A Gradient-Based Method for Optimising the Output of Nondifferentiable Programs

Matthew Sjödin

May 19, 2016

Abstract

Gradient descent is a popular method for finding the maxima and minima of mathematical models of optimisation problems. However, with optimisation of computer programs becoming more common in, for example, machine learning, it would be useful to be able to harness the expressiveness of additional programming features such as conditional statements when modelling optimisation problems. Gradient descent cannot be applied to nondifferentiable functions such as imperative programs containing if statements. This paper proposes an algorithm that interpolates between the branches of if statements to transform a program into a differentiable function that gradient descent can operate on.

The gradients of programs are obtained using a new interpretation of automated differentiation of if statements that interpolates between their branches. The algorithm, inspired by simulated annealing, creates gradually stricter differentiable approximations of a program containing if statements that condition on comparisons of values, applying gradient descent to each approximation.

This algorithm is implemented in Java. Tests performed on 29 basic examples of differentiable and nondifferentiable programs suggest that this method can achieve high performance, requiring few runs of complex programs that could be computationally costly to run.

Contents

1 Introduction

Optimisation problems are ubiquitous in Computer Science, with applications ranging from neural network learning (Graves et al., 2014; Nielsen, 2015; Rumelhart et al., 1986) to computer vision (Läthén et al., 2009). Optimisation is not only a Computer Science problem; "it can determine the dramatic savings available to organisations that could not be achieved by other means" (Kallrath and Wilson, 1997) in the world of business and is used to assess economic policies for sustainable development (Cannon et al., 2005).

A widely used (Chen et al., 2009; Läthén et al., 2009; López-Rodríguez et al., 2007) algorithm for solving optimisation problems is gradient descent, which uses the gradient of a differentiable function (Sz.-Nagy, 1965) at a point to determine how much the function's inputs should be adjusted (Murphy, 2012). This paper adapts gradient descent, which requires differentiable functions, to work on computer programs with potentially disconnected output, so that the expressiveness of imperative programming can be used when creating models of complex systems that need to be optimised.

Gradient descent needs gradients in order to optimise a function. Fortunately, there are efficient techniques for differentiating computer programs and finding their gradients at a point. Automated differentiation is an "accurate and efficient" (Naumann, 2011) method for which there is no "serious competitor" (Naumann, 2011). Programming languages such as Stan (Carpenter et al., 2015) and Stalin∇ (Siskind and Pearlmutter, 2008) have automated differentiation built in, enabling, for example, gradient-based methods in probabilistic computing.

However, these implementations of automated differentiation are not built for optimisation. Conditional statements, in particular if statements, are a key feature of imperative programming languages, but they introduce branch points – discontinuities where the branch taken changes – which are treated as nondifferentiable (Griewank, 2000) or whose gradients are given as intervals (Kearfott, 1996) by traditional automated differentiation systems. This is insufficient to find the inputs that maximise or minimise the outputs of programs such as the following:

```
f(x):
    if x >= 2 then
        return x - 2
    else
        return x^2 + 1
```

The output of this program can be plotted on a graph:

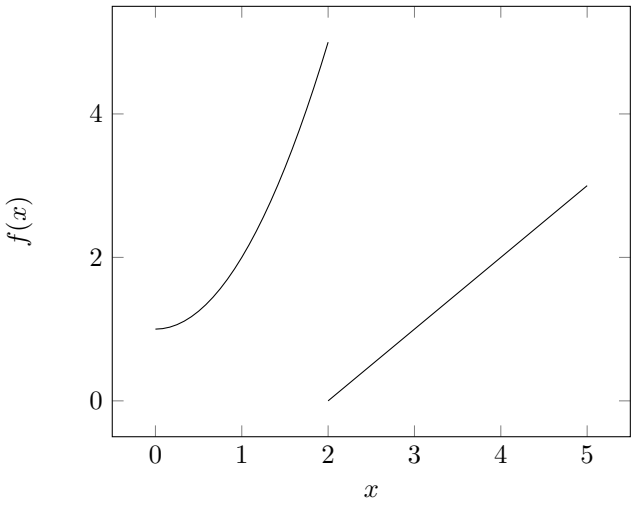

Figure 1: The output of the program `f(x)`

By inspection, the minimum is at `x = 2` and is caused by the downwards "jump" in output, where it is not continuous (Sz.-Nagy, 1965). In this example, the gradient of the output is positive on both sides of the branch point and therefore traditional methods of automated differentiation will not give any indication that there is a local minimum nearby. There has been work on detecting these discontinuities in a computer-aided verification context (Chaudhuri et al., 2010), but where previous work identifies discontinuities and warns of their negative effects, this paper removes their nondifferentiability and optimises the output of programs containing them.

This paper makes a number of contributions. A new interpolative interpretation of if statements in automated differentiation is developed that preserves information about jumps in value caused by discontinuities in output between different branches. These interpolations are defined for two simple classes of if statement conditions. In addition, an algorithm that uses this variant of automated differentiation is designed, implemented and tested. This algorithm uses an iterative approach inspired by simulated annealing to generate differentiable approximations of imperative programs containing if statements and then optimises their outputs. The algorithm can efficiently find values for a program's parameters that output a locally optimal value, to a given precision.

If we were to run the algorithm on the above program (`f(x)`), it would generate successively stricter approximations to perform gradient descent on, as shown in Figure 2.

As demonstrated in the "Implementation and Experimentation" section of this paper, this approach gives positive results. The algorithm is applied to a variety of programs, giving promising results across all of them at high levels of precision. In particular, the iterative approach that creates gradually stricter mathematical models of programs significantly cuts down the amount of time

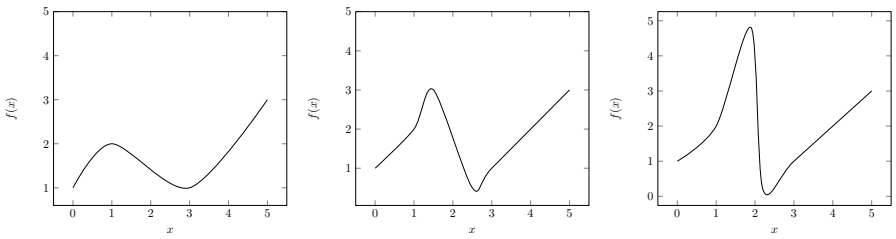

Figure 2: From left to right, 3 successively stricter differentiable approximations of the program `f(x)`, as they might be generated by our algorithm

taken to find a local extremum. At a precision of 7 decimal places, programs that were taking tens of millions of runs to optimise without the outer loop take less than a thousand iterations.

Section 2 of this paper gives an overview of optimisation problems, features of simple imperative programming languages, simulated annealing and gradient descent. It describes how, by using automated differentiation, gradient descent can be used to optimise programs. In §3, the paper demonstrates why standard gradient descent and automated differentiation are insufficient to maximise or minimise the output of computer programs containing basic if statements. In §4, an algorithm taking inspiration from simulated annealing is proposed that iteratively applies gradient descent to successively stricter differentiable mathematical models of a computer program to efficiently calculate inputs that will give locally maximal or minimal output. The algorithm's correctness is then explained. The implementation details of the algorithm are discussed in §5 and a series of experiments conducted testing the efficiency of the algorithm and questioning the design decisions made. Finally, the paper compares the approach taken to related work in §6 and in §7 discusses the successes and failures of this method and suggests future areas of research, such as how this approach could be extended from programs containing basic if statements to full imperative programs.

The Java code written to implement and test the algorithm, including the programs used for testing, is available online at
`https://github.com/MatthewSjo/LocalExtremaOfImperativePrograms`

2 Background

Optimising the output of a function is the problem of determining which inputs make it produce the minimum or maximum output. For $f(x) = x^2$, for example, the minimum output occurs when $x = 0$. We are concerned with functions where the inputs and output are real-valued, so the numbers could take any value in the real numbers. For now we assume that the functions are continuous and can be differentiated. For the function $f(x) = x^2$, $f'(x) = 2x$.

A local minimum occurs at the bottom of a valley in the continuous graph representing output of the function, in that there is some $\epsilon > 0$ such that any change of input within a distance of ϵ will not cause the output to decrease (Binmore, 1982). This means that by determining when $f'(x) = 0$, we can determine where these changes in direction occur and therefore where local maxima and minima lie. For our example ($f'(x) = 2x$), $2x = 0$ when $x = 0$, so this is the minimum. This method works in any number of dimensions and therefore with any number of parameters.

Gradient descent (ascent) is an algorithm for finding local minima (maxima) (Murphy, 2012). Starting from any point in the output, it uses differentiation to determine whether increasing parameters will increase or decrease the output. The variables that track the best input can therefore move "downhill" in the graph of the output until the algorithm hits the bottom (Murphy, 2012). Crucially, the distance it moves at each step is dependent on how steep the change is: at high gradients, there will not be any extrema nearby, so gradient descent can make larger moves. Smaller moves can then be made when closer to an extremum. This makes gradient descent a quick and efficient approach. Furthermore, since the only information it uses is gradient, it can ignore the actual value of the output and does not have to keep track of a large amount of information at any one time.

We can use simple imperative programs to represent mathematical functions. Intuitively, any program that only uses simple combination, such as addition, subtraction, multiplication and division, of variables and constants can be expressed as a mathematical function, and we can express simple mathematical functions using a computer program. For example, the following program represents the mathematical function $f(x) = x^2$:

```
f(x):
    y = x*x
    return y
```

Therefore we seek to interpret computer programs as mathematical functions and use gradient descent to optimise their output. Automated differentiation (Rall, 1981) is a method of determining the gradient of a program's output at any point. It works by redefining real numbers as "dual numbers" (Clifford, 1873), which are a pair (x, c) where x represents the value and c represents the coefficient of ϵ. ϵ is a number sufficiently small such that ϵ^2 is considered equal to 0. ϵ represents a minuscule change in input, and its coefficient is how

much the value would change if the parameter that is being differentiated were changed. Setting the coefficient to 1 for the parameter we are differentiating and 0 otherwise, we can calculate how much changing the parameters would affect the output, and therefore calculate the gradient.

In the program above, $y = (x^2, 2x)$ as $(x + 1\epsilon)(x + 1\epsilon) = x^2 + 2x\epsilon + \cancel{\epsilon^2}$, so the gradient is $2x$. This method of automated differentiation is extremely efficient, as it simply adds a constant amount of computational overhead to every arithmetic operation. By using automated differentiation to find the gradients of programs, gradient descent can find the inputs that optimise their outputs.

Another technique that this paper takes inspiration from is simulated annealing (Kirkpatrick et al., 1983). When metals are forged, they are heated to a high temperature and cooled slowly. When metals are hotter and more liquid they are easier to mould into general shapes and as they cool down finer detail can be added. It is difficult to make large changes to the shape of a hard, cool metal and any fine detail added to a liquid will not keep its shape.

This process, annealing, inspires "simulated annealing" in local search, which experiments with moving in the wrong direction while the "temperature" is "hot" and reduces the probability of movement in the wrong direction over time, during cooling, focusing on the fine detail of the extrema that have been found. This means that if a suboptimal local extremum is found, there is a chance the algorithm will move on and find a better one (Kirkpatrick et al., 1983).

The algorithm this paper proposes also takes inspiration from annealing, by starting with a high "temperature", making simple, rapid optimisation on a simplified mathematical function, and operating at higher detail on a more precise differentiable approximation of the computer program as the system "cools down".

3 Problem

Simple imperative programs built on assignment to variables using basic arithmetic represent differentiable mathematical functions and can therefore be optimised using gradient descent. However, if statements introduce discontinuities that do not meet the core assumptions that gradient descent relies on. An if statement comprises a condition, a true branch and an optional false branch. If the condition evaluates to true given the current state, the true branch is taken, otherwise the false branch is taken.

The issue with using if statements is not simply one of differentiability. There are methods to deal with conditionals in automated differentiation (Kearfott, 1996), but they do not fit the application of gradient descent. Automated differentiation systems typically consider functions containing branches differentiable except for the point where the branch taken changes, at which point the gradient is undefined (Griewank, 2000) or given as an interval (Kearfott, 1996). As the following example demonstrates, allowing disconnected output of functions introduces a new family of extrema that gradient descent cannot find, whichever method of calculating gradient we use at branch points.

Consider the following program:

```
f(x):
    if x >= 2 then
        return x - 2
    else
        return x + 2
```

Some of the output of this program is shown in the graph below.

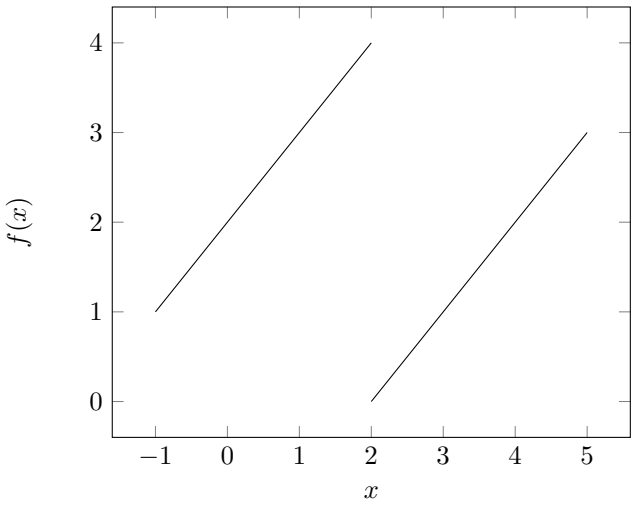

Figure 3: The output of the program `f(x)`

The program `f(x)` clearly has a local minimum at `x = 2`, but the gradient remains constant in its vicinity, so gradient descent cannot find this local minimum. Gradient descent assumes output is connected and changes in value must be reflected in the gradient, whereas it is the jump in the output of `f(x)` that induces its local minimum in this case.

Whatever value for the gradient is given at the branch point, "the chance of ever hitting a crack between two smooth pieces exactly appears to be zero" (Griewank, 2000), so the immediate context needs to indicate that there is a nearby extremum. If we are to reap the benefits of gradient descent when optimising the output of computer programs, we will have to modify the algorithm to detect these implicit extrema without sacrificing the features that make gradient descent so attractive, and without losing its ability to find extrema caused by a change of direction in the graph.

Another key feature of imperative programming languages that our language contains is reassignment. When dealing with the output of different branches of if statements, it is insufficient to deal with the values that the branches return when executed, as the branches may modify variables and return no values themselves. Any method of interpreting if statements and finding extrema, whether zero-gradient or caused by jumps, will have to take reassignment into account, dealing with the changes to state that the branching logic of if statements cause, as well as any values that if statements return.

4 Algorithm

4.1 Design

Standard gradient descent cannot be applied to computer programs as they may represent discontinuous functions containing implicit local extrema. However, gradient descent can be applied to branching programs, in particular those containing if statements, if they are transformed into continuous, differentiable functions. This paper concerns the class of if statements that compare the values that two expressions, such as `5` and `2 times x`, evaluate to given the state of program variables, and test which is the larger or whether they are equal. Whichever test that the condition performs, the "jump" in value – the discontinuity in output caused by the change in branch taken – occurs at the point where the two expressions evaluate to the same value.

The trick, then, is to interpolate between branches when the values these expressions evaluate to are close to one another to bridge the gap between the output of the two branches. If the range of interpolation is made sufficiently smaller than the precision to which we want to find the minimum, we can be certain that any minima the interpolation induces are within the desired precision. This is because the range of acceptable error accounts for every point where the interpolated function returns a different value than the original program.

It is vitally important that the interpolation performed preserves existing extrema in the range of interpolation, creates them when they are implicit in the jumps, and does not introduce them when they should not be there. Great care is taken to meet these conditions in the following interpolation formulae, and their correctness is argued below.

Interpolation between branches involves taking a weighted average of the values that two branches output for any given input. Weighting of the branches should be dependent on the difference in value between the two expressions of the if statement condition. Intuitively, the further from branching, the stronger the influence from the current branch, up to some range of interpolation, past which the current branch is taken on its own.

Interpolation of the output of imperative programs is not, however, as simple as combining the values returned by two branches. In an imperative programming language with reassignment of variables, the branches of individual if statements may not return values themselves, yet they can still have a profound effect on the output of the program on the whole, due to the way they assign values to variables. Therefore, interpolation should be between the outputs of a program as a whole, only with a different branch taken for that if statement.

If statement conditions can be divided into two subcategories, whose interpolation are defined using two sets of formulae below. The first of these subcategories is boundary conditions that branch depending on which of the two expressions being tested evaluates to a higher value. At the point where they are equal the output changes from one function to another, so, at this point, the weighting should be 1:1 (the branches should be weighted equally).

This way, at the point where the branch taken changes, the value returned will remain the same and the output will no longer be discontinuous.

To achieve the correct proportions, over any range of interpolation, T, a useful tool is the exponent adjuster, called `expAdjust` in the implementation. This is referred to as A in the following formulae for conciseness. Its purpose is to adjust the range in which an exponentiation of e changes from 0.5 to 1. More specifically, it is calculated as $\ln(2)/T$, which is a simplification of $-\ln(0.5)/T$. In the formulae below, this will give the correct performance varying between $d = 0$ and $d = T$.

To be able to keep direction of argument consistent between branches when arguing correctness, it is useful to give two formulae, one for the "left" branch and one for the "right". That is, one formula for when the "left" branch is taken where moving towards the branch point is moving "right", and one formula for when the "right" branch is taken and moving to the "right" (by increasing or decreasing variables) increases the distance from the branch point. The ordering of which branch is the left or right is irrelevant, as the two cases result in equivalent interpolation.

Throughout the following definitions, function calls such as $f(x)$ and $g'(x)$ represent output and gradients at the points in the input corresponding to the value d, which takes a value between 0 and T and is used as a measure of distance from the branch point or edge of the range of interpolation. d and x are closely linked but it is written this way for clarity of argument. Note that this relationship means that differentiating with regards to x is the same as differentiating with regards to d. The functions may be multivariate but are written univariate as we differentiate with regards to one parameter at a time so other parameters are considered constants.

If the range of interpolation is T, the function defining the left branch is f, the function defining the right branch is g and the distance towards the branch point from the left end of the range of interpolation is d, then the interpolated output, when the branch that is taken is the left branch, is as follows:

$$e^{-Ad}f(x) + (1 - e^{-Ad})g(x)$$

This interpolation is needed for gradient descent, which is based exclusively on the gradient of the function, not its output values, so the above formula is designed to differentiate easily, giving the following gradient at the point x:

$$g'(x) + (f'(x) - Af(x) - g'(x) + Ag(x))e^{-Ad}$$

The output when taking the right branch is as follows, with d being the distance from the branch point in favour of the right branch:

$$e^{A(d-T)}g(x) + (1 - e^{A(d-T)})f(x)$$

This also differentiates simply, giving the gradient at x of:

$$f'(x) + (g'(x) - Af(x) - f'(x) + Ag(x))e^{A(d-T)}$$

These interpolated gradients are easy to compute so interpolation is rapid. $f(x)$, $g(x)$, $f'(x)$ and $g'(x)$ are all calculated via automated differentiation and A only needs to be calculated once each time T is changed.

The second subcategory of if statement conditions uses equality, rather than boundaries. One branch is taken when two expressions evaluate to the same value, and the other is taken in all other cases. These conditions can be phrased as equalities or inequalities, giving dual results. This definition is written in terms of equality conditions.

When interpolating the branches of equality conditions, the "not equal" branch should have a higher weighting the further from the branch point the input is, and the output should be entirely that of the "equal" branch at the point where those two expressions give the same value.

With T the range of interpolation, d the distance from the branch point (the absolute difference between the two expressions), f the function defining the "equal" branch and g defining the "not equal" branch, the output is as follows:

$$(d/T)^2 f(x) + (1 - (d/T)^2) g(x)$$

This differentiates to:

$$(d^2 f'(x) + 2df(x) + T^2 g'(x) - d^2 g'(x) - 2dg(x))/T$$

This complies with our intuitive understanding of how interpolation of equality conditions should work.

It is insufficient to perform gradient descent with interpolation over sufficiently small ranges to give an answer to the desired degree of precision. As interpolation would be performed between very different values over very small ranges, it would lead to narrow ranges of very steep gradients. To compensate for this, a very small step size needs to be used. If gradient descent is started at a large distance from any minima, there will be a substantial number of redundant gradient checks over a large part of the input space, in case there are sudden large changes in gradient.

A better approach is to start with a larger step size to identify where minima are and then reduce this step size over time to identify their exact positions to within the desired precision. The range of interpolation should vary with this step size so the gradients induced are not too steep for gradient descent to handle.

Hence, the operation of the algorithm is an iterative approach inspired by simulated annealing, where the range of interpolation acts as the temperature. On each iteration, gradient descent is performed with the current range of interpolation and step size, with a precision dependent on these, as it would be wasteful to find an answer to a high precision for a weaker problem.

After each iteration, the step size and range of interpolation are reduced by some factor, which represents the cooling process, and once the sum of the interpolation range and the precision given to inner gradient descent is less than the desired precision (given as a parameter to the algorithm), a true local minimum has been found to the correct precision. In the "Implementation and

Experimentation" section, experiments are performed testing whether iterating over interpolation ranges in this fashion actually leads to an improvement in performance.

4.2 Full Algorithm

The following pseudocode gives the details of the algorithm. It is not immediately obvious where the interpolation is taking place. It is assumed that the program that is passed to the algorithm can be called in "differentiation mode", where automated differentiation takes place during the running of the program. The programming language is designed such that interpolation between branches is performed when an if statement is evaluated. This is why gradient descent is given $expAdjust$ and T, so that it can pass these to the program to perform interpolation over the correct ranges.

The procedure for gradient descent is the standard method, save for these two additional parameters that need to be passed to the program for it to return a gradient at a point. It is assumed that gradient descent will optimise a vector of starting parameter values and return a vector of updated parameter values in the same order. In the pseudocode below, it is assumed that gradient descent's parameters are as follows, in order.

1. The function being minimised
2. The precision to which an answer is required
3. The step size
4. The starting points of the parameters
5. $expAdjust$ as a parameter for interpolation
6. T as a parameter for interpolation

A number of constants are used in the procedure. Any of these could be passed as parameters to the algorithm, but they are not obviously variables in the sense that they should vary by program and user, so they are given as constants. Appropriate values for these constants are explored in the "Implementation and Experimentation" section. These constants are as follows.

startRange The starting value for the range of interpolation T

coolingSpeed The factor by which the range of interpolation T is reduced on each iteration

$\boldsymbol{\gamma}$ The step size from which the effective step size γ' is calculated

The parameters of the algorithm are the values that are variable with each use. These parameters are as follows, in order.

program The program being minimised

precision The precision to which an answer is required

startPointVector The initial guess of optimal parameters as a starting point

The value returned by the algorithm is a vector containing inputs, in order, that will output a local minimum. Adapting the algorithm to find a local maximum instead of a local minimum is trivial. Instead of using gradient descent, gradient ascent would be used. The procedure for finding a minimum is as follows:

```
1: procedure FindMinimum(program, precision, startPointVector)
2:     parameterVector ← startPointVector
3:     T ← startRange
4:     done ← false
5:     while ¬done do
6:         expAdjust ← ln(2)/T
7:         γ′ ← γ/expAdjust
8:         parameterVector ← GradientDescent(program, γ′ × precision,
               γ′, parameterVector, T)
9:         if T < precision then
10:            done ← true
11:        T ← T × coolingSpeed
12:    return parameterVector
```

4.3 Correctness of Boundary Conditions

The purpose of this subsection is to demonstrate some properties of the algorithm to show that it deals with discontinuities correctly. For clarity, the visualisation of "left" and "right" branches continues to be used, and the ordering of the branches is irrelevant, as the two ways give equivalent results. The x-axes of the graphs represent the difference between the expressions of the if statement's condition, and thus distance from the branch point, and the y-axes represent the output. Again, d will be used as in the definitions as a useful measure for the distance x is into the range of interpolation or how far it is from the branch point.

The following argument ignores gradients of zero. These are covered by one of the appropriate positive or negative gradient cases. An explicit matching would be necessary for a formal proof of correctness but is omitted for clarity and conciseness in this argument. It may seem as if branches without a jump are ignored, but in reality the direction of jump simply decides which of the two cases of Assumption 1a is taken. However, a weakening of this assumption would be needed for a proof of correctness that allows discontinuities without jumps. A robust explanation of how multiple parameters are dealt with is also omitted, but would not make the argument much more complicated as we differentiate with regards to each parameter separately.

Correctness is only shown for finding minima. Finding maxima is analogous. It is assumed for now that the programs for which local minima are found do not use equality conditions, which will be discussed later.

First, some assumptions and assertions have to be made.

Assumption 1

$\exists l \in \mathbb{R}$ such that for every point of branching in the program whose left and right branches define functions f and g respectively, l defines the range r of real numbers within distance l of the branch point such that:

(a) $(\forall x \in r. \forall y \in r. f(x) \geq g(y)) \vee (\forall x \in r. \forall y \in r. g(x) \geq f(y))$

i.e., within the range of interpolation, one branch will always output higher values than the other

(b) $\exists k, k'. f'(x) \approx k \wedge g'(x) \approx k'$

i.e., l is sufficiently small that the gradients of the branches are effectively constant

(c) No other condition creates a branch point within a distance l of this branch point

i.e., each range of interpolation can be treated separately

(d) $l > precision$

i.e., these assumptions will hold on the final iteration of the algorithm, when gradient descent's output will be returned

Sufficiently small l keeps this assumption from being too restrictive, and some examples of why the assumption is necessary are given in the appendices. There may be relaxations of this assumption that work, which is a potential extension of the analysis.

Assumption 2

Gradient descent, which is a standard technique extensively covered elsewhere (Murphy, 2012), works correctly on the transformed programs.

Assumption 3

Every minimum in the function can be detected at the given precision. This is necessary as within any arbitrarily small range, there could be any amount of variance that could be skipped over by larger step sizes.

Assertion 1

$$0 \leq d \leq T \implies 0.5 \leq e^{A(d-T)} \leq 1 \wedge 0.5 \leq e^{-Ad} \leq 1$$

Trivial by exponential rules and definition of A (i.e. $expAdjust$).

Assertion 2

Once we perform our transformation, there will be no discontinuities in the resulting transformed program, as asserted when designing the algorithm. We want to know that gradient descent is given a function it can use.

Overall Idea

The purpose of Assumption 1 is to analyse the algorithm at the point where all false minima have been ironed out. If $T \leq l$, which is guaranteed to happen as there is at least one iteration where $T < precision$, then this assumption can be used. As we assume gradient descent to be correct, and as by definition of our interpolation the program operates normally outside the ranges of interpolation, we only need to show that behaviour within the range of interpolation is as desired. There are three things that need to be shown for any branch point's range of interpolation:

Statement 1 If the range of interpolation contains a local minimum, then that local minimum is found

Statement 2 If a local minimum is found in that range of interpolation, then there is a local minimum in that range of interpolation

Statement 3 If a true local minimum is found in that range of interpolation, then the actual local minimum is within a distance of *precision*

Given the nature of the discontinuities and Assumption 1, proofs of these statements will be case analyses ranging over whether the gradients of each of the two branches are positive or negative, and whether the discontinuity is a jump upwards or downwards.

Proof of Statement 1

By Assumption 2, inner gradient descent will correctly find minima that exist in the transformed program.

Therefore it is sufficient to show that discontinuities that cause disconnected minima will, via interpolation, produce local minima for the inner gradient descent to find.

Let this be when $T < l$ from Assumption 1. This is a strict $<$ as this allows us to assert what the sign of the gradient will be at the points where traversal of the graph enters and exits the range of interpolation. Therefore induced minima could be on the edge of the range of interpolation in cases where, for example, the interpolated gradient is negative throughout but a positive gradient is guaranteed when leaving the range of interpolation.

We perform at least one iteration with $T \leq precision$ by the termination condition of the algorithm, and $l > precision$ in Assumption 1d, so $T < l$ on at least the final iteration.

We use a case analysis to show that every time a branch in the program causes a jump that results in a disconnected local minimum, there is a continuous local minimum in the transformed program in the range of interpolation.

The cases are defined by three variables regarding the left and right branches f and g: the sign of $f(x)$, the sign of $g(x)$, and whether $f(x) > g(x)$ of $g(x) > f(x)$. That is, whether each branch slopes up or down and whether the jump is upwards or downwards.

Using the interpolated gradients for the ranges governed by the left and right branches, we then show that one of the left or right branches must introduce a local minimum.

Note that half of the cases do not introduce true local minima, and it is therefore the other four cases that are covered by this case analysis. The rest are covered by the proof of Statement 2.

Cases 1 and 2 Left branch has negative gradient, right branch has positive gradient, jump in either direction.

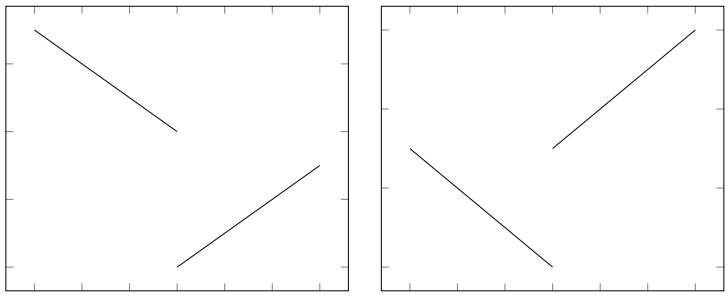

Figure 4: Sample output representing cases 1 and 2

These cases are trivial by Assertion 2; entering the range of interpolation the gradient is negative, and leaving it the gradient is positive, and the graph is connected. Therefore the range must contain a local minimum.

Cases 3 and 4 The other cases are analogous, but are given fully in the appendices.

As gradient descent will find any local minima in the transformed program and every minimum of any kind in the program is reflected by a local minimum in the transformed program, the algorithm will find any local minima in the program.

Proof of Statement 2

By Assumption 2, inner gradient descent will only find local minima that exist in the transformed program.

It is sufficient to show that discontinuities that do not cause disconnected minima will not, when interpolated, produce any local minima.

As in our case analysis for Statement 1, we perform a case analysis using the remaining four cases of the form of disconnects, only this time showing none of them can result in a local minimum in the transformed program.

Case 5 Both branches have positive gradient, jump upwards.

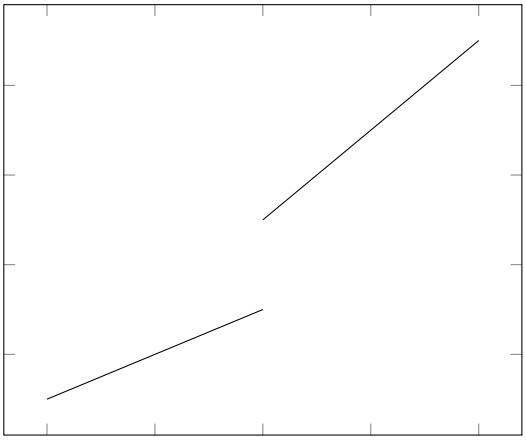

Figure 5: Sample output representing case 5

Prove that for both branches with interpolation, the gradient will always be positive, and therefore no extra minima will be added.

Gradient at left branch:

$$g'(x) + (f'(x) - Af(x) - g'(x) + Ag(x))e^{-Ad}$$

e^{-Ad} positive by Assertion 1, so show coefficients positive.

$f'(x)$ positive by case and Assumption 1b.

$g'(x)(1 - e^{-Ad})$ positive by case, Assumption 1b, and Assertion 1.

$A(g(x) - f(x))$ positive by case and Assumption 1a (by case, $\forall x \in r.g(x) > f(x)$)

Therefore the gradient is always positive when taking the left branch.

Gradient at right branch:

$$f'(x) + (g'(x) - Af(x) - f'(x) + Ag(x))e^{A(d-T)}$$

$e^{A(d-T)}$ positive by Assertion 1, so show coefficient positive.

$g'(x)$ positive by case and Assumption 1b.

$f'(x)(1 - e^{A(d-T)})$ positive by case, Assumption 1b and Assertion 1.

$A(g(x) - f(x))$ positive by case and Assumption 1a (by case, $\forall x \in r.g(x) > f(x)$)

Therefore the gradient is always positive when taking the right branch.

As the gradient is always positive in the range of interpolation, this case does not introduce any false minima.

Case 6 Left branch has positive gradient, right branch has negative gradient, jump upwards.

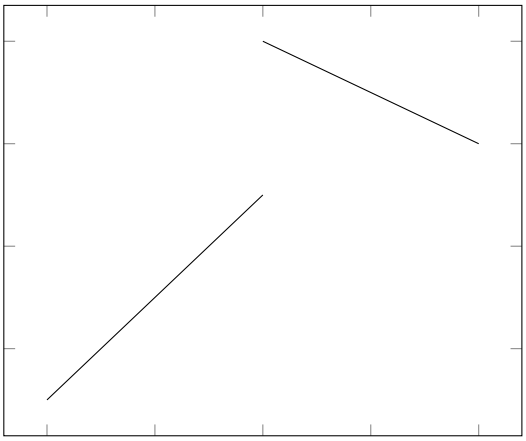

Figure 6: Sample output representing case 6

For this case it is insufficient to show that the gradient is just in one direction in the range of interpolation, as there will clearly be a local maximum.

Instead, we should show that from the first point in either branch with negative gradient, the gradient will be negative for the rest of the range of interpolation.

First assume the first point with negative gradient is in the left branch.

This means, by definition of interpolation for the left branch, that

$$(Af(x) + g'(x))e^{-Ad} > g'(x) + (f'(x) + Ag(x))e^{-Ad}$$

First show that this condition continues for the rest of the left branch.

$g'(x)$ is negative, so the fact it becomes more significant as e^{-Ad} decreases does not invalidate the condition.

For the other terms, e^{-Ad} is, of course, equal on both sides, so show that for any values of d greater than at this maximum:

$$Af(x) + g'(x) > f'(x) + Ag(x)$$

$f'(x)$ and $g'(x)$ are assumed to be constant by Assumption 1b and the dominance of the $f(x)$ term over the $g(x)$ term will only increase (as $f'(x) > 0$ and $g'(x) < 0$), so this holds.

Now show that this negative gradient would continue to the right side.

By definition of interpolation for the right branch, the condition for negative gradient is:

$$(f'(x) + Af(x))e^{A(d-t)} > f'(x) + (g'(x) + Ag(x))e^{A(d-t)}$$

We seek to find the worst case, where the gradient is as likely as possible to be positive, by maximising the magnitude of the positive terms and minimising the magnitude of the negative terms. Therefore assume the first point with negative gradient is at the start of the range of interpolation, which implies the most significant possible $g(x)$:

$$\begin{aligned} Af(x) + \cancel{g'(x)} &> \cancel{g'(x)} + f'(x) + Ag(x) \\ Af(x) &> f'(x) + Ag(x) \end{aligned}$$

This gives us the following inequality at the point most likely to have a positive gradient in the right branch, which is the branch point:

$$\begin{aligned} \frac{1}{2}(f'(x) + Af(x)) &> \frac{1}{2}(f'(x) + Ag(x) + f'(x)) \\ &> f'(x) + \frac{1}{2}Ag(x) \end{aligned}$$

$g'(x) < 0$ by case, so:

$$\begin{aligned} f'(x) + \frac{1}{2}Ag(x) &> f'(x) + \frac{1}{2}Ag(x) + \frac{1}{2}g'(x) \\ \frac{1}{2}(f'(x) + Af(x)) &> f'(x) + \frac{1}{2}(Ag(x) + g'(x)) \\ (f'(x) + Af(x))e^{A(d-t)} &> f'(x) + (g'(x) + Ag(x))e^{A(d-t)} \end{aligned}$$

This is the condition for negative gradient in the worst case.

Therefore if the first point with a negative gradient is in the left branch, the gradient will continue to be negative for the entire range of interpolation.

Now show that the gradient to the right of the first point with a negative gradient will continue to be negative if the first point with a negative gradient is in the right branch.

The condition that becomes true to make the gradient negative is:

$$(f'(x) + Af(x))e^{A(d-t)} > f'(x) + (g'(x) + Ag(x))e^{A(d-t)}$$

Show that this continues to hold once it first holds.

As x increases, so too does $e^{A(d-t)}$, so the contribution of positive $f'(x)$ diminishes. Therefore need to show that:

$$f'(x) + Af(x) > g'(x) + Ag(x)$$

As in the previous case, $f'(x)$ and $g'(x)$ are assumed constant, $f(x)$ is growing, and $g(x)$ is getting smaller.

Therefore, in the range of interpolation, there will be a single maximum, to the left of which the gradient is exclusively positive, and to the right of which the gradient is exclusively negative.

Therefore there are no minima in the range of interpolation.

Cases 7 and 8 The other cases are analogous, but are given fully in the appendices.

4.4 Correctness of Equality Conditions

It is possible to prove a weaker form of correctness for equality conditions, in that the algorithm will correctly find minima within the ranges of interpolation of branch points caused by equality conditions, but due to the nature of the interpolation, it will incorrectly identify additional local minima. For example, consider the following graph of the output of a program:

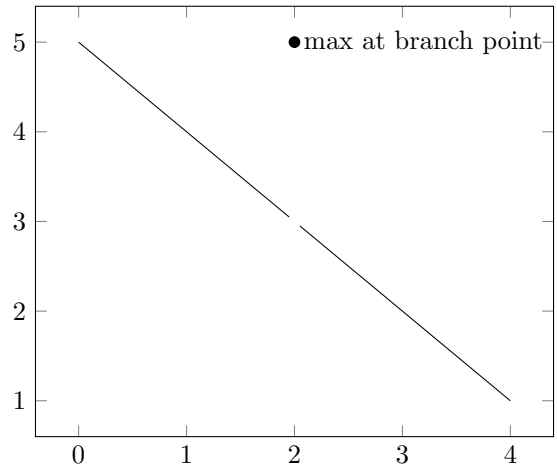

Figure 7: True output of a simple program containing an equality condition

When interpolated, the graph would look something like this:

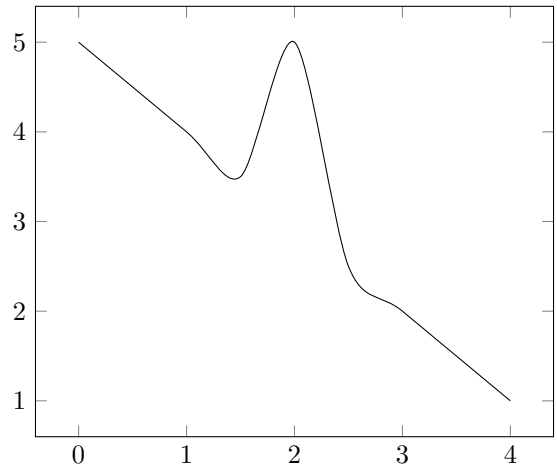

Figure 8: Interpolated output of the same program

The issue in this case is that a local minimum is identified immediately before the local maximum. Though this point fits a certain intuitive notion of what a local minimum is (in that to its immediate left and right the output increases), it is not a true local minimum (Binmore, 1982).

This is mitigated by extensions to gradient descent that find global, rather than local, minima. It is, however, disappointing that this approach to interpolation does not achieve the same success with equality conditions that it does with boundary conditions.

5 Implementation and Experimentation

This section starts by exploring some of the implementation details and optimisations used when implementing the algorithm. The section then examines various design decisions and demonstrates, through experimentation, that they were the correct options. It also presents experiments demonstrating that certain uncontrollable variables that affect the performance of the algorithm, such as the complexity of the program that the user provides, do not have a significant impact on speed. Further tests are given in the appendices.

5.1 Implementation Details

Implementation of the language itself is not important to the operation of the algorithm, so the compilation of written programs is irrelevant. The language is directly defined in an object-oriented manner as abstract trees. There are four kinds of statements:

AssignmentStatement Assign a value to a variable

ReturnStatement Return the value an expression evaluates to

IfStatement Condition leading to a true branch and a false branch

EndStatement A blank statement representing an empty branch or the end of a branch that returns no values

Program structure in terms of successive statements is expressed in `AssignmentStatements` and `IfStatements`, which each store a pointer to the next statement to be executed after they finish. The program is structured this way (rather than, for example, storing statements as a list) as this gives if statements access to the code that follows them so that the algorithm can interpolate using the results that a different branch would lead to.

`AssignmentStatements` and `ReturnStatements` operate using expression trees, which perform basic operations such as extracting the value of a variable from the current state, providing a constant value, or combining the values given by other expressions using basic arithmetic.

Code that defines some of the programs used for testing is given in the appendices.

Other language details are irrelevant, except for two details of program execution. Firstly, when a program is run it is given a state that represents starting parameter values and can be passed from statement to statement and changed, as well as functions for calculating the weightings of each branch for the two types of if statement condition and a value `smoothingRange` that stores the current range of interpolation. Secondly, dual numbers are used instead of real numbers in expressions and state to enable automated differentiation.

As using dual numbers instead of real numbers implicitly performs automated differentiation simply through the definition of their arithmetic operations, the definitions of all the expressions are trivial, as are `ReturnStatements`

and `AssignmentStatements`. `IfStatements` are more complicated, as they have to perform interpolation. This starts with conditions, which return not just a boolean representing whether the condition succeeded (and thus whether the true or false branch should be taken), but also a double precision number representing the distance from the branch point. Intuitively, the higher this number, the higher the weighting of the branch that is actually taken, and if this number is higher than the range of interpolation, interpolation should not occur.

During evaluation, `IfStatements` first evaluate their condition. If the distance from the branch point is larger than `smoothingRange`, the if statement operates as usual, following the appropriate branch. If interpolation should take place, the full program is evaluated from this point with two copies of the state, one generated by each of the branches. Using the distance from the branch point and the functions given for calculating weightings, the results can be interpolated according to the formulae already defined.

The actual interpolation, then, only occurs during the evaluation of `IfStatements` and implicitly through the arithmetic of dual numbers. Therefore, gradient descent can treat the interpolated program as a "black box" that spits out gradient values when given inputs. Thus it is possible to use totally unmodified gradient descent for multiple parameters, but given our method of transforming computer programs to differentiable functions, two optimisations make a substantial difference.

Firstly, given the "sharp corners" of functions such as $f(x) = |x|$ and their lack of variance in gradient, the step size for each parameter is slowed independently each time gradient descent changes direction. Though the functions are now differentiable they are not necessarily smooth curves whose varying gradients will activate gradient descent's built-in slowdown features.

The other optimisation is limiting the "effective" gradient at each step. This means that when dealing with extremely high gradients, such as those caused by very narrow ranges of interpolation, gradient descent will not jump to a distant part of the input space. Experiments supporting the inclusion of both of these features appear in the appendices.

Implementation of the full algorithm reads much like the pseudocode. It feeds parameters to the inner gradient descent, obtains the results and updates its current values for the parameters. Once the temperature T is less than half of precision, the range of interpolation is covered by acceptable error and we can be confident that the answer is accurate to the desired degree of precision.

The smoothing functions are simplified in the implementation provided. A single value is used to represent distance, and the formulae are expressed in terms of functions representing the branch taken and the branch not taken, rather than some notion of "left" and "right" branches. It is clear that this implementation is equivalent. This implementation was not used in the design of the algorithm as the method given in this paper is conceptually clearer and makes the argument for correctness easier to make and understand.

Full implementation code, written in Java 8, including the programs used for testing, is available online at
`https://github.com/MatthewSjo/LocalExtremaOfImperativePrograms`

5.2 Testing Methodology

The purpose of this paper is to propose an algorithm for the optimisation of the output of arbitrarily complex computer programs. As running them could take significantly longer than any algorithmic overhead, the most suitable metric for determining efficiency of the algorithm is the number of times programs have to be run to determine a correct answer (and indeed, whether a correct answer is reached at all). The main variable in terms of analysing variation in complexity is the precision to which an answer is required.

`ProgramTracker` is a wrapper for statements that counts the number of times the program it contains is run. Building on this, `OptimisationTest` represents a full test, allowing the creator of a testing suite to give the name of a test, the program being tested, a flag of whether it's a maximisation or minimisation test, and an expected answer. By parametrising these variables a test can be run iteratively on a large number of different programs and the results aggregated.

`TestPrograms` gives a list of 29 `OptimisationTests`. These are designed to cover every case from the explanation of correctness, every comparison operator for conditionals, and maximisation and minimisation variants for each test. They also vary greatly in terms of complexity, to determine how much of a difference that makes to runtime, which is explored more fully in the first test. There are also a number of fine implementation details that differ across the programs, such as whether constants or variables come first in comparisons. This successfully highlighted subtle implementation bugs caused, for example, by failure to make deep copies of states when running variants of the same program.

`TestingSuite` is, effectively, a static class containing a number of standardised tests that can be run on lists of `OptimisationTests` such as those produced by `TestPrograms`. These can produce tables of results given in count of runs of programs and highlight any specific tests that failed and the reason why, allowing for effective debugging and analysis.

Most of the tables of results examined in the rest of this section aggregate the results for all of the 29 tests in `TestPrograms`, but they were also run individually so any outlier programs could be found. A number of tests gave good results for all except a few programs. One case where this happened was high precision with multiple variables. This uncovered bugs caused by poor step size cooldown for multiple variables, which was originally applied uniformly across all parameters, and is now independent for each. For all of the tests with aggregated results in this version of the system, results are consistent across programs, with no outliers.

The first two tests examine variance caused by factors outside of the algorithm's control. These are the program given by the user and the distance from the correct answer of the starting point (which is arbitrarily chosen). The next test questions the fundamental choice of the simulated annealing-based approach. Additional tests, examining how the constants used in operation of the algorithm are chosen and exploring other algorithmic design decisions, are given in the appendices.

5.3 Test 1: How does time taken to find an answer vary with the complexity of the program?

Before we can perform large-scale aggregated tests over our 29 programs, we have to be sure results are relatively consistent across those programs. More specifically, if one kind of program performs very poorly but the others are very efficient, poor results could be hidden by the good and the algorithm would seem more efficient than it is.

Table 1 shows number of runs of programs required to optimise them over a range of precisions for all 29 testing programs.

	Precision						
Name	$\mathbf{10^{-2}}$	$\mathbf{10^{-3}}$	$\mathbf{10^{-4}}$	$\mathbf{10^{-5}}$	$\mathbf{10^{-6}}$	$\mathbf{10^{-7}}$	$\mathbf{10^{-8}}$
minimise basic x squared	72	98	121	147	170	195	219
maximise 3 - (x-3)^2	41	57	72	87	102	118	133
maximise 3 - x^2	49	65	80	96	110	126	141
minimise x*y	276	376	472	572	668	768	864
maximise 3 - ((x - 3)^2 + (y - 4)^2 + (z - 5)^2)	132	177	222	270	315	360	405
minimise if x >0 return x else return -x	113	117	120	125	129	135	139
maximise if x <0 return x else return -x	80	83	86	91	96	101	105
minimise if x >0 return x else return -x (reassign)	113	117	120	125	129	135	139
maximise if x <0 return x else return -x (reassign)	80	83	86	91	96	101	105
disconnected minimum using less than	211	390	425	660	721	1000	1046
disconnected minimum using less than or equal to	211	390	425	660	721	1000	1046
disconnected minimum using greater than	211	390	425	660	721	1000	1046
disconnected minimum using greater than or equal to	211	390	425	660	721	1000	1046
disconnected maximum using less than	161	307	351	552	605	850	912
disconnected maximum using less than or equal to	161	307	351	552	605	850	912
disconnected maximum using greater than	161	307	351	552	605	850	912
disconnected maximum using greater than or equal to	161	307	351	552	605	850	912
min passing through positive gradient, jump up	76	79	80	83	85	88	90
min, disconnect of positive gradient, jump down	381	738	833	1258	1400	1939	2034
max, disconnect of pos then neg gradient, jump up	347	703	802	1228	1371	1921	2019
max, disconnect of pos then neg gradient, jump down	211	396	432	662	728	1003	1051
min, disconnect of neg then pos gradient, jump up	352	650	716	1101	1234	1686	1852
min, disconnect of neg then pos gradient, jump down	419	773	869	1294	1434	1991	2109
max, disconnect from neg gradient jump up	304	598	668	1062	1195	1626	1765
max, pass through region of neg gradient jump down	76	79	82	87	92	97	102
minimum at equality point in program	180	361	398	632	691	965	1015
maximum at equality point in program	304	598	668	1062	1195	1626	1765
maximum with repeated conflicting downward jumps	320	619	686	1079	1211	1666	1836
minimum with repeated conflicting upward jumps	101	105	108	113	117	123	127

Table 1: Runs required, by precision, for each of the 29 programs used for testing

The results demonstrate that the required precision has an insignificant effect on the number of runs of the program (with optimal values for algorithm constants). The results seem to support the argument that the algorithm essentially works in two stages, first identifying a place where a local extremum occurs and then taking a roughly constant number of steps per parameter per level of precision to drill down.

Increasing the number of parameters leads to a slight decrease in performance, but not a significant one. Note, in particular, the fact that the test optimising over three variables takes less time than that optimising over two, suggesting the number of variables isn't the most significant factor.

The most significant factor affecting performance seems to be implicit extrema. Fortunately, the basic tests of if statements were very fast, as were optimisations that had to pass through a range of interpolation, rather than finding an extremum within such a range. This means that each parameter will only take this performance hit once, as this form of the algorithm seeks to find one local extremum. It is not the interpolation or the branching that causes the inefficiency, rather it is drilling down on the implicit extremum once it has been found.

5.4 Test 2: How much slower is optimisation when starting further from the true answer?

The purpose of the algorithm is to find the inputs that give the best outputs. Though the user may have some idea of what the optimal input is, the arbitrarily chosen starting point could be very far from the true answer. It is therefore important to test how many more times the program has to be run when starting further away from any local extrema.

The optimal inputs for all 29 programs are in the range 0 to 5, so by ranging upwards from a starting point of 5, the distance from the correct answer could be correctly parametrised. Tests were run at two different precisions to see if this compounds any slowdown. The secondary purpose of this test is to find an appropriate starting range. The hope is that with a sufficiently large initial step-size, the gap to any local extrema could be closed quickly.

This test was also run using a number of starting ranges above 0.1, starting at 0.2, all of which failed for at least one program, jumping past and failing to find local extrema. Table 2 displays results for varying starting points and starting ranges, for two levels of precision.

Precision 10^{-3}	**Starting Range**				
Starting Point	**0.0001**	**0.001**	**0.01**	**0.05**	**0.1**
5	74945	7638	897	409	333
6	87027	8847	1018	433	345
7.5	104862	10631	1197	468	361
10	134509	13596	1493	529	391
15	193804	19525	2086	646	450
20	253099	25454	2679	765	511
50	608865	61031	6237	1477	867

Precision 10^{-6}	**Starting Range**				
Starting Point	**0.0001**	**0.001**	**0.01**	**0.05**	**0.1**
5	308087	9960	112096	713	616
6	108320	11169	1602	738	629
7.5	288530	13055	22755	775	645
10	317710	16171	50437	834	674
15	376955	21846	18793	952	733
20	436199	27775	3263	1069	793
50	792551	63352	58307	1781	1151

Table 2: Runs taken for different starting points and starting ranges, for two precisions

As expected, with lower starting ranges, the lower step size causes the algorithm to take a substantial number of steps to arrive at the local minimum. For larger, optimal starting ranges, distance from the true start point is an insignificant factor. The results are very encouraging, as we can choose a starting range such as 0.1.

5.5 Test 3: Why have an outer loop, rather than immediate full-precision gradient descent?

This paper introduces the use of interpolation between branches of if statements and the use of a simulated annealing-style system for cooldown of detail using gradient descent. It is important to demonstrate that the outer loop implementing the cooldown gives a sufficiently large gain in performance to justify its existence.

The two tables of Table 3 compare results between normal gradient descent using the interpolation proposed by this paper and the algorithm including the outer loop. Note that one result is negative, which is caused by integer overflow while counting the number of runs required.

Without outer loop (basic gradient descent)

Starting Point	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
5	132	1381	15018	163835	1777644	19172785
6	156	1622	17434	187998	2019265	21588838
7.5	191	1979	21002	223671	2375989	25156298
10	251	2572	26931	282965	2968934	31085741
15	369	3758	38789	401554	4154822	42944625
20	488	4945	50649	520142	5340710	54803508
50	1200	12059	121802	1231676	12456041	-22145506

With outer loop (actual algorithm)

Starting Point	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
5	174	190	333	373	555	616
6	187	202	345	387	568	629
7.5	203	219	361	403	584	645
10	231	248	391	433	613	674
15	292	308	450	492	674	733
20	351	368	511	552	733	793
50	708	724	867	908	1090	1151

Table 3: Comparison of naïve approach to full algorithm for a number of starting points and precisions

Clearly, the method using an outer loop is significantly faster than normal gradient descent. This is because it cannot be assumed that gradients will become shallower closer to extrema, as there are implicit extrema. Because an implicit extremum could happen anywhere in the output, the step size has to be very small. However, if we immediately start at a small enough step size for correct results to be found at full precision, the input space would be traversed far too slowly to be able to use this interpolative approach to optimising programs containing if statements. Note that this improvement in performance over standard gradient descent is specifically for the small step sizes necessary for this method of interpolation, and does not mean that this approach would perform better than ordinary gradient descent in all cases.

5.6 Additional Tests

Additional tests question design decisions such as limiting maximum step size, as well as testing which values for the constants are most appropriate. They are explained in more detail and their results are given in the appendices.

6 Related Work

6.1 Functional Programming Languages using Automated Differentiation

This paper explores automated differentiation in an imperative programming context, but many authors feel that automated differentiation is "most naturally embodied in a functional programming language" (Manzyuk, 2012) because "the differentiation operator is almost a paradigmatic example of a higher-order function" (Manzyuk, 2012). Siskind and Pearlmutter (2008) say that "numerical computing could greatly benefit from more expressive programming languages" in a functional style. They invented STALIN∇ to be a higher-order functional programming language with efficient differentiation using automated differentiation at its core.

Hence, work on using automated differentiation in a functional context is focused on one of the key elements of functional programming: functions as first-class citizens. In STALIN∇, "derivative" is defined as a higher-order function. Manzyuk (2012) takes this one step further, by creating an "extension of the simply typed λ-calculus with pushforward operators" that can "incorporate forward automatic differentiation... into functional programming". By rooting automated differentiation in the simply typed λ-calculus, theoretical guarantees can be made about use of automated differentiation with higher-order functions.

Research into automated differentiation in functional and imperative programming need not be separate. Work on automated differentiation of higher-order functions in functional languages can be applied to higher-order imperative languages. There is nothing about the work on if statements in this paper that is not applicable to if statements in functional programming languages, and implementation of the algorithm in a functional programming setting is an interesting area of future work.

6.2 Neural Networks

Many of the concepts examined in this paper are also used in machine learning applications, in particular neural networks. Gradient descent has been useful in neural networks for years; Rumelhart et al. (1986) took advantage of gradient descent's efficiency in learning harder neural network mappings, such as XOR, in the 1980s. They use gradient descent for learning using training examples: "the system first uses the input vector to produce its own output vector and then compares this with the desired output". If these values differ, "the weights are changed to reduce the difference" by gradient descent. Using gradient descent is a natural and widely used (Graves et al., 2014; Nielsen, 2015; Sutskever et al., 2013) approach to machine learning, and any improvements to gradient descent's power, such as being able to find the extrema of the output of imperative programs, could lead to further improvements to neural networks.

Work in this paper also bears similarities to work on multivalued recurrent neural networks. López-Rodríguez et al. (2007) use "Functional Annealing... to

help optimization techniques to avoid some local minima of the objective function". They define a sequence of approximations to the function being optimised and minimise these in turn. As in the system presented in this paper, these approximations tend towards the true function.

López-Rodríguez et al. (2007) use functional annealing to simplify functions and allow global, rather than local, minima to be found. This paper, in contrast, uses approximation of the functions being minimised to make a function continuous and differentiable. Though functional annealing is used to improve two different aspects of gradient descent, the two uses of functional annealing are not necessarily incompatible. In a system where the first few iterations already work on approximations to a function, further approximation to solve additional problems could be convenient. More specifically, it would be interesting to see how these methods of approximation interact, and whether their approach to avoiding local minima could be applied to the algorithm presented in this paper.

The motivator for the work in this paper was recent work in Neural Turing Machines, which were invented by Graves et al. (2014), researchers at Google DeepMind, to "extend the capabilities of neural networks by coupling them to external memory resources". This system combines neural networks with key features of computers, while still being "differentiable end-to-end, allowing it to be efficiently trained with gradient descent". Other attempts have been made to store information in neural networks, such as "Long Short-Term Memory" (Hochreiter and Schmidhuber, 1997), which also learns using gradient descent.

Previous researchers sought to extend mathematical, differentiable neural networks with features of programming languages, while this paper takes the opposite approach, by starting with computer programs and making approximations that allow them to be optimisable via gradient descent.

6.3 Optimisation of Nondifferentiable Functions

There has been much work done on the optimisation of various classes of nondifferentiable functions. For example, the "subgradient method is a simple algorithm for minimizing a nondifferentiable convex function" (Boyd et al., 2003). It operates by using "step lengths that are fixed ahead of time, instead of an exact or approximate line search as in the gradient method" (Boyd et al., 2003). This is just one of many examples of methods for optimising convex functions (Boyd and Vandenberghe, 2004). Unfortunately, imperative programs are not necessarily expressible as convex functions.

Many authors focus on finding classes of nondifferentiable functions they can create optimisation algorithms for. Murray and Overton (1978), for example, identify "common types of nondifferentiable functions" arising from certain "approximation problems". They construct variants of gradient descent that find minima of nondifferentiable functions defined by their mathematical properties.

This paper's key difference to previous work is that the class of functions being optimised is the practically applicable class of functions represented by imperative programs, rather than classes of nondifferentiable functions with certain mathematical properties.

7 Conclusion

7.1 Achievements

This paper demonstrates that the discontinuities caused by the branching logic of if statements present a significant problem when maximising or minimising the output of imperative programs. Interpolating between these branches, however, gives promising results, and it has been shown that it is possible to optimise gradient descent for use with these differentiable approximations of imperative programs. Though this method of interpolation is only applied to one feature of advanced imperative programs, it could be extended to deal with other imperative constructs. It is encouraging that the difficult problem of optimising the output of imperative programs can be transformed into the simpler task of optimising differentiable mathematical functions.

When testing its ability to optimise the output of 29 sample imperative programs, the algorithm took, in the worst case, roughly 500 more steps for each decimal point of precision. At 7 decimal places the algorithm required little more than 2000 runs to optimise the output of any program. Though these experiments are not, of course, conclusive, these results suggest that this method can achieve high performance, requiring few runs of complex programs that could be computationally costly to run.

This approach to dealing with if statements in automated differentiation of imperative programs is not incompatible with work on functional programs. The strategy of finding a way to optimise programming constructs, rather than classes of nondifferentiable functions, was successful, leading to a practical solution to a programming problem. There are many fields that perform gradient descent on mathematical models of practical problems, from image segmentation (Läthén et al., 2009) to analysing grain-size in sedimentary deposits (Chen et al., 2009), and this work is a step towards enabling those models to be expressed as imperative computer programs.

7.2 Limitations and Future Work

This algorithm ensures that output will be within a certain precision from a true local extremum, but given the nature of disconnected maxima and minima, running the program with the inputs returned by the algorithm may give suboptimal results. This is because if a maximum or minimum is at a branch point, both sides of that branch point are represented in the range of precision. If one of those branches gives suboptimal performance, there is a risk that this branch may be taken with the input returned by the algorithm. A quick fix may be to perform a scan of values of inputs near those returned, which finds what branch returns the better output. This does not, however, scale to large numbers of parameters.

The algorithm as it is defined in this paper only finds local extrema, not global. There are many extensions to gradient descent that help it escape local extrema, such as momentum-based approaches (Sutskever et al., 2013), and as

gradient descent is essentially unmodified, it should be possible to use these in conjunction with the existing algorithm without any further modifications. Techniques such as the momentum method may also make the algorithm significantly faster (Qian, 1999). It is encouraging that the algorithm performs successfully without such speed optimisations. It would be an interesting line of future enquiry to see which improvements to gradient descent are most compatible with our algorithm.

As mentioned when discussing correctness, the method used to interpolate equality conditions returns secondary local extrema that are not useful, and indeed are not true local extrema in the mathematical sense. Any extension that finds global, rather than local, extrema should ensure that it discards false extrema such as these.

The conditions of the if statements discussed in this paper simply compare the values of one expression to another. In reality, if statements can contain boolean logic, using conjunction, disjunction and negation of conditions. It might not be difficult to add these features. Conjunction might be expressed using nested if statements and negation can be pushed to the deepest level using De Morgan's laws, negating the comparison operators used. Disjunction may be more complicated, however. It would require an analysis of which conditions are within the range of interpolation and which are not, and perhaps a variant on the interpolation formulae that takes multiple conditions into account.

If statements are just one feature of imperative programming; it is of particular interest also to be able to optimise programs that contain while loops. As the algorithm relies on running the program whose output is being optimised, the algorithm would have the stipulation that it is only guaranteed to terminate when the program itself is guaranteed to terminate, given any input. In such an extension of the algorithm, while loops might be unfolded into if statements. Then, with the assumption that all of these loops would terminate, and the fact that automated differentiation is performed while running the program, this unfolding would only happen a finite number of times per while loop. Once the true run of the program leaves the while loop, the run representing the other branch for the purposes of interpolation would be unable to repeat the loop. It is unclear whether this would be sufficient to give appropriate results and this is an area for future research.

References

Binmore, K. G. (1982). *Mathematical Analysis: a straightforward approach*, chapter 11, page 101. Cambridge University Press, 2 edition.

Boyd, S. and Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press.

Boyd, S., Xiao, L., and Mutapcic, A. (2003). Subgradient methods. In *Notes for EE392o*. Stanford University.

Cannon, M., Kouvaritakis, B., and Huang, G. (2005). Modelling and optimisation for sustainable development policy assessment. *European Journal of Operational Research*, 164:475–490.

Carpenter, B., Lee, D., Brubaker, M., Riddel, A., Gelman, A., Goodrich, B., Guo, J., Hoffman, M., Betancourt, M., and Li, P. (2015). Stan: A probabilistic programming language. *Journal of Statistical Software*.

Chaudhuri, S., Gulwani, S., and Lublinerman, R. (2010). Continuity analysis of programs. In *Principles of Programming Languages (POPL)*, pages 57–70.

Chen, H., Lv, X., and Qiao, Y. (2009). Application of gradient descent method to the sedimentary grain-size distribution fitting. *Journal of Computational and Applied Mathematics*, 233:1128–1138.

Clifford, W. K. (1873). Preliminary sketch of bi-quaternions. In *Proceedings of the London Mathematical Society, 4*, pages 381–395.

Graves, A., Wayne, G., and Danihelka, I. (2014). Neural turing machines. In *arXiv preprint arXiv:1410.5401*.

Griewank, A. (2000). *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Philadelphia, Penn.: Society for Industrial and Applied Mathematics.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9:1735–1780.

Kallrath, J. and Wilson, J. (1997). *Business Optimisation Using Mathematical Programming*. Macmillan Press Ltd.

Kearfott, R. B. (1996). Automatic differentiation of conditional branches in an operator overloading context. In *Computational Differentiation: Techniques, Applications, and Tools*, pages 75–81. SIAM.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *SCIENCE*, 220(4593).

Läthén, G., Andersson, T., Lenz, R., and Borga, M. (2009). Momentum based optimization methods for level set segmentation. In *Scale Space and Variational Methods in Computer Vision, Second International Conference*, pages 124–136.

López-Rodríguez, D., Mérida-Casermeiro, E., Galán-Marín, G., and de Lazcano-Lobato, J. M. O. (2007). Stochastic functional annealing as optimization technique: Application to the traveling salesman problem with recurrent networks. In *30th Annual German Conference on AI, KI 2007*.

Manzyuk, O. (2012). A simply typed λ-calculus of forward automatic differentiation. In *Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVIII)*.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. The MIT Press.

Murray, W. and Overton, M. L. (1978). Steplength algorithms for minimizing a class of nondifferentiable functions. Technical Report CS-TR-78-679, Stanford University, Department of Computer Science.

Naumann, U. (2011). *The art of differentiating computer programs: an introduction to algorithmic differentiation*. Society for Industrial and Applied Mathematics.

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press.

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural Networks 12*, pages 145–151.

Rall, L. B. (1981). *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer, Berlin.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). *Learning Internal Representations by Error Propagation*, chapter 8, pages 318–362. MIT Press Cambridge.

Siskind, J. M. and Pearlmutter, B. A. (2008). Using polyvariant union-free flow analysis to compile a higher-order functional-programming language with a first-class derivative operator to efficient fortran-like code. In *ECE Technical Reports*.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning*.

Sz.-Nagy, B. (1965). *Introduction to Real Functions and Orthogonal Expansions*, chapter 2. Oxford University Press.

Appendices

A Justification of Assumptions in Correctness of Boundary Conditions

This appendix gives examples of edge cases that make it necessary to include the restrictive Assumption 1a and Assumption 1b in the explanation of the correctness of boundary conditions.

$$(\forall x \in r. \forall y \in r. f(x) \geq g(y)) \vee (\forall x \in r. \forall y \in r. g(x) \geq f(y))$$

In the following graph, the filled lines represent the functions that define the branch currently taken, and the dashed lines are their extensions used for interpolation.

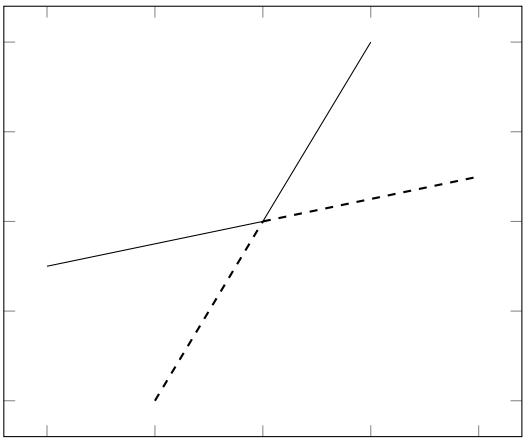

Figure 9: Output of a program. Filled lines are the actual output and dashed lines are the extensions of branches

Interpolation of the above graph could potentially give results such as the following:

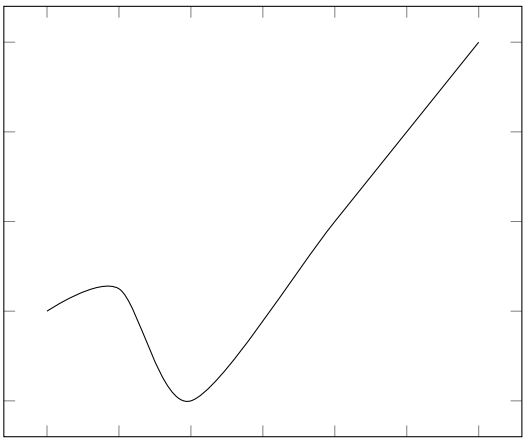

Figure 10: Results of interpolating the branches of the program whose output is shown in the previous figure

The right branch is sufficiently steep that its influence creates a false local minimum in the left. If, throughout the range of interpolation, one branch gives a higher output than the other, this kind of incorrect interpolation cannot occur.

$$\exists k, k'.f'(x) \approx k \wedge g'(x) \approx k'$$

A constant gradient is required as otherwise additional extrema could be added. Consider the following graph, where a local minimum appears. The hard corners are unrealistic, but are used to demonstrate the point by making the interpolation simple. The interpolation – the dashed line – is given as a direct average (so very near the branch point). This produces a local minimum even though the signs of the gradients of both lines do not change.

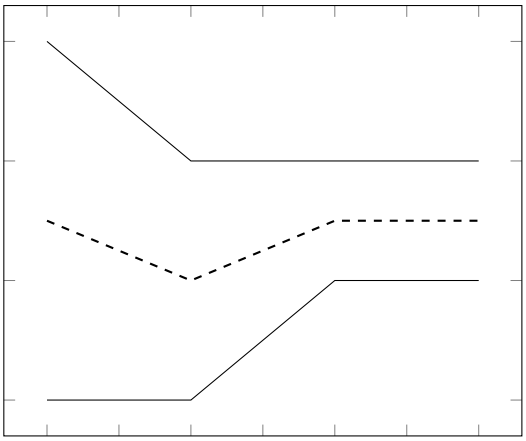

Figure 11: Interpolation showing that variable gradients can produce false minima. Filled lines are true functions and the dashed line is their interpolation

B Additional Cases from Correctness of Boundary Conditions

There were some analogous cases in the case analyses of the proof, which were omitted for conciseness. These are given here in full.

Additional Cases of Statement 1

Case 3 Both branches have negative gradient, jump upwards.

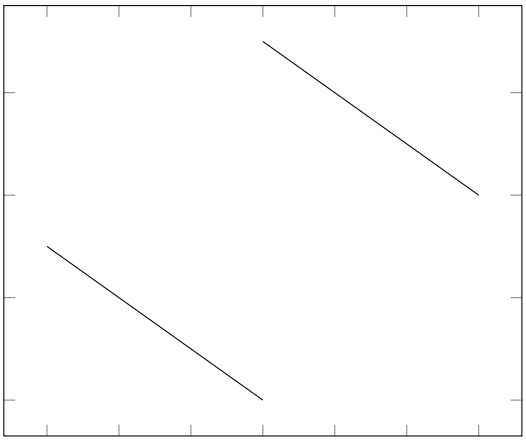

Figure 12: Sample output representing case 3

When entering and leaving the range of interpolation, the gradient is negative.

However, by Assumption 1a, the output when leaving the range of interpolation is higher than the output when entering the range of interpolation.

As the graph is connected (Assertion 2), there must be a point in the range of interpolation where the gradient is positive.

As there is a point in the range of interpolation with positive gradient and the gradient is negative at the start of the range of interpolation, there must be a local minimum in the range of interpolation.

Case 4 Both branches have positive gradient, jump downwards.

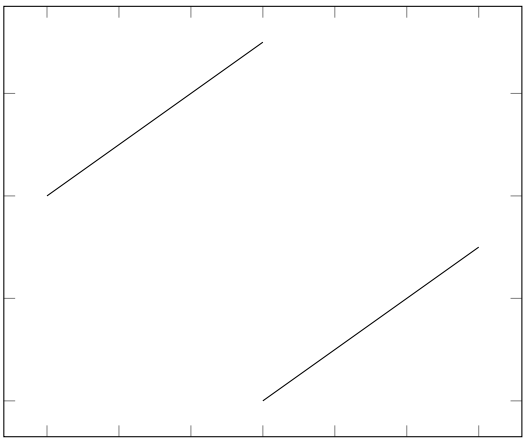

Figure 13: Sample output representing case 4

When entering and leaving the range of interpolation, the gradient is positive.

By Assumption 1a, the output when leaving the range of interpolation is less than the output when entering the range of interpolation.

As the graph is connected (Assertion 2), there must be a point in the range of interpolation where the gradient is negative.

As there is a point in the range of interpolation with negative gradient and the gradient is positive at the end of the range of interpolation, there must be a local minimum in the range of interpolation.

Additional Cases of Statement 2

Case 7 Both branches have negative gradient, jumps downwards.

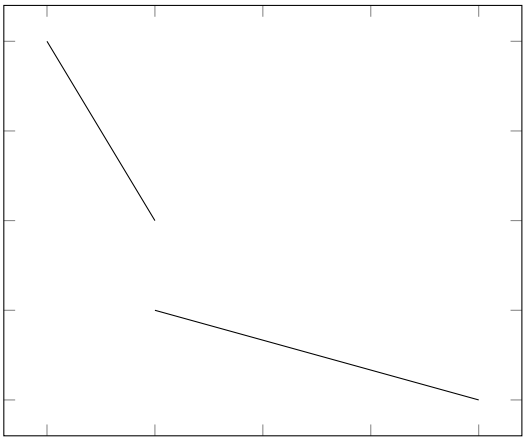

Figure 14: Sample output representing case 7

Similarly to case 5, show that within the range of interpolation, the gradient is always negative, so no false minima will be introduced.

The gradient when taking the left branch is:

$$g'(x) + (f'(x) - Af(x) - g'(x) + Ag(x))e^{-Ad}$$

By Assertion 1, e^{-Ad} is positive and less than 1, so $g'(x)(1 - e^{-Ad}) < 0$
$f'(x) < 0$ by case and Assumption 1b.
$A(g(x) - f(x)) < 0$ as $g(x) < f(x)$ by case and Assumption 1a.
Therefore the interpolated function has a negative gradient when taking the left branch.

The gradient when taking the right branch is:

$$f'(x) + (g'(x) - Af(x) - f'(x) + Ag(x))e^{A(d-T)}$$

By Assertion 1, $e^{A(d-T)}$ is positive and less than 1, so $f'(x)(1-e^{A(d-T)}) < 0$
$g'(x) < 0$ by case and Assumption 1b.
$A(g(x) - f(x)) < 0$ as $g(x) < f(x)$ by case and Assumption 1a.
Therefore the interpolated function has a negative gradient when taking the right branch.
Therefore, within the range of interpolation, the gradient will always be negative so no minima are introduced.

Case 8 Left branch has positive gradient, right branch has negative gradient, jump downwards.

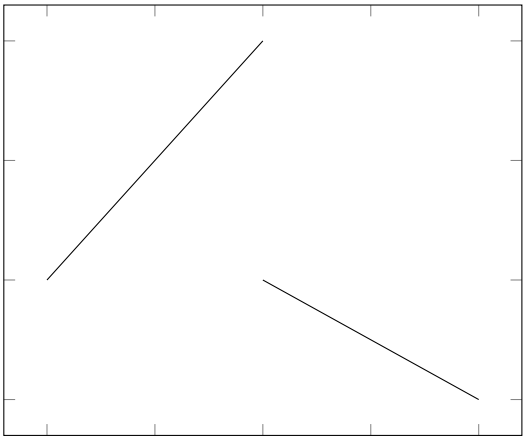

Figure 15: Sample output representing case 8

This is very similar to case 6: show that there is a single maximum, to the right of which the gradient is exclusively negative and to the left of which the

gradient is exclusively positive, by looking at the first point with a negative gradient.

If the first point with a negative gradient appears in the left branch then it is the first point that fulfils:

$$(Af(x) + g'(x))e^{-Ad} > g'(x) + (f'(x) + Ag(x))e^{-Ad}$$

Show that this continues to be the case for the rest of the left branch.

As e^{-Ad} decreases, $g'(x)$ becomes more dominant. As $g'(x) < 0$, this does not invalidate the condition.

$g'(x)$ and $f'(x)$ assumed constant by Assumption 1b.

As $f(x)$ is increasing and $g(x)$ is decreasing (by case), the condition will continue to be fulfilled.

Now show that this continues into the right branch.

Gradient of the right branch is negative if:

$$(Af(x) + f'(x))e^{A(d-t)} > f'(x) + (g'(x) + Ag(x))e^{A(d-t)}$$

Choose the worst case of the condition for the left branch (to imply the highest possible $f'(x)$) by setting e^{-Ad} to $\frac{1}{2}$. Then:

$$\frac{1}{2}Af(x) > \frac{1}{2}(g'(x) + f'(x) + Ag(x))$$

Now look at worst case for the second condition. Here we maximise the influence of $f'(x)$ by setting $e^{A(d-T)}$ to $\frac{1}{2}$.

Therefore to show that the gradient remains negative, we need to show that:

$$\frac{1}{2}(f'(x) + Af(x)) > f'(x) + \frac{1}{2}(g'(x) + Ag(x))$$

$$\frac{1}{2}Af(x) > \frac{1}{2}(g'(x) + f'(x) + Ag(x))$$

This was our worst case for condition 1.

Therefore there is a single local maximum and no local minimum in the range of interpolation if the first point with a negative gradient is in the left branch of the interpolated function.

If the first point with a negative gradient appears in the right branch, then it is the first point to fulfil:

$$(f'(x) + Af(x))e^{A(d-t)} > f'(x) + (g'(x) + Ag(x))e^{A(d-t)}$$

Show this continues to be the case.

$e^{A(d-t)}$ is increasing, so influence of $f'(x)$ is decreasing.

Assuming $f'(x)$, $g'(x)$ constant by Assumption 1b.

$f(x)$ is increasing and $g(x)$ is decreasing, so $f(x)$ continues to be dominant.

Therefore there is a single local maximum and no local minimum in the range of interpolation if the first point with a negative gradient is in the right branch of the interpolated function.

C Sample Code

This appendix gives 3 examples of how the programs used for testing are expressed in an object-oriented fashion as abstract trees.

The program `f(x) = 3 - (x-3)(x-3)`:

```
new AssignmentStatement(0, new SubtractionExpression(new
    GetValueExpression(0), new ConstantExpression(new DualNumber(3.0,
    0.0))),
new AssignmentStatement(1, new MultiplicationExpression(new
    GetValueExpression(0), new GetValueExpression(0)),
new AssignmentStatement(2, new SubtractionExpression(new
    ConstantExpression(new DualNumber(3.0, 0.0)), new
    GetValueExpression(1)),
new ReturnStatement(new GetValueExpression(2)))))
```

The program `f(x,y) = x*y`:

```
new AssignmentStatement(0,new MultiplicationExpression(new
    GetValueExpression(0),new GetValueExpression(1)),
new ReturnStatement(new GetValueExpression(0))
)
```

A program with a disconnected maximum induced by negative gradient with a jump upwards in the middle:

```
new IfStatement(
ConditionWithDistance.lessThan(new GetValueExpression(0), new
    ConstantExpression(new DualNumber(2.0, 0.0))),
new ReturnStatement(new SubtractionExpression(new
    ConstantExpression(new DualNumber(0.0, 0.0)), new
    GetValueExpression(0))),
new ReturnStatement(new SubtractionExpression(new
    ConstantExpression(new DualNumber(5.0, 0.0)), new
    GetValueExpression(0))),
new EndStatement()
)
```

D Additional Tests

The most important tests of the design decisions of the algorithm have already been covered, but it is important to analyse other design decisions, including the values chosen for constants.

Test 4: How fast should the cooling speed be?

"Cooling speed" refers to how much the range of interpolation is reduced on each iteration of the algorithm. In Table 4, the optimal cooling speed at each precision is highlighted using boldface, to point out patterns in the results.

	Precision							
Cooling	$\mathbf{10^{-1}}$	$\mathbf{10^{-2}}$	$\mathbf{10^{-3}}$	$\mathbf{10^{-4}}$	$\mathbf{10^{-5}}$	$\mathbf{10^{-6}}$	$\mathbf{10^{-7}}$	$\mathbf{10^{-8}}$
0.001	1005	862	878	1684	1729	1776	2624	2671
0.005	267	264	479	520	565	837	892	1168
0.01	174	190	333	373	555	**616**	833	**891**
0.05	100	170	**216**	**322**	**459**	626	**707**	892
0.1	91	155	244	355	496	667	859	1056
0.2	**86**	**147**	277	397	617	807	1117	47691
0.5	98	222	429	693	1112	1560	2072	2734

Table 4: How cooling speed of range of interpolation affects time taken for a number of precisions (best results in bold)

The trend in the results is clear; low precision is fastest with slow cooling. Faster cooling may be useful for higher precisions as it takes fewer iterations to reach the necessary precision, while the benefits of faster movement makes sense for lower precisions.

In the rest of the tests a cooling speed of 0.01 is used, as performance at higher precisions is the main concern. The findings of this test suggest, however, that some sort of parametrisation might work best, with a cooling speed based on precision. Finding a formula for the most efficient cooling speed for any precision that still gives a correct result would be an interesting extension to this algorithm.

Test 5: How fast should the slowdown in the inner gradient descent be?

Slowdown is necessary in the inner gradient descent as functions such as $f(x) = |x|$ have a "sharp corner" as a minimum. The gradient has the same magnitude directly either side of the extremum, so the size of moves made will be constant. This means that gradient descent could infinitely loop, alternating between a point on each side of the local extremum. Therefore it makes the most sense to slow on change in direction, but it is important to test the appropriate speed of this slowdown.

This test was also run for step sizes of 0.1, 0.2 and 0.4, but these all gave incorrect results. The tests that succeeded are shown in Table 5.

	Precision							
Cooldown	$\mathbf{10^{-1}}$	$\mathbf{10^{-2}}$	$\mathbf{10^{-3}}$	$\mathbf{10^{-4}}$	$\mathbf{10^{-5}}$	$\mathbf{10^{-6}}$	$\mathbf{10^{-7}}$	$\mathbf{10^{-8}}$
0.5	202	209	355	405	598	649	880	950
0.6	185	195	334	382	559	620	**828**	896
0.65	174	190	333	**373**	**555**	**616**	833	**891**
0.7	169	**186**	**332**	377	568	626	856	917
0.75	**166**	191	342	393	591	658	896	968
0.9	190	252	487	590	924	1063	1487	1632
0.95	251	375	762	957	1533	1791	2544	2817

Table 5: How step size cooldown of gradient descent affects number of runs taken for a range of precisions (best results in bold)

The results suggest a slowdown of 0.65, which is used throughout the rest of the tests. Tests with faster cooldown are probably failing due to the reduction in step size falsely indicating a gradient close to 0.

Test 6: Why is the step size for inner gradient descent calculated in the way it is?

There are two subtle variants on our method for calculating the step size for inner gradient descent, and all of the tests in this section were performed for both. Table 6 gives a quick summary aggregating over basic tests.

	Precision							
Method	$\mathbf{10^{-1}}$	$\mathbf{10^{-2}}$	$\mathbf{10^{-3}}$	$\mathbf{10^{-4}}$	$\mathbf{10^{-5}}$	$\mathbf{10^{-6}}$	$\mathbf{10^{-7}}$	$\mathbf{10^{-8}}$
effectiveGamma = gamma/expAdjust	174	190	333	373	555	616	833	891
effectiveGamma = gamma*T	232	251	431	473	691	755	1004	1066

Table 6: How two methods of calculating the step size for inner gradient descent affect time taken for a number of precisions

Clearly, dividing by the adjustment value we generated for interpolation is faster than multiplying by the temperature. This makes sense considering the way the adjustment is calculated. More importantly, both methods give correct answers, as expected considering our analysis of the design of the algorithm.

Test 7: Why limit step size in inner gradient descent?

It seems to go against the design of gradient descent to limit movement in areas of high gradient where gradient descent expects no nearby extrema, but disconnected extrema are, by definition, sudden, and we cannot risk gradient descent jumping over any areas of interpolation. Indeed, though it is important

to have a large enough maximum step size to make use of the advantages of gradient descent, the programs we test tend not to have large gradients and perform significantly better with lower maximum step sizes.

The "limit" in Table 7 refers to the maximum gradient used. That is, the limit of what the step size can be multiplied by when calculating displacement during the actual gradient descent.

	Precision							
Limit	$\mathbf{10^{-1}}$	$\mathbf{10^{-2}}$	$\mathbf{10^{-3}}$	$\mathbf{10^{-4}}$	$\mathbf{10^{-5}}$	$\mathbf{10^{-6}}$	$\mathbf{10^{-7}}$	$\mathbf{10^{-8}}$
2	150	166	273	296	419	450	586	614
5	174	190	333	373	555	616	833	891
10	207	227	426	497	775	885	1233	1337
20	268	298	613	740	1210	1417	2027	2225
25	297	333	705	860	1426	1681	2420	2662
100	543	745	1924	2456	4403	5314	7956	8827
1000	2029	4181	14316	18711	36511	44522	69300	77049
10000	5088	11948	95537	125100	285849	349175	579928	642610
No Limit	6529	16572	2781648	3567935	DNF	DNF	DNF	DNF

Table 7: How limiting maximum move size affects number of runs required at different levels of precision (DNF means test took too long and was cancelled)

The fact that limiting displacement has such a profound effect seems counter-intuitive, but the reason was discovered on further investigation. On later iterations, the range of interpolation is very small. The gradients this induces will therefore be significant. When drilling down on an extremum, what matters is the direction moved, not the amount, and the very large jumps caused by these steep, interpolated gradients led to most of the work being conducted on returning to the area in which the extremum occurs, not on drilling down in more detail.