM approach can be markedly more sample-efficient in human time: once trained, generation is rapid, can incorporate domain conventions, and can be guided by natural-language specifications directly. However, it offers weaker guarantees: functional correctness is not guaranteed, outputs may be brittle under small prompt variations, and “hallucination,” whereby LLMs output data that is false, non-existent, or otherwise incorrect, is an issue which plagues LLMs to this day [79]. This necessitates external verification (unit tests, static analysis, execution-based checking, or even manual review by a human) and often extensive post-processing. By contrast, GP typically provides a clearer coupling between the objective and the synthesis procedure: correctness and performance can be embedded directly in fitness evaluation. Another difference is that GP can discover unconventional solutions not anchored to common human coding patterns or knowledge. In contrast, LLMs are largely constrained to outputting patterns that were part of the training data; it is uncommon that novel solutions outside of the training data are produced [80]. The corresponding disadvantages of GP are well known: evolutionary search can be computationally expensive, sensitive to representation and operator/terminal choices, and may require careful fitness shaping to avoid deceptive landscapes. In practice, these paradigms are increasingly complementary: LLMs provide strong priors and a convenient interface, whereas GP/CGP provide an optimisation and verification framework with explicit behavioural grounding, and significantly faster candidate generation, allowing orders of magnitude more solutions to be attempted in a given time period.

2.3 Machine Learning approaches to Computer Vision

Many varieties of machine learning have been applied to computer vision problems. Genetic programming is certainly a worthwhile paradigm, and while neural networks dominate the literature of today where computer vision research is concerned, much research has nevertheless been done on the subject of applying GP to computer vision.

2.3.1 Genetic Programming Approaches

In general, GP applications for Computer Vision fall into three categories: classification, enhancement (or processing/modifying images in some way), and feature extraction. Some applications may combine two or even all three of these categories. Some of the first GP research to focus on Computer Vision was performed by Tackett in 1993 [58], shortly after GP was popularised by Koza's first book. This research used GP to classify features which had been extracted from images, rather than operating on the images directly. Soon after, Andre used GP to evolve feature detectors [81], and Koza evolved detectors that were able to distinguish between the letters "I" and "L" [42]. Both pieces of research evolved 3×3 filters which were applied to an image. In general, this approach seems to be the most common way of applying GP to Computer Vision: filters are easy to evolve, and use numerical types, so can be produced as part of a single-type tree. Breunig attempted to evolve pattern recognition algorithms, with an emphasis on ensuring evolved algorithms were location independent [82]. Although relatively few runs produced successful results (roughly 30%), GP was able to evolve location independent algorithms.

The GP Computer Vision research mentioned thus far has focused on classification (with the exception of Tackett [58] who also used GP for feature discovery). Feature extraction is equally well represented[1]: in one of the earliest examples, Ebner attempted to evolve an interest operator identical to that of Moravec, and found that a close approx-

[1]Not just for Computer Vision: GP has been used for feature extraction for non-image domains by several researchers [83, 84].

imation could be generated [85]. Shao et al. more recently used GP to extract features from images [86]. The multiobject GP feature extractor was used to classify features in a number of known datasets. The results matched or outperformed many state-of-the-art hand-designed features, as well as features generated by feature learning algorithms.

Image enhancement is a step which is used in nearly all hand-designed computer vision algorithms [87]. One of the earliest examples of applying GP to enhancement is the 1996 work by Poli et al. in which the problem was approached from the perspective of image filtering [88]. GP was used to segment the human brain from MRI images by evolving low-level filters and applying them to images. The experiment demonstrated significantly better performance than the state-of-the-art neural networks at the time. It was concluded that GP was promising and deserving of more research in the Computer Vision domain. More recently, Zhang developed a pre-processing stage using GP which used feature extraction to produce a maximally separating adaptive thresholding algorithm [89]. This was used to segment images, as well as being applied as a proof-of-concept to edge detection, where the results outperformed the Canny edge detection algorithm based on minimal Bayes risk.

The majority of GP Computer Vision research has used tree-based GP as the representation. However, CGP has been demonstrated to work equally well. Montes and Wyatt [90] used CGP to find centroids of objects in images. Similar work was done by Paris et al. [91] in which image filters were automatically learned by a hardware CGP system. The evolved filters used sequences of morphological and logical operators. Sekanina et al. [72, ch. 6] used CGP to evolve a number of low-level filters for FPGAs, as well as evolving more complex and advanced image operators such as dilation/erosion using basic mathematical functions such as sin, square root etc. An image classification task was also attempted: CGP was used to define transformations on medical images which significantly improved classification accuracy when using predefined image features.

Many of the CGP applications to Computer Vision used direct pixel values and evolved low level filters. Wilson et al. [46] used an extended version of MT-CGP [74] to allow matrix input types. This was the first time that CGP had been used to process a whole-image input. It was demonstrated that this input format allowed expressive programs to be evolved. The programs were used for Atari game playing and were shown to be competitive with state-of-the-art Atari benchmark controllers, while requiring less processing time.

2.3.2 Deep Learning Approaches

It would be remiss not to briefly discuss deep learning in computer vision, given its near-ubiquity in modern CV pipelines. A foundational and highly influential example of neural networks applied to image recognition is LeCun et al.'s 1998 work on Convolutional Neural Networks (CNNs) for document recognition [92]. However, it was not until 2012—with Krizhevsky et al.'s AlexNet result on ImageNet [93]—that the modern era of **deep** learning in computer vision began in earnest. This seminal work demonstrated that very large CNNs, trained using GPU acceleration on datasets containing *millions* of labelled images, could dramatically improve large-scale visual recognition performance.

This was quickly followed by VGGNet in 2014 [94] and ResNet in 2015 [95], both of which became widely used as "backbone" classification models (i.e., feature extractors) underpinning many later architectures. Segmentation was also transformed by deep learning; for example, U-Net [96] was an early and highly influential encoder–decoder model introduced for biomedical segmentation, and it (and its variants) remains widely used [97, 98, 99].

A common characteristic of these deep-learning approaches is their data and compute requirements. Deep neural networks can achieve excellent accuracy, but typically require substantial labelled datasets and expensive training (often multi-GPU). They are also commonly regarded as "black boxes" [100], making their internal decision processes difficult to interpret, despite extensive work on interpretability and explainability methods [101, 102].

These developments provide a useful point of comparison for the methods developed in this thesis. Rather than competing with deep networks on maximum benchmark accuracy, the aim here is to explore *program synthesis*: evolving explicit, auditable image-processing pipelines (and pipelines for other problem domains) whose structure and operators can be inspected and constrained without specialised inference runtimes. This is particularly relevant in settings where labelled data are limited, where objectives include non-differentiable operators or hard constraints (e.g., morphology, conditional logic, latency/throughput budgets), or where model transparency is a requirement. The strongly-typed CGP variants introduced in this thesis further constrain the search to semantically valid programs when operating over multiple data modalities (e.g., scalars, vectors, and whole-image/matrix inputs), aiming to improve search efficiency and produce pipelines that are both interpretable and practically deployable.

2.3.3 Summary

A primary advantage of Genetic Programming (GP) for computer vision is *interpretability*: the learned solution is usually an explicit, human-readable program composed of known operators, rather than an opaque set of learned parameters. This makes it feasible to inspect, debug, and reason about failure modes (for example, sensitivity to illumination, noise, or specific shapes or structures) [103]. In traditional, handcrafted feature-based pipelines, interpretability exists at the level of manually chosen components, but performance relies on hand-crafted design choices that may not transfer well across datasets. GP retains transparency while also reducing the need for manual redesign by *automatically constructing end-to-end pipelines* tailored to the task.

In comparison with deep-learning approaches, GP (and CGP) is often attractive in situations where data, compute, or deployment constraints dominate. As mentioned, deep neural networks have a large data requirement, and produce models that are difficult to audit or compress without additional engineering. GP can be comparatively *sample-efficient* when strong primitives are available [104], and can optimise directly for non-differentiable objectives (discrete morphology or conditional logic, for instance) without surrogate losses. CGP strengthens these advantages through its graph-based representation; its ability to reuse intermediate computations via a directed acyclic graph is well-suited to complex CV pipelines.

Chapter 3

Strongly Typed Cartesian Genetic Programming

The words of the well-known saying “Everything should be as simple as possible, but not simpler,” ring particularly true when it comes to programming, where the aim is often to build systems that are at once robust, efficient, and yet not excessively complicated. This “Occam’s razor-driven approach” is a delicate balance to strike, and the introduction of a strong typing system to Cartesian Genetic Programming embodies this concept, providing a method of simplifying the system in ways that are inherently robust.

Let us consider a metaphor in the form of the cultivation process of a lemon tree. A gardener chooses a very specific set of care procedures, nutrients, and pruning techniques based on the species of tree. This approach allows the tree to produce the best possible lemons and minimises any issues that may arise. In a similar vein, if traditional CGP can be compared to a “one-size-fits-all” cultivation approach due to its type-agnostic methodology, then ST–CGP can be likened to a specific, tailored gardening plan, where the unique “type” of each part of the system is taken into account, resulting in more robust plants and better yields.

The key novelty of ST–CGP, which this chapter seeks to explore, is the introduction of a robust typing system akin to those employed by high-level programming languages.

Unlike conventional CGP, which interprets all data uniformly due to its lack of type awareness, ST–CGP uses *strong typing* to delineate data and functions. This introduces a structure that constrains and guides the generation of programs, reducing the search space by discarding programs that do not comply with these constraints. The analogy with the well-cultivated lemon tree comes full circle here: just as the gardener ensures productive growth by guiding the tree's development, ST–CGP directs program generation towards greater efficiency and correctness.

In complement to the strong typing system, ST–CGP also integrates additional techniques that typically find less utilisation in the CGP approach. One instance is the incorporation of crossover, which is seldom employed in the traditional CGP methodology, but finds its place in the ST–CGP framework. The latter segments of this chapter further explore these supplemental methodologies, explaining their purpose and function within the ST–CGP system.

3.1 Architecture

Genetic Programming is an inherently abstract domain, which can make for a challenging implementation process. The process of designing a Genetic Programming system entails a multitude of decisions regarding how each constituent part should be implemented. This characteristic is not exclusive to tree-based Genetic Programming, but extends to Cartesian Genetic Programming and, in turn, to the Strongly-Typed Cartesian Genetic Programming proposed in this thesis.

In this section, we explore the architecture of the Strongly-Typed Cartesian Genetic Programming system in depth, presenting a detailed exposition of each element and explaining the reasoning underpinning each design choice. The ultimate objective of this exploration is to describe a "blueprint" that may be implemented by any reader possessing the requisite technical expertise, thus making the application of Strongly-Typed Cartesian Genetic Programming accessible beyond the bounds of this work. The section concludes by highlighting the differences between this architecture and that of the original, untyped Cartesian Genetic Programming. Throughout this section notes will be made about the "reference implementation" of ST–CGP. This refers to the implementation created by the author during this research.

3.1.1 Types

This thesis focuses on the application of strong typing to Cartesian Genetic Programming, on the basis that applying strong typing to tree-based Genetic Programming resulted in many benefits. Before explaining how ST–CGP uses types, it is important to clarify the concept of strong typing and to understand the nature of types.

Types & Strong Typing

In programming, "types" are a way of categorising values or expressions based on their characteristics, behaviour, and interactions. Put simply, they allow for the definition of the set of possible values and operations that can be applied to an expression or value. Types serve as a fundamental concept in most programming languages and help to ensure correctness and reliability in software engineering. They enable the compiler (or interpreter) to enforce constraints on how values can be combined and manipulated, and can therefore help to detect errors early in the development process by ensuring syntactical correctness. The appropriate use of types contributes to robust and maintainable software, by facilitating easily readable, reusable code.

The author shall focus mainly on object-oriented languages in this thesis, as that is what was used for the reference implementation of ST–CGP. Generally, object-oriented programming languages break types into three categories:

- primitive types—the simplest types, from which other types are constructed—such as integers, floating point numbers, and booleans.
- composite types—more complex types, formed using the primitive types of the language—such as arrays, classes, and structures.

- abstract types—used specifically in object oriented languages, and cannot be instantiated directly, but rather serve as an aid to writing typed programs—such as interfaces and abstract classes.

The definition and implementation of types varies widely across programming languages. Although there is some argument about the exact classification of languages, a commonly accepted method is to designate typed languages as either strongly or weakly typed, and either statically or dynamically typed. In this classification, strong typing refers to type systems in which strict type checking is performed. Typically, explicit type declarations are required, and type conversion is not performed implicitly. This ensures that operations are only performed on values of compatible types, thereby reducing the likelihood of unexpected behaviour and exceptions. Weak typing, on the other hand, refers to type systems in which type conversions typically occur implicitly, and type checking is less strict. This can lead to unexpected behaviour if the types are not handled carefully. Static typing refers to languages which perform type checking at compile time—before the program ever executes. Dynamic typing refers to languages which perform type checking at runtime, as the program executes. Generally speaking, statically typed languages are more likely to be strongly typed, and dynamically typed languages are more likely to be weakly typed, but there are languages which do not fit this trend: dynamic, strongly typed languages are fairly common, for example.

The selection between strong and weak typing, as well as static and dynamic typing, relies on the specific objectives of the programming language's design, the characteristics of the problems expected to be tackled using the language, and various other considerations. Each approach encompasses advantages and disadvantages, including factors such as development simplicity and speed, code comprehensibility, ease of maintenance, execution efficiency, and type safety.

The name "Strongly Typed Cartesian Genetic Programming" reveals the nature of its typing system—it is strongly typed! It is also statically typed; thereby guaranteeing that the evolved programs are both type-safe and syntactically valid prior to their execution.

Types in ST–CGP

As previously mentioned, the programs generated by ST–CGP exhibit both strong and static typing. However, the requirement for constructing syntactically valid programs extends beyond the programs themselves. To ensure the generation of valid programs, the entire ST–CGP system must possess an awareness of types[1]. Therefore, all elements of the underlying CGP engine are equipped with type annotations, including function inputs and outputs, as well as input and output variables of the system as a whole. Furthermore, evolutionary operations such as mutation and crossover are designed to maintain type-awareness, guaranteeing that these operations always produce individuals that are syntactically correct. A more comprehensive exploration of type usage in functions is presented in subsection 3.1.2, while an extensive explanation of the type-aware nature of evolutionary operators is provided in subsection 3.2.1.

The representation of types can vary across different implementations. In the reference implementation, the programmer is required to assign a unique string to each type. Each component of an individual—i.e. each gene in the phenotype—possesses a string property that stores one of these "type strings." To illustrate, let's consider an example where the programmer designates two types: an integer value and a floating-point value. In this scenario, the programmer might choose to assign the string `"INT"` to the integer type and the string `"FLOAT"` to the float type. If these are the only specified types, then all functions, inputs, and outputs must be either `"INT"` or `"FLOAT"`.

3.1.2 Functions

Functions play a crucial role in the design and use of programming languages and systems. Programming languages can be classified into distinct categories based on their approach to function construction and utilisation. These categories can be further subcategorised. The three most common categories are as follows:

[1]Technically, the generation of syntactically invalid programs could be permitted. However, in testing, it was found that failing to constrain program generation to valid programs resulted in such a dramatic degradation in performance that it was not worth pursuing further.

- Imperative programming languages, like C, employ a series of statements to manipulate the state of a program. Procedural programming, a common subset of imperative programming, employs both functions and sequences of individual statements to execute program logic.

- Functional languages, such as Lisp, utilise “pure” functions that do not alter the program’s state. The output of a function remains constant when provided with the same input. Programs in functional languages are often composed of interconnected function calls applied to input data, rather than a sequence of isolated statements.

- Object-oriented languages facilitate program organisation through the use of classes, which can incorporate both functions and properties (variables). These classes can be instantiated, and typically support polymorphism, where subclasses can extend other classes.

Programming style in ST–CGP

The programs generated by ST–CGP follow a functional style akin to Lisp; an example is given in Fig. 3.1. Consequently, they share the characteristics associated with functional languages, as mentioned earlier: lack of state within programs, outputs remaining consistent for a given input, and the absence of isolated statements. This approach offers both advantages and disadvantages. On the positive side, the resulting programs are relatively readable for human comprehension, and interpreters can be easily developed. Additionally, the absence of state simplifies the process of reasoning about and debugging program execution. However, the lack of state and the absence of more intricate paradigms like classes may restrict the expressive potential of the programs. Nevertheless, this research has discovered that such a programming style proves more than sufficient for the selected problems.

Despite the functional nature of the output produced by ST–CGP, the implementation of ST–CGP itself is not bound by the same style requirement. In fact, the reference

```
(add 2 (sub 3 (round 1.2)))
```

Figure 3.1: An example of a program produced by ST–CGP. Whilst the program is simple, note the lack of state, and its functional nature.

implementation is coded in C#, an object-oriented language. The author discovered that the advantages offered by object-oriented programming were particularly valuable during the implementation of the ST–CGP system. Notably, all system components were defined as separate classes, facilitating a faster development process and greatly simplifying debugging procedures. Additionally, employing an object-oriented approach allowed for polymorphic implementation of functions in C#, even though the outputs of ST–CGP do not inherently possess polymorphic characteristics. A brief discussion on this matter is presented towards the end of this subsection.

Node Functions

In ST–CGP, a distinction is made between "Node Functions" and "Implementation Functions," in order to differentiate between functions written in the implementation language and functions used in the ST–CGP outputs. For example, the function which determines program fitness is an implementation function, whereas a function contained in an ST–CGP program to add two numbers together is a node function. Node functions are a proxy to implementation functions; in other words, each node function has a corresponding implementation function.

Node Functions are the main constituents of individuals within ST–CGP. Each node function must have an output type specified; void functions that do not return a value are not permitted. Additionally, node functions can have any number of inputs, each of which also have a type. Unlike CGP, which allows up to two parameters per function, ST–CGP allows node functions of any arity to be supplied. This is because the mutation mechanism described in subsection 3.2.1 is able to take into account the varying arities of all node functions in the system. Function arity is determined from the number of input types specified. It is assumed that node functions are pure—although, in practice, this is hard to enforce.

As mentioned, node functions are proxies to implementation functions—each node function has a corresponding implementation function, and that is what is actually executed when the *node function* is executed by the ST–CGP interpreter. The ST–CGP interpreter automatically supplies the inputs to the implementation function and receives the output, before passing that to the next function in the program.

Finally, each node function is given a name. This name must be unique to each node function from the perspective of ST–CGP outputs, so even if polymorphic implementation functions are used, a separate node function must be used if different types are to be supplied to the function. Fig. 3.2a demonstrates this concept with a polymorphic implementation function (Fig. 3.2a which adds two numbers together. Due to the polymorphic nature of C#, both the inputs to, and the output of, the function can be specified as the primitive type `object`. Two node functions can then be defined: one which uses the implementation function with `"INT"` parameters (Fig. 3.2b) and one which uses it with `"FLOAT"` parameters (Fig. 3.2c). In practice, the implementation function performs all calculations using floating point numbers.

In the reference implementation, C#'s ability to provide function references is leveraged to allow the `NodeFunction` class to refer to the implementation function which is executed. This is in contrast to CGP where a function table and lookup index is used.

3.1.3 Genotype & Phenotype Representation

In GP, "*genotype*" refers to the encoded representation of a candidate solution, whereas "*phenotype*" refers to the decoded, executable program or expression that the genotype represents. In tree-based GP, the genotype is typically the tree-structured encoding (nodes as functions/terminals and their connections), while the phenotype is the executable program/function that the tree evaluates to; in CGP, the genotype is a fixed-length genome formed of nodes (either arranged in a "grid" or a single linear "row") encoding a feed-forward directed acyclic graph. The genotype contains node functions plus

```
object _Add(object arg1, object arg2) {
    return Convert.ToDouble(arg1) + Convert.ToDouble(arg2);
}
```

(a) An implementation function that adds two numbers together, after converting them to floating point numbers.

```
var addInt =
  new NodeFunction("ADD_I", _Add, ["INT", "INT"], "INT");
```

(b) A node function that calls the `"_Add"` implementation function with two parameters of type `"INT"`, returning an `"INT"` value.

```
var addFloat =
  new NodeFunction("ADD_F", _Add, ["FLOAT", "FLOAT"], "FLOAT");
```

(c) A node function that calls the `"_Add"` implementation function with two parameters of type `"FLOAT"`, returning a `"FLOAT"` value.

Figure 3.2: A polymorphic implementation function with several node functions, as used in the reference implementation written in C#.

connection genes, often with inactive "junk" nodes, while the phenotype is the resulting expressed computation graph/program formed by the active nodes. The process of converting genotype to phenotype is called the genotype–phenotype conversion (also commonly referred to as genotype-phenotype mapping, expression, or decoding). In CGP, the genotype–phenotype conversion involves decoding the fixed-length genome into a feed-forward DAG by assigning each node its function and input connections (respecting levels-back/acyclic constraints), then tracing backwards from the designated output genes to identify the active nodes and assembling only this active subgraph into the executable program.

Strongly Typed Cartesian Genetic Programming introduces a series of substantial modifications designed to enrich the original CGP representation, thus facilitating the new features such as strong typing, variable function arity, and so on. The way ST–CGP implements evolutionary operations such as mutation and crossover, while strictly adhering to type constraints, distinguishes it from conventional CGP.

This subsection is devoted to detailing the fundamental constituents of ST–CGP's genotype-phenotype representation, along with its distinctive characteristics and the rationale behind the necessary adaptations from the classic CGP model. The author will describe how ST–CGP presents the genotype as a sequence of complex nodes, and the phenotype as a directed acyclic graph, as well as what this means in practice. Special attention will be paid to the type constraints of the nodes and their role in the construction and execution of individuals. By the end of the subsection, the reader should understand how the genotype and phenotype are represented and converted in ST–CGP, and by extension, in CGP (which is a broadly similar, if simpler, representation and conversion process).

Genotype - The Grid

Cartesian Genetic Programming is named after its use of a two-dimensional grid to represent the genotype. This "grid" representation changed over time to be a grid with a single row with many columns, as opposed to a grid with many rows and columns. ST–

CGP adopts this single row representation[2]. An important distinction to highlight is that ST–CGP does not utilise the levels-back parameter, which normally constrains the level to which a node can reference previous nodes. In contrast, a particular node within the ST–CGP grid can make references to any other node that holds a lower index.

Each grid in ST–CGP can have one or more output node. Each output node must produce a value of the specified type—this type is excluded from mutation, so the grid will always return values that correspond to the supplied output types. In the reference implementation, when constructing a grid, the output types are specified as an array, for example: `["INT", "INT"]`. This will result in two output nodes, each of which must return an `"INT"` value.

Nodes

While nodes in Cartesian Genetic Programming comprise three integers and the entire genotype of an individual is a simple string of integers, ST–CGP adopts a more intricate method. In ST–CGP, nodes possess a reference to a node function, not an index to a function table. As hinted at previously, ST–CGP's nodes are not limited to a fixed number of inputs. Rather, they can have as many inputs as the node function necessitates. For instance, a node featuring a function of 1-arity will require one input, while a node with a function of 3-arity will necessitate three inputs. Notably, mutations may alter the node function reference, leading to changes in the number of inputs for the node. A convenient technique to accommodate this fluctuating number of inputs is to use arrays. Grid outputs are a special case of node. Unlike program nodes, which have 0-N inputs and always have exactly one node function, output nodes have *no node function*, and always have a *single input*. This means that they are essentially just a pointer to a node in the program. Output nodes have a fixed type, set by the user; as such, an output node with type T must always point to a program node that outputs type T for the program to be valid. An output node can have its pointer-like input modified by mutation much

[2]The reference implementation allows a two-dimensional grid to be specified if required, however in testing this showed no advantage over a single row.

like the inputs of any other node.

In the reference implementation, the `List` class provided by C# is employed to represent the input array, in preference to primitive arrays, offering the runtime additional convenience functions. Besides the node function and inputs, each node is assigned an index, signifying its position in the overall grid. This index serves as the reference for nodes to other nodes within the grid, assisting the interpreter in navigating through the grid during execution. Consequently, each node's input array becomes a `List<int>`. It is crucial to ensure these "node references" stay within the boundaries of the overall grid. The reference ST–CGP interpreter carries out checks before grid execution to guarantee this, and the evolutionary operators are designed to prevent node references from exceeding grid boundaries.

The node function of a specific node defines its output type. Consequently, altering the node function, perhaps through mutation, changes the output type of that node. As previously discussed, output nodes are shielded from type changes, though alterations to the node reference remain permissible. A situation could arise wherein evolutionary operators modify the node function so that it demands inputs of a specific type, but no nodes with lower indices provide the necessary output type. There are two potential solutions to this challenge:

1. Adopt a fail-fast strategy: should an evolutionary operator lead to an invalid state in which a node fails to locate a node of lower index outputting the correct type, the operator should be discarded and the process retried.

2. Introduce a constraint to the system that mandates a zero-arity function for each type. In a system with N pre-defined node functions and M types, populate the first N nodes by applying all known node functions, assigning one to each node. Lastly, exclude these nodes from the influence of evolutionary operators. As N is always greater than or equal to M, it guarantees that for any modifiable node (a node with $index > N$) there is always at least one node of lower index available that can provide the required output type.

Although the first solution appears simpler to implement and the second introduces an additional constraint, the second solution proves to be superior in performance. The initial solution tends to result in excessive computational waste during the evolution process, whereas the second solution guarantees, subject to evolutionary operators respecting the unmodifiable nodes, there will be no failed operations. Consequently, the mutation process is faster, reducing the generation time significantly. This, in turn, accelerates the overall evolutionary process, leading to more efficient performance.

Nodes which can be modified by evolutionary operators are referred to as "program nodes." New genetic material is introduced to the population through changes to the program nodes over the course of the evolutionary process. The output nodes and unmodifiable nodes never change, and will be the same (apart from the input to the output nodes) for each individual in the population.

In the context of evolutionary operators, each input for a node and the node function are collectively considered as genes. This implies, for instance, that a node with a 2-arity node function, consists of three genes: the node function itself and two inputs. These genes can be the target of evolutionary operators, which may act on zero, one, or more of these elements, depending on the operator.

Genotype to Phenotype Conversion

Despite the genotype being depicted as a two-dimensional grid, the phenotype, once converted, is represented as a Directed Acyclic Graph, also known as a DAG. A DAG is a type of graph comprising nodes and directed edges. Nodes typically encapsulate either information or operations. In the context of ST–CGP, nodes equate to the ones discussed in section 3.1.3. Directed edges can be visualised as arrows linking several nodes. Importantly, the term "acyclic" denotes the absence of loops or cycles within the graph, implying that once a node has been traversed, it is not possible to return to the same node by following a series of edges.

Given the constraints imposed by ST–CGP, it is feasible to convert a DAG into a tree-based representation. Following this transformation, code can be generated using in-order depth-first tree traversal. However, it should be highlighted that the conversion from a DAG to a tree is not strictly required. Indeed, under the constraints set by ST–CGP, code could directly be generated from the DAG. However, this functionality was not implemented as part of this thesis. One disadvantage of the DAG-to-tree conversion is the resulting duplication of both any reused nodes, whether they are terminal or an internal node. This means that, during evaluation, the corresponding result of that node will be recalculated as many times as it is reused. Fig. 3.3 demonstrates the entire process of conversion from genotype to phenotype.

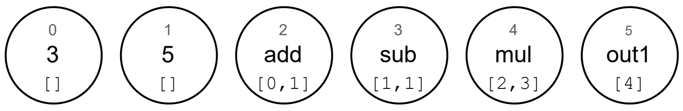
The reference implementation generates code in a functional programming style, where all function calls are inlined. However, alternative programming styles could also be adopted. Fig. 3.4a provides pseudocode that closely mirrors the code that would be generated by the reference implementation, given the tree depicted in Fig. 3.3c. Conversely, Fig. 3.4b presents a potential procedural rendition of the same pseudocode.

3.2 Evolutionary Mechanisms

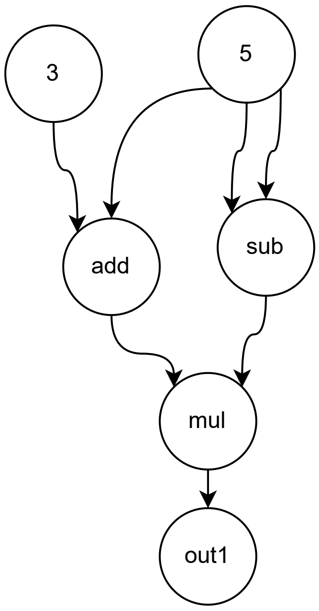
Evolutionary mechanisms hold a pivotal role in the design of any evolutionary algorithm framework. These mechanisms, acting as the foundation and guiding principles of evolutionary processes, precisely delineate the manner in which the framework progresses and refines its solutions. They serve to articulate the operation of evolutionary operators, those being the drivers of diversity and innovation within the population.

The primary evolutionary operators in CGP—mutation and selection—all interact with the evolving solutions in unique ways, generating variation and ensuring the survival and propagation of the most effective solutions. Mutation introduces spontaneous changes, prompting new explorations in the solution space. Selection, meanwhile, acts as a filter, prioritising solutions that exhibit a higher degree of fitness.

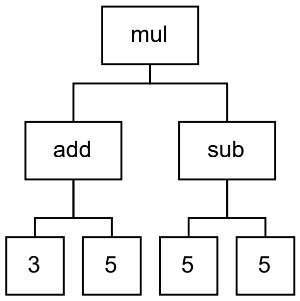
Furthermore, these mechanisms incorporate the overarching evolution strategy utilised to guide the development of the population over generations. This strategy determines the core mechanics of how new solutions are generated, evaluated, and then



(a) A simple grid in ST–CGP. Each node has an index, a function (here represented by the function name), and a set of inputs. Note that node 2 is redundant; it is not actually used in producing the output.



(b) The directed acyclic graph resulting from the grid. Note the reuse of nodes.



(c) The tree produced from the directed acyclic graph. Note that terminal nodes are now duplicated - a drawback to this representation.

Figure 3.3: The process of Genotype-Phenotype conversion in ST–CGP.

```
mul(add(3, 5), sub(5, 5))
```

(a) Functional pseudocode generated from the tree.

```
// Nodes
node0() {
  return 3;
}

node1() {
  return 5;
}

node3() {
  return add(node0(), node1());
}

node4() {
  return sub(node1(), node1());
}

node5() {
  return mul(node3(), node4());
}

// Outputs
node6() {
  return node5();
}

main() {
  return node6();
}
```

(b) Procedural pseudocode generated from the tree.

Figure 3.4: Code generated from a tree.

introduced back into the population. This influences the trajectory of the evolution, steering it towards areas of the solution space that exhibit potential.

Additionally, evolutionary mechanisms have a hand in the selection of hyperparameters, the auxiliary numerical values that tune the behaviour of the evolutionary algorithm. Hyperparameters might define population size, mutation rate, or the degree of selection pressure, among other aspects. Their values can considerably impact the performance of the evolutionary process.

The aim of this section is two-fold: first, to explain in simple terms how the evolutionary operators in ST–CGP function, and second, to share the practical reasons behind the selection of different methods and parameters. This discussion should make the evolutionary specifics of ST–CGP more understandable, giving a clearer picture of how its evolutionary mechanisms work.

3.2.1 Mutation

Similarly to CGP, mutation in ST–CGP is the primary method of introducing new genetic material into populations. One of the standard methods of mutation in CGP is called "until active" mutation, which seeks to mutate genes at random until an active gene (one that is part of the set of active nodes) is changed:

1. The list of *active* nodes is recorded.
2. A node is selected at random from the program nodes.
3. A gene belonging to the node is selected at random.
4. One of the node's genes is changed—either an input reference, or the node function. The new gene is selected at random.
5. The list of *active* nodes is recorded once more.
6. If the new list of active nodes matches original list, the process is repeated. Otherwise, the mutation process concludes.

This approach of modifying individuals was originally devised for systems that do not incorporate typing. Naturally, the introduction of strong typing adds an extra layer of complexity to such a function. To produce a semantically correct program tree, it is essential that all nodes, including their arguments, align with the correct type. There are two strategies by which this requirement can be met:

Option 1: When altering a function for a specific node, restrict the selection of potential new functions to those which share an identical signature with the current function[3]. When altering an input that is supplied as an argument to the node, ensure that the newly chosen input has the same type as the previous input.

The main advantage of this option lies in its simplicity of implementation—one just needs to select a new function or node that aligns with the types. However, upon more careful examination, it becomes evident that this seemingly advantageous option carries significant drawbacks. By confining the selection of new functions to those with matching return types, the overall "type structure" of the individual remains static. This drastically narrows down the search space accessible to that individual, often hindering the individual from ever attaining an optimal solution.

Option 2: When altering a function for a specific node, permit *any* function to substitute the current function. In a manner analogous to Option 1, when changing an input presented as an argument to the node, it is required again that the newly chosen input matches the type of the previous input. The advantage and disadvantage of this option stand in contrast to Option 1—it offers an individual the capacity to traverse a substantially larger portion of the search space by enabling the type structure of the individual to evolve over time. However, the task of preserving semantic correctness escalates to a significantly more complex challenge.

In designing ST–CGP, the author devised a technique that ensures the maintenance of semantic correctness of an individual, even while allowing free substitution of functions

[3]This implies that both the return type and the argument types must be an exact match. For instance, if the current function returns an `"INT"` and accepts `["INT", "INT"]` as inputs, any replacement function must also return an `"INT"` and accepts `["INT", "INT"]` as arguments.

as depicted in Option 2. This can be applied as part of a modified *until active* mutation process, a detailed explanation of which follows.

Step 1: Record the list of active nodes

The initial step, as stated in the preceding list, is to record a list of the active nodes prior to mutation. Active nodes are those which contribute to the generation of an output from the individual, given a specific input. If a node is absent from the active list, it is referred to as inactive or *redundant*. Research by Miller [71] revealed that a considerable degree of redundancy is advantageous to the evolutionary process. In fact, an optimum level of redundancy can be as high as 95%.

In order to collect the active node list for a specific output node, an in-order traversal technique is used. This traversal begins at the output node, and the active list is initialised to the empty set. Observant readers may find a similarity with the actual evaluation of an output, which also utilises in-order traversal. As each node is visited during the traversal, the index of that node is added to the active list if it isn't already part of it. Once the traversal is complete, the active list encompasses all active nodes that are connected to the initial node. If an individual possesses multiple output nodes, the active node list is assembled for each node. Subsequently, these lists are combined, with any duplicate entries being removed.

In the reference implementation, the traversal function used to evaluating outputs, gather active node lists, and ensure semantic type correctness leverages a stack to help with traversal. For each node that is visited, its inputs are added to the top of the stack. Then, the first item on the stack is popped, and processed by a callback supplied to the traversal function. The callback is provided with a reference to the node that has been popped. Once the callback has finished executing, any nodes used as inputs to the current node are added to the stack. This is repeated until the stack is empty. This method of traversal is by no means obligatory but presents a memory-efficient way of performing depth-first in-order traversal.

Step 2: Select a node at random

Following the collection of the active list, a node is randomly selected from the individual's program node list. This selection process employs a uniform probability distribution, meaning that each node possesses an equal likelihood of being chosen. Expressed mathematically,

$$p(selection) = \frac{1}{|G|},$$

where G symbolises the set of program nodes.

It should be noted that for the purposes of mutation the initial set of function nodes kept at the beginning of the set of nodes are considered sacrosanct—they may not be selected for mutation, and are not included in the selection process described above. The reader is referred to Section 3.1.3 if a reminder of the purpose of these nodes is required.

An important point to remember for mutation in this context is the unmodifiability of the initial set of function nodes. These nodes, located at the start of the grid, are excluded from mutation and are not part of the selection process.

Step 3: Select a gene from the node at random

Once a node has been selected, the next step involves choosing a gene associated with that particular node for mutation. In the context of ST–CGP, the term "gene" encompasses both the inputs supplied to the node and the node function assigned to the node. During each mutation cycle, precisely one gene is chosen for mutation. It is important to note that altering the function assigned to the node can result in more than one change to the grid, as explained later. Similar to the process of node selection, a uniform probability distribution is employed to select the gene. Each gene has an equal probability of being chosen, ensuring a fair selection. To illustrate, consider a scenario where a node has two inputs. In this case, there are three genes: the function and each of the inputs. Consequently, each gene has a one in three chance of being selected.

Step 4: Modify the gene

The modification of the chosen gene represents a crucial step in the process as it introduces fresh genetic material into the individual. The specific method employed to modify the gene varies based on whether an input or a function has been selected.

If the selected gene happens to be the node function assigned to the node, a new node function is randomly chosen from the list of node functions that were defined during the creation of the individual. It is worth noting that the specific type or arity of the function is not significant at this stage—any discrepancies or inconsistencies in terms of type or arity are addressed and resolved in Step 5.

Alternatively, if the selected gene corresponds to an input, a node is chosen at random from the list of program nodes to replace the input. The replacement node must adhere to the following constraints:

- The index of the replacement node must be strictly lower than the index of the node being mutated.
- The replacement node must possess the same return type as the input being substituted.

While the new input will always have the correct type, there may be situations where the inputs of that particular node are not syntactically correct. This can occur, for instance, if an input to the node undergoes a mutation that alters its function and changes its type accordingly. If such a situation arises, it will be rectified in Step 5.

Step 5: Ensure syntactic correctness

Ensuring type correctness represents the most intricate stage of mutation in ST–CGP and serves as the most significant enhancement compared to conventional CGP. The preceding step, which involves modifying the selected gene, has the possibility of yielding a program that is syntactically incorrect. To maintain compliance with the prerequisite that all programs must be syntactically correct, an additional traversal of the individual is necessary to address any problems. The procedure is carried out as follows:

1. Initialise a stack of nodes containing the list of output nodes.
2. Initialise a set of nodes—which will keep track of nodes which have been visited—to the empty set.
3. While the stack is not empty:
    (a) Pop the top node from the stack (refer to this as the current node).
    (b) For each input type required by the node function on the current node (refer to this as the current input):
        i. If the return type of the node pointed to by the current input does not match the required type, select a new node of the correct type with an index less than the current node, and set the current input to the newly selected input.
        ii. If the current input (after the above step has been carried out) is not present in the visited node list, push it onto the stack.
    (c) If the input list contains more inputs than are required by the function, remove excess inputs.
    (d) Add the node to the set of visited nodes.

Following the completion of this procedure, it is guaranteed that the active nodes constitute a program that is syntactically correct. However, the same guarantee does not extend to redundant nodes, as there may exist incorrect type structures within the unused portions of the list of program nodes. Nevertheless, this is not a concern since future mutations will once again enforce syntactic correctness by adhering to the aforementioned procedure. In fact, granting the redundant nodes the freedom to disregard type constraints yields two advantages. Firstly, it allows for a wider exploration of the search space. Secondly, it reduces the time required for a mutation, as only the active nodes are taken into consideration.

Step 6: Determine if mutation must continue

Due to the random selection of a node from the complete set of program nodes in Step 2, it is possible for mutations to exclusively occur within redundant sections of the program nodes set, resulting in no impact on the actual program. In ST–CGP, an “until active” mutation approach is utilised. This means that the mutation process is *repeated until a change is observed in the program*. To initiate this step, the list of active nodes is once again collected. If the newly obtained list of active nodes is identical to the one collected at the beginning of the mutation, it indicates that the program remains unchanged, and thus, mutation must be performed again. Conversely, if the lists differ, it signifies that mutation has concluded.

Listing 1 is a pseudocode listing for this mutation technique. Note that Step 5 described above begins on Line 23 of the listing.

```
1  originalActive = graph.activeNodes
2  activeChanged = false
3  while (activeChanged == false) {
4      selectedNode = select random node from graph
5      if (selectedNode is an output node) {
6          // output nodes have no function and just one input, so can only
            ↪  change the input
7          requiredType = selectedNode.outputType
8          selectedNode.inputs[0] = select random node where outputType ==
            ↪  requiredType and index < selectedNode.index
9      } else {
10         target = random int between 0 and num of inputs for selectedNode
11         if (target == 0) {
12             // change the function
13             newFunc = randomly choose any known node function
14             selectedNode.func = newFunc
15         } else {
16             // change an input
17             requiredType = selectedNode.inputs[target-1]
```

```
            selectedNode.inputs[target-1] = select random node where
             ↪  outputType == requiredType and index < selectedNode.index
        }
    }

    // fix inputs of all active nodes
    nodesToFix = stack containing all output nodes
    while (nodesToFix is not empty) {
        currNode = nodesToFix.pop()
        ensure currNode.inputs is the same length as currNode.func expected
         ↪  input types
        for each input in currNode.inputs:
            if (input is the wrong type) {
                input = select random node where outputType == requiredType
                 ↪  and index < selectedNode.index
                nodesToFix.push(input)
            }
    }
    newActive = graph.activeNodes
    if (newActive != originalActive) {
        activeChanged = true
    }
}
```

Listing 1: Pseudocode for the typed until-active mutation technique used in ST–CGP.

3.2.2 Evolution Strategy

The term “evolution strategy” (ES) is commonly used to describe the parent–offspring selection scheme by which candidate solutions are generated and a new set of parents is chosen each generation. Broadly, an ES specifies how many parents are used to generate offspring, how many offspring are produced, and whether selection is performed from offspring only or from the union of parents and offspring. The most common schemes are:

- $(1+1)$**-ES**: One parent $\rightarrow$ one offspring. The next parent is chosen as the better of the parent and offspring (elitist selection).
- (μ,λ)**-ES**: μ parents $\rightarrow$ λ offspring. The comma indicates that selection is performed from the offspring only; all parents are discarded (non-elitist replacement).
- $(\mu+\lambda)$**-ES**: μ parents $\rightarrow$ λ offspring. Selection is performed from parents $\cup$ offspring (elitist replacement), preserving the best-so-far individual.

Cartesian Genetic Programming commonly employs a $(1+\lambda)$-ES selection scheme, with a small λ often less than five. CGP typically allows **neutral drift**, whereby an offspring with fitness equal to its parent may replace the parent. This permits changes to inactive genes without loss of fitness. While much of the CGP literature adopts this scheme, ST–CGP uses a slightly more intricate variant. The following procedures are followed to generate a new population in each generation:

1. The individual with the highest fitness score from the previous generation is added to the new population.
2. Half of the remaining population size is filled by generating individuals through mutation of an individual selected via tournament selection.
3. A quarter of the remaining population size is filled by generating individuals through mutation of the individual with the best fitness score from the previous generation.
4. The remaining portion of the population size is filled by randomly generating individuals.

- The preservation of the best individual “so far” throughout the evolution process, ensuring that it is not lost between generations.
- The generation of the new population is not excessively biased towards the elite members of the previous generation’s population, as approximately 50% of the new population is generated through tournament selection.

- Nevertheless, some preference is given to the elite members of the population through the 25% elite selection.
- Additionally, each generation introduces entirely new genetic material, thereby enhancing the diversity. This effect is particularly noticeable in the early stages of evolution when programs tend to be shorter and less complex, making the newly generated programs more likely to be relevant.

In the initial testing phase of ST–CGP, the effectiveness of the $(1 + \lambda)$-ES strategy was evaluated, but it was found to be less successful compared to utilising a larger population with the aforementioned technique. The author suggests that this discrepancy may be attributed to the introduction of types, which, despite increasing the relevance of the search space, also significantly increases its size. As such, a $(1 + \lambda)$-ES strategy alone proves insufficient in adequately exploring the entire search space and achieving good coverage. The technique described above proved to be a good compromise between elitism and diversity.

3.2.3 Hyperparameters

As is customary in most GP frameworks, ST–CGP allows for the fine-tuning of hyperparameters prior to commencing a run. However, in pursuit of user-friendliness and ease of implementation, ST–CGP simplifies the hyperparameter selection process by omitting certain hyperparameters typically associated with CGP. Therefore, ST–CGP presents a concise set of exposed hyperparameters that can be adjusted to customise the system according to specific requirements.

- **Population size**—the number of individuals in the population.
- **Node count**—the fixed [4] number of nodes contained in each individual.

[4]In the reference implementation, the number of nodes is fixed, though there is no technical reason preventing it from being variable. Of course, the number of *active* nodes is variable due to nature of the genetic operators.

- **Max. Generations**—the maximum number of generations to be performed.
- **Max. Time**—the maximum amount of time[5] for the entire evolution process.
- **Fitness threshold**—a value below which ST–CGP considers the problem "solved" and will exit early.

Allowing both a maximum number of generations and a maximum evolution time to be specified has several advantages. First, it supports fair comparisons between runs and across techniques: a generation cap provides a clear, step-based measure of computational effort (i.e., how *hard* the evolutionary process was allowed to work), which is useful when comparing search dynamics under similar evaluation models. In contrast, a time cap reflects a real-world resource constraint: given the same wall-clock budget, can one technique achieve more progress than another, especially when evaluation cost varies due to implementation details (e.g., interpreters, caching, or parallelism)? Using both criteria also makes termination more robust, preventing very slow runs from consuming excessive resources and very fast runs from implicitly gaining an advantage by simply completing far more generations.

3.3 Advanced Optimisation Techniques

Genetic Programming systems are often used to tackle complex problems. To ensure a timely exploration of such large and complex search spaces, advanced optimisation techniques can be used. These techniques, which span from algorithmic enhancements to computational shortcuts, play a pivotal role in equipping ST–CGP to effectively tackle complex problems. They serve to enhance the efficiency of computations, bolster the quality of solutions, and offer more flexibility in handling diverse problem domains. Through the optimisation of ST–CGP, we can drive the evolution of solutions that are not only optimal but are also found with greater computational efficiency. Hence, the

[5] In the reference implementation, specified in seconds.

exploration and application of these advanced optimisation techniques are of crucial importance in the progressive evolution of ST–CGP.

In this section, we explore the intricacies of Strongly Typed Cartesian Genetic Programming (ST–CGP) in more detail, specifically focusing on the deployment of various advanced optimisation techniques that serve to augment its performance.

Firstly, we explore the implementation of crossover, a prevalent concept in traditional Genetic Programming (GP) yet somewhat uncharted in the realm of Cartesian Genetic Programming (CGP). The focus then shifts to multi-objective optimisation. The challenge here is to handle competing objectives within the evolutionary process and achieve a balance that satisfies, as far as possible, all objectives. Through this exploration, we seek to illuminate the effects and potential benefits of multi-objective optimisation within ST–CGP.

The concept of "subgrids" is also examined, a technique that allows for the creation and evolution of reusable small grids of nodes. These subgrids, which offer the prospect of greater computational efficiency and modularity, can be seen as a unique tool in the evolution of ST–CGP solutions.

Lastly, we investigate efficiency improvements, such as the application of memoisation caching. By storing the results of expensive function calls and reusing them when the same inputs occur, memoisation caching presents an avenue for enhancing computational performance.

3.3.1 Crossover

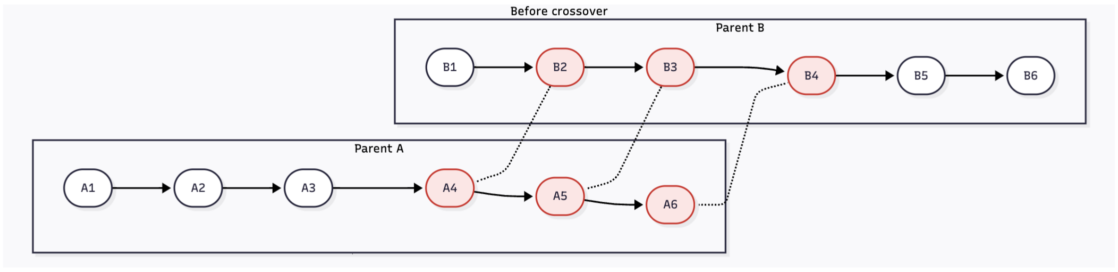
In the realm of Genetic Programming, the technique of crossover has been extensively studied and applied. However, its application within Cartesian Genetic Programming has received considerably less attention. Unlike GP, CGP predominantly relies on mutation operators to drive the evolution of its solutions. Crossover, on the other hand, allows for the recombination of diverse solutions, thereby facilitating the exchange and blending of successful characteristics. It is reasonable to assume that the benefits observed in GP through the use of crossover could potentially extend to CGP if crossover were to be incorporated. Julian Miller, the late creator of CGP, noted that while the omission of crossover was a deliberate design choice in the original form of CGP, it would still be advantageous to investigate its potential benefits.

The author explored the application of two types of crossover to ST–CGP. The first type, named "full crossover" in this study, closely resembles the standard two-point crossover commonly employed in GP. The second type introduces a novel approach to crossover called "**genetic rewiring**," which enables crossover operations without altering the structural composition of an individual.

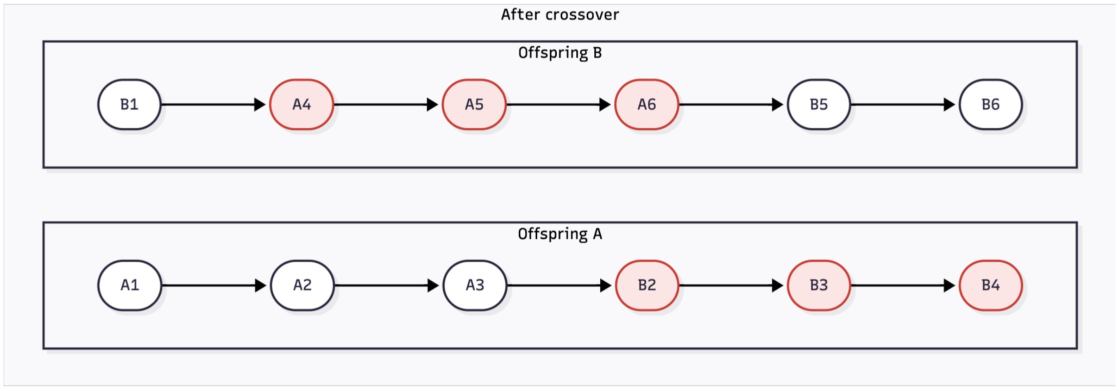
Full Crossover and Genetic Rewiring

In tree-based GP, two-point crossover is a genetic operator which operates by first choosing one crossover point in each of the two parent trees. Each point marks the root of a subtree. These subtrees, comprising the selected node and all its descendants, are then exchanged between the parents. The swap replaces each chosen subtree with the corresponding subtree from the other parent, creating two new offspring while preserving the overall tree structure (and "shape") of each.

ST–CGP includes a similar mechanism. Because its genome is linear rather than tree-structured, two positions in the genotype are chosen at random for each individual. Two parent individuals, drawn according to the chosen selection scheme (e.g., tournament or random selection), then swap the sequence of nodes lying between these positions. The type-fixing routine is subsequently executed to ensure syntactic validity. Fig. 3.5 illustrates the application of the *full crossover* operator to two linear program individuals. A contiguous block of three consecutive nodes is selected in each parent (A4-A6 in Parent A and B2–B4 in Parent B; highlighted in red) and these blocks are exchanged. The dotted lines indicate the correspondence between swapped nodes (A4 $\leftrightarrow$ B2, A5 $\leftrightarrow$ B3, A6 $\leftrightarrow$ B4). Although the starting positions differ between the parents, the block length is chosen **once** and enforced for both parents, so the same amount of genetic material is exchanged and the offspring remain the same length as their parents. The relative order of nodes within each exchanged block is preserved.



(a) **Before applying full crossover**. Selected equal-length blocks to be exchanged are highlighted (Parent A: A4–A6; Parent B: B2–B4). Dotted lines indicate node-to-node correspondence.



(b) **After applying full crossover**. Offspring produced by exchanging the highlighted blocks, preserving node order and individual length.

Figure 3.5: Illustration of Full Crossover applied to two individuals.

Genetic rewiring refines the full crossover method outlined above. Whereas full crossover selects crossover points randomly, without regard to the node types that lie between them, genetic rewiring adds an additional constraint: a segment is eligible for crossover only if the **sequence of node types** within that segment is identical in both parents. This type-aware selection method increases the likelihood that the exchanged segments will remain syntactically valid, with one caveat: as the segment length increases, the likelihood of finding a matching segment of the same length in two individuals decreases. The segment length is initially chosen at random; if a matching segment is not found (either due to the length being too long, or simply due to there not being a matching segment in the two individuals), the operation is retried with a different length, strictly less than the length of the failed operation.

3.3.2 Multi-objective Optimisation

Complex real-world problems often exhibit a remarkable characteristic: their solutions, despite the intricacy of the underlying challenges, can sometimes possess elegant simplicity (though this is not universally the case, and many natural phenomena do not admit a suitably simple model). This observation has motivated researchers in the field of evolutionary computation to seek optimisation techniques that strike a balance between the complexity of solution representations and their performance. In the realm of Genetic Programming, in particular, where programs are represented as trees, the issue of program length, or "bloat," has been a recurring concern. Bloat refers to the tendency of evolved programs to grow excessively in size without proportionate improvements in performance.

Numerous attempts have been made to address the issue of excessive code growth in GP by modifying its operational mechanisms. However, an alternative approach called Multi-objective Optimisation offers another avenue for tackling this problem. Unlike traditional single-objective optimisation, which aims to find a single optimal solution, multi-objective optimisation strives to identify a diverse set of solutions that represent various trade-offs between competing objectives. For instance, when addressing the concern of code growth, one may opt to employ the length of the program generated by an individual as a minimisation objective. Or, consider the case of algorithm stagnation observed in CGP, where population diversity decreases over time. One can introduce the age of an individual as an objective to preserve and enhance diversity [105].

NSGA–II

ST–CGP introduces multi-objective optimisation to CGP using the same technique established in [105]: the Non-dominated Sorting Genetic Algorithm II (NSGA–II). The NSGA–II algorithm allows for multiple objectives to be optimised at the same time [106]. [105] demonstrated that using NSGA–II with CGP resulted in fewer fitness evaluations, higher rates of success, and smaller, more generalised programs. Two objectives were used in addition to fitness: the age of the program and its size (the number of *active* nodes; the total number of nodes was fixed). Employing age as an optimisation objective is a concept which has been explored previously in the literature (see, for example, [107, 108]) and is known to be a good way of enforcing genetic diversity. It does this by ensuring "age-fair" competition, whereby individuals of a wide range of ages are kept in the population, preventing individuals of a narrow range of ages from "taking over."

NSGA–II is an extension of the original NSGA algorithm, which incorporates a novel approach to sorting individuals based on non-domination levels. While NSGA–II can generally be applied to any evolutionary algorithm, the implementation in ST–CGP has some minor differences to the conventional algorithm. When NSGA–II is used, it replaces the evolutionary strategy described in section 3.2.2. It operates as follows:

1. Initialisation: The first generation begins by randomly generating an initial population of individuals.

2. Evaluation: The fitness of each individual is evaluated by assessing its performance with respect to the multiple objectives of the optimisation problem. This evaluation involves computing the objective function values for each individual.

3. Non-dominated sorting: The individuals in the population are sorted into different fronts based on their non-domination levels. An individual is said to dominate another individual if it performs better in at least one objective and does not perform worse in any other objective. This sorting process creates a hierarchy of individuals based on their dominance relationships.

4. Crowding distance calculation: The crowding distance of each individual is computed to determine its diversity within a front. The crowding distance reflects how much the surrounding individuals are spread out in the objective space. Individuals with larger crowding distances are preferred, to maintain diversity.

5. Selection: This is the primary means by which ST–CGP's implementation of NSGA–

II differs from the reference. The population for the next generation is created through a combination of non-dominated sorting, crowding distance comparison, and elitism. First, The first front (the Pareto front) is always added to the new population to ensure the best individual for each objective is not lost. A small percentage of the total population size is produced by random initialisation of individuals, much like the strategy in section 3.2.2. Finally, a weighted random selection is used to select a set of individuals, whereby individuals with higher non-domination levels and greater crowding distances are given higher priority for selection. This helps to ensure a diverse set of solutions. These individuals are used for reproduction to fill the rest of the new population.

6. Reproduction: The selected individuals undergo genetic operations to create offspring. The operators always include mutation, and may include crossover, depending on whether crossover has been enabled.

7. Termination: Steps 2-6 are repeated for multiple generations until a termination criterion is satisfied. This criterion can be a fixed number of generations, a maximum runtime, or reaching a predefined convergence level of one or more objectives.

By iteratively applying the above steps, NSGA–II gradually evolves a population of solutions that offers a good trade-off between conflicting objectives. The final result is a set of non-dominated solutions called the Pareto front, representing the optimal trade-off options for the given multi-objective optimisation problem.

3.3.3 Subgrids

Any software engineer who has spent even the briefest time writing code commercially will be familiar with the DRY concept: "Don't repeat yourself." By developing modular, reusable components of code, programmers ensure their work remains streamlined, efficient, and less prone to error. As an analogy, consider a chef who prepares a popular dish many times over the course of a night. Instead of creating the dish from scratch each time, they prepare essential components in advance—say, a rich tomato sauce or a set of chopped vegetables—to expedite the cooking process and maintain consistency in taste and quality. This notion of reusability not only saves time but also makes it easier to modify or rectify any individual component, improving overall efficiency and reducing the chance of errors.

The importance of modularity and reusability is mirrored in Genetic Programming, where the principle has found its expression in several ways. Perhaps the most well-known is in the form of Automatically Defined Functions (ADFs). ADFs are reusable sub-trees that can be called from various points in the genetic program, mirroring the way in which methods or functions are reused in traditional programming. However, another powerful way of expressing reusability is in the non-parametrised reuse offered by Poli's Parallel Distributed Genetic Programming [67] (as discussed in Chapter 2). PDGP achieves reuse not by calling a subroutine with arguments, but by structurally sharing sub-computations inside a program graph: the same node or subgraph can be provided to multiple downstream consumers, so intermediate results are computed **once** and re-used wherever needed. This is referred to as "*non-parametrised*" because the reused fragment is not invoked with different inputs at different call sites; instead, it produces a single value per evaluation, and multiple parts of the program can reference that computed value. In contrast, ADFs support parametrised reuse by being re-evaluated *every time* they are called, allowing the same evolved logic to be applied to different inputs via arguments. This trades simple value sharing for more flexible, function-like abstraction, at the cost of increased computational effort (multiple calls require multiple evaluations).

In the context of Strongly Typed Cartesian Genetic Programming, the concept of reusability is enabled through "subgrids." Subgrids are an advanced optimisation technique aimed at evolving reusable code modules within the ST–CGP framework, much like the ADFs of tree-based GP.

ST–CGP treats subgrids in an identical manner to regular grids. They undergo mutation, can undergo crossover, and are executed using the same interpreter. Each subgrid consists of a single output node, which shares the output type with the main grid. Subgrids are assigned an index, enabling a subgrid with a higher index to access any subgrid with a lower index, if multiple subgrids are employed per individual. In higher subgrids and the main grid, each subgrid is represented by a zero-arity proxy node function. To illustrate, suppose there are three subgrids. The first subgrid operates solely on the node functions provided by the user. The second subgrid incorporates the same node functions as the first subgrid, along with an additional node function named `SUBGRID1`. Similarly, the third subgrid includes all the node functions available to the second subgrid, as well as `SUBGRID2`. Lastly, the main grid encompasses all the node functions supplied by the user, as well as `SUBGRID1`, `SUBGRID2`, and `SUBGRID3`.

The utilisation of subgrids in the context of ST–CGP holds the potential to introduce a highly expressive and modular programming paradigm. Nevertheless, a significant drawback associated with subgrids is the exponential increase in execution time. To illustrate this, consider a scenario where a subgrid at the fifth level initiates multiple invocations to a subgrid at the fourth level, which in turn calls a subgrid at the third level, and so forth. Each invocation of the fifth-level subgrid leads to a cascade of additional calls to subgrids at lower levels. Consequently, the size and complexity of the program can expand at an astonishingly rapid rate. To mitigate this issue, it is advisable to impose constraints on the number of subgrids employed, with the author having achieved particular success by restricting it to a maximum of as few as two subgrids.

3.3.4 Efficiency Improvements

Regardless of the specific variant, GP demands an *immense* volume of computation: a single run may evaluate millions or even billions of candidate solutions. Consequently, to ensure that runs finish in good time, it can become necessary to implement various optimisations, or “shortcuts” to improve efficiency.

Memoisation is one such example. Here, each newly generated individual is represented by a canonical form, in the case of ST–CGP, a hash of its parse tree, and its fitness value is stored in a cache after computation. When the same individual, or one that is structurally identical, is encountered later in the run, the algorithm retrieves the previously computed fitness from the cache instead of performing a fresh evaluation. This simple lookup can dramatically reduce redundant computation, especially in search spaces where duplicates arise frequently through mutation and crossover.

3.3.5 The Reference Implementation

We have seen several advanced techniques designed to facilitate faster, more relevant, and less computationally intensive evolution in ST–CGP. It is worth noting at this juncture that each technique covered is optional in the reference implementation: any, all, or none of the techniques can be enabled when performing evolution. As such the use of each technique represents the addition of a new hyperparameter to the system, thus expanding the hyperparameter list to that shown in Table 3.2.

While the intention of additional optimisation techniques is to improve performance, this is not guaranteed across all problem types. In practice, individual techniques may introduce trade-offs (e.g., improved convergence speed at the cost of solution diversity) and their effects can depend on properties of the task, such as noise, class imbalance, or the dimensionality of the search space. Optimisations can also interact: a technique that is beneficial in isolation may be neutral or even harmful when combined with others, making it difficult to predict the net effect *a priori*.

A systematic evaluation could be used to quantify these contributions, most naturally via an ablation study. ST–CGP would be run with all options enabled and then re-run with individual techniques disabled, followed by runs in which pairs (and larger subsets) of techniques are disabled, thereby testing combinations of settings (up to a full factorial design, if computationally feasible). This would yield a matrix of configurations and performance metrics—ideally reported over multiple random seeds—to identify robustly effective settings and to assess whether the best configuration differs by task. Unfortunately, time constraints prevented a full ablation study from being conducted in this research, with all combinations of settings, however a minimal set of ablation testing was performed on the even parity test, the toy problem in described in Chapter 4, and the experiment described in Chapter 5. This resulted in a set of hyperparameters which appear to perform well on multiple tasks. These were used in all experiments that follow, unless otherwise stated, and can be found in Table 3.3. This set of hyperparameters features a notably larger population size compared to conventional CGP implementations, which in the empirical testing performed better than small populations.

Parameter name	Description	Type	Notes
Population size	The number of individuals in each population	integer	
Node count	The fixed number of nodes contained in each individual	integer	
Max. Generations	The maximum number of generations to be performed	integer	-1 indicates no max
Max. time	The maximum amount of time for the entire evolution process, in seconds	integer	-1 indicates no max
Fitness threshold	A value below which ST–CGP considers the problem "solved" and will exit early	float	-1 indicates no threshold
Use NSGA-II	Whether to use NSGA-II multi-objective optimisation	boolean	
Use full crossover	Whether to use full crossover to generate some individuals	boolean	
Use genetic rewiring	Whether to use genetic rewiring to generate some individuals	boolean	
Number of subgrids	The number of subgrids to use in each individual	integer	0 disables subgrids
Subgrid size	The number of nodes each subgrid should have	integer	
Use function memoisation	Whether to use function memoisation	boolean	Assumes all functions are pure

Continued on next page

Continued from previous page

Parameter name	Description	Type	Notes
Use individual caching	Whether to use cached fitness evaluations for individuals	boolean	

Table 3.2: The hyperparameters exposed by the reference implementation of ST–CGP.

Hyperparameter	Selected value
Population size	300
Node count	250
Max. Generations	500
Max. time	-1
Fitness threshold	1×10^{-6}
Use NSGA-II	true
Use full crossover	true
Use genetic rewiring	true
Number of subgrids	0
Subgrid size	N/A (not used)
Use function memoisation	true
Use individual caching	true

Table 3.3: The default hyperparameters chosen empirically through minimal ablation testing.

3.4 Testing ST–CGP

To verify that ST–CGP actually works, the well-known even-parity test was chosen. The even parity test has long been considered a *de facto* standard for evaluating the performance of GP and its derivatives [109, 41, 70]. Despite the development of more complex benchmarks in recent years, it remains a relevant benchmark for comparison, particularly in light of the fact that CGP has itself been evaluated using this test [110].

The even parity test aims to evolve a program that calculates the even parity bit of a string of bits, starting with a string length of three bits and incrementing the length each time a successful solution is found. The even parity bit is a simple method of error checking in which the parity bit is equal to 0 if the number of 1-bits in the string is even, or 1 if the number of 1-bits in the string is odd. This can be calculated by performing an exclusive-OR (XOR) on each bit in the string, an example of which is demonstrated in this equation:

$$ep(1001) = 1 \oplus 0 \oplus 0 \oplus 1 = 0.$$

While the even parity problem may not be complex in nature, traditional GP techniques have struggled to generate solutions for longer strings, and have rarely been able to generate a solution that works for any input length. A generic solution for untyped techniques must first evolve an XOR function and then apply it successively to each bit in the input string. This presents a significant challenge, particularly given the limited capacity for loops in traditional GP approaches.

It is not just GP that struggles with the parity problem. Indeed, even neural networks often fail to solve the problem reliably. A key reason is that parity defines a highly non-smooth input–output mapping: in the input space, adjacent bitstrings (Hamming distance 1) are mapped to opposite outputs. Consequently, local similarity in the representation does not translate into local similarity in the output, so small moves in the search space induce maximal changes in behaviour. This undermines gradient-based learning and, more generally, any optimisation process that relies on incremental improvements driven by local neighbourhood structure.

The introduction of types allows for a different approach to the problem. By incorporating the “array,” “bit,” “bool,” and “integer” types, as well as a few standard array and mathematical operations, the system can operate at a higher level instead of relying solely on the low-level exclusive-OR functions. For example, it is possible to calculate the even parity bit of an array of bits by summing all non-zero bits using a for-each loop and then calculating the result modulo 2. This solution is not only significantly simpler than evolving a low-level solution, but it also works for input strings of any length.

To assess the impact of adding types on the evolution of a solution to the even parity problem, two different configurations of ST–CGP were tested, one with multiple types, and the other with a single type. In the latter case, a single type was provided to ST–CGP: “bit”; this is essentially the same as an untyped approach. Each test was repeated

Table 3.4: Genetic parameters used in tests.

Parameter	Value
Max Generations	500
Population Size	300
Node Count	250

on bit strings of multiple lengths, from 3 to 8 inclusive. The default hyperparameters referenced earlier were used for this test. It is important to note that the maximum number of generations and the population size were fixed, resulting in a fixed number of possible fitness evaluations, regardless of whether multiple types were used. If a perfect solution was discovered during a run, the run was stopped and no further fitness evaluations were performed. The tests were repeated 100 times to ensure robust results.

The parameters chosen for these experiments were determined through extensive testing during the development of ST–CGP. It is noteworthy that the 1+4-ES[6] evolutionary search, commonly used with CGP, was not employed in this study, as was similarly reported in [74]. Instead, larger populations were utilised. Preliminary experiments indicated that, at least for ST–CGP, larger populations produced considerably better results than the 1+4-ES approach. This finding informed the selection of parameters for these experiments.

For all tests, a set of basic operators was provided. These operators were designed to operate on the "bit" type and are outlined in Table 3.5. For tests that employed multiple types, additional operators were supplied to enable the utilisation of these types. These operators are listed in Table 3.6. The additional operators facilitated the effective evolution of solutions, including generic solutions. Of particular significance is the "INPARR" operator, which presents all input bits to the individual as a single array object. This

[6]A 1+4 evolutionary search is where in each generation four offspring are produced by one parent. The parent is returned if all offspring have a lower fitness value. Otherwise, the offspring with the highest fitness is returned.

object can be utilised without the need for recursion or modification of the individual structure, unlike conventional GP and CGP approaches which often struggle to evolve generic solutions without these more complex options.

Table 3.5: The operators supplied for every even parity test.

Operator Name	Input Types	Output Type	Description
AND	“bit,” “bit”	“bit”	AND of two bit values
OR	“bit,” “bit”	“bit”	OR of two bit values
NOT	“bit”	“bit”	NOT of a bit value (inverse)
NAND	“bit,” “bit”	“bit”	NAND of two bit values
NOR	“bit,” “bit”	“bit”	NOR of two bit values
ZERO	-	“bit”	0-arity function returning 0-bit value
ONE	-	“bit”	0-arity function returning 1-bit value
INP	-	“bit”	Returns input at the current counter index, and increments counter

Table 3.6: Additional operators supplied for multi–type even parity tests.

Operator Name	Input Types	Output Type	Description
NEWARR	-	“array”	0-arity function returning empty array
INPARR	-	“array”	0-arity function returning an array containing the input as separate ints
ZERO_2	-	“int”	0-arity function returning the value 0 as an int
ONE_2	-	“int”	0-arity function returning the value 1 as an int
INPLEN	-	“int”	0-arity function returning the input length
FALSE	-	“bool”	0-arity function returning the false boolean value
TRUE	-	“bool”	0-arity function returning the true boolean value
BIT2BOOL	“bit”	“bool”	Casts a bit to a bool. 0 = false, 1 = true
BIT2INT	“bit”	“int”	Casts a bit to an int
MOD	“int,” “int”	“int”	Modulus of two ints
ARRPUSH	“array,” “int”	“array”	Appends an int to an array
COUNTNONZERO	“array”	“int”	Returns number of non–zero elements in array
ADD	“int”, “int”	“int”	Returns the sum of two ints
SUB	“int,” “int”	“int”	Subtracts one int from another and returns the result
EQ	“int,” “int”	“bool”	Returns true if two ints are equal
GT	“int,” “int”	“bool”	Returns true if the first int is greater than the second
SUMARR	“array”	“int”	Returns the sum of the elements of an array

Table 3.7: Results from single type configuration.

input length	mean gens	s.d gens	% solved	% generic
3	444.34	41.26	85	0
4	481.72	19.30	60	0
5	—	—	0	0
6	—	—	0	0
7	—	—	0	0
8	—	—	0	0

3.4.1 Results

Tables 3.7 and 3.8 contain statistics on the number of generations required to solve the problem and how often each configuration was able to evolve a program that solved the problem. As NSGA–II was used, the output from evolution is a Pareto front of individuals; for the purposes of these results, the individual with the best fitness score is chosen. The most striking result is the improvement in the number of cases for which a solution could be evolved when multiple types are used. With multiple types, a working solution was always generated, whereas with only the "bit" type, this was only the case for short bit-strings. With a single type, no successful solution to bit-strings of length 5 or greater was found in any of the 100 runs. Moreover, there was a major reduction in the number of generations required to evolve a solution when using multiple types. This reduction is clearly a consequence of the more sophisticated functions available with multiple types.

It is also interesting to observe how often the solution evolved was generic, able to work on a bit-string of any length. For a length of 6 or greater, ST–CGP *always* evolved a generic solutions; while for shorter lengths, length-specific solutions were found in a proportion of cases.

Table 3.9 presents results on the even parity problem from previous GP studies. It can be seen that ST–CGP matches or outperforms all these previous state-of-the-art approaches, especially on the higher input lengths. This comparison strengthens the notion

Table 3.8: Results from multiple type configuration.

input length	mean gens	s.d gens	% solved	% generic
3	7.64	6.82	100	91
4	20.66	12.39	100	93
5	21.96	17.50	100	99
6	21.18	18.90	100	100
7	21.21	18.25	100	100
8	22.84	19.44	100	100

that ST–CGP is a worthwhile technique.

It should be noted that, while the runs with the larger primitive set were more successful, this is likely due to the added ability for the resulting program to count the number of bits (and it is especially important to acknowledge that many of the studies referenced in 3.9 did not have access to such a function). As mentioned earlier, the difficulty in the parity problem lies in each input bit being treated individually—when one flips, the output flips too. When the input bits are able to be summed/counted, this simplifies the challenge considerably. However, this also serves to prove the point: giving ST–CGP access to such primitives means that it is able to solve problems more easily. If the goal of ST–CGP is to have “general problem solving ability,” then it makes sense to give as strong a set of priors as possible; for example, a sum function for bits, or a suite of computer vision operators for CV tasks.

Table 3.9: Results from previous studies.

Input Length N	Standard GP [41]	GP with ADFs [111]	GP with LEF [112]	SMCGP [110]	ST–CGP
N <5	Solvable	Solvable	Solvable	Solvable	Solvable
5	Solvable	Solvable	Solvable	Solvable	Solvable
6	Not solvable	Solvable	Solvable	Solvable	Solvable
7	Not solvable	Solvable	Solvable	Solvable	Solvable
8	Not solvable	Solvable	Not consistently solvable	Solvable	Solvable
9	Not solvable	Solvable	Not solvable	Solvable	Solvable
10	Not solvable	Solvable	Not solvable	Solvable	Solvable
N >10	Not solvable	Not consistently solvable	Not solvable	Solvable	Solvable

Chapter 4

Application: Computer Vision

Chapter 3 provided a detailed examination of the theory and implementation of ST–CGP. Having established this foundation, this chapter demonstrates the application of ST–CGP to a real-world problem. The key advantage of strong typing is that it enables us to tackle complex problems involving diverse data types, allowing rapid adaptation to different problem domains whilst maintaining semantic correctness throughout the evolutionary process.

Given the author's background in computer vision, it seemed natural to evaluate whether ST–CGP could effectively address problems within this domain. Computer vision presents a particularly challenging problem space, primarily due to the high dimensionality of visual data. Even a relatively small image of 512×512 pixels contains over $262,000$ individual pixel values, each of which may have multiple "channels" of information (typically three), each potentially contributing important information to the overall task. This dimensional complexity requires computational approaches that can extract meaningful features whilst managing the large search space effectively.

Whilst searching for an appropriate real-world application, the author discovered a dataset containing images of thin blood smears treated with Giemsa staining–a standard technique used in the clinical detection of malaria parasites. This dataset was accompanied by a study investigating the use of pre-trained convolutional neural networks for detecting infected cells [113]. This represented an ideal candidate for testing ST–CGP's computer vision capabilities, as it provided both a well-established dataset and state-of-the-art performance benchmarks against which ST–CGP's results could be systematically compared.

The chapter is structured as follows: section 4.2 describes the experimental setup, with results presented in 4.3. First, however, the necessary operators and adaptations to allow ST–CGP to support computer vision problems are discussed in 4.1.

4.1 Adapting ST–CGP for Computer Vision

Computer vision represents a challenging and complex problem domain within artificial intelligence research. The field encompasses several distinct yet interconnected categories of computational tasks, including image segmentation (the process of distinguishing foreground elements from background regions), whole-image classification, object detection, and object-level classification. These problem categories frequently need to be used in conjunction with each other, operating in hierarchical relationships, where successful execution of higher-level tasks depends upon the completion of prerequisite lower-level operations. For instance, accurate object classification typically requires object detection to be performed first, which in turn often relies upon preliminary segmentation procedures to isolate regions of interest within an image.

A fundamental challenge in computer vision lies in developing computational methods that enable machines to extract meaningful information from visual data. This challenge is particularly pronounced when contrasted with other data modalities, such as tabular datasets, where the processing methodologies employed by humans and computers are often not too dissimilar. Visual information processing in humans is fundamentally different due to the sophisticated neural architecture of the human visual system, which has undergone millions of years of evolutionary refinement. Humans possess highly specialised capabilities for object detection and segmentation tasks, conceptualising visual scenes primarily in terms of discrete objects and their relationships and meaning. For example, when a human is presented with an image of a kitchen, there is an immediate *expectation* that there will be objects related to a kitchen present in the image. In contrast, computational vision systems encounter images as arrays of pixel values, with no prior knowledge or ability to think about what the image represents. As such, they require explicit algorithmic frameworks to extract higher-level semantic understanding from this low-level numerical representation.

A typed computer vision system can offer a practical advantage over an untyped CV system because realistic handcrafted CV workflows routinely involve a mix of representation types: they manipulate not only images but also intermediate values such as scalars (e.g., summary statistics), boolean masks, feature vectors, and geometric objects (e.g., contours/region descriptors). Many handcrafted CV feature-extraction and post-processing operators therefore consume and produce non-image types, and treating these values as first-class citizens simplifies pipeline construction and reuse. Systems such as ST–CGP, which support multiple types, and make the valid ways of connecting pipeline stages explicit, reduce type-mismatch errors and ambiguous operator usage. This can support richer mixed-type pipelines while improving interpretability, reliability and robustness.

The fundamental approaches to image interpretation differ markedly between human and artificial visual systems. Human vision operates predominantly through a *top-down* processing paradigm, whereby initial visual input undergoes rapid high-level analysis to identify salient features and regions of interest. Attention-based procedures subsequently direct detailed analysis towards these prioritised areas. This cycle repeats; an iterative and adaptive interpretation process that proves highly effective in natural environments. Conversely, contemporary computer vision systems, particularly convolutional neural networks which represent the current state-of-the-art in the field, employ a fundamentally different approach. These systems process *all pixel information simultaneously* across the entire visual field, propagating this data through sequential layers, each designed to extract increasingly complex features. Initial layers typically identify low-level features such as edges, corners, and basic geometric primitives, whilst subsequent layers combine these elements to recognise more complex structures including shapes and textures, before final classification layers produce semantic interpretations (classifying an image or pixel as belonging to a particular class). This computational process is entirely *feed-forward* and deterministic, ensuring consistent outputs for identical inputs. This contrasts with human visual processing, which operates as an iterative feedback system wherein interpretation emerges through multiple cycles of hypothesis generation and refinement.

Despite these fundamental differences in processing, parallels do exist between artificial and biological visual systems. Human visual processing is primarily performed by the occipital lobe, which contains functionally distinct regions responsible for different aspects of visual analysis. The Primary Visual Cortex (V1) specialises in processing low-level features including orientation-selective responses to lines and edges, whilst the Inferior Temporal (IT) cortex processes higher-level visual attributes such as texture, shape, and colour information. The Secondary Visual Cortex (V2) contributes to the semantic interpretation of visual stimuli. This hierarchical organisation resembles the layered architecture of convolutional neural networks, and whilst the underlying computational mechanisms differ substantially, both systems demonstrate the principle of hierarchical feature extraction, where elementary visual features are progressively combined to form increasingly complex representations capable of supporting high-level visual understanding.

4.1.1 Making ST–CGP understand images

Modern computer vision applications predominantly use convolutional neural networks trained on extensive datasets comprising thousands to tens of thousands of labelled examples per class. This approach relies upon the assumption that sufficient training data will enable the network to learn optimal weight configurations capable of performing the desired task, whether segmentation, classification, object detection, or other visual recognition challenges. The success of this methodology has largely replaced the traditional approach, establishing deep learning as the dominant paradigm in modern computer vision research.

Traditional computer vision methodologies, developed prior to the widespread adoption of deep learning techniques, followed fundamentally different principles rooted in explicit feature engineering. These approaches revolved around the identification and extraction of discriminative visual features from input images, following the three broad categories outlined above: shape, colour, and texture. For example, shape-based features such as edges and contours, colour-based features including specific colour ranges or pixel counts within defined colour spaces, and texture-based features characterising surface properties and patterns, can be used in combination to provide a very comprehensive summary of an image or region in an image. The computational workflow typically comprised three distinct stages: image preprocessing to ensure that input data was clean and as uniform as possible, systematic feature extraction using carefully designed algorithms, and subsequent classification based upon the extracted features. This three stage pipeline required substantial domain expertise to design effective feature extractors, but provided interpretable and controllable processing stages.

Given the demonstrated capability of ST–CGP—and genetic programming methodologies more broadly—to effectively replicate and frequently surpass human-written programs, the expectation is that similar performance advantages would result in computer vision applications if GP were to be encouraged to follow a human-like pipeline process. As such, a wrapper around ST–CGP was written, supporting two different pipelines designed to address different categories of computer vision problems. Pipeline 1 implements a direct whole-image classification approach, wherein ST–CGP receives images as input and produces class predictions without any explicit intermediate processing stages. Pipeline 2 employs a more sophisticated hierarchical architecture, generating two models which are run in sequence: the first model performs image segmentation followed by object detection on the segmented regions, whilst the second model receives the detected objects as input and executes object-level classification. This two-stage approach is akin to the biological and deep-learning processes described above.

Crucially, both pipeline configurations provide ST–CGP with an identical set of primitives. These operators operate on images, either returning information about those images or in many cases outputting a modified version of that image. This allows functional chaining of operators to perform complex transformations on images. This in turn allows ST–CGP the potential to evolve novel programs that may include “mini pipelines.” Indeed, the author has observed several instances where ST–CGP autonomously evolved sophisticated multi-stage procedures combining segmentation, object detection, and object classification operations within the whole-image classification pipeline. These emergent behaviours demonstrate the system’s capacity to discover complex visual processing strategies that mirror the hierarchical decomposition employed in traditional computer vision approaches, whilst potentially identifying novel algorithmic combinations that would not typically be considered by human programmers.

CGP has been successfully applied to image processing tasks in previous research [90, 114], and the image processing capabilities developed for ST–CGP share similarities with those implemented in CGP–IP [115], which aims to replicate human image processing methodologies using the OpenCV library. However, ST–CGP’s strongly typed architecture and support for varying function arities provide greater flexibility compared to CGP–IP, enabling the generation of more expressive programs beyond simple chains of image processing operators. For instance, a typical ST–CGP program might convert an image between colour spaces, calculate statistical measures, perform mathematical operations on those statistics, apply image thresholding based on the computed values, and subsequently execute additional calculations before producing the final output, demonstrating the enhanced compositional capabilities afforded by the system’s architectural design.

Image Operators

Table 4.1 shows the operators provided to ST–CGP. In the table, several data types are referenced. These are:

- **`IMG3`** - A three-channel image, such as an RGB image.
- **`IMG1`** - A single-channel image. This is the primary means by which features are

extracted and transformations are performed.

- **`FLOAT`** - A floating point number.
- **`INT`** - An integer.
- **`BOOL`** - A boolean .(`true/false`)
- **`COOC`** - A special type of single-channel image that represents a grey-level co-occurrence matrix, used for texture analysis.
- **`CONTOURS`** - An object that a list of detected contours, or shapes, within an image. This object is used by contour-processing functions in conjunction with an input image to either extract features or perform a transformation.

It is worth noting that the set of primitives supplied to ST–CGP is deliberately large. This is intentional: by providing a broad collection of priors that encode domain knowledge, we give ST–CGP the best opportunity to solve a wide range of CV problems. As discussed earlier in this chapter, many CV tasks benefit from combining shape, colour, and texture information; accordingly, the primitive set includes multiple operators targeting each of these measures, increasing the likelihood that suitable building blocks exist for the task at hand. Overall, the aim is to maximise ST–CGP's problem-solving ability by ensuring the search space contains rich, task-relevant components.

In addition to these image-specific primitives described, the standard suite of arithmetic, boolean, and array-based primitives described in Chapter 3 are also included for computer vision problems.

Table 4.1: The image operators provided to ST–CGP for all computer vision problems.

Operator Name	Category	Description	Input Types	Output Type
Otsu	Colour	Applies Otsu thresholding to produce a binary-inverse mask.	IMG1	IMG1
BinaryThresh	Colour	Applies a fixed binary threshold to the image at a given threshold value.	IMG1, FLOAT	IMG1
Mean	Colour	Computes the mean pixel value of the input image.	IMG1	FLOAT
SetBetween	Colour	Replaces pixel values in a given open interval (lower, upper) with specified constant.	IMG1, INT lower, INT upper, INT val	IMG1
SetBelow	Colour	Replaces all pixel values below a specified upper bound with a constant.	IMG1, INT upper, INT val	IMG1
SetAbove	Colour	Replaces all pixel values above a specified lower bound with a constant.	IMG1, INT lower, INT val	IMG1
Equalise	Colour	Applies histogram equalisation to enhance image contrast.	IMG1	IMG1
SetEqual	Colour	Replaces all pixels equal to a given value with another constant.	IMG1, INT eq, INT val	IMG1
ContourMean	Colour	For each contour, computes mean value within contour and fills it with that mean value.	IMG1, CONTOURS	IMG1
ContourRange	Colour	For each contour, computes (max – min) value within contour and fills contour with that difference.	IMG1, CONTOURS	IMG1

Table 4.1 continued from previous page

Operator Name	Category	Description	Input Types	Output Type
ContourMin	Colour	For each contour, computes minimum pixel value within contour and fills contour with that value.	IMG1, CONTOURS	IMG1
ContourMax	Colour	For each contour, computes maximum pixel value within contour and fills contour with that value.	IMG1, CONTOURS	IMG1
PixelwiseMax	Colour	Combines two single-channel images by taking the maximum value at each pixel.	IMG1, IMG1	IMG1
PixelwiseMin	Colour	Combines two single-channel images by taking the minimum value at each pixel.	IMG1, IMG1	IMG1
CountInRangeGlobal	Colour	Counts globally how many pixels lie within a specified [lower, upper] range and returns the count.	IMG1, FLOAT lower, FLOAT upper	FLOAT
InRange	Colour	Produces a binary mask by thresholding image pixels within [lower, upper].	IMG1, FLOAT lower, FLOAT upper	IMG1
CountInRangeContour	Colour	For each contour, counts number of pixels within [lower, upper] and fills contour with that count.	IMG1, FLOAT lower, FLOAT upper, CONTOURS	IMG1
BitwiseAnd	Colour	Computes bitwise AND of two single-channel images.	IMG1, IMG1	IMG1
BitwiseOr	Colour	Computes bitwise OR of two single-channel images.	IMG1, IMG1	IMG1

Table 4.1 continued from previous page

Operator Name	Category	Description	Input Types	Output Type
BitwiseNot	Colour	Computes bitwise NOT (inversion) of a single-channel image.	IMG1	IMG1
BitwiseXor	Colour	Computes bitwise XOR of two single-channel images.	IMG1, IMG1	IMG1
Invert	Colour	Inverts a single-channel image (alias for BitwiseNot).	IMG1	IMG1
LinearTransform	Colour	Apply linear gain and bias to the input image.	IMG1, FLOAT gain, FLOAT bias	IMG1
PixelwiseRange	Colour	Computes absolute difference (pixel1-pixel2) between two images at each pixel.	IMG1, IMG1	IMG1
HLS	Colour	Converts a three-channel image into HLS colour space.	IMG3	IMG3
HSV	Colour	Converts a three-channel image into HSV colour space.	IMG3	IMG3
RGB	Colour	Converts a three-channel image into RGB colour space.	IMG3	IMG3
i-IMG3	Other	For each pixel in each channel of a three-channel image, subtract the pixel value from an integer.	int value, IMG3	IMG3
IMG3-i	Other	For each pixel in each channel of a three-channel image, subtract an integer from the pixel value.	int value, IMG3	IMG3

Table 4.1 continued from previous page

Operator Name	Category	Description	Input Types	Output Type
i+IMG3	Other	For each pixel in each channel of a three-channel image, add the pixel value to an integer.	int value, IMG3	IMG3
i*IMG3	Other	For each pixel in each channel of a three-channel image, multiply the pixel value by an integer.	int value, IMG3	IMG3
i/IMG3	Other	For each pixel in each channel of a three-channel image, (protected) divide an integer by the pixel value.	int value, IMG3	IMG3
IMG3/i	Other	For each pixel in each channel of a three-channel image, (protected) divide the pixel value by an integer.	int value, IMG3	IMG3
i-IMG1	Other	For each pixel in a single-channel image, subtract the pixel value from an integer.	int value, IMG1	IMG1
IMG1-i	Other	For each pixel in a single-channel image, subtract an integer from the pixel value.	int value, IMG1	IMG1
i+IMG1	Other	For each pixel in a single-channel image, add the pixel value to an integer.	int value, IMG1	IMG1
i*IMG1	Other	For each pixel in a single-channel image, multiply the pixel value by an integer.	int value, IMG1	IMG1
i/IMG1	Other	For each pixel in a single-channel image, (protected) divide an integer by the pixel value.	int value, IMG1	IMG1

Table 4.1 continued from previous page

Operator Name	Category	Description	Input Types	Output Type
IMG1/i	Other	For each pixel in a single-channel image, (protected) divide the pixel value by an integer.	int value, IMG1	IMG1
NOOP	Other	No-op, simply returns any input provided.	ANY	ANY
GetChannel	Other	Gets a single channel from a three channel image.	IMG3, INT channel	IMG1
Chunkify	Other	Divides the input image into a list of non-overlapping blocks of specified width and height.	IMG1, INT width, INT height, INT discardRemainder, INT outputAsMask	ARRAY<IMG1>
FindContours	Shape	Finds contours (shapes) in the input image using OpenCV's built in contour finding routine.	IMG1	CONTOURS
Canny	Shape	Performs Canny edge detection using two thresholds and a gradient option.	IMG1, FLOAT thresh1, FLOAT thresh2, INT l2gradient (0 or 1)	IMG1
Erode	Shape	Erodes the image using a 3×3 structuring element of specified shape (type 0–2).	IMG1, INT type	IMG1
Dilate	Shape	Dilates the image using an elliptical structuring element of given radius.	IMG1, INT elemSize	IMG1
Opening	Shape	Applies morphological opening (erosion followed by dilation) with a 3×3 elliptical element.	IMG1	IMG1

Table 4.1 continued from previous page

Operator Name	Category	Description	Input Types	Output Type
Closing	Shape	Applies morphological closing (dilation followed by erosion) with a 3×3 elliptical element.	IMG1	IMG1
Blackhat	Shape	Applies black-hat morphological operation (closing minus original) to highlight small dark spots.	IMG1	IMG1
Tophat	Shape	Applies top-hat morphological operation (original minus opening) to highlight small bright spots.	IMG1	IMG1
Hitmiss	Shape	Applies the hit-or-miss morphological operation to detect specific patterns.	IMG1	IMG1
Gradient	Shape	Computes the morphological gradient (difference between dilation and erosion) to emphasise edges.	IMG1	IMG1
FillRoundness	Shape	Calculates contour roundness (area / enclosing-circle area) and fills contour pixels with that value.	IMG1, CONTOURS	IMG1
FillFormFactor	Shape	Calculates form factor $((4\pi \cdot area)/(perimeter^2))$ for each contour and fills it accordingly.	IMG1, CONTOURS	IMG1
FillExtent	Shape	Computes contour extent (area / area of minimum bounding rectangle) and fills contour with that value.	IMG1, CONTOURS	IMG1
FillArea	Shape	Fills each contour with its area value (number of pixels).	IMG1, CONTOURS	IMG1

Table 4.1 continued from previous page

Operator Name	Category	Description	Input Types	Output Type
FillPerimeter	Shape	Fills each contour with its perimeter length (arc length).	IMG1, CONTOURS	IMG1
FillNumPoints	Shape	Counts number of vertices in polygonal approximation of each contour and fills contour with that count.	IMG1, CONTOURS	IMG1
FillMeanDefectDepth	Shape	Computes mean convex-defect depth for each contour and fills contour with a scaled value.	IMG1, CONTOURS	IMG1
FillIsConvex	Shape	Tests whether each contour is convex; fills convex with 1 and non-convex with 0.	IMG1, CONTOURS	IMG1
FillM00	Shape	Computes zero-order spatial moment (area) for each contour and fills contour with that moment.	IMG1, CONTOURS	IMG1
FillM01	Shape	Computes first-order spatial moment M01 for each contour and fills contour with that moment.	IMG1, CONTOURS	IMG1
FillM02	Shape	Computes second-order spatial moment M02 for each contour and fills contour with that moment.	IMG1, CONTOURS	IMG1
FillM03	Shape	Computes third-order spatial moment M03 for each contour and fills contour with that moment.	IMG1, CONTOURS	IMG1
FillM10	Shape	Computes first-order spatial moment M10 for each contour and fills contour with that moment.	IMG1, CONTOURS	IMG1
FillM11	Shape	Computes mixed spatial moment M11 for each contour and fills contour with that moment.	IMG1, CONTOURS	IMG1

Table 4.1 continued from previous page

Operator Name	Category	Description	Input Types	Output Type
FillM12	Shape	Computes mixed spatial moment M12 for each contour and fills contour with that moment.	IMG1, CONTOURS	IMG1
FillM20	Shape	Computes second-order spatial moment M20 for each contour and fills contour with that moment.	IMG1, CONTOURS	IMG1
FillM21	Shape	Computes mixed spatial moment M21 for each contour and fills contour with that moment.	IMG1, CONTOURS	IMG1
FillHu1	Shape	Computes first Hu moment for each contour and fills contour with that value.	IMG1, CONTOURS	IMG1
FillHu2	Shape	Computes second Hu moment for each contour and fills contour with that value.	IMG1, CONTOURS	IMG1
FillHu3	Shape	Computes third Hu moment for each contour and fills contour with that value.	IMG1, CONTOURS	IMG1
FillHu4	Shape	Computes fourth Hu moment for each contour and fills contour with that value.	IMG1, CONTOURS	IMG1
FillHu5	Shape	Computes fifth Hu moment for each contour and fills contour with that value.	IMG1, CONTOURS	IMG1
FillHu6	Shape	Computes sixth Hu moment for each contour and fills contour with that value.	IMG1, CONTOURS	IMG1
FillHu7	Shape	Computes seventh Hu moment for each contour and fills contour with that value.	IMG1, CONTOURS	IMG1

Table 4.1 continued from previous page

Operator Name	Category	Description	Input Types	Output Type
FillRectangularity	Shape	Computes rectangularity ratio (max side / min side) of minimum bounding rectangle and fills contour.	IMG1, CONTOURS	IMG1
FillRatio	Shape	Computes contour perimeter / area ratio (clipped to [0,255]) and fills contour with that value.	IMG1, CONTOURS	IMG1
SobelX	Shape	Computes horizontal gradient (Sobel X) of input image.	IMG1	IMG1
SobelY	Shape	Computes vertical gradient (Sobel Y) of input image.	IMG1	IMG1
Sobel	Shape	Computes combined gradient magnitude by averaging Sobel X and Sobel Y.	IMG1	IMG1
Median3	Texture	Applies a 3×3 median filter to reduce salt-and-pepper noise.	IMG1	IMG1
Median5	Texture	Applies a 5×5 median filter to reduce salt-and-pepper noise.	IMG1	IMG1
MedianN	Texture	Applies a N×N median filter to reduce salt-and-pepper noise.	IMG1	IMG1
Gabor	Texture	Convolves the image with a Gabor kernel (sigma, theta, lambda, gamma, psi) to extract oriented texture.	IMG1, FLOAT sigma, FLOAT theta, FLOAT lambda, FLOAT gamma, FLOAT psi	IMG1

Table 4.1 continued from previous page

Operator Name	Category	Description	Input Types	Output Type
Gaussian3	Texture	Applies a 3×3 Gaussian blur with specified sigma to smooth the image.	IMG1, FLOAT sigma	IMG1
Gaussian5	Texture	Applies a 5×5 Gaussian blur with specified sigma to smooth the image.	IMG1, FLOAT sigma	IMG1
Bilateral	Texture	Apply a bilateral filter to the input image.	IMG1, FLOAT d, FLOAT sigmacolour, FLOAT sigmaspace	IMG1
MatCooc	Texture	Builds a normalised grey-level co-occurrence matrix using offsets (dX, dY).	IMG1, INT dX, INT dY	COOC
HaralickDissimilarity	Texture	Computes Haralick dissimilarity from a grey-level co-occurrence matrix.	COOC	FLOAT
HaralickContrast	Texture	Computes Haralick contrast from a grey-level co-occurrence matrix.	COOC	FLOAT
HaralickHomogeneity	Texture	Computes Haralick homogeneity from a grey-level co-occurrence matrix.	COOC	FLOAT
HaralickEnergy	Texture	Computes Haralick energy (Angular Second Moment) from a co-occurrence matrix.	COOC	FLOAT
HaralickEntropy	Texture	Computes Haralick entropy from a grey-level co-occurrence matrix.	COOC	FLOAT

Table 4.1 continued from previous page

Operator Name	Category	Description	Input Types	Output Type
HaralickInverseDifference	Texture	Computes Haralick inverse difference from a grey-level co-occurrence matrix.	COOC	FLOAT
L5E5	Texture	Convolves with a 5×5 L5E5 kernel (Laplacian of Gaussian variant) for edge/texture extraction.	IMG1	IMG1
E5L5	Texture	Convolves with a 5×5 E5L5 kernel (edge orientation filter) for edge/texture extraction.	IMG1	IMG1
L5R5	Texture	Convolves with a 5×5 L5R5 kernel (Laplacian-like operator) to highlight texture features.	IMG1	IMG1
R5L5	Texture	Convolves with a 5×5 R5L5 kernel (reverse of L5R5) to highlight texture features.	IMG1	IMG1
E5S5	Texture	Convolves with a 5×5 E5S5 kernel (edge-sensitive pattern) for directional texture extraction.	IMG1	IMG1
S5E5	Texture	Convolves with a 5×5 S5E5 kernel (structure-enhancement filter) for texture analysis.	IMG1	IMG1
S5S5	Texture	Convolves with a 5×5 S5S5 kernel (symmetric edge detector) for texture analysis.	IMG1	IMG1
R5R5	Texture	Convolves with a 5×5 R5R5 kernel (rotated Laplacian filter) for texture extraction.	IMG1	IMG1
L5S5	Texture	Convolves with a 5×5 L5S5 kernel (Laplacian combined with symmetric structure filter).	IMG1	IMG1

Table 4.1 continued from previous page

Operator Name	Category	Description	Input Types	Output Type
S5L5	Texture	Convolves with a 5×5 S5L5 kernel (symmetric structure combined with Laplacian).	IMG1	IMG1
E5E5	Texture	Convolves with a 5×5 E5E5 kernel (double edge detector) to accentuate edges and texture.	IMG1	IMG1
E5R5	Texture	Convolves with a 5×5 E5R5 kernel (edge-reverse operator) for directional edge enhancement.	IMG1	IMG1
R5E5	Texture	Convolves with a 5×5 R5E5 kernel (reverse edge operator) for directional edge enhancement.	IMG1	IMG1
S5R5	Texture	Convolves with a 5×5 S5R5 kernel (structure-reverse operator) for directional texture capture.	IMG1	IMG1
R5S5	Texture	Convolves with a 5×5 R5S5 kernel (reverse-structure operator) for directional texture capture.	IMG1	IMG1

4.1.2 Painter

To facilitate the creation of accurately annotated ground-truth datasets, the author developed an annotation tool designed to enable visual labelling of image regions through a “painting” interface. The painter annotation tool serves two purposes within the annotation pipeline, accommodating both segmentation and classification annotation within a single interface. For segmentation tasks, the tool enables users to delineate background and foreground regions at the pixel level, whilst for classification problems, it allows each foreground region to be “painted” with a different class label. Upon completion of annotation, the tool generates mask files that encode pixel-level class assignments in a format directly compatible with the computer vision pipelines. These mask files serve as the ground-truth labels for a dataset during model training and evaluation. Figure 4.1 shows a screenshot of painter’s user interface.

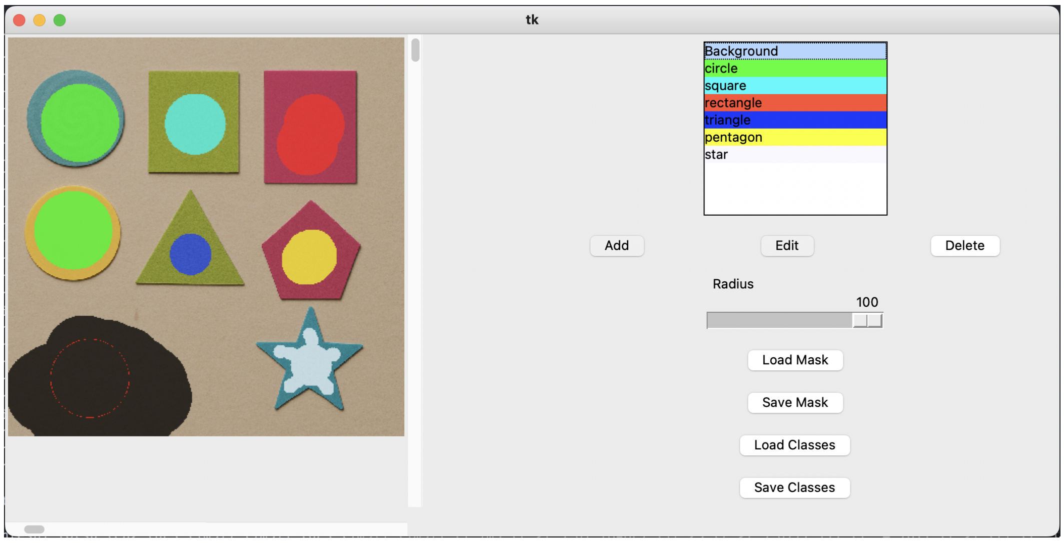


Figure 4.1: A screenshot of the painter interface.

4.1.3 Proof-of-concept toy problem

To ensure that ST–CGP was able to address simple computer vision challenges during the development phase, a controlled synthetic dataset was constructed to systematically evaluate the three distinct operator categories and both pipeline architectures. The dataset comprised a collection of synthetic images specifically designed to enable comprehensive testing of all operator types. Each image contained between four and nine geometric shapes, with each shape selected from a predefined set of six primitive shapes: circle, triangle, square, rectangle, pentagon, and star. The shapes were one of four colours: magenta, green, cyan, and yellow. These specific colours were deliberately selected to ensure that certain hues would activate multiple channels within the RGB colour space (for instance, magenta is represented as (255, 0, 255) in RGB coordinates), thereby introducing additional complexity to colour-based classification tasks.

Four distinct texture patterns were applied to the shapes: smooth (representing an absence of texture), light noise, heavy noise, and a “spiral” pattern. This combination of shape, colour, and texture attributes enabled independent evaluation of each visual feature, as well as assessment of operator performance when processing combinations of these characteristics. Simple classification tasks could be formulated based on individual attributes—such as “which shape is present?”—whilst more complex multi-attribute classification problems could be constructed, for example: “is this shape a lightly textured magenta star, or a smooth green rectangle?” To test ST–CGP, seven experiments were performed, with reduced numbers of some categories to keep runtime short:

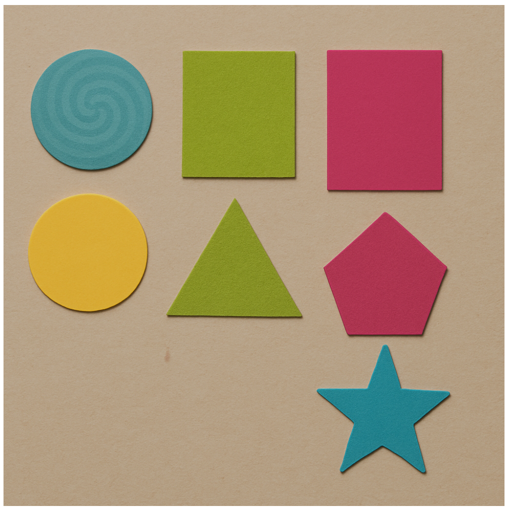
- Label by **colour only** (4 colours)
- Label by **texture only** (4 textures)
- Label by **shape only** (6 shapes)
- Label by **shape and texture** (3 shapes $\times$ 3 textures $=$ 9 labels)
- Label by **shape and colour** (3 shapes $\times$ 3 colours $=$ 9 labels)
- Label by **texture and colour** (3 textures $\times$ 3 colours $=$ 9 labels)
- Label by **shape, colour, and texture** ($3 \times 3 \times 3 = 27$ labels)

These tests were chosen because they allowed each attribute type to be evaluated both in isolation and in combination with other attributes. Real-world CV problems typically involve interactions between shape, colour, and texture, so it was important to verify that the system could handle tasks involving these attributes at a deliberately simplified level. Had the system proved unable to solve even these controlled problems, it would have suggested that a richer set of primitive operators was required to provide ST–CGP with the information necessary to discriminate between classes.

The dataset was intentionally crafted in such a way to challenge the ML process. Accurate shape classification often necessitates sophisticated combinations of different operator types due to the inherent similarities between certain shapes. Squares and rectangles, for instance, both possess four corners, requiring additional discriminative features beyond simple corner detection. Similarly, shapes such as squares, circles, stars, and pentagons of similar size have similar bounding box dimensions, necessitating more nuanced geometric analysis. Consequently, distinguishing between circles, squares, and rectangles requires examination not only of corner count, but also consideration of bounding box aspect ratios and the proportion of the bounding box area occupied by the shape itself. Colour classification presented additional challenges, as the yellow colour selected for shapes was intentionally similar to the background colour in the RGB, HLS, and HSV colour spaces. Whilst the human eye is able to tell apart the shapes from the background easily, the numerical values representing these colours are sufficiently close that standard automatic thresholding techniques, including Otsu's method and adaptive thresholding, prove insufficient for reliable foreground-background segmentation across all objects. Finally, to ensure that texture analysis remained appropriately challenging, a controlled amount of noise was introduced to all images, simulating the visual characteristics commonly encountered in real-world photographic data.

The two pipeline architectures were evaluated using slightly different dataset configurations. The multi-model pipeline was tested on a dataset comprising 100 images, each containing a random assortment of shapes with randomly selected colours and textures from the sets defined above. In contrast, the whole-image pipeline utilised a dataset of

500 images, with each image containing a *single shape* of randomly assigned colour and texture attributes. Figure 4.2 shows an example image from each dataset.


(a) An image from the multi-model pipeline dataset.

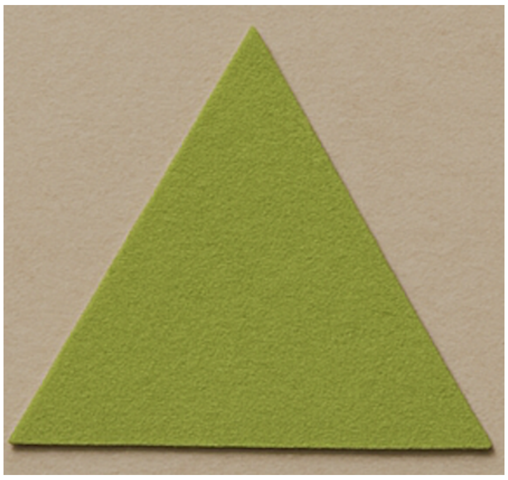

(b) An image from the whole-image pipeline dataset.

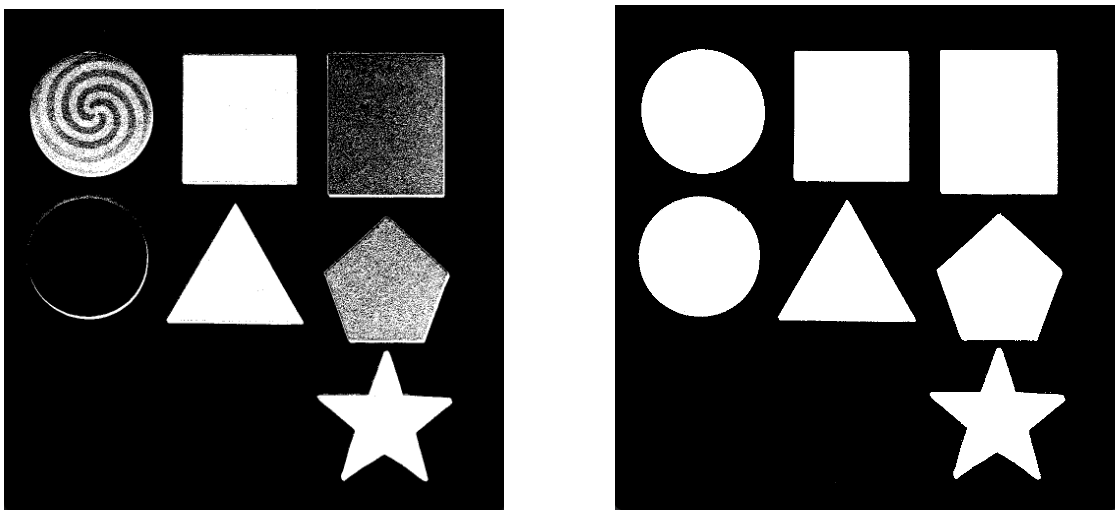
Figure 4.2: Sample images from both the multi-model and whole-image pipelines.

To demonstrate the operation of ST–CGP on this simple computer vision problem, let us consider a run of the multi-model pipeline. The first stage is segmentation: separating background from foreground. The segmentation stage produces a binary mask of the image, where white pixels denote foreground objects and black pixels denote background pixels. As mentioned, this is made more complex by the colour choices for the shapes in the dataset. However, ST–CGP was able to consistently (in every run!) evolve a segmenter that was able to perfectly segment the images, even when only provided with a single image as a training set. The hyperparameters used were the default set explained earlier, which are repeated in Table 4.2 for ease of access. Fig. 4.3 shows the segmentation mask produced by the best program from the first generation of a run, followed by the segmentation mask produced by the best program from the last generation. Fig. 4.4 shows the corresponding programs for the segmentation masks, in LISP form and then as Python pseudocode. It can be seen that the plain Otsu invocation, which was the best outcome of the first generation, is insufficient to discriminate between the yellow circle and the background. However, within a few generations, a more complex program had been evolved which could correctly segment all the shapes perfectly. The program achieves this by converting the colour space from RGB to HLS, after making a few small arithmetic adjustments to the pixel values. This is then followed by a few more arithmetic operations, before finally performing Otsu's thresholding on the third channel of the transformed image (the saturation channel).

Hyperparameter	Selected value
Population size	300
Node count	250
Max. Generations	500
Max. time	-1
Fitness threshold	1×10^{-6}
Use NSGA-II	true
Use full crossover	true
Use genetic rewiring	true
Number of subgrids	0
Subgrid size	N/A (not used)
Use function memoisation	true
Use individual caching	true

Table 4.2: The default hyperparameters.

Across the entire dataset for the multi-model pipeline, ST–CGP was able to perfectly segment every ground-truthed image. After this, the object detection and classification model was evolved. For this part of the pipeline, three examples of each class were provided (so, for shape based problems, three of each shape were provided, for colour, three of each colour, and so on). Fig. 4.5 shows the bounding boxes of all detected objects in the example image. It can be seen that the excellent segmentation results



(a) The segmentation mask produced by the best program from the first generation.

(b) The segmentation mask produced by the best program from the last generation.

Figure 4.3: Segmentation masks produced by the segmentation model in the multi-model pipeline.

allowed very easy detection of the objects in the image. Fig. 4.6 shows images which have been run through the classifier programs for two different problems: the first shows a “colour” based classifier, while the second shows a “shape” based classifier. It can be seen that in both cases, each different class has been coloured differently, thereby corresponding to different class labels.

Across all tests, for both pipeline architectures, ST–CGP was able to maintain an accuracy of 100% on both train and test sets. While this sounds high, the problems were relatively trivial compared to typical computer vision tasks, and were designed only to test that ST–CGP was functioning correctly across a diverse range of computer vision tasks. As such, this toy problem served its purpose, and was particularly useful during development of the different categories of image operators.

```
(Otsu IMAGE)
```

(a) Best segmentation program from the first generation.

```
(Otsu (GetChannel (IMG3-i (i+IMG3 (i+IMG3 (IMG3-i (HLS (i+IMG3
     (HLS (i+IMG3 (i+IMG3 (i-IMG3 (f->i 5.0) IMAGE) 5) 1)) (i-
   5 16))) 50) 4) 256) 50) 2))
```

(b) Best segmentation program from the last generation.

```
leaf12 = i-IMG3(5, IMAGE)
leaf11 = i+IMG3(leaf12, 5)
leaf10 = i+IMG3(leaf11, 1)
leaf9 = HLS(leaf10)
leaf8 = i+IMG3(leaf9, -11)
leaf7 = HLS(leaf8)
leaf6 = IMG3-i(leaf7, 50)
leaf5 = i+IMG3(leaf6, 4)
leaf4 = i+IMG3(leaf5, 256)
leaf3 = IMG3-i(leaf4, 50)
leaf2 = GetChannel(leaf3, 2)
leaf1 = Otsu(leaf2)
result = leaf12
```

(c) Best segmentation program from the last generation, in python pseudocode.

Figure 4.4: Segmentation programs generated by the segmentation stage of the multi-model pipeline.

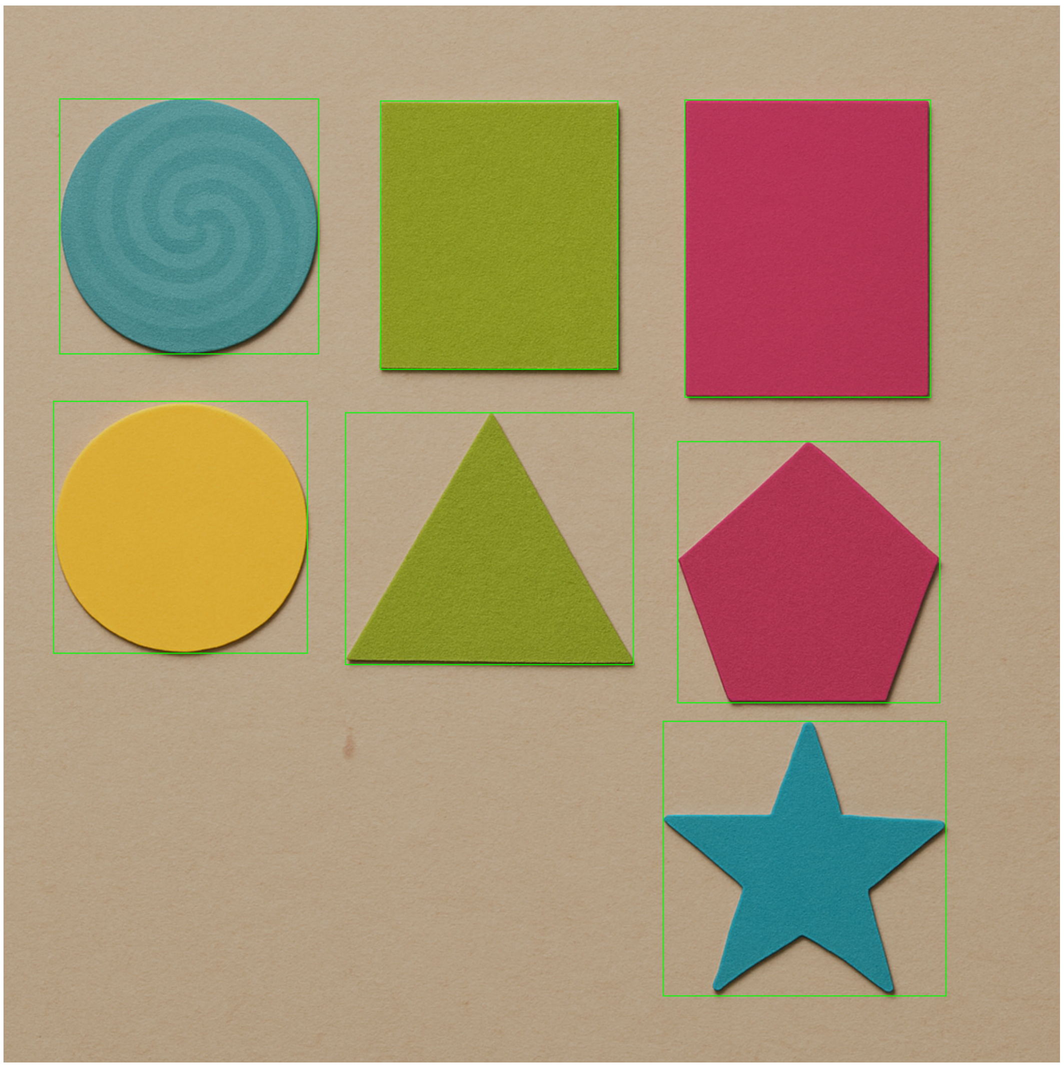
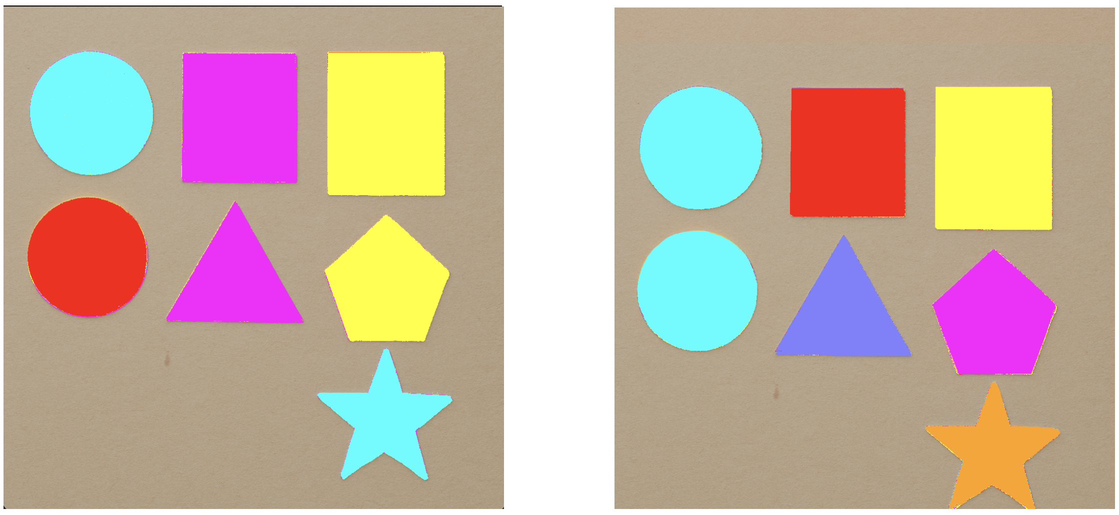


Figure 4.5: Bounding boxes detected using the segmentation mask, applied to the actual image.



(a) An image showing objects classified by colour.

(b) An image showing objects classified by shape.

Figure 4.6: Objects classified by colour vs objects classified by shape.

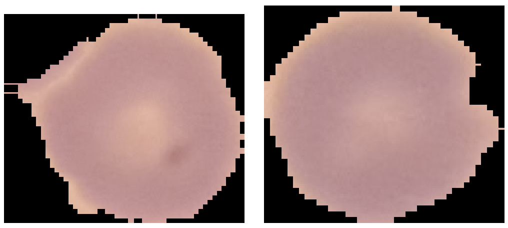
4.2 A Real-world Experiment: Detecting Malaria

Malaria is a serious tropical disease which causes millions of deaths worldwide every year. Many previous studies have tried various machine learning techniques to detect malaria in thin blood smears. A comprehensive review of such literature is available in [116], but most recent publications have used deep learning—specifically convolutional neural networks—to attempt to solve the problem.

4.2.1 Dataset

To test whether ST–CGP could approach the performance of convolutional neural networks, the author used the dataset provided by the Lister Hill National Center for Biomedical Communications, of the National Institute for Health, used in a recent set of studies [113, 117]. This dataset consists of 27,558 images, each containing a single cell. The images are split equally between two classes: uninfected and infected, and are drawn from 193 patients, of which 148 were infected and 45 were uninfected. The images range quite considerably in dimension from 46 × 79 pixels to 394 × 291 pixels. As the images were pre-segmented by the authors of the study, the most appropriate pipeline for this task was the *whole-image* pipeline. As such, a segmenter was not evolved for this task.

Fig. 4.7 shows a few examples of infected and uninfected cells. A typical way to detect infected cells is to look for dark purple regions; the Giemsa stain used on the thin blood smears from which these images are extracted typically stains the parasitic regions dark purple. However, this method is not foolproof: some infected cells do not show a dark purple region, and conversely, some uninfected cells can have darker regions that do not necessarily relate to an infection. As such, human categorisation of these cells is a specialised job requiring extensive training and expertise, and the diagnostic accuracy can still be impacted by inter-observer variability.


(a) Two uninfected cells.

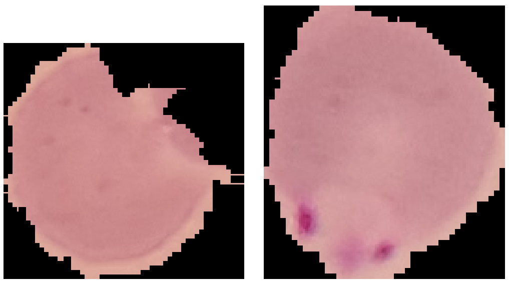

(b) Two infected cells.

Figure 4.7: Examples of uninfected and infected cells.

4.2.2 Methodology

The dataset was split into distinct training and test sets, and a series of experiments was conducted using varying quantities of training images per class. Four experimental configurations were tested: 5, 10, 100, and 500 training images per class. For each configuration, 100 independent runs were executed, with each run having the training images randomly selected from the training set using a different seed. The test set remained fixed across all runs and configurations and was balanced, comprising 10,000 images per class. Metrics were calculated from the predictions on the test set for each run and were aggregated across all runs for the final analysis.

For this whole-image classification task, each ST–CGP individual was configured with a single output type: `BOOL`, establishing a binary decision framework for Malarial infection detection. Upon program execution, an output value of *true* indicated that the analysed cell exhibited characteristics consistent with infection, whilst a value of *false* signified that the cell was classified as free from infection, thereby creating a straightforward binary classification system for automated pathological assessment.

The default hyperparameters were once again used for this experiment (Table 4.2. The entire suite of operators discussed earlier was provided to ST–CGP. This includes: arithmetic, array-based, boolean, and image-based operators. In total, this operator library comprised approximately 130 functions, providing ST–CGP with an extensive repertoire of primitives from which to construct solutions.

4.3 Results

Table 4.3 presents the experimental results obtained across varying training set sizes, encompassing performance statistics for mean accuracy, sensitivity, specificity, F1-score, and Matthew's Correlation Coefficient (MCC), as well as the standard deviation of each across all runs. For comparative analysis, the optimal results for each metric reported in [113] are included as benchmark values. This study was chosen because it is the original study that produced the dataset. Subsequent studies have extended the dataset, but the extended datasets were not available to the author while producing this research.

The experimental findings demonstrate that utilising five training images per class yielded the poorest performance across all evaluated metrics, establishing this configuration as the baseline for comparison. Increasing the training set size to ten images per class improved performance across all metrics. However, further increases in training set size beyond ten images per class did not yield commensurate gains. In particular, the configuration with 500 training images per class exhibited slightly lower mean performance across all metrics than the 10- and 100-image-per-class configurations (Table 4.3).

Differences in MCC between the 5-training-image-per-class configuration and the other configurations **were** statistically significant (Table 4.4), indicating that increasing the training set beyond 5 images per class yields a statistically reliable improvement in MCC. In contrast, the differences between the 10-image-per-class configuration and the 100- and 500-image-per-class configurations **were not** statistically significant after Holm correction ($p_{\text{Holm}} > 0.05$ for all pairwise MCC comparisons), suggesting that MCC performance largely saturates by 10 training images per class in this experimental setting. In practical terms, increasing the number of training images per class beyond 10 did not produce a statistically detectable improvement in MCC on the fixed test set, and thus 10 images per class appears to provide a good trade-off between annotation effort and performance.

Comparative analysis with the original study that provided the dataset reveals that the experimental configurations utilising 10, 100, and 500 training images per class

Table 4.3: Per-config performance over 100 independent runs (mean ± SD).

Config	Accuracy	Sensitivity	Specificity	F1	MCC
5	0.945 ± 0.0110	0.951 ± 0.0167	0.939 ± 0.0169	0.945 ± 0.0155	0.890 ± 0.0216
10	0.966 ± 0.0128	0.963 ± 0.0244	0.971 ± 0.0217	0.967 ± 0.0125	0.934 ± 0.0250
100	0.968 ± 0.0118	0.964 ± 0.0167	0.970 ± 0.0175	0.968 ± 0.0113	0.935 ± 0.0234
500	0.965 ± 0.00924	0.962 ± 0.0134	0.968 ± 0.0140	0.966 ± 0.00917	0.930 ± 0.0189
Best result from [113]	0.959	0.960	0.972	0.959	0.917

Table 4.4: All-pairs MCC comparisons over 100 independent runs per config. Δ denotes the difference in mean MCC (B−A) across runs. 95% CIs for Δ are computed by bootstrap resampling over runs. Two-sided Welch *t*-tests are used for *p*-values, with Holm correction across all 6 pairwise comparisons. 'Significant?" indicates Holm-adjusted $p < 0.05$.

Pair (A vs B)	Δ (B−A)	95% CI for Δ	Welch t	p (raw)	p (Holm)	Significant?
5 vs 10	0.0440	[0.0377, 0.0502]	13.3	4.33×10^{-29}	1.73×10^{-28}	Yes
5 vs 100	0.0450	[0.0394, 0.0506]	14.1	1.09×10^{-31}	6.54×10^{-31}	Yes
5 vs 500	0.0400	[0.0342, 0.0456]	13.9	4.57×10^{-31}	2.28×10^{-30}	Yes
10 vs 100	0.00100	[−0.00560, 0.00750]	0.292	0.771	0.771	No
10 vs 500	−0.00400	[−0.0105, 0.00229]	−1.28	0.204	0.407	No
100 vs 500	−0.00500	[−0.0108, 0.000811]	−1.66	0.0980	0.294	No

outperform the previously reported results across the majority of evaluated metrics. This outcome is particularly noteworthy given the scarcity of instances in the literature where GP-based methodologies demonstrate superior performance compared to convolutional neural network-based approaches, which typically dominate computer vision benchmarks. The observed performance advantage is especially encouraging when considered in the context of the substantially reduced training data requirements, as the present study achieved competitive results with minimal training samples, whereas convolutional neural networks conventionally require thousands of training images per class to achieve optimal performance. Furthermore, the significance of these findings is amplified by the fact that the comparative study employed sophisticated methodological enhancements, including feature extraction techniques, extensive pre-processing procedures, and pre-trained model architectures, to achieve their reported results. The ability of the GP-based approach to exceed these performance levels whilst operating under more constrained conditions demonstrates the potential efficacy of evolutionary computation methods for image classification tasks in data-limited scenarios.

One plausible explanation for the competitive performance of ST–CGP compared to CNNs, despite having access to limited training examples, may be that ST–CGP has the opportunity to search a hypothesis space with stronger built-in priors. In this particular CV workflow, solutions are composed from a library of hand-designed operators which implicitly encodes domain structure and invariances that a CNN would need to learn directly from data; if these priors align with the task, fewer labeled examples may suffice (as discussed earlier in 4.1.1). In addition, GP can behave like simultaneous architecture discovery and feature construction/selection, explicitly optimising which operator compositions and derived features are used, rather than fitting a large end-to-end parameterisation. In simple terms, the learned programs may have substantially fewer effective degrees of freedom than typical CNNs, potentially reducing variance and overfitting despite the low levels of data.

4.4 Discussion

The experimental results are particularly encouraging and demonstrate significant potential for other practical applications. The capacity of ST–CGP to achieve superior performance compared to convolutional neural networks whilst utilising substantially fewer training images and operating without the computational requirements of expensive GPU hardware establishes it as a compelling alternative approach for certain computer vision applications. These characteristics suggest broader applicability to scenarios where computational resources are constrained, particularly given that ST–CGP can be trained effectively on comparatively low-cost hardware with substantially reduced power consumption requirements. Such attributes could make the approach particularly attractive for deployment in field-based applications where time-critical analysis is required and access to high-performance computing infrastructure may be limited or unavailable.

Furthermore, the solutions generated by ST–CGP offer a significant advantage over traditional deep learning approaches in terms of interpretability, as the resulting programs are not "black boxes" but rather produce human-readable code that can be analysed, understood, and potentially incorporated into larger computer vision systems where transparency is essential. Figure 4.8 presents an example of a classifier program generated during the experimental evaluation, illustrating the complex yet interpretable operations that ST–CGP can construct. This particular program demonstrates a sequence of operations including the selective utilisation of the red channel from the RGB representation of the input image, the application of conditional thresholding whereby pixels with values below 187 are reassigned to 157, the identification and processing of contours with each detected region filled using the minimum value found within that contour, the computation of the E5S5 texture measure across the processed image, the subsequent filling of each contour with the range of pixel values contained within it, and finally the application of a binary threshold at a value of 128. The resulting output constitutes a binary mask wherein white pixels indicate the presence of infected cells

```
threshold(
  cnt_range(
    E5S5(
      cnt_min(
        setbelow(im_r, 187.0, 157.0),
        im_contours)
    ),
    im_contours),
  128)
```

Figure 4.8: An example cell classifier generated by ST–CGP.

and black pixels denote healthy cells. This program effectively demonstrates both the expressive capabilities of ST–CGP within the computer vision domain and the significant advantage of producing interpretable solutions that enable researchers to comprehend *precisely how* the classification decision is reached, potentially informing future research directions in both computational and manual analysis.

Traditionally, research in GP (and CGP) has predominantly focused on the implementation of low-level computational operations, avoiding the incorporation of higher-level programming constructs such as iterative loops operating on data structures like arrays. However, the author proposes a fundamental reorientation of GP objectives, and evolutionary computation more broadly, from the conventional goal of problem-solving towards the more ambitious pursuit of "automatic programming," a perspective that has gained considerable support in recent publications [118, 119]. Human programmers characteristically operate at higher levels of abstraction, leveraging pre-existing function libraries and frameworks that provide access to complex functions and algorithms, thereby simplifying the programming process. The Malaria detection task demonstrates a similar idea: image processing and analysis functionality was realised through operators that encapsulate widely-utilised functions from the OpenCV library, providing high-level access to sophisticated computer vision algorithms. To conduct a meaningful evaluation of the potential for novel evolutionary computing techniques to generate solutions that are genuinely competitive with human-developed programs, it is essential that these computational methods be equipped with equivalent tools and resources to those available to human programmers. The results obtained in this experiment provide compelling evidence that such an approach can facilitate the evolution of solutions that demonstrate enhanced generalisability and adaptability compared to those produced using traditional low-level operator sets.

4.5 Conclusion

This chapter has demonstrated that ST–CGP extends beyond a purely pedagogical research exercise, exhibiting practical applicability to real-world computational tasks, including those of considerable complexity such as computer vision problems. The chapter presented a tool for the annotation of image-based datasets and successfully addressed a toy problem with synthetic data, yielding encouraging experimental results that validate the system's fundamental capabilities.

Furthermore, when applied to a genuine real-world task, ST–CGP demonstrated performance that proved competitive with established convolutional neural networks and deep learning approaches. This achievement is particularly noteworthy given that the target task is recognised as exceptionally challenging for human practitioners, whilst ST–CGP accomplished this level of performance utilising significantly fewer computational resources and requiring substantially smaller training datasets compared to conventional deep learning methodologies.

A recent study discovered that a small proportion of the data in the dataset used for this experiment was mislabelled. Unfortunately, time did not allow for these findings to be incorporated into the work for this PhD, and so the results presented here are obtained using the mislabelled dataset. Nonetheless, in future work, it would be interesting to see whether using a corrected dataset improves results. Perhaps the samples "incorrectly classified" by ST–CGP may actually have been correctly classified!

ST–CGP demonstrated an additional significant advantage over CNNs: the generation of interpretable programs. Machine learning researchers have long expressed concern regarding the opaque, black-box nature of neural networks, and the recent surge in artificial intelligence adoption, driven in large part by the widespread availability of large language models, has elevated this issue to a position of prominence within the research community. ST–CGP has demonstrated that, consistent with previous GP-based techniques, it can produce human-interpretable code that enables researchers and practitioners to understand the underlying decision-making processes. This interpretability provides substantial benefits, including the ability to validate that the system has learned appropriate patterns rather than exploiting spurious correlations, facilitating debugging and refinement of the evolved solutions, and enabling domain experts to assess whether the discovered relationships align with established theoretical understanding, thereby increasing confidence in the system's reliability and appropriateness for deployment in critical applications.

Chapter 5

Soil Health Analysis: Yield

Soil health plays a fundamental role in agriculture, serving as the foundation for food production and ecosystem sustainability. Healthy soil supports plant growth, regulates water, filters pollutants, and cycles nutrients, making it essential for maintaining both agricultural productivity and environmental resilience. However, soil degradation, resulting from intensive farming practices, deforestation, and climate change, has become a critical global issue. The degradation of soil quality, through loss of organic matter, compaction, erosion, and chemical contamination, threatens not only agricultural yields but also long-term food security and the health of natural ecosystems.

In response to these challenges, modern agriculture is increasingly focused on soil health management as a way to ensure sustainable farming practices. The ability to monitor and assess soil health accurately is crucial for informing land management decisions, optimising crop growth, and mitigating the damage caused by intensive farming methods. Traditional soil analysis methods, which often rely on laboratory testing of physical and chemical properties, can be time-consuming, expensive, and may fail to capture the dynamic nature of soil ecosystems. Alternatively, in-field methods of analysis can be used, but these are typically suffer from poor levels of accuracy. Consequently, there is a growing demand for innovative, real-time approaches to soil health monitoring that can capture both the complexity of soil conditions and the variability introduced by factors such as seasonality, moisture levels, and biological activity.

During the course of this PhD, the author was introduced to PES Technologies, a small startup which has developed an electronic nose sensor that can collect a gas fingerprint from soil samples over the course of five minutes, in the field. This device measures volatile organic compounds and other gases emitted by the soil, providing a rapid, non-invasive method of capturing information which can be useful in the prediction and management of soil health. However, the raw gas fingerprint data is complex, and requires sophisticated analysis to interpret.

A preliminary, proof-of-concept project sponsored by InnovateUK set out to investigate whether the gas fingerprint data produced by the PES sensor provided enough information to predict indicators of soil health. Ultimately, ST–CGP proved to be such a successful technique for analysing and interpreting these gas fingerprints that it has since been applied commercially by PES. The project as a whole was split into two phases. In the first, agronomists gathered soil samples along with ancillary data. Part of each sample was retained for use with PES's sensor while the remainder was subject to conventional biological, physical, and chemical analysis. This chapter focuses on the latter because it allows the establishment of the baseline performance that is to be expected of a machine learning system in predicting soil health, the kind of performance that is expected from today's technologies. The next chapter then explores how well machine learning, and ST–CGP in particular, is able to predict soil health using the data from PES's sensor.

Section 5.1 will cover the background and context of the project, while section 5.2 covers the experimental setup. Section 5.3 shows the results obtained from this initial study.

5.1 Background and Context

5.1.1 The Importance of Soil Health

Soil health refers to the condition of soil in terms of its ability to perform essential functions that support both agricultural productivity and broader ecosystem sustainability.

It is often understood as the capacity of soil to sustain plant and animal productivity, maintain environmental quality, and promote plant, animal, and overall ecosystem health [120]. Put simply, a healthy soil system is one that effectively regulates water, sustains plant growth by cycling nutrients, supports microbial and other biological activity, and resists erosion and degradation. Unlike a static resource, soil is a dynamic living system composed not only of minerals and organic matter but also of billions of microorganisms that drive key ecological processes [121]. The concept of soil health captures the holistic and long-term functionality of soil, beyond immediate fertility for crop production, highlighting its role in maintaining critical ecosystem services such as carbon sequestration, water filtration, and biodiversity support.

Soil health is a cornerstone of agricultural productivity, providing the essential conditions that allow plants to thrive. Healthy soil acts as a medium that supplies plants with key nutrients, water, and a stable structure for root development. The organic matter in healthy soils enhances nutrient cycling, ensuring that essential elements such as nitrogen, phosphorus, and potassium are made available to crops in forms they can readily absorb [122]. In addition to nutrient supply, well-structured soil with good porosity and texture allows water to infiltrate and be retained, preventing both drought stress and waterlogging. This balance is critical for the growth of crops and contributes to the long-term sustainability of agricultural systems [123]. Furthermore, healthy soil supports a diverse community of microorganisms, including bacteria, fungi, and other organisms that break down organic matter, fix nitrogen, and protect plants from pathogens [124]. This complex biological network within the soil helps build natural resilience, reducing the dependency on synthetic inputs like fertilisers and pesticides, and ultimately supporting more sustainable farming practices [125].

Soil degradation poses a significant threat to agricultural productivity and sustainability. When soil health deteriorates, the key functions typically performed by a healthy soil system are severely compromised, leading to reduced crop yields. For instance, compacted soils limit root growth and restrict water infiltration, which stresses crops during dry periods. Erosion strips away the topsoil—rich in nutrients and organic matter—

leaving behind less fertile land. As a result, farmers are forced to rely on chemical fertilisers and soil amendments to maintain productivity, increasing both costs and environmental impacts, such as nutrient runoff and pollution of water bodies.

Degraded soils are more susceptible to environmental stresses, such as droughts and heavy rainfall, which can exacerbate yield losses. This makes agriculture less resilient to climate variability, further threatening food security, especially in regions already prone to extreme weather conditions. In the long term, continuous degradation can lead to the "desertification" of arable land, rendering it unproductive and leading to significant economic and social challenges for communities dependent on farming. It therefore follows that maintaining healthy soils is crucial not only for immediate agricultural output, but also for the long-term viability and resilience of agricultural systems globally.

5.1.2 Traditional Soil Health Measurement Techniques

Assessing soil health is fundamental to sustainable agricultural practices and environmental management. Traditional soil health measurement techniques have long been the cornerstone of this assessment, relying on established methodologies to evaluate various soil properties. Understanding these conventional approaches provides a foundation for recognising their strengths and limitations, thereby highlighting the need for faster, cheaper solutions such as in-field testing using electronic nose sensors.

The process of collecting soil samples is vitally important in order to accurately assess soil health, given the inherent variability of soil properties across different locations, even those within the same field. A commonly employed method involves a technique known as "walking the W," where the sampler moves systematically across the field in a W-shaped pattern. At predefined points along this path, individual soil "sub-samples" are taken, with these sub-samples being combined at the end of the path to form a composite sample. This approach ensures that samples are taken from various points, which helps to mitigate the effects of localised variability and provides a more representative sample for a given field or agricultural plot. By averaging the properties of multiple subsamples, researchers can obtain a clearer picture of factors such as moisture content, nutrient distribution, and soil structure, which might otherwise be obscured by the natural fluctuations present in the soil. This meticulous sampling strategy is essential for producing reliable data, as it accounts for the complex nature of soil, thereby increasing the accuracy of subsequent measurements and analyses.

Assessing soil health comprehensively necessitates the evaluation of three primary measurement categories: physical, chemical, and biological. Physical measurements examine characteristics such as soil texture, structure, moisture content, and bulk density. These indicators provide essential information about the soil's ability to retain water, facilitate root growth, and support plant structures. Chemical measurements involve analysing parameters such as soil pH, nutrient concentrations (including nitrogen, phosphorus, potassium, and magnesium), and the presence of any contaminants or pollutants. These assessments are crucial for determining soil fertility, nutrient availability, and potential toxicities that could affect plant and microbial life. Finally, biological measurements focus on the diversity and activity of soil microorganisms, including microbial biomass, enzyme activities, and the presence of soil fauna such as earthworms and arthropods. These biological indicators are vital for understanding the soil's biological vitality, nutrient cycling processes, and overall ecosystem functionality. By integrating data from these three categories, researchers can obtain a holistic view of soil health, enabling more informed decisions regarding soil management and agricultural practices. Each category contributes unique insights, collectively ensuring a thorough and accurate assessment of the soil's condition.

While certain physical measurements of soil health can be conducted directly in the field using portable instruments or test kits, the majority of chemical and biological assessments require specialised laboratory facilities to ensure accuracy and reliability. Laboratory-based analyses are essential for precisely determining soil pH, nutrient concentrations, and the presence of contaminants, as these parameters require sophisticated equipment such as spectrophotometers, chromatographs, and mass spectrometers. Additionally, biological measurements, such as microbial biomass and enzyme activities, demand controlled environments and advanced techniques like DNA sequencing and highly controlled ignition of soil samples to accurately assess soil biodiversity and microbial functions. The complexity of these analyses means that they cannot be effectively performed outside of a laboratory setting, where environmental variables can be meticulously managed. Furthermore, laboratory procedures often involve intricate sample preparation processes, including drying, sieving, and chemical extraction, which are critical for obtaining valid and reproducible results. Expertise in soil science, chemistry, and microbiology is also imperative, as skilled technicians are required to operate the specialised equipment, interpret the data correctly, and ensure that the methodologies adhere to standardised protocols. Consequently, the reliance on laboratory-based analyses introduces logistical considerations, such as the need for proper sample storage and transportation to prevent degradation or contamination.

Given that the laboratory analyses required for the assessment of soil health are so specialised, it stands to reason that they would be both costly and time-intensive. Typically, the process from sample collection to the delivery of comprehensive analysis can span as many as two to three months. This prolonged timeframe poses substantial limitations for farmers, land managers, and agronomists who rely on timely data to make informed decisions about soil management practices. During this waiting period, soil conditions can undergo considerable changes due to factors such as weather fluctuations, crop growth cycles, and ongoing agricultural activities. As a result, the information provided by delayed laboratory analyses may no longer accurately reflect the current state of the soil, reducing its practical utility. Additionally, the lag in obtaining results makes it more challenging to implement corrective measures or amendments at the right time, potentially exacerbating soil degradation or nutrient imbalances. This delay can also can affect the scheduling of planting and harvesting activities, as decisions based on outdated data can lead to suboptimal crop performance and yield.

The financial implications of traditional laboratory-based soil health assessments significantly influences the frequency and extent of soil testing undertaken by farmers and land managers. While basic chemical analyses can be performed quite cost effectively, costs rise dramatically as biological analyses are introduced (or even just slightly less common chemical and physical analyses)[1] due to the complexity of the processes used, in addition to the costly reagents. These high costs can be prohibitive, especially for small-scale farmers or organisations with limited budgets, limiting their ability to perform regular soil health evaluations. As a result, comprehensive soil testing may only be conducted periodically or when specific issues arise, rather than as part of a routine monitoring framework. This infrequent testing approach can lead to delayed detection of soil degradation, nutrient deficiencies, or contamination, making it more challenging to implement timely and effective amendments. Additionally, the high cost per test discourages extensive spatial sampling across larger agricultural fields or multiple plots, potentially overlooking areas that may be experiencing different soil health conditions.

So, to conclude, while traditional soil health measurement techniques allow a comprehensive piture of soil health to be produced, the economic constraints imposed by traditional measurement techniques, as well as the time-intensive nature of both collecting the samples and analysing them, hinder the ability to achieve widespread and continuous soil monitoring, which is essential for maintaining sustainable agricultural practices and ensuring long-term soil fertility and productivity.

5.2 Experimental setup

This section outlines the experimental setup for the Innovate UK project. It details the data collection process, the rationale for the selected collection parameters, and the choice of chemical, physical, and biological assessments. Additionally, the section examines the machine learning training approaches explored during the study and evaluates their resulting accuracy.

[1]A standard NPK test can cost as little as £15-20, whereas a full suite of biological indicators can be upwards of £500.

5.2.1 Yield as a Proxy for Soil Health

The objective of this project was to develop a machine learning model that predicts soil health based on the analysis of collected soil sample data. As discussed, assessing soil health is inherently complex due to the multitude of interacting factors that influence its condition; soil cannot be categorically defined as simply "healthy" or "unhealthy" because variables such as nutrient content, moisture levels, organic matter, and microbial activity all contribute significantly to its overall state. Furthermore, the significant spatial variability displayed by soil mean that the characteristics that define a healthy soil in one field may not be directly applicable to another field, and even within a single field, different areas may display markedly different conditions. This complexity necessitates a careful approach to data analysis and model development, ensuring that the model can account for a wide range of influencing factors and spatial disparities.

A practical abstraction of soil health is to instead consider the specific area of the field from which a soil sample originates. It is expected that different areas within a field will exhibit different levels of soil quality, roughly correlating with the intensity of farming practices as well as other factors such as the influence of soil amendments, cultivation methods, etc. For instance, it is common for a field to contain an unfarmed area, such as a grass margin or hedgerow, known as the "reference zone" where, due to remaining undisturbed during the farming process, it is expected that the soil quality would be higher compared to the actively farmed regions. In contrast, soil samples taken from cultivated parts of the field can be indirectly assessed by examining the yield from those areas, under the assumption that higher yield generally correlates with better soil health; we logically expect that higher yielding areas will be healthier than lower yielding ones.

Based on these considerations, for the purposes of this project, each soil sample is assigned one of three yield labels: "low," "high," or "reference." These labels serve as a proxy for soil health, with yield acting as an indirect measure of the underlying quality of the soil. It is these labels that the machine learning models developed during this project are designed to predict.

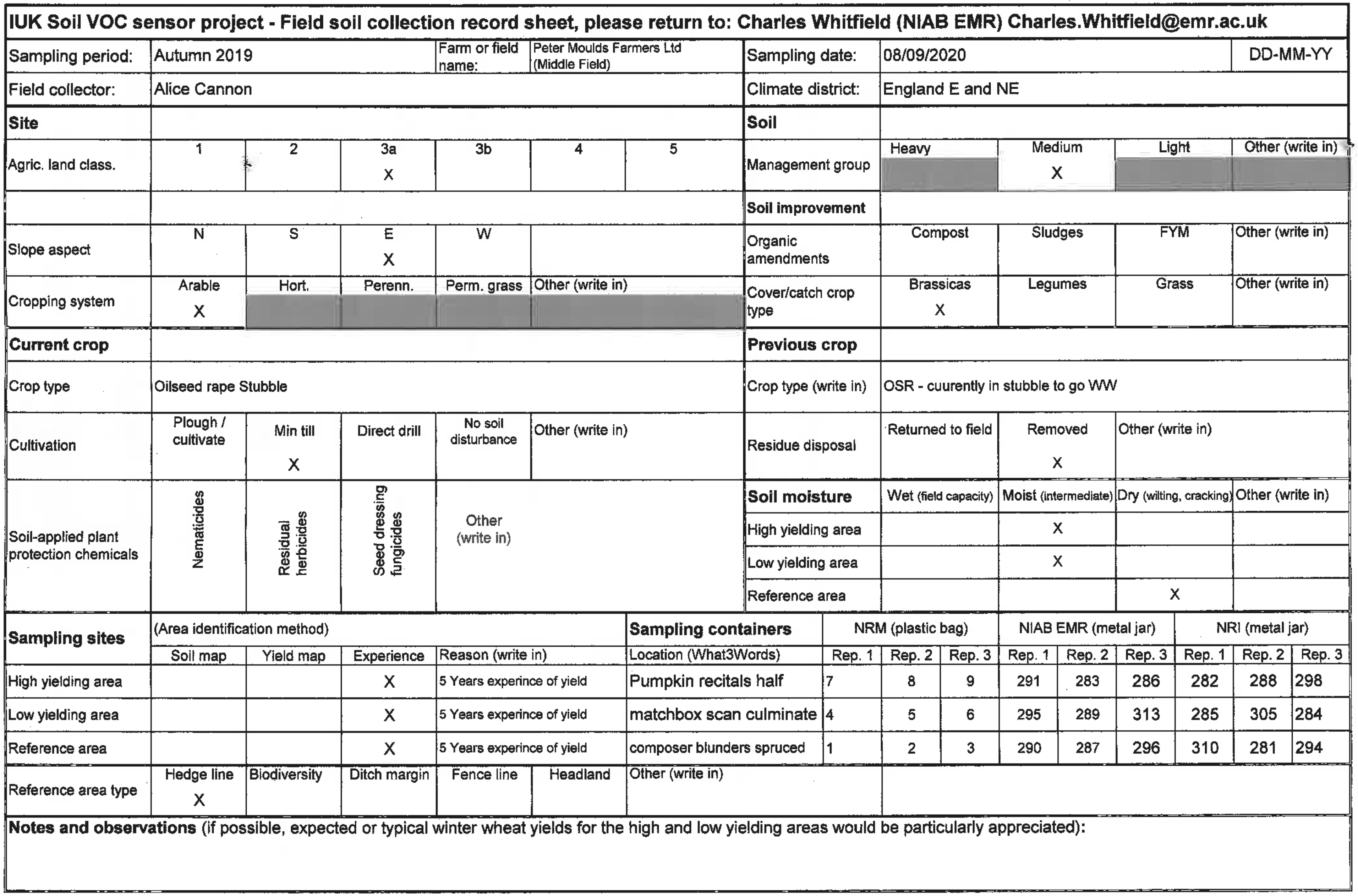
5.2.2 Data Collection

As explained earlier, conventional soil sampling methodologies are widely recognised as being both costly and time intensive. In light of these challenges, the data collection strategy for this project was carefully designed to balance the acquisition of detailed, informative data with the practical limitations imposed by available resources. This approach aimed to ensure that sufficient information could be gathered to support a rigorous analysis while maintaining cost efficiency across the study.

Data were collected from a total of 45 fields, with each field undergoing a structured sampling protocol aimed at capturing spatial variability. To allow the reliability (and error margin) of laboratory measurements to be measured, samples were taken in triplicate at each designated zone, thereby allowing for the calculation of measurement error. This method yielded nine data points per field—three per zone—resulting in an overall dataset comprising 405 data points. Although this dataset is modest in size relative to larger agricultural surveys and typical machine learning studies, it was considered adequate to serve as a proof of concept for the study's objectives.

It is worth noting that in order to accommodate the yield labelling methodology described earlier, the sampling protocol adopted in this project departs from the traditional "walking the W" method described earlier, where samples are collected at regular intervals along a transect and subsequently homogenised. Instead, the three aforementioned discrete locations within each field were sampled, a choice that offers an additional key advantage beyond allowing a direct comparison of soil health indicators across different yield zones; by refraining from homogenising the samples, the protocol allows the variation in biological activity between locations to be measured, acknowledging that this can vary significantly within a single field. Biological activity is perhaps the most important predictor of soil health, so understanding how this varies with yield could prove to be very useful for agricultural practice. However, while this approach increases the analytical rigour, it also leads to a higher number of samples per field (at least three, versus a single homogenised sample from a "W"), which, due to cost constraints, limits the overall number of fields that can be included in the study.

Each soil sample was collected by a trained agronomist following a detailed soil sampling protocol developed by one of the project's academic partners. In addition to the specific set of steps for the physical collection of the soil, the protocol required the recording of extensive metadata for each sample, including GPS coordinates, soil texture, moisture content, crop history, cultivation practices, and additional relevant parameters. This comprehensive data collection framework was designed with the hypothesis that such metadata could have significant implications for soil health, particularly regarding biological activity. By systematically recording these diverse factors, the study aimed to rigorously assess the predictive value of the metadata in relation to soil health indicators. Fig. 5.1 shows the data collection sheet used by agronomists during soil sample collection.



IUK Soil VOC sensor project - Field soil collection record sheet, please return to: Charles Whitfield (NIAB EMR) Charles.Whitfield@emr.ac.uk

Sampling period:	Autumn 2019	Farm or field name:	Peter Moulds Farmers Ltd (Middle Field)	Sampling date:	08/09/2020	DD-MM-YY
Field collector:	Alice Cannon			Climate district:	England E and NE	

Site						
Agric. land class.	1	2	3a	3b	4	5
			X			
Slope aspect	N	S	E	W		
			X			
Cropping system	Arable	Hort.	Perenn.	Perm. grass	Other (write in)	
	X					

Soil				
Management group	Heavy	Medium	Light	Other (write in)
		X		
Soil improvement				
Organic amendments	Compost	Sludges	FYM	Other (write in)
Cover/catch crop type	Brassicas	Legumes	Grass	Other (write in)
	X			

Current crop					
Crop type	Oilseed rape Stubble				
Cultivation	Plough / cultivate	Min till	Direct drill	No soil disturbance	Other (write in)
		X			
Soil-applied plant protection chemicals	Nematicides	Residual herbicides	Seed dressing fungicides	Other (write in)	

Previous crop				
Crop type (write in)	OSR - cuurently in stubble to go WW			
Residue disposal	Returned to field	Removed	Other (write in)	
		X		
Soil moisture	Wet (field capacity)	Moist (intermediate)	Dry (wilting, cracking)	Other (write in)
High yielding area		X		
Low yielding area		X		
Reference area			X	

Sampling sites	(Area identification method)				**Sampling containers**	NRM (plastic bag)			NIAB EMR (metal jar)			NRI (metal jar)		
	Soil map	Yield map	Experience	Reason (write in)	Location (What3Words)	Rep. 1	Rep. 2	Rep. 3	Rep. 1	Rep. 2	Rep. 3	Rep. 1	Rep. 2	Rep. 3
High yielding area			X	5 Years experince of yield	Pumpkin recitals half	7	8	9	291	283	286	282	288	298
Low yielding area			X	5 Years experince of yield	matchbox scan culminate	4	5	6	295	289	313	285	305	284
Reference area			X	5 Years experince of yield	composer blunders spruced	1	2	3	290	287	296	310	281	294

Reference area type	Hedge line	Biodiversity	Ditch margin	Fence line	Headland	Other (write in)	
	X						

Notes and observations (if possible, expected or typical winter wheat yields for the high and low yielding areas would be particularly appreciated):

Figure 5.1: The data collection sheet used by agronomists during sample collection.

Soil samples were collected across two seasons: spring and autumn. This approach reflects standard agronomic practice, as sampling and testing are typically conducted during these periods when soil conditions are more conducive to accurate (and useful!) analysis. In contrast, winter conditions render the soil excessively hard, with low or no biological activity, due to low temperatures, while summer conditions, characterised by high temperatures and low moisture levels, can lead to overly dry soils and inconsistent biological activity. The season in which the samples were collected was recorded, and serves as another piece of metadata to be used in analysis and training.

5.2.3 Soil Health Indicators

It turns out that defining soil health is a very tricky thing to do, and is something that requires a good deal of data. As alluded to earlier, these data typically comprise a set of "indicators" that, when taken as a holistic picture with metadata such as that collected using the data sheet in Fig. 5.1, can give an overall picture of soil health. For this project, the following set of indicators was chosen, as they cover physical, chemical, and biological analyses, and are sufficient to provide a comprehensive picture of soil health in combination with the metadata collected using the agronomists data sheet, and, of course, the interpretation of an expert agronomist.

Microbial Biomass

Soil microbial biomass refers to the living microbial population in soil. It plays many important roles in soil processes such as nutrient cycling and plant growth.

Respiration

The basal respiration rate of the microbial communities in the soil sample. This indicator measures the flux of CO_2 released from microorganisms in the soil during respiration. It is crucial for understanding soil ecosystems and creating effective management strategies for soil health and agricultural productivity.

pH

Soil pH is a measure of the acidity or alkalinity of the soil. Different plants have different pH requirements and it is important to maintain the optimal pH range for the specific plants being grown.

Organic Matter (Loss on ignition)

Soil organic matter percentage determined using the loss-on-ignition method. It is an important indicator of soil quality and fertility, and can influence plant growth, nutrient availability, and soil structure.

K Available

The portion of total soil potassium that plants can take up and use. Potassium availability is crucial for plant growth. However, excess levels of potassium can negatively affect plant growth as it interferes with uptake of other nutrients, particularly zinc, magnesium, iron, and calcium, causing small, curled foliage, thin stems, and acidic fruit (in fruiting crops).

Mg Available

The portion of total soil magnesium that plants can take up and use. Magnesium availability is crucial for chlorophyll production. However, much like potassium, too much magnesium can negatively affect plant growth by inhibiting calcium uptake, leading to stunted growth and foliage that is too dark.

P Available

The portion of total soil phosphorus that plants can take up and use. Phosphorus availability is crucial for root growth. Excess amounts of phosphorus can prevent plants from absorbing iron and zinc, causing yellowing of foliage and stunted growth.

Ammonium Nitrogen

The amount of ammonium (NH_4^+) ions that can be extracted from soil. Ammonium is a key source of nitrogen for plants. Too little can lead to stunted or slow growth, but too much can lead to toxicity and nutrient imbalances.

Nitrate Nitrogen

Also known as soil extractable nitrate, nitrate nitrogen refers to the amount of nitrate that can be extracted from the soil. It is important for determining the amount of fertilizer needed for crops, and high levels could lead to possible contamination of groundwater due to leaching.

Field water content

The current amount of water in a soil sample, presented as a weight percentage. This also allows field soil content to be calculated by subtracting the water content value from 100.

Water Holding Capacity water content

The upper limit of water capacity (WHC) for a given soil sample, presented as a weight percentage. A higher WHC value for soil denotes better moisture retention, while a lower WHC indicates improved drainage. Differences in soil composition typically result in differing WHC values; for instance, a sandy soil will hold less water than a soil with a high clay percentage.

Sand %

The percentage of the soil composition which is considered to be sand, measured using the laser diffraction method. Together with silt and clay percentage, this allows the soil texture to be categorised. Sand particles are defined as particles with a diameter between 2.00mm and 0.063mm.

Silt %

The percentage of the soil composition which is considered to be silt, measured using the laser diffraction method. Together with sand and clay percentage, this allows the soil texture to be categorised. Silt particles are defined as particles with a diameter between 0.063mm and 0.002mm.

Clay %

The percentage of the soil composition which is considered to be clay, measured using the laser diffraction method. Together with sand and silt percentage, this allows the soil texture to be categorised. Clay particles are defined as particles with a diameter less than 0.002mm.

Geomsin

A volatile organic compound (VOC) contained in soil. This is the VOC associated with the “smell” of fresh soil, as well as the smell released when rain falls on dry ground. Geosmin is produced by bacterial communities in the soil, and higher concentrations are generally associated with more biologically active soil (though this does not necessarily mean that soil is *healthier*).

5.2.4 Initial Analysis

Understanding the dataset is a crucial preliminary step in any machine learning task. Initial data analysis provides valuable insights into the structure and characteristics of the dataset, enabling an early assessment of its potential performance on a given problem. This preliminary exploration can reveal trends, anomalies, and patterns that suggest which features may prove most informative. Such analysis often highlights the strengths and limitations inherent to the datase, thereby guiding the selection of appropriate modelling techniques and feature engineering strategies.

Data visualisation represents an effective starting point for this initial analysis. As seen in Figures 5.2 to 5.4, charts depicting the relationship between yield and three key variables—Microbial Biomass, Extractable Nitrate, and Geosmin—offer a visual means of assessing potential correlations. Ideally, when a variable is plotted against the target label, distinct clusters should be visible that could indicate clear, predictive patterns. In these charts, however, such clusters are not evident. This lack of distinct groupings provides an early indication that these individual variables, when considered in isolation, may be insufficient to accurately predict the yield label. This observation suggests the need for an approach that may involve the integration of additional data (or metadata) or the application of more sophisticated feature selection techniques.

The next stage of the initial analysis involves examining the relationship between the input variables and the target label through correlation analysis. This analysis serves as an important step in identifying which variables exhibit strong associations with the target label, either in a positive or negative direction. When a variable shows a significant correlation, it is considered a promising candidate for predictive purposes, as it suggests that changes in the variable may be closely linked to changes in the target label. Conversely, variables that display little to no correlation are likely to contribute less to the predictive model. In this manner, correlation analysis can function as an informal

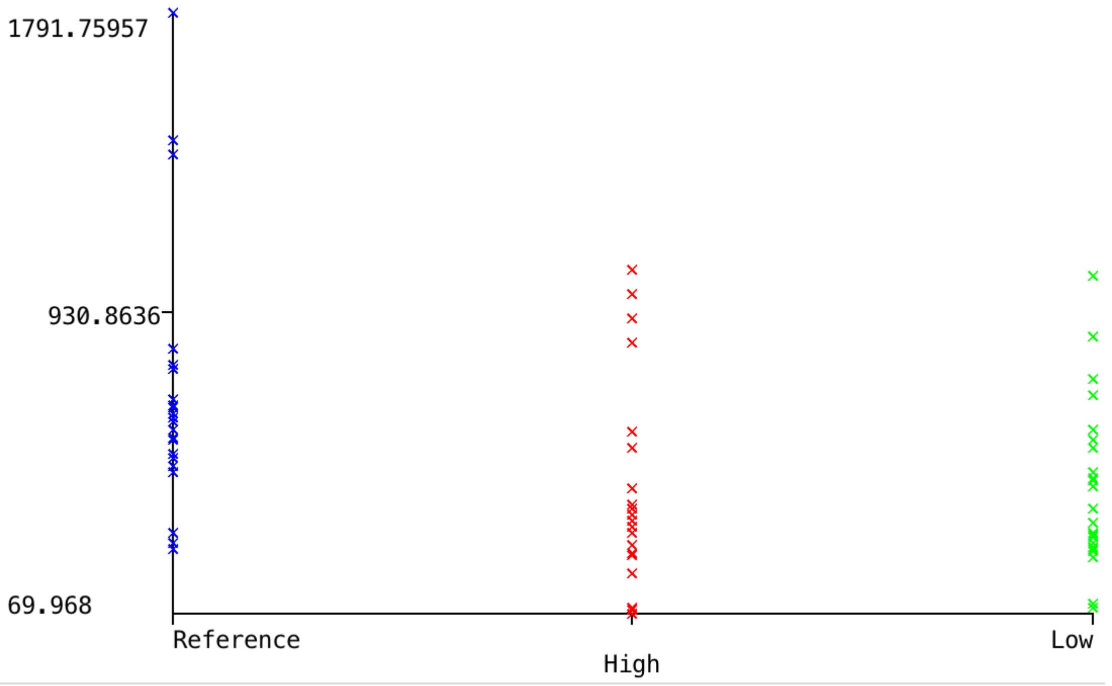


Figure 5.2: A cluster chart showing Yield vs Microbial Biomass.

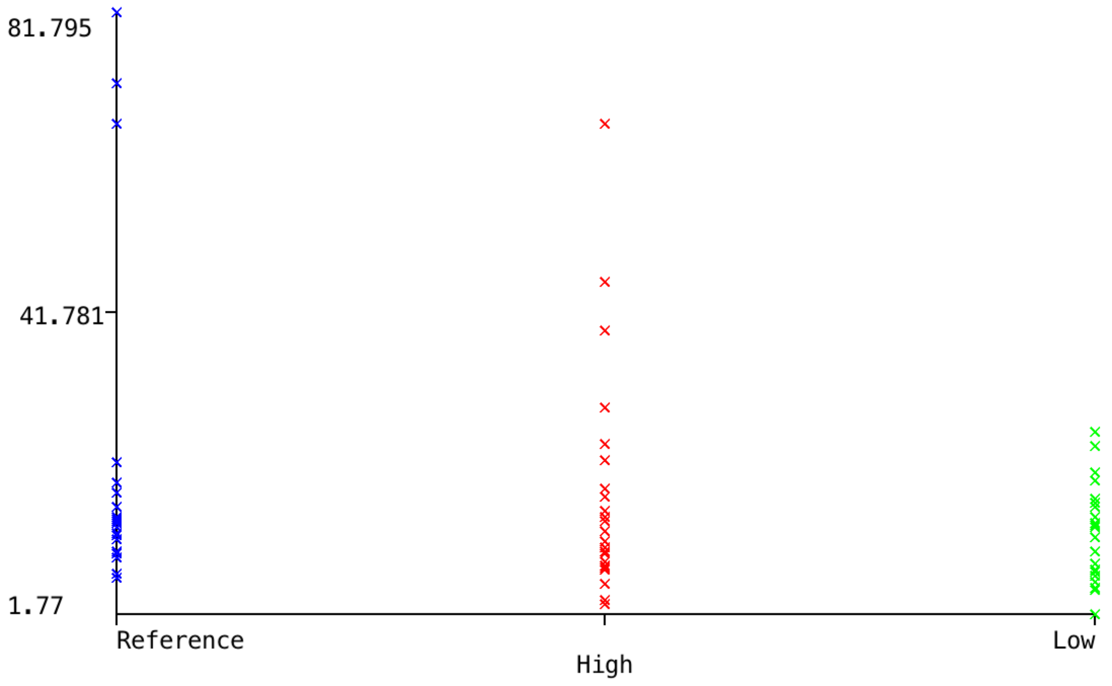


Figure 5.3: A cluster chart showing Yield vs Extractable Nitrate.

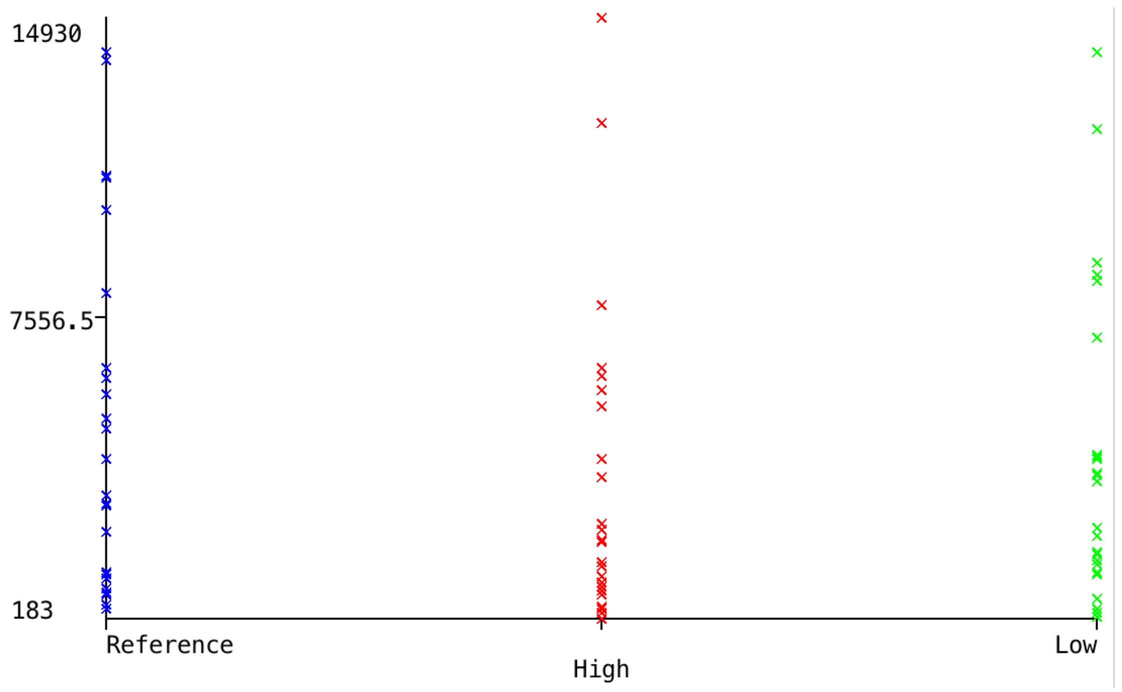


Figure 5.4: A cluster chart showing Yield vs Geosmin.

method of *dimensionality reduction*. By reducing the number of variables provided to a machine learning algorithm, "input noise" is reduced which can improve the results of the classification process. Those variables that maintain a strong association with the target—regardless of the direction of the correlation—are retained for further analysis, while those with negligible or weak correlations are excluded from subsequent modelling efforts.

Table 5.1 presents the Pearson's correlation coefficient for each variable with respect to yield. Note that the "water" and "soil" content for the Field and WHC variables have been merged, as the soil and water constituents of these readings sum to a total value, so the correlation is identical. It can be seen that microbial biomass exhibits the strongest correlation with yield, while geosmin shows the weakest correlation. However, even microbial biomass, the most strongly correlated variable, does not exhibit a particularly high correlation coefficient; a value of ≈ 0.25 is indicative of a weak correlation at best. This reinforces the notion that the variables collected in their raw state may not offer

Variable	Correlation Coefficient
Microbial Biomass	0.257 ± 0.022
100% WHC Soil/Water Content	0.223 ± 0.019
Field Soil/Water Content	0.196 ± 0.024
2-methyl-iso-borneol	0.136 ± 0.025
Extractable Nitrate	0.119 ± 0.021
Respiration-MeanCO2	0.112 ± 0.024
Extractable Ammonium	0.068 ± 0.025
Geosmin	0.072 ± 0.024

Table 5.1: Correlation coefficient of each variable wrt Yield.

sufficient predictive information for accurate yield classification, and that the inclusion of additional metadata is crucial to enhance model performance.

5.2.5 Pre-processing

Several pre-processing techniques were implemented and rigorously evaluated to ensure their suitability in the machine learning pipeline. Each technique was systematically applied to the data, and subsequent analyses compared the performance of the machine learning algorithms with and without the application of these techniques. This approach was adopted to verify that the introduction of any pre-processing step did not adversely affect the accuracy or reliability of the produced models.

Removal of incomplete variables

Several of the collected variables were incomplete, meaning that not every one of the 405 datapoints contained a recorded value for each variable. Although a small proportion of missing values is often not too much of a problem, variables with a majority of missing data may adversely affect the performance of machine learning models. To assess this impact, tests were conducted using datasets that included the incomplete variables, as well as datasets where these variables were excluded. The evaluations revealed that incorporating the incomplete variables did not enhance predictive performance; in several instances, performance metrics declined, and the computational runtime increased significantly. Consequently, it was determined that excluding the incomplete variables from further analyses would be the most effective strategy to optimise both model performance and efficiency. Including the lab analyses and the metadata, there were 11 complete variables for the dataset; the 8 variables seen in table 5.1, and three further pieces of metadata: soil moisture, soil texture, and the season in which the sample was collected. Moisture and texture were categorised broadly into three categories: for moisture, “dry,” “moist,” and “wet;” and for texture, “light,” “medium,” and “heavy” were the categories chosen, as they are relatively easy for even untrained samplers to determine.

Normalisation

Continuous variables often span extensive ranges that differ significantly among variables. Machine learning algorithms generally exhibit improved performance when the scale of each variable is the same. Normalisation is the process of transforming continuous variables from their original ranges to a predefined scale, most commonly 0–1, in an attempt to mitigate the influence of different scales. This technique involves applying a transformation to each variable based on its minimum and maximum values, ensuring that all values are proportionately scaled within the chosen range.

Such standardisation usually helps because it reduces the dominance of variables with larger numerical ranges and ensures that each variable contributes equally during model training. By aligning the scales of the variables, the normalisation process not only facilitates more stable and efficient convergence but can also improve the overall accuracy of the machine learning model. This approach is particularly beneficial when dealing with datasets where the variability in measurement units is quite pronounced, something which is true of the dataset used in this project.

Principal Component Analysis

Principal Component Analysis (PCA) is a common method of dimensionality reduction, similar in purpose to the correlation-based feature selection performed earlier. Unlike approaches that simply discard certain variables, PCA transforms the original set of variables by combining them to form new variables known as *principal components*. These components are constructed in such a way that they capture the maximum possible amount of the variance within the data, given an allowable number of principal components. In this project, PCA was applied to transform data originally comprising the 11 complete variables into 6 principal component variables; so chosen because 6 components explained $> 95\%$ of the variance in the original variables.

Data Augmentation

Data augmentation refers to the process of creating additional data records by duplicating existing ones and introducing subtle modifications. This technique is particularly useful when the available dataset is limited, as is the case here. In this project, machine learning techniques were evaluated with varying levels of data augmentation, with a maximum of one synthetic record generated for each original record. The rationale behind this limit is that generating more synthetic records may lead to an over-representation of augmented data, which can undermine the reliability of the analysis. Excessive augmentation can introduce biases and distort the patterns present in the original data, potentially impairing the model's ability to generalise effectively. Augmentation was performed by adding small Gaussian perturbations to each continuous input feature. Specifically, each continuous feature had zero-mean noise injected with a standard deviation equal to 1% of that feature's own standard deviation.

5.2.6 Tests

The analysis was based on three distinct sets of variables: the original dataset variables, those derived from principal component analysis (PCA), and a combination of both.

Each overall configuration (of pre-processing, hyperparameters, and set of variables) was tested 10 times to ensure that the results were representative and to reduce variability due to stochastic effects. Due to the limited sample size, k-fold cross-validation was used with $k = 5$ for each test in order to obtain a more robust estimate of generalisation performance. Metrics were computed as the mean across all k folds, and then averaged across the 10 repeated runs.

A comprehensive evaluation of multiple machine learning techniques was conducted in this project. For each algorithm, all combinations of the aforementioned pre-processing methods were evaluated to identify the best-performing pipeline. Hyperparameter optimisation was performed using k-fold cross-validation. Because hyperparameters were tuned and performance was estimated using the same cross-validation procedure (i.e., without nested cross-validation), the resulting cross-validated scores may be optimistically biased due to selection over many configurations. Nevertheless, since all models were optimised under an identical protocol, the relative comparisons between methods remain informative. The algorithms were selected to span a range of modelling paradigms and inductive biases, including linear probabilistic classifiers, non-parametric instance-based learning, single-tree and ensemble tree-based methods, probabilistic graphical models, neural networks, and an evolutionary/program-synthesis approach. The techniques examined were as follows:

- ST–CGP.
- Random Forest Classifier.
- Logitboost.
- K-Nearest-Neighbours Classifier.
- Multilayer Perceptron Neural Network.
- J48 Decision Tree.
- Logistic Regression.

- Bayesian Network.

Table 5.2 shows the hyperparameter search space for each algorithm. To ensure fair comparison between methods, hyperparameter search was conducted under a fixed computational budget: for each algorithm, up to $N = 1000$ hyperparameter configurations were evaluated, subject to a maximum runtime of two hours per algorithm (with most methods completing earlier). All algorithms were evaluated using the same cross-validation protocol, repetition count, pre-processing options, feature-set variants, and performance metrics. Additionally, ST–CGP was used with a single type, so that a fair comparison could be made with untyped algorithms. Most features were continuous, with a few categorical features which were encoded to be integers. As such, the behaviour of ST–CGP was largely similar to untyped CGP for this experiment, though with the uncommon addition of crossover and the additional *genetic rewiring* technique.

5.3 Results and Discussion

Table 5.3 shows the best performing run for each of the tested machine learning techniques. For each technique, the table shows the accuracy, the F-Measure value and Matthews Correlation Coefficient (MCC). The F-measure is a measure commonly used to compare performances of classifiers. A higher score indicates a better classifier. Matthews Correlation Coefficient shows the extent to which a classifier is able to classify a given sample compared to chance—a value of 0 means the classifier is no better than chance, while the closer to 1 the value is, the more likely it is to correctly predict the result. MCC is generally considered to be a better metric than accuracy because it takes into account class weightings; if one class is very over- or under–represented the accuracy metric can be thrown off, while MCC is less affected.

It can be seen from the table that the random forest produced the best performing classifier, yielding a maximum accuracy of 65%, an F-Measure of 0.615 and a Matthews Correlation Coefficient of 0.472. However, this technique was also one of the most variable in performance: the standard deviation of the performance was nearly 3%. ST–

Algorithm	**Parameter**	**Min Value**	**Max Value**
Random Forest	Number of Trees	10	500
	Number of Features	1	30
	Maximum Depth	5	20
LogitBoost	Number of Iterations	10	100
	Shrinkage	0.1	1.0
	Weight Threshold	50	200
KNN	Number of Neighbours	1 (odd only)	29 (odd only)
Multilayer Perceptron	Learning Rate	0.01	0.5
	Momentum	0.0	0.9
	Training Epochs	100	1000
	Hidden Neurons (1 layer)	5	50
J48 Decision Tree	Confidence Factor	0.05	0.5
	Min Instances per Leaf	1	20
Logistic Regression	Ridge Parameter	10^{-10}	10^{-1}
	Max Iterations	50	500
Bayesian Network	Max Number of Parents	1	5
	Smoothing (alpha)	0.1	1.0

Table 5.2: The hyperparameter search space for each machine learning algorithm.

CGP performed nearly as well as random forests with a much lower variation. While on the lower end of the accuracy scores, decision trees were very consistent, showing almost no variation in the results.

These findings are promising, with several techniques achieving accuracies exceeding 60% and a Matthews correlation coefficient approaching 0.5. Such results suggest that, although the performance is not flawless, the methods demonstrate a statistically significant ability to classify samples more reliably than would be expected by random chance.

Technique	Accuracy	F-Measure	MCC
Random Forest	65.0% $\pm$ 2.89%	0.644	0.472
ST–CGP	62.5% $\pm$ 0.42%	0.615	0.436
Logitboost	57.5% $\pm$ 0.78%	0.571	0.361
K-Nearest-Neighbours Classifier	50.0% $\pm$ 0.38%	0.505	0.270
Multilayer Perceptron Neural Network	48.8% $\pm$ 1.26%	0.483	0.230
J48 Decision Tree	47.5% $\pm$ 0.69%	0.463	0.210
Logistic Regression	41.3% $\pm$ 1.10%	0.413	0.121
Bayesian Network	31.3% $\pm$ 2.21%	0.288	-0.043

Table 5.3: Results of the machine learning techniques tested.

Further analysis revealed a consistent pattern across all techniques: the optimal performance was obtained in every case with the following configuration: incomplete variables excluded, the remaining variables normalised, and principal components derived from PCA omitted. This outcome is particularly noteworthy given that PCA is frequently employed to *enhance* performance by reducing dimensionality. The results may imply that, for this particular dataset, the variables exhibit limited intercorrelation, thereby diminishing the benefits of PCA.

Although it was not the highest-scoring algorithm, ST–CGP offered the additional benefit of producing largely interpretable programs. Given the complexity of soil health processes, the ability to interrogate why a model produces a particular prediction was considered valuable by project stakeholders. Many insights were able to be drawn from qualitative inspection of the evolved programs; for example, soil texture was frequently influential—many programs branched early on texture-related inputs—whereas moisture appeared less consistently and was often absent from the final program structure. Insights of this kind can be translated into agronomic hypotheses and decision support, helping end users relate model outputs to practical soil management actions and field observations.

5.4 Following Work

This project used yield as a proxy for soil health. In actuality, soil health is significantly more complex than a simple three-label category: it is derived from a holistic view of both the indicators and metadata used as variables in this project. By showing that these indicators are loosely able to translate into a picture of soil health, the next logical step is to try to predict those indicators, given the sensor data obtained from the PES sensor. In this way, we can say that, since these indicators can be used to predict soil health, if we can predict the indicators from the sensor data, then overall the sensor data can predict soil health. This indirect approach was necessary because the PES sensor was not yet operational when the soil samples were collected, and so it was not possible at the time to predict yield directly from the sensor outputs. Had the sensor been operational, a more direct approach could have been investigated. Once operational, the PES sensor was used to produce data for all soil samples that had been collected and stored, and so the subsequent phase of research focused on using this sensor data to attempt to predict these individual soil health indicators from the samples, in order to complete the indirect approach. This work is covered in the next chapter.

Chapter 6

Soil Health Analysis: Time Series Sensor Data

This chapter will describe in detail the application of ST–CGP to soil health through use of the PES sensor data. Through this research, the author has shown that ST–CGP can predict the values of the soil health indicators described in the previous chapter. ST–CGP is able to predict these indicators accurately enough that the predictions can be used by agronomists as part of their routine soil health monitoring processes, offering a faster, more cost-effective alternative to lab-based analyses and enabling more informed decision-making in agricultural management. In many cases, the predictions are as accurate as those produced by laboratories, with lower variance. The initial Innovate UK project which explored the application of ST–CGP to this data was so successful that the company has integrated ST–CGP into its product, and it is now being used commercially across the UK by agronomists and farmers in the management of their soil health.

The chapter starts by describing in detail the structure of the data, and how it is produced, before describing the data processing pipeline and ST–CGP setup used to predict the indicators. Finally, results are presented and discussed.

6.1 The PES Soil Test

6.1.1 Anatomy of a PES Test

The PES Soil test employs a novel approach to soil health assessment by effectively "sniffing" the soil to detect volatile organic compounds (VOCs). The process involves the gentle heating of a soil sample over a roughly five-minute period, a method designed to release VOCs that are otherwise trapped within the soil. These emitted VOCs are then captured by a proprietary single-use sensor embedded within the Electronic Reader Unit (ERU). The sensor comprises six distinct sensing channels, each doped with different semiconductors at varying densities. This diversification allows for a broad spectrum of VOC detection, which in theory allows for a more sensitive detection of VOC profiles.

During the five-minute test, the ERU continuously monitors changes in both resistance and capacitance across the six sensing channels as well as fluctuations in environmental parameters such as humidity and temperature, which can be vitally important for accurate VOC analysis. The dynamic data captured during this period generates a "VOC fingerprint" unique to the soil sample being tested. This fingerprint serves as a rich source of data, in theory capturing the complex interactions between soil properties and emitted VOCs.

In the initial project with PES Technologies, the author was tasked with the primary objective of determining the feasibility of mapping the acquired VOC fingerprints to various soil health indicators through machine learning. This involved comparing the effectiveness of ST–CGP on the problem to other machine learning techniques, in order to establish which was best suited to analysing this type of data. The hope was that at least one machine learning technique would be able to capture the intricate patterns within the sensor data.

6.1.2 Data characteristics

The sensor produced by the PES Sensor contans several challenging characteristics. One of these is the substantial variation in the scales of its constituent measurements. Specifically, the resistance data span multiple orders of magnitude; the circuit has a very high initial resistance, typically starting around $100G\Omega$ and dropping to $1M\Omega$ by the end of the test. In contrast, the capacitance measurements are confined to an exceptionally narrow range, operating within the picofarad range. The temperature and humidity data are different again—as they sit within ranges that are easily understood by human interpretation. This heterogeneity in the data characteristics necessitates meticulous construction of feature extraction techniques; such techniques must be carefully tailored to accommodate the distinct properties of each data type, ensuring their applicability and accuracy across the diverse range of measurements. Fig. 6.1 shows an example of the resistance readings produced by the sensor. The VOC "fingerprint" which is unique to each sample analysed is made up of six such channels, in addition to six capacitance channels and several environmental readings such as temperature and humidity.

Perceptive readers may notice an additional complication: the presence of noise. The sensors used in the PES reader unit are connected to highly sensitive electronic circuitry, which, while essential for detecting subtle changes in signal from the semiconductors, inevitably captures a significant amount of electrical noise. This noise can arise from various sources, including thermal fluctuations, electromagnetic interference, and inherent variability in electronic components. Such unwanted signals can obscure the true data signal, making it more challenging to obtain accurate and reliable measurements. The issue is further complicated by the fact that this noise can vary widely in both frequency and amplitude, making its identification and removal a complex task.

Unlike the human visual system, which has evolved to effectively filter out irrelevant information and focus on important details, computational algorithms are much more vulnerable to noise. Humans can ignore background noise with relative ease and concentrate on salient features, a capability that algorithms do not necessarily possess. Developing algorithms that are robust against such noise is particularly challenging when the noise spans multiple orders of magnitude. As a result, to minimise the effect of the noise it is essential to apply noise reduction and smoothing techniques to minimize the impact of noise on the data. These techniques help in improving the signal-to-noise ratio, thereby enhancing the accuracy and reliability of subsequent data analyses, including by machine learning.

To address the issue of noise, a robust preprocessing pipeline has been implemented, comprising several protocols tailored to the specific characteristics of each component of the dataset, and the types of noise present. These techniques include data cleaning, noise removal, and normalisation. A detailed description of these methodologies is provided in 6.2.

6.2 Preprocessing

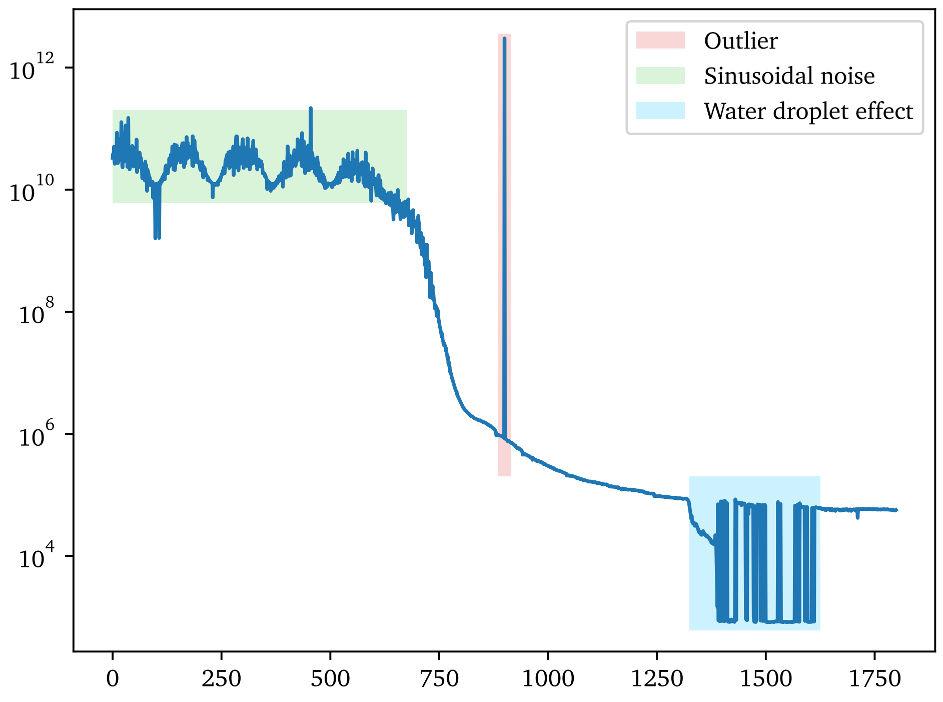


Figure 6.1: An example of a resistance curve with some areas of concern highlighted.

In order to devise a robust data processing pipeline it is necessary to perform a comprehensive examination of the collected data to determine which components require processing, and what the processing should aim to achieve. Figure 6.1 presents the resistance curve recorded from a single sensor element during a test. A preliminary analysis of this figure reveals four distinct issues that could potentially disrupt any subsequent data analysis. These issues have been marked in the accompanying figure for ease of reference and further discussion.

The first issue arises from the extensive range covered by the data, which spans several orders of magnitude. This wide range has the potential to pose significant scaling problems, as the initial resistance values are substantially larger than the final ones; on average the resistance data spans 10 orders of magnitude. Such discrepancies can lead to difficulties in accurately interpreting the data, as algorithms may struggle to maintain sensitivity across the entire range. Additionally, previous research has shown that the critical information lies in the overall shape and characteristics of the resistance curve rather than the absolute values.

The second issue identified is the presence of noise—both the random noise found throughout the entire curve, due to the sensitive measurement of an electronic circuit, as well as the repetitive sinusoidal noise found most noticeably at the higher end of the reading. In both cases the noise amplitude increases in proportion to the true signal's magnitude. This variability means that a simple, fixed noise removal approach would be ineffective, as it cannot adapt to the changing noise levels across different signal strengths. Additionally, the sinusoidal noise pattern means that traditional smoothing algorithms struggle to completely eliminate the effects of noise.

The third issue is that a sudden spike occurs at approximately index 700, representing an unexpectedly large value that deviates sharply from the surrounding data points. Such spikes can distort the analysis by introducing outliers that can skew extracted features and interfere with analysis. This particular issue is intermittent—not all resistance curves exhibit it—but it occurs frequently enough that it is necessary to mitigate it.

The fourth issue is a period of rapid fluctuations or "jumping" variations towards the end of the signal. These erratic changes, caused by water droplets forming on the sensor, can complicate the extraction of meaningful features from the data, as they obscure the true signal and make it difficult to approximate the shape of the true curve numerically.

In response to these identified challenges, the preprocessing pipeline for resistance curves must incorporate several essential steps to ensure the data is clean and suitable for further analysis.

The first step involves scaling or normalising the data to mitigate the effects of the wide range of values. This can be achieved effectively by applying a logarithmic transformation, specifically using $\log_{10}$, which brings all values into the same order of magnitude. Such normalisation not only simplifies the data but also enhances the performance of subsequent processing stages by ensuring that all data points are within a comparable scale.

The second step focuses on data cleaning, which involves the removal of the anomalous spike at index 700 and the stabilization of the rapid variations observed towards the end of the signal. Techniques such as outlier detection algorithms can be employed to identify and eliminate spikes, while a combination of smoothing algorithms and iterative methods can help reduce the impact of water-droplet-induced variation. These methods are covered in 6.2.1.

The final step is noise removal, which requires the implementation of adaptive filtering methods that can account for the variable noise levels across the signal. By systematically applying these preprocessing techniques, the integrity of the resistance data is preserved, allowing for more accurate and meaningful downstream analysis.

6.2.1 Data cleaning

A relatively simple technique is used for the initial cleaning of the data. The method starts by taking the derivative of the dataset to identify points where sudden changes occur. A basic threshold is then applied to detect spikes in the data; any values that exceed or fall below this threshold are considered outliers. Once these problematic indices are identified, they are removed from the original data.

Following the removal of these outliers, the original data is interpolated at the miss-

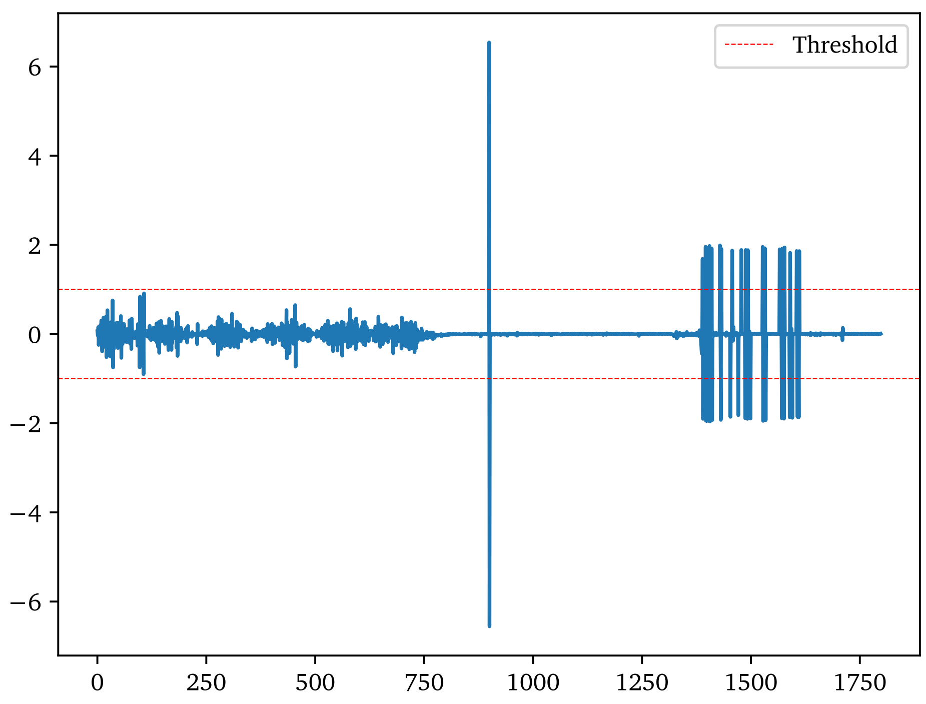


Figure 6.2: The derivative of the resistance curve, before data cleaning.

ing points to ensure that the curve shows a smooth transition around the removed indices. In practice, this approach effectively removes the random high outlier spikes as well as some of the water droplet effects that can appear towards the end of the curve. Fig. 6.2 shows the derivative for the example data. It can be seen that the random outlier, as well as many of the water droplet spikes, are detected by this method and would therefore be successfully removed.

6.2.2 Smoothing and noise removal

Moving Average

One of the most straightforward and commonly used smoothing algorithms in data analysis is the moving average. This method is particularly effective for reducing small fluctuations caused by noise, thereby providing a clearer representation of the underlying trend or true signal within the data. The moving average achieves this by calculating the average of a specific number of consecutive data points, which smooths out short-term variations and highlights longer-term patterns. This simplicity makes the moving average a valuable tool across various disciplines, including economics, finance, and environmental studies, where identifying genuine trends is essential.

Mathematically, the moving average can be defined thus: Consider a time series dataset $\{x_1, x_2, \ldots, x_n\}$, where n is the total number of observations. A simple moving average (SMA) with a window size of k is calculated by taking the arithmetic mean of each set of k consecutive data points. Specifically, the moving average at position t is given by:

$$\mathrm{SMA}_t = \frac{1}{k} \sum_{i=t-k+1}^{t} x_i.$$

In this formula, t ranges from k to n, ensuring that each moving average value incorporates exactly k data points. The choice of window size k is crucial as it determines the extent of smoothing. A larger k results in a smoother resultant signal, effectively reducing more noise but potentially overlooking short-term changes. On the other hand, a smaller k makes the moving average more responsive to recent data points, allowing for shorter-term shifts to be represented, but retaining more of the original variability (and potentially, some of the noise!).

Selecting the appropriate window size involves balancing the need for smoothness with the need to remain sensitive to important changes in the data. Broadly speaking, a longer moving average might be used to identify long-term trends, while a shorter moving average could help detect shorter-term, or more sudden changes.

The simple moving average serves as a foundational technique upon which more advanced smoothing methods are built. Variations such as the weighted moving average (WMA) and the exponential moving average (EMA) introduce different weighting schemes to give more importance to certain data points. A detailed explanation of these techniques is beyond the scope of this thesis. In total three varieties of moving average were tested on the data: simple moving average (SMA), exponential moving average (EMA), and weighted moving average (WMA). Figures 6.3 to 6.5 show the results of applying SMA, EMA, and WMA to the resistance curve with three different window sizes k: $k = 10$, $k = 50$, and $k = 100$.

The analysis of the smoothed data shows a common issue across all the moving average techniques tested: each method introduces an offset in the smoothed line that corresponds to the chosen window length k. This offset occurs because of the sliding window used to calculate the average of a set number of points. When calculating the moving average on a series of data points, until k points have occurred, there aren't enough preceding data points to fully apply the window. There are two common paproaches to ameliorate this: the first is to simply ignore the first k points, and not calculate an average for them. This results in an output which is $k - 1$ points shorter than the input. This truncation means that some important information from the beginning of the signal may be lost, potentially affecting the overall analysis.

Alternatively, the initial section of the smoothed signal can be calculated using an expanding window approach, whereby for every point p with index less than k, the average is calculated on the preceding $p - 1$ points. While this method preserves the

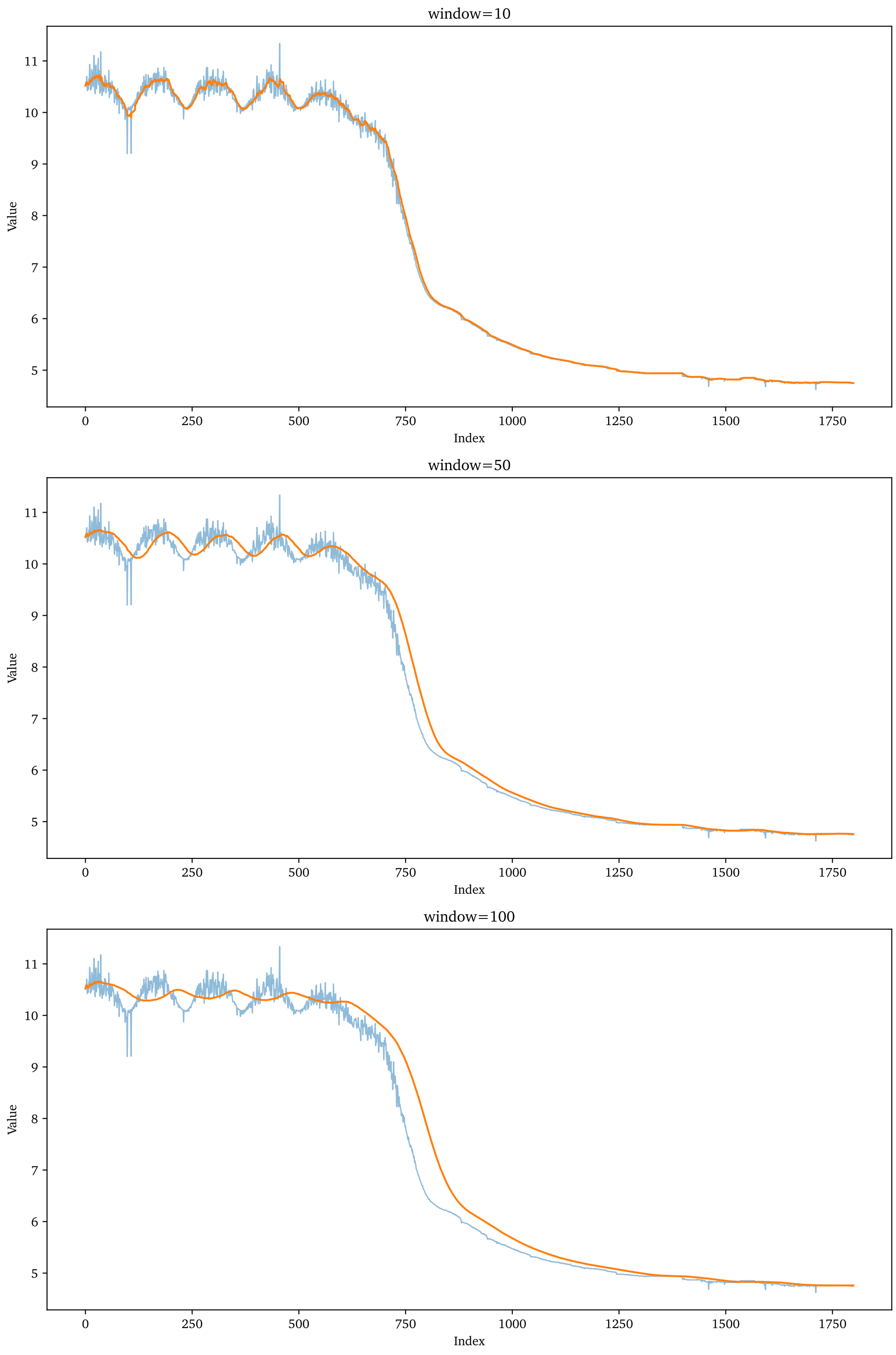


Figure 6.3: SMA.

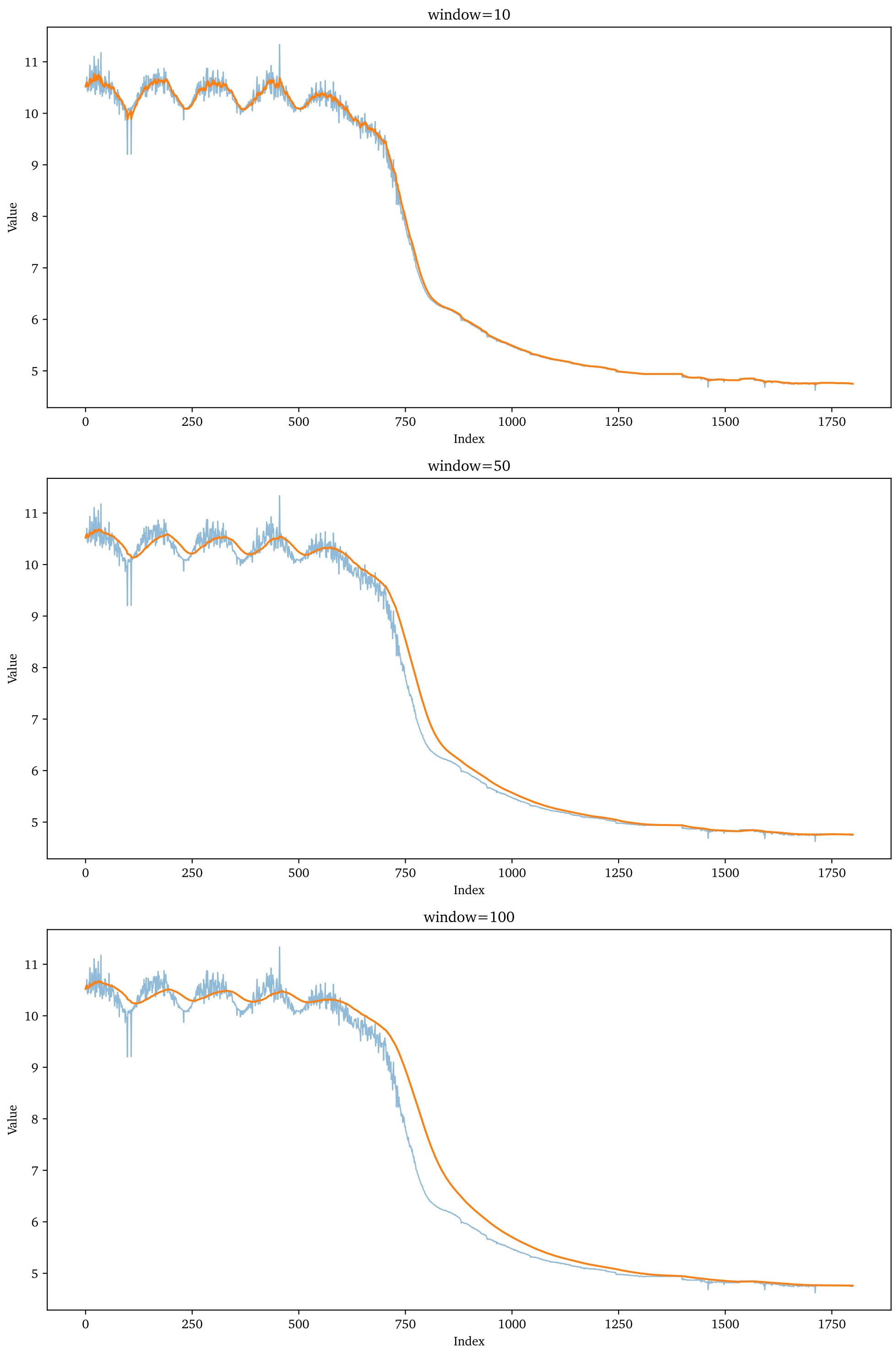


Figure 6.4: EMA.

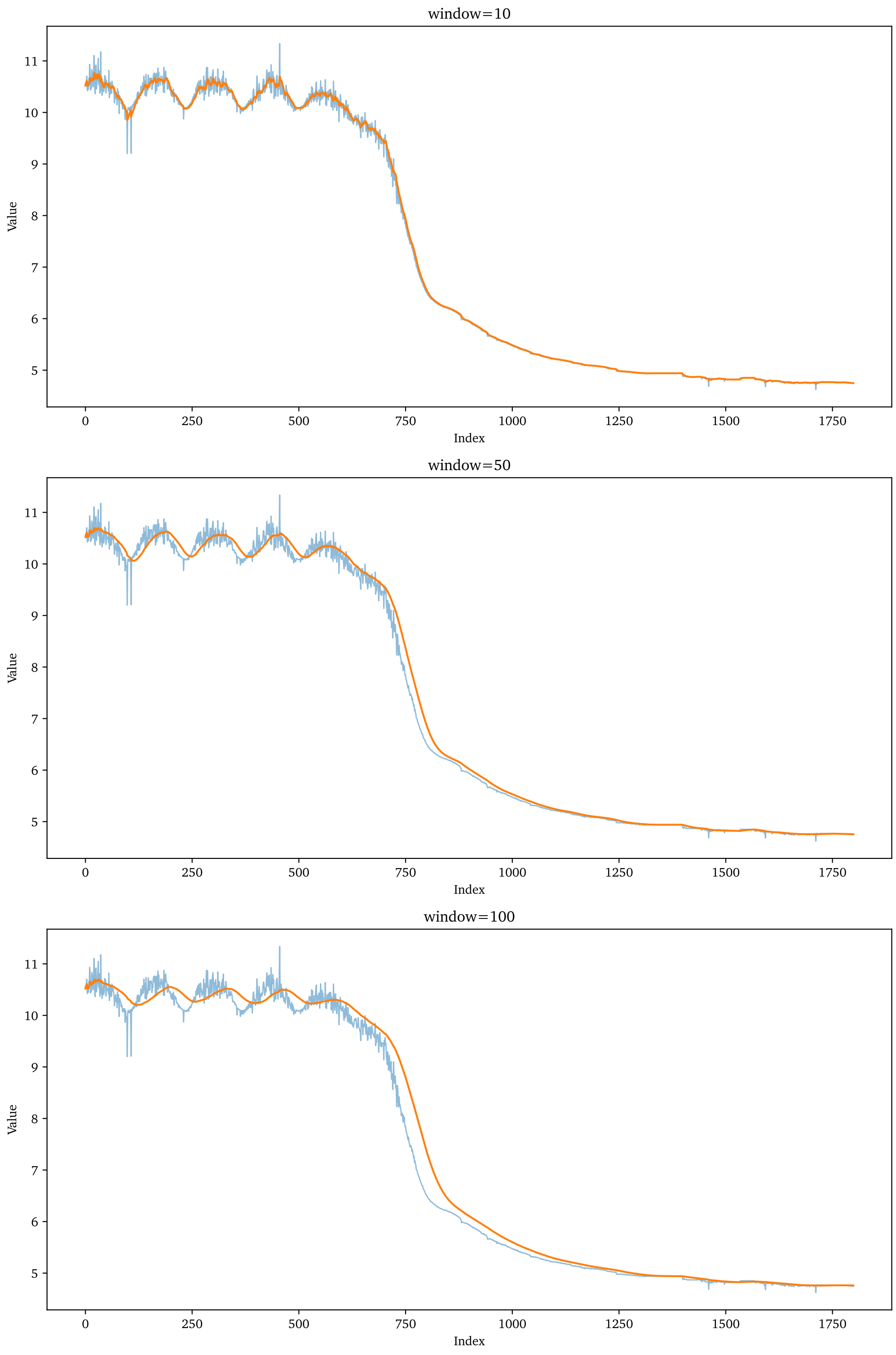


Figure 6.5: WMA.

length of the data, it does not provide a consistent smoothing effect; the early data points are smoothed less than later points.

Whether the output is shortened or padded, the result is a displacement of the smoothed signal relative to the original data by an offset of k. This displacement causes the smoothed output to lag behind the true signal, meaning that changes or fluctuations in the data take longer to be reflected in the smoothed line. In contexts where timely reactions to data changes are essential, this delay can affect subsequent data analysis. When the smoothed signal is used in feature extraction or building predictive models, the delay can distort the true timing of these events.

Another important observation is that moving averages struggled to remove the sinusoidal noise at the start of the signal. This type of noise posed a challenge for all the moving average techniques evaluated. The amplitude of this sinusoidal noise is too large to be removed, even when using a substantial window size of 100 points. As a result, this type of structured noise remains part of the smoothed signal, making it difficult to distinguish from the actual signal.

Despite this difficulty with sinusoidal noise, all three moving average techniques performed well in reducing the random noise present in the signal. It can be seen that SMA, EMA, and WMA were similarly effective in removing random noise and smoothing the signal.

To address the two primary limitations associated with moving averages—the lag and the inadequate removal of sinusoidal noise—two additional techniques were explored. The first technique employed was the Savitzky-Golay filter, a widely recognised method in signal processing known for its ability to smooth data while preserving important features such as peaks and edges. By fitting successive subsets of data points with a low-degree polynomial, the Savitzky-Golay filter effectively reduces noise without introducing significant lag, thereby enhancing the responsiveness of the smoothed signal. The second technique involved the application of the Fast Fourier Transform (FFT) to analyse and mitigate variations in the frequency domain. This approach targets the removal of sinusoidal noise by transforming the time-domain signal into its frequency components, allowing for the selective filtering of unwanted frequencies. By addressing noise in the frequency domain, FFT-based methods can more precisely eliminate structured noise patterns that moving averages fail to remove.

Savitzky-Golay

The Savitzky-Golay algorithm is a widely utilised digital filtering technique that performs local polynomial regression on subsets of data to achieve effective smoothing while preserving important signal features. Its principal mechanism involves fitting a low-degree polynomial to a window of data points via a least-squares criterion, ensuring that key aspects of the original signal, such as peak amplitudes and widths, are maintained without introducing significant lag. This makes it particularly useful for applications where immediate tracking of the data is critical and where excessive delay or distortion of features would be detrimental. This characteristic seems well-suited to the task at hand.

At a high level, each “step” of the Savitzky-Golay filter follows this process:

1. **Window Selection**: A contiguous window of data points is selected from the overall signal. The window size, which must be an odd number, is chosen based on a trade-off between noise reduction and the preservation of detailed signal features; larger window sizes reduce noise to a greater degree, but risk losing finer (non-noise) details in the original signal.

2. **Polynomial Fitting**: Within each window, a polynomial of a chosen degree (typically low, such as 2 or 3) is fitted to the data using least squares.

3. **Coefficient Derivation**: The coefficients of the polynomial fit are computed, and these coefficients are then used to derive the convolution coefficients (or filter coefficients) that characterise the Savitzky-Golay filter.

4. **Convolution**: The derived filter coefficients are convolved with the data in the moving window, generating a smoothed output that approximates the true signal values at the central point of the window.

This process is repeated for each position of the window across the entire dataset, ensuring that each point in the filtered signal reflects the local polynomial fit of its surrounding data.

While this method has the advantage of minimising lag and preserving local features, it is not without its drawbacks. The algorithm's reliance on local polynomial fitting means that small-scale variations, such as the sinusoidal noise mentioned earlier, may not be effectively removed unless a larger window is used; which in turn could result in some loss of detail that is important in the prediction of soil health indicators.

Fig. 6.6 shows the Savitzky-Golay filter applied to the data with varying window lengths. Unfortunately, while it does an excellent job at removing typical noise, and shows absolutely no lag, the sinusoidal noise is still present even with a large window size. As such, while this technique is clearly very effective, it does not quite perform well enough on the upper portion of data to be used as the sole noise removal method.

Fourier Smoothing with Detrending

There exist many methods using Fourier Transforms to remove noise from signals. Central to these methods is the Fast Fourier Transform (FFT), an algorithm that converts signals from the time domain into the frequency domain. This transformation is simplifies the process of identifying and isolating various components of the signal based on their distinct frequencies. By analysing the signal in the frequency domain, it becomes significantly easier to pinpoint specific frequencies where noise is concentrated.

Noise within signals frequently occurs at particular high frequencies, making it identifiable using FFT-based methods. Once these noisy high-frequency components are detected, they can be selectively removed within the frequency domain. After this noise removal, the inverse FFT is applied to transform the signal back to the time domain. By removing the frequencies in the frequency domain, the result after the inverse transformation is that the noise associated with the targeted frequencies is eliminated in the time domain, while the integrity of the original signal is largely preserved. This selective filtering capability of FFT methods stands in contrast to time domain approaches,

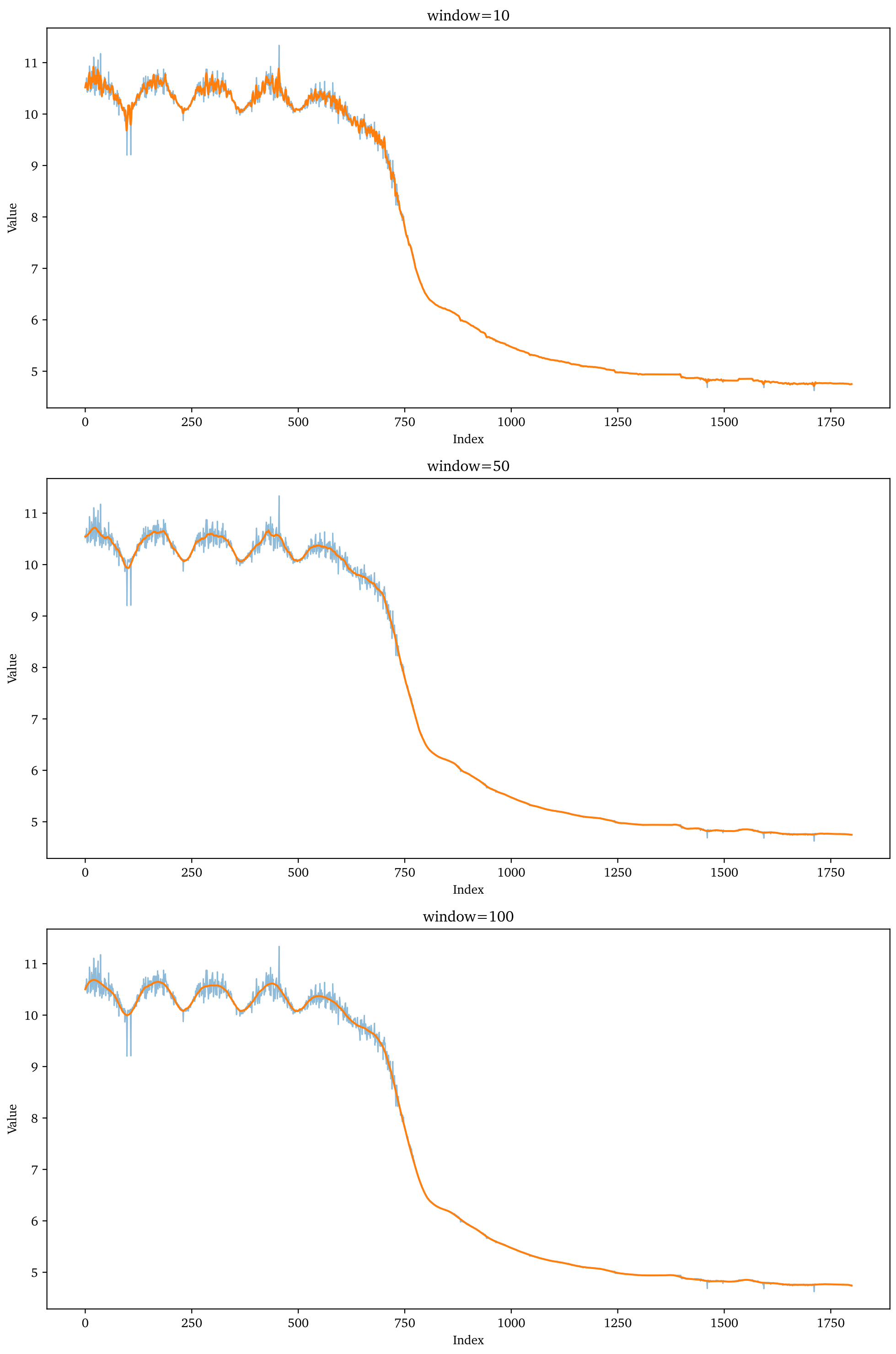


Figure 6.6: Savitzky-Golay.

such as the moving average and Savitzky-Golay methods seen earlier, which lack the precision to filter out specific frequencies without affecting others. As seen, the window-based averaging used by these methods can inadvertently lose fine details and important high-frequency components of the signal. As a result, while these techniques are quite effective at reducing random noise, they often lose the subtle nuances of the input signal as a side effect.

In the context of the resistance curves analysed in this study, the sinusoidal noise occurs at a consistent frequency of approximately 0.1Hz. Given that FFT-based methods excel at removing noise at particular frequencies, they present an ideal solution for addressing this type of structured noise. By transforming the resistance curves into the frequency domain, the sinusoidal noise can be accurately identified and isolated based on its frequency, and then removed along with any other high frequency noise. This type of filter is known as a *low-pass filter*, because it lets low frequencies pass while attenuating higher frequencies.

An adaptation of the low-pass parabolic fast fourier transform filter [126] (PFFTF) with added trend robustness and simplified parabola calculation was implemented. At a high level, the filter follows this process:

1. **Identify trend and construct baseline:** Calculate the overall trend (and direction) of the data by drawing a straight line from the start point to the end point. For robustness, calculate the start point as the average of the first few points, and the end point as the average of the last few points, to minimise the effect of noise on this calculation.

2. **Detrend:** Subtract the baseline calculated above from the input data. This removes the underlying trend, leaving behind only the fluctuations around that baseline.

3. **Transform to frequency domain:** Apply the Fast Fourier Transform to the detrended data, thereby transforming from the time domain to the frequency domain.

4. **Apply a low-pass parabolic filter:** Apply a low-pass parabolic filter to the data with smoothing factor n, to attenuate high frequencies while preserving low frequencies, see eq. (6.1). A parabolic filter is chosen to smoothly decrease the influence of higher frequencies, rather than using a hard cutoff.

5. **Inverse transform:** Transform the filtered data back to the time domain using the Inverse Fast Fourier Transform (IFFT). This produces a smoothed version of the detrended data.

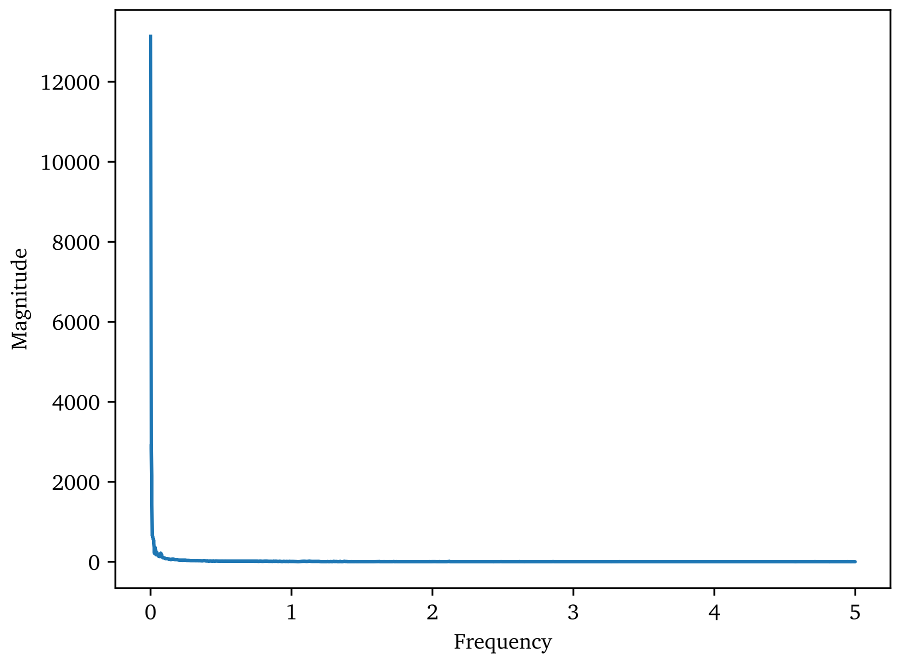
6. **Restore the trend:** Add the baseline back to the smoothed detrended data. This results in a smoothed version of the original data that maintains the original trend, but has reduced noise.

$$\begin{aligned}
f_c &= \frac{1}{2n\Delta t} \\
f_k &= \frac{k}{N\Delta t}, \quad \text{for } k = 0, 1, \ldots, N-1 \\
H(f_k) &= \begin{cases} 1 - \left(\frac{f_k}{f_c}\right)^2 & \text{if } |f_k| \leq f_c, \\ 0 & \text{otherwise} \end{cases} \\
Y_{\text{filtered},k} &= Y_k \cdot H(f_k), \quad \forall k = 0, 1, \ldots, N-1.
\end{aligned} \tag{6.1}$$

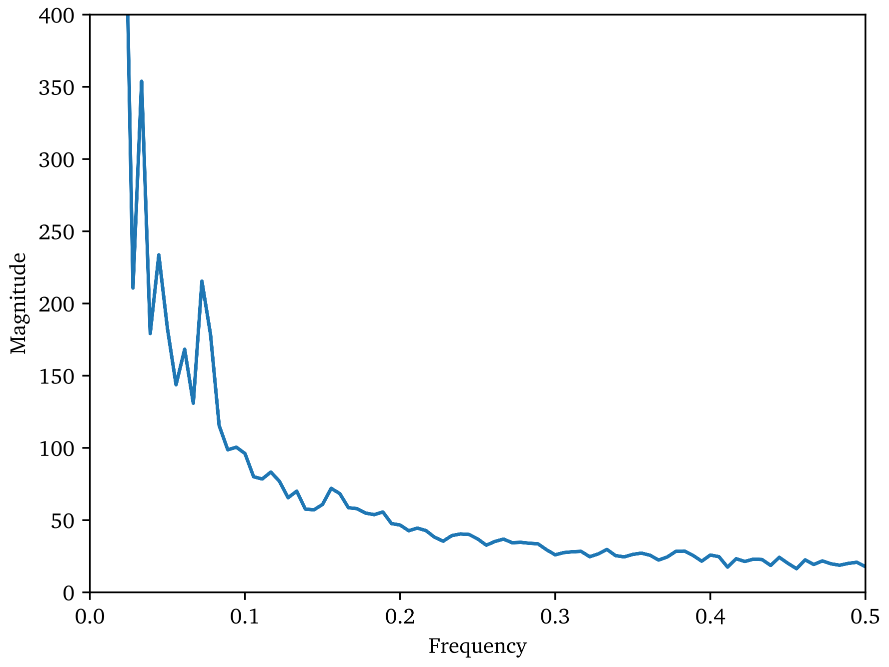
To understand why this works, it is important to see the process applied to real data. Let us first consider the low-pass parabolic FFT filter, and then apply the detrending process to it.

Figure 6.7 shows the frequency spectrum. Recall that the sinusoidal noise is approximately 0.1Hz; this should appear as a peak in this frequency spectrum. fig. 6.7a shows the full spectrum, while fig. 6.7b shows a zoomed portion of the low frequencies. It can be seen that there is a peak just below 0.1Hz—this is the sinusoidal noise, and by attenuating this frequency in the frequency domain the sinusoidal noise can be eliminated in the time domain.

Figure 6.8 shows the calculated cutoff frequency parabolas for various smoothing factors, overlaid on the frequency spectrum. It can be seen that the higher the smoothing



(a) Full frequency spectrum.



(b) Zoomed low frequencies.

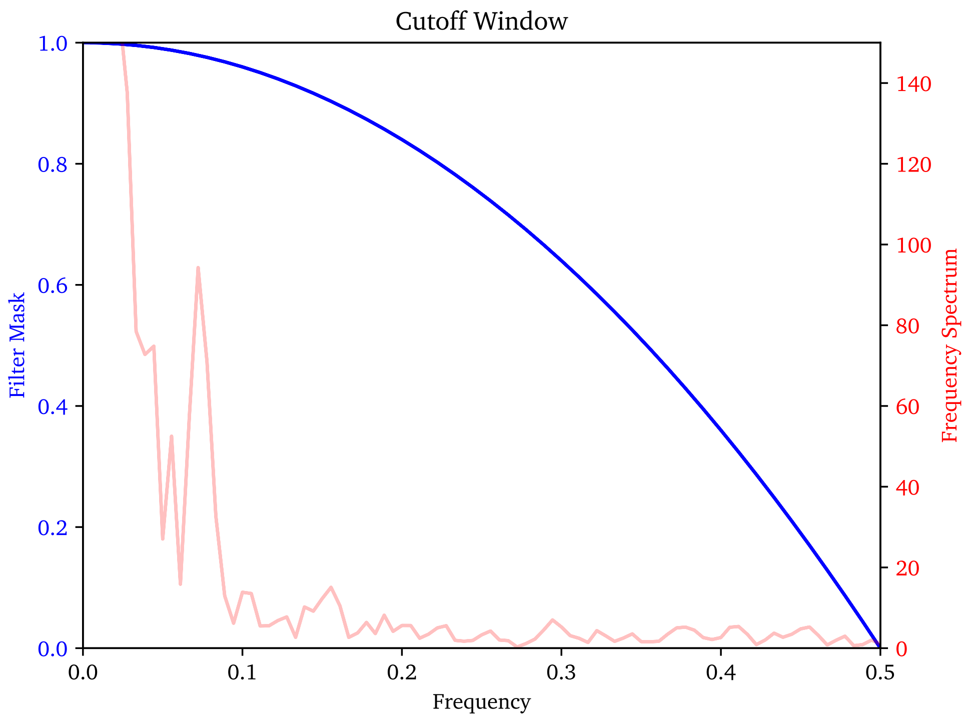
Figure 6.7: Frequency spectrum for the resistance data (without detrending.)

factor, the lower the “final” cutoff frequency, and therefore the more frequencies are attenuated. In other words, a higher smoothing factor results in a smoother output, as one would expect. It can also be seen that for a smoothing factor $n = 100$, the cutoff frequency sits below the $\approx$0.1Hz sinusoidal noise, while the lower smoothing factors attenuate, but don’t completely remove these frequencies. Therefore, when performing the IFFT, it should be expected that lower values of n do not completely remove the sinusoidal noise, while $n = 100$ should.

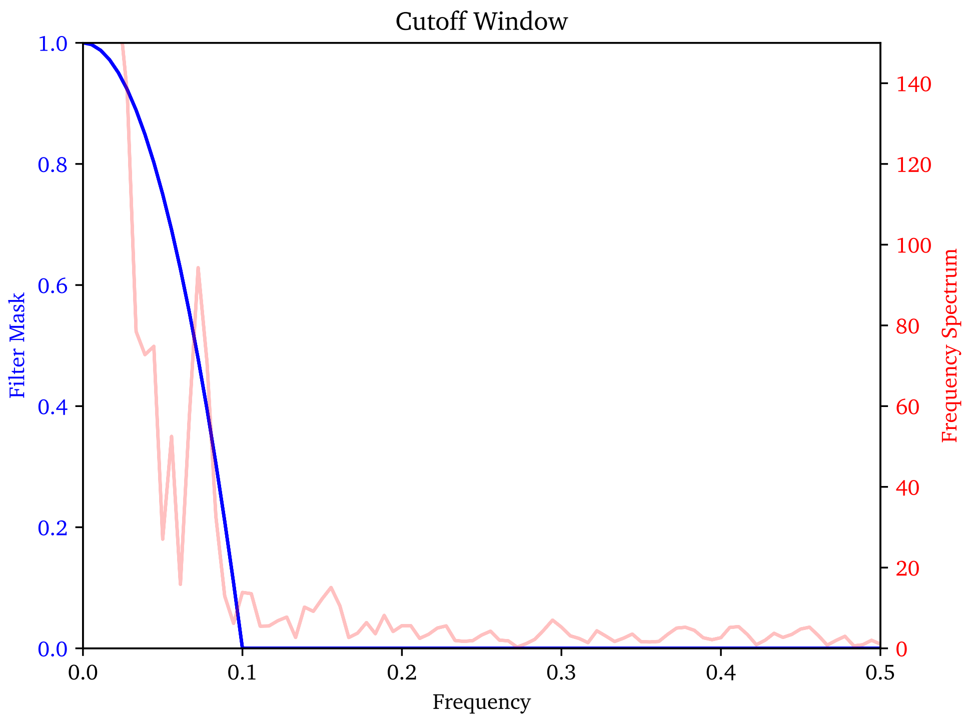
Figure 6.9 shows the smoothed data after applying the low-pass parabolic FFT filter with $n = 100$, overlaid on the original data. Astute readers will notice that the smoothed data does not really represent the original data at the start and end. This is because the data was not detrended before being supplied to the filter, and as such the large drop in the data interferes with the filtering process. Nevertheless, it is encouraging that the sinusoidal noise is largely removed by the filter. We shall now apply the detrending process and see how it affects the result.

Figure 6.10 shows the data once the detrending process has been applied; this is the *fluctuation around the baseline*, the straight line drawn from the start to the end of the data. By feeding this into the low-pass filter the frequency spectrum can be observed. Figure 6.11 shows the frequency spectrum of this curve, and it can be seen that when detrending is applied, the peak at $\tilde{0}$.1Hz is even more evident than before. As such, the frequency cutoffs used previously, when applied to this curve, are even more effective: fig. 6.12 shows the inverse FFT of the detrended curve, with the baseline added, and it can be seen that not only has the random noise been removed from the entire curve, but the sinusoidal noise has also been removed.

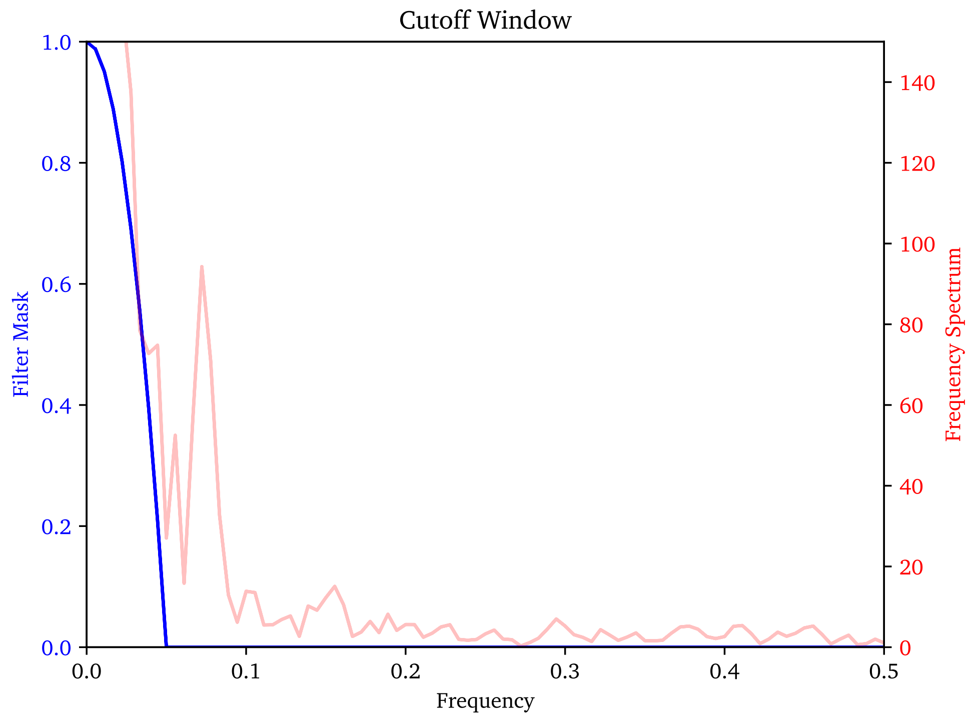
Figure 6.13 shows the results of various smoothing factors when detrending is used. As expected, it can also be seen that this technique does not suffer from any lag or shifting in the smoothed data, because the technique does not use a windowed average. A further benefit is that the inflection point does not change—this is one of the key features that can be extracted from this type of curve (which roughly follows an inverse logistic shape).


Cutoff Window
Filter Mask
Frequency Spectrum
Frequency
1.0
0.8
0.6
0.4
0.2
0.0
140
120
100
80
60
40
20
0
0.0
0.1
0.2
0.3
0.4
0.5

(a) $n = 10$.


Cutoff Window
Filter Mask
Frequency Spectrum
Frequency
1.0
0.8
0.6
0.4
0.2
0.0
140
120
100
80
60
40
20
0
0.0
0.1
0.2
0.3
0.4
0.5

(b) $n = 50$.



(c) $n = 100$.

Figure 6.8: Cutoff frequency parabolas for various smoothing factor n.

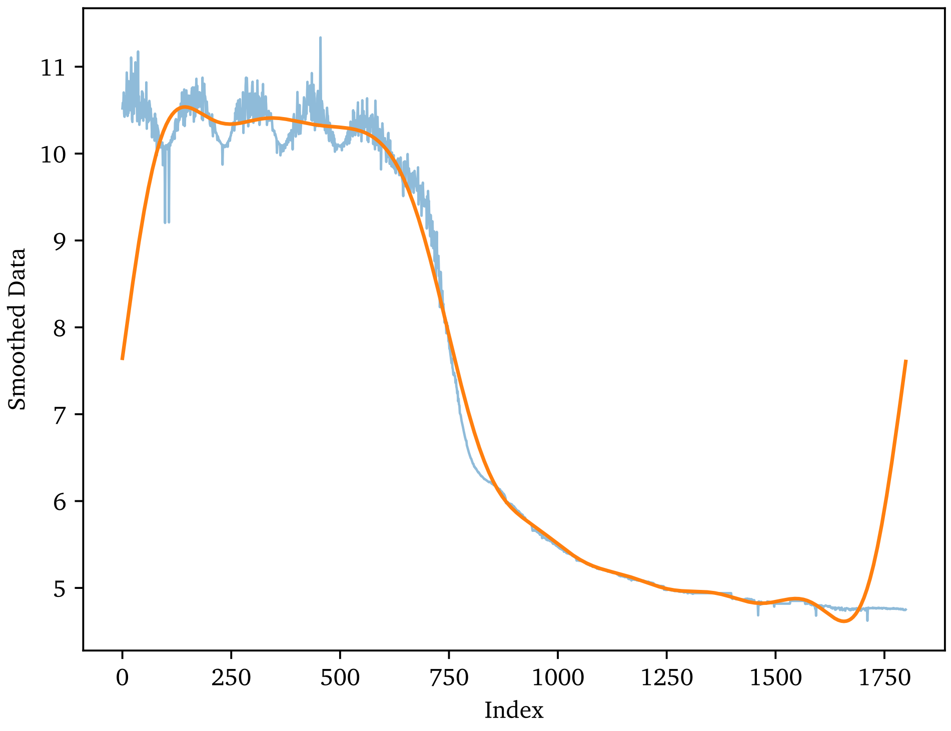


Figure 6.9: Inverse FFT (without detrending.)

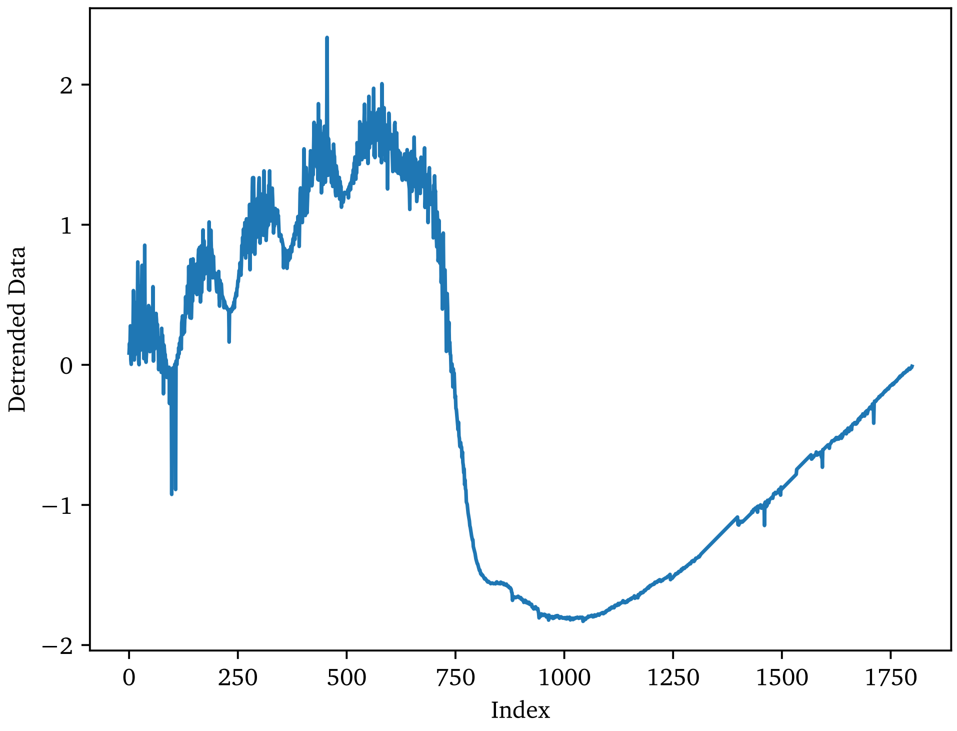
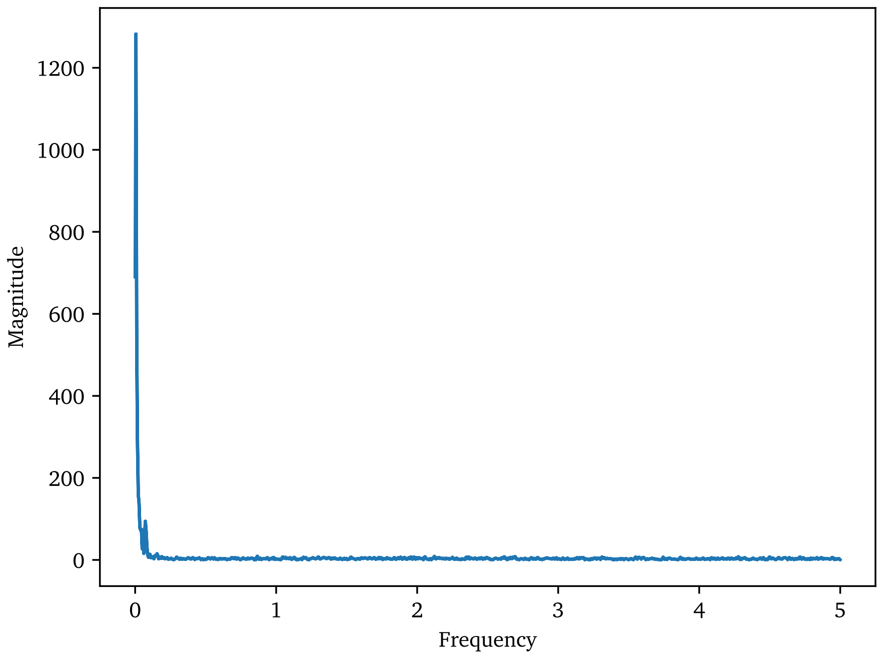
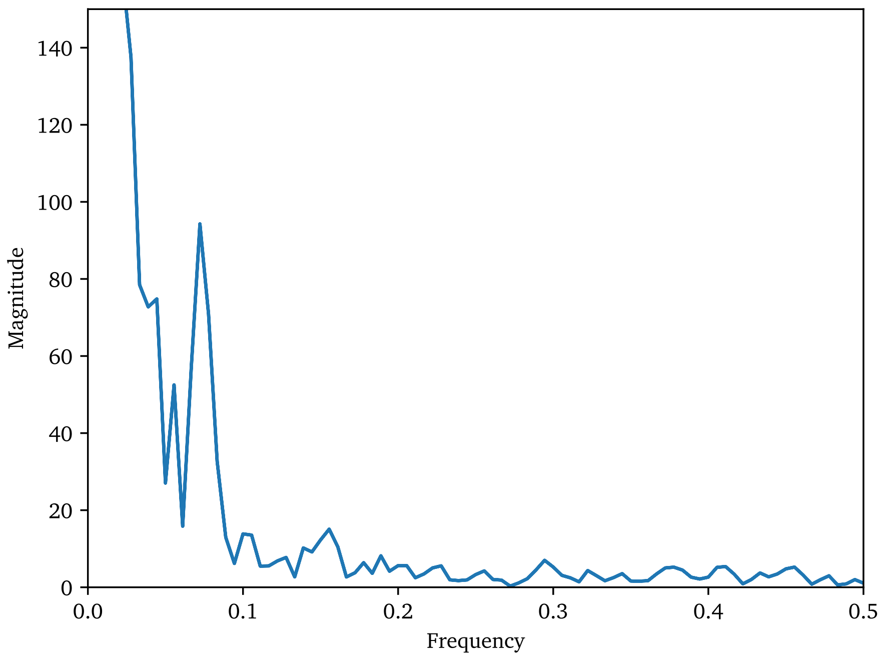


Figure 6.10: Detrended data.



(a) Full frequency spectrum.



(b) Zoomed low frequencies.

Figure 6.11: Frequency spectrum for the resistance data (with detrending)

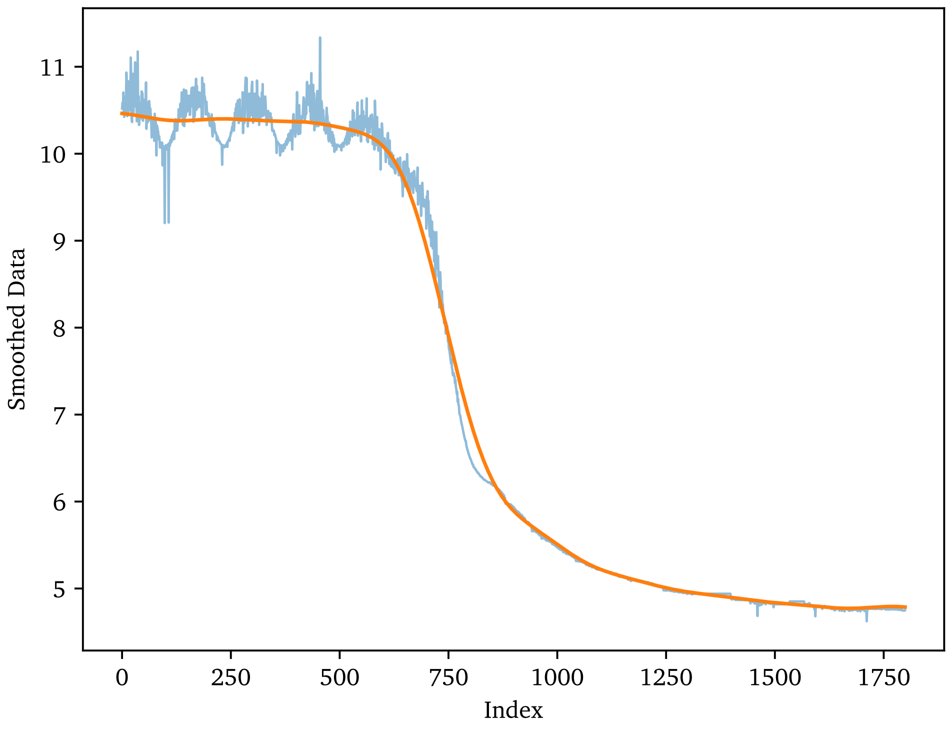


Figure 6.12: Inverse FFT (with detrending.)

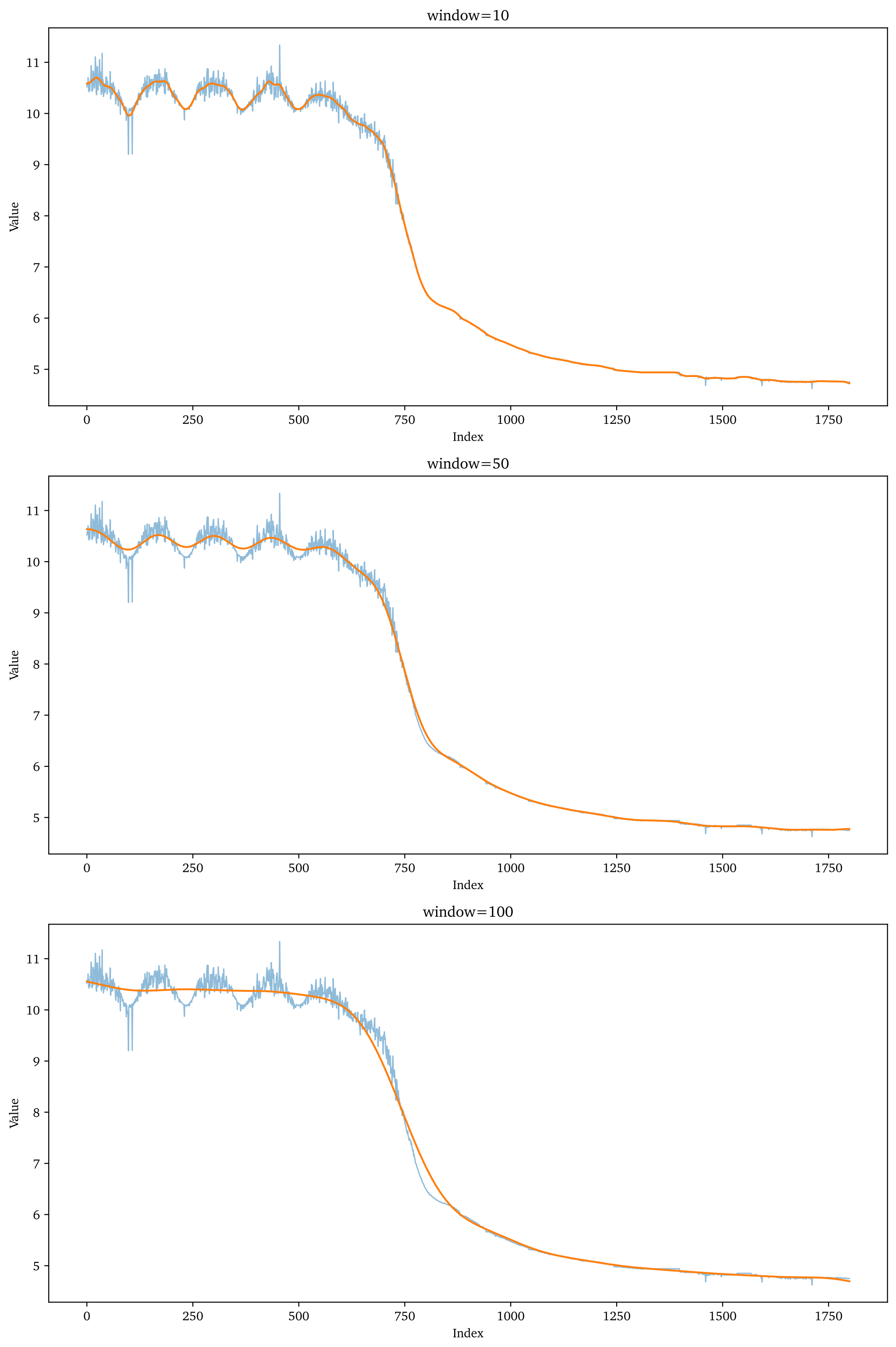


Figure 6.13: Curves smoothed using the low-pass parabolic Fast Fourier Transform filter.

Overall, this smoothing technique has several benefits over the other techniques tested; most notably, the elimination of the sinusoidal noise and lack of lag. As such, this FFT-based technique was selected as the smoothing method of choice for the pre-processing pipeline.

6.3 Final experiment

Each of the indicators described in the previous chapter was evaluated using k-fold cross validation with $k = 5$, with the exception of Geosmin, which was not available for a large proportion of samples due to the cost of measuring it. Multiple evaluations were made, each with a subset of N features selected using "mutual information" values, with N taking values from 10 to 150, in steps of 10. For each value of N, the cross validation performance was assessed by calculating the average root mean squared error (RMSE). The subset of N features yielding the best performance during this phase was then applied to the test set, in order to ensure that the model was generalisable to unseen data.

This process was repeated across three distinct algorithms: ST–CGP, Random Forests, and Neural Networks. These three techniques were chosen for the following reasons: Random Forests and ST–CGP were the two highest performing techniques in the previous experiment, and the project requirements stipulated that neural networks must be used as a point of comparison, given the relative ubiquity of NN-based approaches across other disciplines. To ensure that the comparison between the techniques was fair, the computational budget and stopping criteria for each technique were kept as consistent as possible. By doing so, each method was given a broadly comparable opportunity to optimise its performance. The computational budget settings were as follows:

- The neural network employed a standard multi-layer architecture, incorporating dense layers augmented with dropout and regularisation techniques, and early stopping was used (based on validation loss) to prevent overfitting, with a patience of 25 epochs. A maximum of 500 epochs was permitted.

- ST–CGP was constrained to a maximum of 500 generations, with an early stopping mechanism added, which monitored the validation root mean squared error (RMSE) with a patience of 25 generations.

- The Random Forest algorithm was executed using default scikit-learn settings. As it does not train iteratively in epochs or generations, no early stopping criterion was applied in this case.

As the other techniques used do not support typing, ST–CGP was operated with a single type, in order to keep the comparison fair. As discussed earlier, this has the effect of essentially making ST–CGP operate similarly to a standard CGP system, with the addition of the advanced optimisiation techniques described earlier, such as full crossover, genetic rewiring, multi-objective optimisation, memoisation, and varying arities.

6.4 Results

Section 6.4 shows the results of training the three selected algorithms using k-fold cross validation. As this is a regression task, the metrics used to assess the performance of each algorithm are the root mean square error (RMSE) and r-squared value, which is a goodness of fit measure. The closer r^2 is to 1, the better the performance of the algorithm. A value of 0 indicates that the algorithm performs no better than chance; no variance in the results is explained by the algorithm. A negative value indicates an algorithm that performs worse than this.

The results in this table are the mean average of the metrics for each out-of-sample fold used in the k-fold cross validation. So, for a value $k = 5$, five folds are generated, and each algorithm is trained five times, with each fold being used as out-of-sample data once. The results on each out-of-sample fold are averaged here.

It can be seen from the table that ST–CGP was competitive on all indicators—it either outperformed the other algorithms, or came close to the results of the best performing algorithm. Random Forests performed best on the most indicators overall: nine of the fourteen indicators were most performant with this algorithm compared to five with ST–CGP. Notably, neural networks underperformed considerably on every indicator. The most likely explanation for this is that the training dataset is simply too small; as discussed earlier, neural networks benefit from a significant amount of training data and significant computational resources dedicated to the training process. Neither were available here.

Table 6.1: Performance of ST–CGP, Random Forest, and Neural Networks on various soil health indicators. Results are the average of k-fold cross validation, with $k = 5$.

	ST–CGP		**Random Forest**		**Neural Network**	
Indicator	RMSE	r^2	RMSE	r^2	RMSE	r^2
Microbial Biomass	149 ± 9.6	0.72 ± 0.04	**147 ± 14.4**	**0.72 ± 0.05**	223 ± 7.6	0.38 ± 0.05
Respiration	0.52 ± 0.06	0.49 ± 0.04	**0.51 ± 0.06**	**0.50 ± 0.04**	0.68 ± 0.04	0.10 ± 0.14
WHC Water Content	2.39 ± 0.20	0.77 ± 0.04	**2.32 ± 0.18**	**0.78 ± 0.04**	4.60 ± 0.07	0.16 ± 0.08
Field Water Content	2.29 ± 0.20	0.81 ± 0.04	**2.21 ± 0.25**	**0.82 ± 0.04**	3.54 ± 0.12	0.55 ± 0.06
Soil Organic Matter (LOI)	**1.23 ± 0.20**	**0.82 ± 0.06**	1.30 ± 0.21	0.79 ± 0.07	2.01 ± 0.15	0.51 ± 0.07
Nitrate Nitrogen	7.20 ± 0.35	0.59 ± 0.04	**7.14 ± 0.38**	**0.60 ± 0.04**	9.33 ± 0.50	0.31 ± 0.04
Ammonium Nitrogen	**0.79 ± 0.14**	**0.40 ± 0.12**	0.83 ± 0.09	0.33 ± 0.07	0.92 ± 0.15	0.18 ± 0.09
Mg Available	**22.6 ± 1.5**	**0.78 ± 0.02**	23.4 ± 2.5	0.76 ± 0.05	39.3 ± 2.4	0.34 ± 0.08
K Available	**63.5 ± 3.5**	**0.46 ± 0.07**	63.9 ± 3.3	0.45 ± 0.07	86.0 ± 2.6	0.01 ± 0.08
P Available	**9.75 ± 0.69**	**0.60 ± 0.08**	10.3 ± 0.66	0.55 ± 0.07	14.2 ± 0.79	0.16 ± 0.04
pH	0.33 ± 0.01	0.86 ± 0.02	**0.32 ± 0.02**	**0.87 ± 0.02**	0.61 ± 0.04	0.52 ± 0.08
Sand %	8.43 ± 1.0	0.90 ± 0.03	**8.05 ± 1.0**	**0.91 ± 0.03**	13.6 ± 0.70	0.76 ± 0.04
Silt %	8.37 ± 0.91	0.69 ± 0.07	**5.06 ± 0.62**	**0.88 ± 0.03**	9.31 ± 0.48	0.61 ± 0.05
Clay %	7.58 ± 0.46	0.69 ± 0.05	**4.77 ± 0.48**	**0.88 ± 0.02**	8.80 ± 0.73	0.59 ± 0.05

Training performance does not tell the full story. 20% of the entire dataset was kept back for the training process before cross-validation was performed, to be used as a final test set for each algorithm. This allows the performance of each algorithm to be assessed on *truly unseen* data.

For this test, the *k* models were all evaluated on each data point. For each data point, the *k* results predicted by the models were averaged to give the final value. This method of creating an averaging ensemble model is a common technique to try to maximise results when small training sets are used. Section 6.4 presents these results for each algorithm.

It can be seen that for nearly every indicator, the performance on the test set was better than the averaged training performance. This is likely due to the fact that the ensemble model is less influenced by variations within the dataset.

The most interesting observation of these results is that for the ensemble models, ST–CGP is the best performing algorithm on *nearly every* indicator, in some instances by quite a margin. It outperformed random forests for twelve of the fourteen indicators tested. This indicates that ST–CGP is better able to generalise to unseen data; it does not overfit as much as the other algorithms. This echoes the results of the even parity test seen earlier.

Table 6.2: Performance of ST–CGP, Random Forest, and Neural Networks on the test set.

	ST–CGP		**Random Forest**		**Neural Network**	
Indicator	RMSE	r^2	RMSE	r^2	RMSE	r^2
Microbial Biomass	**138**	**0.74**	139	0.73	217	0.37
Respiration	**0.39**	**0.61**	0.44	0.50	0.52	0.28
WHC Water Content	**2.18**	**0.80**	2.48	0.74	3.62	0.43
Field Water Content	**1.85**	**0.83**	2.00	0.81	3.05	0.55
Soil Organic Matter (LOI)	**1.02**	**0.87**	1.31	0.78	1.75	0.61
Nitrate Nitrogen	**7.07**	**0.56**	7.38	0.52	8.35	0.39
Ammonium Nitrogen	**0.82**	**0.26**	0.90	0.13	0.89	0.13
Mg Available	**23.6**	**0.78**	30.4	0.64	38.3	0.43
K Available	**58.9**	**0.56**	64.6	0.47	85.5	0.08
P Available	**9.75**	**0.57**	12.3	0.32	13.2	0.22
pH	**0.32**	**0.88**	0.36	0.84	0.58	0.59
Sand %	**8.57**	**0.90**	9.54	0.88	12.8	0.79
Silt %	7.16	0.79	**5.64**	**0.87**	9.36	0.64
Clay %	7.06	0.74	**5.68**	**0.82**	9.31	0.54

One of the main advantages of ST–CGP is its ability to utilise multiple data types. Algorithms like random forests and neural networks are purely numerical—inputs must all be either integers or floating point numbers. In order to make a fair comparison between ST–CGP and the other algorithms, ST–CGP was constrained to this same restrictive behaviour for the results seen so far; the same 50 numerical features were provided to all algorithms.

This phase of the project was initially intended to close the loop of the indirect approach described in the previous chapter (sensor predicts soil health indicators, health indicators predict yield). However, one of the early outcomes of the project, derived from discussions with agronomists, soil scientists, and farmers, was that it is actually *more useful* to be able to predict the soil health indicators than the yield. This is because the yield labels—low, high, and reference—collected during this project are *localised*, meaning that what is considered "high" in one field could be "low" in another. In contrast, the soil health indicators are *global*; for example, 35ppm of phosphorous is the same in one field as another. Agronomists can use the soil health indicators to provide a holistic picture of soil health on a per-field basis, which is more valuable to farmers than a simple yield label. As such, it was not explored whether feeding the soil health indicator predictions back into the classifiers from the first phase of the project produced accurate classifications or not.

Chapter 7

Conclusion

This thesis set out to investigate whether adding an explicit, compile-time type system to CGP could deliver a reliable, interpretable, and broadly speaking *useful* evolutionary-programming framework. The resulting method—ST–CGP—integrates input and output types, and also holds a number of other improvements over standard CGP, supporting operators of arbitrary arity, adding full crossover and a new variant of crossover, both in a way that ensures offspring remain syntactically valid and type-safe, and adding multi-objective optimisation, and speed improvements from memoisation and caching. By screening out ill-typed graphs before execution and after genetic operations, the approach narrows the search space, shortens run times, and preserves the elegant genome representation that distinguishes CGP. Importantly, the type rules are lightweight: they demand no problem-specific tuning and coexist easily with the neutral mutations that are characteristic of CGP.

Empirical studies across computer vision, arable agriculture, and soil chemistry show that ST–CGP is applicable to a wide range of problems. In particular, type information does more than guarantee syntactic correctness. Because each operator declares exactly which data it accepts, numeric, Boolean, image, and time-series primitives can mix freely within a single population. That flexibility enables expressive, but more importantly, *powerful* programs: with only ten labelled images per class, ST–CGP matched a convolutional-network baseline on a large image-based dataset while running on a single CPU core. ST–CGP also performs well when typing is not involved: in yield-zone classification for forty-five UK fields, it trailed a tuned random forest by barely three percentage points yet exhibited an order-of-magnitude lower variance between runs, signalling a more reliable search. When supplied with five-minute gas-sensor traces, the same codebase produced regression models that met laboratory standards for twelve soil-health indicators and is already in commercial use. Across domains where typing was tested, successful programs remained markedly shorter than those produced by untyped CGP, suggesting a pruning effect of strong typing.

Several themes emerge. **First**, type constraints act as an intrinsic bias that guides evolution without heavy-handed heuristics. **Second**, modest crossover improves the evolutionary search performed by ST–CGP, regardless of typing. **Third**, the evolved graphs remain human-readable: clinicians can trace each image transformation, and agronomists can see which sections of a sensor trace or field history drive a prediction. This interpretability is key in fields where opaque "black box" models are unacceptable.

A limitation of this research is the scope and isolation of the evidence. Not every experiment involved typing, as mentioned, and the thesis does not include a full ablation that cleanly separates the effects of strong typing, crossover (including rewiring), and other optimisation choices; nor are the benefits of crossover/optimisation quantified across a wider problem suite. This may leave threats to internal and external validity, since observed gains may partly reflect operator/hyperparameter interactions and the limited set of domains studied. That said, the malaria results show statistically significant improvements across repeated runs, suggesting the effect is not purely due to noise.

Future work should replicate this level of statistical validation on more tasks that make heavier use of typing, alongside systematic ablations to attribute gains to each component. To test these themes more rigorously, a controlled factorial study could be designed in which the only experimental factors are: (i) the presence or absence of type constraints (i.e., typed vs. untyped ST–CGP) and (ii) the use and magnitude of crossover (e.g., crossover probability $p_c \in \{0, 0.05, 0.10, 0.20\}$), while holding all other settings fixed (operator set, mutation, selection, population size, initialisation procedure, and, critically, an identical evaluation budget). Each condition would be repeated over a sufficiently large number of independent random seeds and across multiple task families to ensure that any observed effects are robust and not dataset-specific. Performance and search efficiency could then be compared statistically between conditions (e.g., final test performance, fitness–evaluation learning curves, and evaluations-to-threshold), enabling an objective assessment of the contribution of type constraints and crossover to ST–CGP's behaviour.

This thesis began by setting out two main objectives: adding strong typing to CGP, and determining whether this addition yielded any benefits. Both of these objectives have been accomplished: the first explained in Chapter 3, and the second demonstrated across Chapters 4, 5, and 6. Furthermore, the addition of advanced optimisation techniques has allowed ST–CGP to perform competitively even in the absence of multiple types.

Overall, the evidence suggests that ST–CGP is a useful practical technique for data-sparse, computation-limited, and explanation-critical settings, especially when the problem benefits (or suffers!) from multiple data types. The framework thus offers a credible alternative to both classic GP and black box deep learning, and its modular design invites future work, especially with regard to richer data types, parallel evaluation, and automated discovery of domain primitives.

Bibliography

[1] Robert J Schalkoff. *Digital image processing and computer vision*, volume 286. Wiley New York, 1989.

[2] Marcus D Bloice, Christof Stocker, and Andreas Holzinger. Augmentor: An Image Augmentation Library for Machine Learning. *The Journal of Open Source Software*, 2(19):432, 2017.

[3] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *CoRR*, abs/1712.04621, 2017.

[4] Mark Nixon and Alberto Aguado. *Feature extraction and image processing for computer vision*. Academic press, 2019.

[5] Hanife Kebapci, Berrin Yanikoglu, and Gozde Unal. Plant image retrieval using color, shape and texture features. *Computer Journal*, 54(9):1475–1490, 2011.

[6] Syed Inthiyaz, P. V.V. Kishore, and B. T.P. Madhav. Pre-informed level set for flower image segmentation. In *Smart Innovation, Systems and Technologies*, volume 78, pages 11–20. Springer, 2018.

[7] C. Cavina-Pratesi, R. W. Kentridge, C. A. Heywood, and A. D. Milner. Separate channels for processing form, texture, and color: Evidence from fMRI adaptation and visual object agnosia. *Cerebral Cortex*, 20(10):2319–2332, 2010.

[8] Ming-Kuei Hu. Visual pattern recognition by moment invariants. *IRE Transactions on Information Theory*, 8(2):179–187, 1962.

[9] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[10] Jamie Shotton, Andrew Blake, and Roberto Cipolla. Contour-based learning for object detection. In *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, volume 1, pages 503–510. IEEE, 2005.

[11] Daekeun You, Sameer Antani, Dina Demner-Fushman, and George R Thoma. A contour-based shape descriptor for biomedical image classification and retrieval. In *Document Recognition and Retrieval XXI*, volume 9021, pages 194–205. SPIE, 2014.

[12] N. Kumar, P. N. Belhumeur, A. Biswas, D. W. Jacobs, W. J. Kress, I. C. Lopez, and J. V. B. Soares. Leafsnap: a computer vision system for automatic plant species identification. *Lecture Notes in Computer Science*, pages 502–516, 2012.

[13] J. Florindo, A. Backes, and O. Bruno. Leaves shape classification using curvature and fractal dimension. pages 456–462, 2010.

[14] J. Ma'touq and N. Alnuman. Comparative analysis of features and classification techniques in breast cancer detection for biglycan biomarker images. *Cancer Biomarkers*, 40:263–273, 2024.

[15] M. Obayya, A. Alhebri, M. Maashi, A. S. Salama, A. M. Hilal, M. I. Alsaid, A. E. Osman, and A. A. Alneil. Henry gas solubility optimization algorithm based feature extraction in dermoscopic images analysis of skin cancer. *Cancers*, 15:2146, 2023.

[16] C. Zhang, Y. Zheng, B. Guo, C. Li, and N. Liao. Scn: a novel shape classification algorithm based on convolutional neural network. *Symmetry*, 13:499, 2021.

[17] M. N. Abdullah, M. A. M. Shukran, M. R. M. Isa, N. S. M. Ahmad, M. A. Khairuddin, M. S. F. M. Yunus, and F. Ahmad. Colour features extraction techniques and

approaches for content-based image retrieval (cbir) system. *Journal of Materials Science and Chemical Engineering*, 09:29–34, 2021.

[18] Meenu Dadwal and Vijay Kumar Banga. Color image segmentation for fruit ripeness detection: A review. In *2nd International Conference on Electrical, Electronics and Civil Engineering (ICEECE'2012)*, 2012.

[19] J. Wu, B. Zhang, J. Zhou, Y. Xiong, B. Gu, and X. Yang. Automatic recognition of ripening tomatoes by combining multi-feature fusion with a bi-layer classification strategy for harvesting robots. *Sensors*, 19:612, 2019.

[20] Abdul Mueez. A cost-effective framework to predict the ripeness of any fruit based on color space. In *2020 IEEE Region 10 Symposium (TENSYMP)*, pages 1729–1733. IEEE, 2020.

[21] Rong Zhou, Lutz Damerow, Yurui Sun, and Michael M Blanke. Using colour features of cv.'gala'apple fruits in an orchard in image processing to predict yield. *Precision Agriculture*, 13(5):568–580, 2012.

[22] SA Oyewole and OO Olugbara. Product image classification using eigen colour feature with ensemble machine learning. *Egyptian Informatics Journal*, 19(2):83–100, 2018.

[23] Ruturaj Kulkarni, Shruti Dhavalikar, and Sonal Bangar. Traffic light detection and recognition for self driving cars using deep learning. In *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, pages 1–4. IEEE, 2018.

[24] Husam A Almusawi, Mohammed Al-Jabali, Amro M Khaled, Korondi Péter, and Husi Géza. Self-driving robotic car utilizing image processing and machine learning. In *IOP Conference Series: Materials Science and Engineering*, volume 1256, page 012024. IOP Publishing, 2022.

[25] Michael J Swain and Dana H Ballard. Color indexing. *International journal of computer vision*, 7(1):11–32, 1991.

[26] Greg Pass, Ramin Zabih, and Justin Miller. Comparing images using color coherence vectors. In *Proceedings of the fourth ACM international conference on Multimedia*, pages 65–73, 1997.

[27] Robert M Haralick and Linda G Shapiro. Glossary of computer vision terms. *Pattern Recognit.*, 24(1):69–93, 1991.

[28] R. Haralick, K. Shanmugam, and I. Dinstein. Texture features for image classification. *IEEE Transactions on Systems, Man, and Cybernetics*, 3(6), 1973.

[29] Robert M Haralick. Statistical and structural approaches to texture. *Proceedings of the IEEE*, 67(5):786–804, 1979.

[30] Dennis Gabor. Theory of communication. part 1: The analysis of information. *Journal of the Institution of Electrical Engineers-part III: radio and communication engineering*, 93(26):429–441, 1946.

[31] Goesta H Granlund. In search of a general picture processing operator. *Computer Graphics and Image Processing*, 8(2):155–173, 1978.

[32] Bruno A Olshausen and David J Field. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, 381(6583):607–609, 1996.

[33] AD Belsare, MM Mushrif, MA Pangarkar, and N Meshram. Classification of breast cancer histopathology images using texture feature analysis. In *Tencon 2015-2015 IEEE Region 10 Conference*, pages 1–5. Ieee, 2015.

[34] Sergio Varela-Santos and Patricia Melin. Classification of x-ray images for pneumonia detection using texture features and neural networks. In *Intuitionistic and type-2 fuzzy logic enhancements in neural and optimization algorithms: Theory and applications*, pages 237–253. Springer, 2020.

[35] Geun-Ho Kwak and No-Wook Park. Impact of texture information on crop classification with machine learning and uav images. *Applied Sciences*, 9(4):643, 2019.

[36] Stefania Barburiceanu, Serban Meza, Bogdan Orza, Raul Malutan, and Romulus Terebes. Convolutional neural networks for texture feature extraction. applications to leaf disease classification in precision agriculture. *IEEE Access*, 9:160085–160103, 2021.

[37] Facundo Pieniazek, Ana Sancho, and Valeria Messina. Texture and color analysis of lentils and rice for instant meal using image processing techniques. *Journal of Food Processing and Preservation*, 40(5):969–978, 2016.

[38] Richard Forsyth. Beagle - A darwinian approach to pattern recognition. *Kybernetes*, 10(3):159–166, 1981.

[39] Nichael Lynn Cramer. A representation for the Adaptive Generation of Simple Sequential Programs. *International Conference on Genetic Algorithms and the Applications*, pages 183–187, 1985.

[40] John R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. *Evolutionary Computation: The Fossil Record*, pages 578–584, 1989.

[41] John R. Koza. *Genetic Programming: On the Programming of Computers By Means of Natural Selection Complex Adaptive Systems*. MIT Press, 1992.

[42] John R Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA, 1994.

[43] John R Koza, David Andre, Forrest H Bennett, and Martin A Keane. *Genetic Programming III: Darwinian Invention & Problem Solving*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[44] John R Koza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[45] M. Sipper. Attaining human-competitive game playing with genetic programming. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4173 Lncs, page 13, 2006.

[46] Dennis G. Wilson, Hervé Luga, Sylvain Cussat-Blanc, and Julian F. Miller. Evolving simple programs for playing atari games. In *GECCO 2018 - Proceedings of the 2018 Genetic and Evolutionary Computation Conference*, pages 229–236. Acm, 2018.

[47] Sean Luke. Genetic Programming Produced Competitive Soccer Softbot Teams for RoboCup97. *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 214–222, 1998.

[48] Denis Robilliard and Cyril Fonlupt. Towards human-competitive game playing for complex board games with genetic programming. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9554, pages 123–135, 2016.

[49] John R. Koza. Human-competitive machine invention by means of genetic programming. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing: AIEDAM*, 22(3):185–193, 2008.

[50] John R. Koza, Forrest H. Bennett, and Oscar Stiffelman. Genetic programming as a darwinian invention machine. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1598(June):93–108, 1999.

[51] John R. Koza. Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, 11(3-4):251–284, 2010.

[52] David J. Montana. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[53] Thomas Haynes, Roger L Wainwright, Sandip Sen, and Dale A Schoenefeld. Strongly Typed Genetic Programming in Evolving Cooperation Strategies. In *Icga*, volume 95, pages 271–278, 1995.

[54] Thomas Haynes, Dale Schoenefeld, and Roger Wainwright. Type Inheritance in Strongly Typed Genetic Programming. 1998.

[55] Amaury Hazan, Rafael Ramirez, Esteban Maestre, Alfonso Perez, and Antonio Pertusa. Modelling expressive performance: A regression tree approach based on strongly typed genetic programming. In *Workshops on Applications of Evolutionary Computation*, pages 676–687. Springer, 2006.

[56] Walter Alden Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, 1994.

[57] Peter J Angeline. Advances in Genetic Programming. chapter Genetic Pr, pages 75–97. MIT Press, Cambridge, MA, USA, 1994.

[58] W a Tackett. Genetic programming for feature discovery and image discrimination. *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 303–309, 1993.

[59] W Banzhaf and W B Langdon. Some Considerations on the Reason for Bloat. *Genetic Programming and Evolvable Machines*, 3:81–91, 2002.

[60] Nicholas Freitag McPhee and Justin Darwin Miller. Accurate Replication in Genetic Programming. *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 303–309, 1995.

[61] W. B. Langdon and R. Poli. Fitness causes bloat: Mutation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1391(September):37–48, 1998.

[62] Riccardo Poli, Leonardo Vanneschi, William B. Langdon, and Nicholas Freitag McPhee. Theoretical results in genetic programming: the next ten years? *Genetic Programming and Evolvable Machines*, 11(3):285–320, Sep 2010.

[63] Maarten Keijzer and James Foster. Crossover bias in genetic programming. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4445 Lncs(May):33–44, 2007.

[64] Byoung-Tak Zhang and Heinz Mühlenbein. Balancing Accuracy and Parsimony in Genetic Programming. *Evolutionary Computation*, 3(1):17–38, 1995.

[65] Terence Soule. *Code Growth in Genetic Programming*. PhD thesis, University of Idaho, 1998.

[66] T. Soule and J. A. Foster. Effects of code growth and parsimony pressure on populations in genetic programming. *Evolutionary computation*, 6(4):293–309, 1998.

[67] Riccardo Poli. *Parallel distributed genetic programming*. University of Birmingham, Cognitive Science Research Centre Birmingham, UK, 1996.

[68] Riccardo Poli. Parallel distributed genetic programming applied to the evolution of natural language recognisers. In *AISB International Workshop on Evolutionary Computing*, pages 163–177. Springer, 1997.

[69] Riccardo Poli. Evolution of recursive transition networks for natural language recognition with parallel distributed genetic programming. *COGNITIVE SCIENCE RESEARCH PAPERS-UNIVERSITY OF BIRMINGHAM CSRP*, 1996.

[70] Jilian F. Miller. An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming approach. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1135–1142. Morgan Kaufmann Publishers Inc., 1999.

[71] Julian F. Miller and Stephen L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, 2006.

[72] Julian Francis Miller. *Cartesian Genetic Programming*, volume 43. Springer, 2003.

[73] Julian Miller. What Bloat? Cartesian Genetic Programming on Boolean Problems. *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 295–302, 2001.

[74] Simon Harding, Vincent Graziano, Jürgen Leitner, and Jürgen Schmidhuber. MT-CGP: Mixed type cartesian genetic programming. In *GECCO'12 - Proceedings of the 14th International Conference on Genetic and Evolutionary Computation*, pages 751–758. Acm, 2012.

[75] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[76] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[77] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter,

Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[78] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022.

[79] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 43(2):1–55, January 2025.

[80] Adrian de Wynter, Xun Wang, Alex Sokolov, Qilong Gu, and Si-Qing Chen. An evaluation on large language model outputs: Discourse and memorization. *Natural Language Processing Journal*, 4:100024, 2023.

[81] David Andre. Advances in Genetic Programming. chapter Automatica, pages 477–494. MIT Press, Cambridge, MA, USA, 1994.

[82] Markus M Breunig. Location independent pattern recognition using genetic programming. *Genetic algorithms and genetic programming at Stanford*, 1995:29–38, 1995.

[83] Hong Guo, L B Jack, and A K Nandi. Feature generation using genetic programming with application to fault classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 35(1):89–99, 2 2005.

[84] Jamie R Sherrah, Robert E Bogner, and Abdesselam Bouzerdoum. The Evolutionary Pre-Processor: Automatic Feature Extraction for Supervised Classification using Genetic Programming. *Genetic Programming 1997: Proceedings of the Second Annual Conference*, (July):304–312, 1997.

[85] Marc Ebner. On the Evolution of Interest Operators using Genetic Programming. *Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming*, pages 6–10, 1998.

[86] Ling Shao, Li Liu, and Xuelong Li. Feature learning for image classification via multiobjective genetic programming. *IEEE Transactions on Neural Networks and Learning Systems*, 25(7):1359–1371, 2013.

[87] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer, 2010.

[88] Riccardo Poli. Genetic programming for image analysis. In *Proceedings of the 1st annual conference on genetic programming*, pages 363–368. MIT Press, 1996.

[89] Yang Zhang and Peter I Rockett. Evolving Optimal Feature Extraction Using Multi-objective Genetic Programming: A Methodology and Preliminary Study on Edge Detection. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, Gecco '05, pages 795–802, New York, NY, USA, 2005. Acm.

[90] Héctor A. Montes and Jeremy L. Wyatt. Cartesian Genetic Programming for Image Processing Tasks. *Proceedings of the IASTED International Conference on Neural Networks and Computational Intelligence*, pages 185–190, 2003.

[91] Paulo Cesar Donizeti Paris, Emerson Carlos Pedrino, and M C Nicoletti. Automatic learning of image filters using Cartesian genetic programming. *Integrated Computer-Aided Engineering*, 22(2):135–151, 2015.

[92] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[93] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[94] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[95] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[96] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[97] Umut Tatli and Cafer Budak. Biomedical image segmentation with modified u-net. *Traitement du Signal*, 40(2):523–531, 2023.

[98] Hasib Zunair and A Ben Hamza. Sharp u-net: Depthwise convolutional network for biomedical image segmentation. *Computers in biology and medicine*, 136:104699, 2021.

[99] Narinder Singh Punn and Sonali Agarwal. Modality specific u-net variants for biomedical image segmentation: a survey. *Artificial Intelligence Review*, 55(7):5845–5889, 2022.

[100] Vanessa Buhrmester, David Münch, and Michael Arens. Analysis of explainers of black box deep neural networks for computer vision: A survey. *Machine Learning and Knowledge Extraction*, 3(4):966–989, 2021.

[101] Seong Joon Oh, Bernt Schiele, and Mario Fritz. Towards reverse-engineering black-box neural networks. In *Explainable AI: interpreting, explaining and visualizing deep learning*, pages 121–144. Springer, 2019.

[102] Zhongheng Zhang, Marcus W Beck, David A Winkler, Bin Huang, Wilbert Sibanda, Hemant Goyal, et al. Opening the black box of neural networks: methods for interpreting neural network models in clinical applications. *Annals of translational medicine*, 6(11):216, 2018.

[103] Yi Mei, Qi Chen, Andrew Lensen, Bing Xue, and Mengjie Zhang. Explainable artificial intelligence by genetic programming: A survey. *IEEE Transactions on Evolutionary Computation*, 27(3):621–641, 2022.

[104] Andreas Margraf, Henning Cui, Anthony Stein, and Jörg Hähner. Evolving processing pipelines for industrial imaging with cartesian genetic programming. In *2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 133–138. IEEE, 2023.

[105] Roman Kalkreuth, Gunter Rudolph, and Jorg Krone. More efficient evolution of small genetic programs in Cartesian Genetic Programming by using genotypie age. *2016 IEEE Congress on Evolutionary Computation, CEC 2016*, pages 5052–5059, 2016.

[106] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 4 2002.

[107] Michael D. Schmidt and Hod Lipson. Age-Fitness Pareto optimization. *Proceedings of the 12th Annual Genetic and Evolutionary Computation Conference, GECCO '10*, (2):543–544, 2010.

[108] Michael Schmidt and Hod Lipson. Age-Fitness Pareto Optimization. In Rick Riolo, Trent McConaghy, and Ekaterina Vladislavleva, editors, *Genetic Programming Theory and Practice VIII*, pages 129–146. Springer New York, New York, NY, 2011.

[109] Riccardo Poli, W.B. Langdon, and N.F. McPhee. *A Field Guide to Genetic Programing*. Number March. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008.

[110] S. Harding, J. F. Miller, and W. Banzhaf. Self modifying cartesian genetic programming: Parity. *2009 IEEE Congress on Evolutionary Computation, CEC 2009*, pages 285–292, 2009.

[111] John R Koza. Hierarchical automatic function definition in genetic programming. In *Foundations of Genetic Algorithms*, volume 2, pages 297–318. Elsevier, 1993.

[112] Chris Gathercole and Peter Ross. Tackling the Boolean even n parity problem with genetic programming and limited-error fitness. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 119–127, Stanford University, CA, USA, 13-16 jul 1997. Morgan Kaufmann.

[113] Sivaramakrishnan Rajaraman, Sameer K. Antani, Mahdieh Poostchi, Kamolrat Silamut, Md A. Hossain, Richard J. Maude, Stefan Jaeger, and George R. Thoma. Pre-trained convolutional neural networks as feature extractors toward improved malaria parasite detection in thin blood smear images. *PeerJ*, 2018(4):1–17, 2018.

[114] Simon Harding. Evolution of image filters on graphics processor units using cartesian genetic programming. *2008 IEEE Congress on Evolutionary Computation, CEC 2008*, pages 1921–1928, 2008.

[115] Simon Harding, Jürgen Leitner, and Jürgen Schmidhuber. Cartesian Genetic Programming for Image Processing. In Rick Riolo, Ekaterina Vladislavleva, Marylyn D Ritchie, and Jason H Moore, editors, *Genetic Programming Theory and Practice X*, pages 31–44. Springer New York, New York, NY, 2013.

[116] Mahdieh Poostchi, Kamolrat Silamut, Richard J. Maude, Stefan Jaeger, and George Thoma. Image analysis and machine learning for detecting malaria. *Translational Research*, 194:36–55, 2018.

[117] Sivaramakrishnan Rajaraman, Stefan Jaeger, and Sameer K. Antani. Performance evaluation of deep neural ensembles toward malaria parasite detection in thin-blood smear images. *PeerJ*, 7:e6977, 2019.

[118] Michael O'Neill and David Fagan. The Elephant in the Room: Towards the Application of Genetic Programming to Automatic Programming. pages 179–192, 2019.

[119] Michael O'Neill and Lee Spector. Automatic programming: The open issue? *Genetic Programming and Evolvable Machines*, 21(1-2):251–262, 2020.

[120] J. Zhang, M. G. A. v. d. Heijden, F. Zhang, and S. Bender. Soil biodiversity and crop diversification are vital components of healthy soils and agricultural sustainability. *Frontiers of Agricultural Science and Engineering*, 7:236, 2020.

[121] C. Wagg, K. Schlaeppi, S. Banerjee, E. E. Kuramae, and M. G. A. v. d. Heijden. Fungal-bacterial diversity and microbiome complexity predict ecosystem functioning. *Nature Communications*, 10, 2019.

[122] B. Mamatha, C. Mudigiri, G. Ramesh, P. Saidulu, N. Meenakshi, and C. L. Prasanna. Enhancing soil health and fertility management for sustainable agriculture: a review. *Asian Journal of Soil Science and Plant Nutrition*, 10:182–190, 2024.

[123] M. M. Tahat, K. M. Alananbeh, Y. A. Othman, and D. I. Leskovar. Soil health and sustainable agriculture. *Sustainability*, 12:4859, 2020.

[124] Review on the role of soil microorganisms on soil physico-chemical properties and plant growth. *Journal of Natural Sciences Research*, 2023.

[125] H. S. Sheoran, R. Kakar, N. Kumar, and .. Seema. Impact of organic and conventional farming practices on soil quality: a global review. *Applied Ecology and Environmental Research*, 17:951–968, 2019.

[126] Zhongke Jiao, Bo Liu, Enhai Liu, and Yongjian Yue. Low-pass parabolic fft filter for airborne and satellite lidar signal processing. *Sensors*, 15(10):26085–26095, 2015.