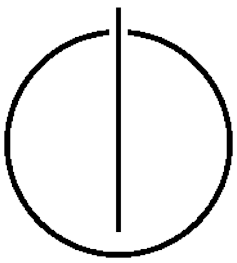
TECHNISCHE UNIVERSITÄT MÜNCHEN

DEPARTMENT OF INFORMATICS

Bachelor's Thesis in Informatics

Formalisation of Interval Methods for Nonlinear Root-Finding

Daniel Seidl



TECHNISCHE UNIVERSITÄT MÜNCHEN

DEPARTMENT OF INFORMATICS

Bachelor's Thesis in Informatics

Formalisation of Interval Methods for Nonlinear Root-Finding

Formalisierung nichtlinearer Intervall-Verfahren zur Nullstellensuche

Author:	Daniel Seidl
Supervisor:	Prof. Tobias Nipkow
Advisor:	Dr. Manuel Eberl
Submission Date:	15.05.2021

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.05.2021 Daniel Seidl

Acknowledgements

First of all, I would like to thank my advisor Dr. Manuel Eberl for his fast responses to all my questions. Especially his help, when I got stuck in a proof, saved many hairs from being pulled out of my head – I honestly could not think of better support!

Next, I want to thank Andreas Kruck, who submitted his bachelor's thesis shortly before me and supplied me with many pieces of information regarding formality!

Also, a big shout-out to Philipp Schwarz for proof-reading this thesis!

Lastly, I would like to thank my family for always giving me the emotional support I needed – especially in this weird time right now!

Abstract

English

The interval Newton method is an extension of the well-known Newton's method for finding roots of differentiable functions. While the latter approximates at most one root of a function, the interval Newton method finds all roots in a given interval. In this thesis, the method is formalised for arbitrary-precision floating point numbers analogous to Moore [2] using the proof assistant Isabelle/HOL. With different examples the results of the implementation are analysed and evaluated. Using the `Approximation` package, the implementation is easily usable to approximate the roots of a large class of univariate real-valued functions.

Deutsch

Das Newton-Intervallverfahren ist eine Erweiterung des bekannten Newtonschen Näherungsverfahren, um Nullstellen ableitbarer Funktionen zu bestimmen. Während letzteres maximal eine Wurzel approximiert, bestimmt das Newton-Intervallverfahren alle Nullstellen in einem angegebenen Intervall. In dieser Arbeit wird die Methode analog zu Moore [2] mit dem interaktiven Theorembeweiser Isabelle/HOL für Gleitkommazahlen formalisiert. Mit verschiedenen Beispielen werden die Ergebnisse der Implementierung analysiert und bewertet. Durch die Einbindung des „`Approximation` package“ ist die Implementierung einfach nutzbar, um Nullstellen von vielen verschiedenen reellen univariaten Funktionen zu approximieren.

Contents

1 Introduction

When analysing and talking about functions, it is almost impossible to skip the calculation of roots. Because this is such a central problem, there are many different methods for calculating numbers x_0 meeting this condition for a function f:

$$f(x_0) = 0$$

During my time in school, we talked about two main methods for finding the roots of a function. The first was the quadratic formula, which we called "Mitternachtsformel" (midnight formula), the second and more general one was Newton's method.

This method had quite some flaws, as we get at most one root of the function, and even this is not guaranteed and we do not know before applying the method. Both these issues are solved by the interval Newton method, an extension of the original Newton's method: If the method does not find any root in a given interval, the function has no root in this interval. However, if the function has roots, the method results in a list of intervals, the union of which contains all roots.

In this thesis, the interval Newton method is implemented for arbitrary-precision floating point numbers and later proven with the proof assistant Isabelle/HOL.

Outline

In Chapter 2 we show the mathematical background of the method.

Following this, Chapter 3 contains the main part of this thesis, the formalisation of the method: To begin with, we look at the different parts of the implementation and then go through the formal correctness proof the same way.

After that, in Chapter 4, we analyse the results of the implementation using different examples.

2 Mathematical Background

Before going into details of the implementation, it is important to clarify the mathematics behind it. This chapter is based on chapter 8.1 of "Introduction to Interval Analysis" by Moore et al. [2].

Let f be a function $f : \mathbb{X} \to \mathbb{R}$ which is differentiable on $\mathbb{X} = \left[\underline{X}, \overline{X}\right]$ with the derivative f'. A variation of the mean value theorem shows that if f' is the derivative of f on an interval $\mathbb{X}$, there exists an $s \in \mathbb{X}$ so that:

$$f(\overline{X}) - f(\underline{X}) = f'(s) \cdot \left(\overline{X} - \underline{X}\right) \tag{2.1}$$

Of course the theorem also shows for $x, y \in \mathbb{X}$, that:

$$\exists s \in [x, y] : f(y) - f(x) = f'(s) \cdot (y - x) \tag{2.2}$$

Now let x be a root of f, so that $f(x) = 0$. Following (2.2) there is an s so that:

$$f(y) - f(x) = f(y) = f'(s) \cdot (y - x) \quad \Longrightarrow \quad f(y) + f'(s) \cdot (x - y) = 0 \tag{2.3}$$

In order to isolate x we assume $\exists y \in \mathbb{X} : f(y) \neq 0$. If a y like this does not exist this means f is constant on $\mathbb{X}$ and $\forall s \in \mathbb{X} : f'(s) = 0$. That constant is either 0 or not; in both cases, the resulting interval is known: In the first case there is an infinite number of roots in $\mathbb{X}$ and therefore the result is the same as the starting interval, as we cannot narrow it down any more. And if f is a constant function $\neq 0$ there are no roots.

Therefore we can safely assume that the image of f is not $\{0\}$. Following (2.3) we get for all $y \in \mathbb{X}$ with $f(y) \neq f(x) = 0$:

$$\frac{f(y)}{f'(s)} + (x - y) = 0 \Longrightarrow x = y - \frac{f(y)}{f'(s)} \tag{2.4}$$

This is similar to the well-known formula of Newton's method:

$$x_{n+1} \coloneqq x_n - \frac{f(x_n)}{f'(x_n)}$$

This can be interpreted in a graphical way: The mean value theorem states, that there is a point s, so that the tangent at s has the same slope as the secant connecting the function values of the interval bounds: $f'(s)$. In our case, one of these bounds is a root and its function value is 0. Now we can recreate the secant by transforming the tangent and therefore find the root.

However, we do not know the exact value of s and more importantly $f'(s)$. But we know, that $s \in \mathbb{X}$, therefore we replace $f'(s)$ with an enclosure F' of f' on $\mathbb{X}$ where $F' \supseteq [\min\{f'(\chi) : \chi \in \mathbb{X}\}, \max\{f'(\chi) : \chi \in \mathbb{X}\}]$. Now the method takes all possible slopes into consideration for further calculation. As $f'(s) \in F'$ it follows that:

$$x \in y - \frac{f(y)}{F'} \tag{2.5}$$

The problem in (2.5) is the division with an interval, which could contain 0. Still assuming $F' \neq [0,0]$ we have to distinguish between four cases:

$$\frac{1}{F'} = F'^{-1} = \begin{cases} \left[\overline{F'}^{-1}, \underline{F'}^{-1}\right] & \text{if } 0 \notin F' \\ \left(-\infty, \underline{F'}^{-1}\right] & \text{if } \underline{F'} < 0 = \overline{F'} \\ \left[\overline{F'}^{-1}, \infty\right) & \text{if } \underline{F'} = 0 < \overline{F'} \\ \left(-\infty, \underline{F'}^{-1}\right] \cup \left[\overline{F'}^{-1}, \infty\right) & \text{if } \underline{F'} < 0 < \overline{F'} \end{cases} \tag{2.6}$$

The second, third and forth case follow extended interval arithmetic. As already shown it is not an issue to exclude the case $F' = [0,0]$, as for every root x there is always a $y \in \mathbb{X}$ where $f(y) \neq 0$ if $F' \neq [0,0]$ and therefore $f'(s) \neq 0$.

Since, in general, we will not be able to perform all computations exactly, the next step is to replace $f(y)$ by a bounding enclosure $F(y)$. As $f(y) \in F(y)$ per definition, the equation (2.5) can easily be rewritten:

$$x \in y - \frac{F(y)}{F'} \tag{2.7}$$

In order to transform this into an algorithm, the selection of y has to follow a rule. Suitable for this is amongst others the midpoint of $\mathbb{X}$:

$$\mathrm{m}(\mathbb{X}) = \frac{\underline{X} + \overline{X}}{2} =: m \tag{2.8}$$

However, this can lead to a possible problem: What if $F(m) =: F$ is zero, so $F = [0,0]$? This was not allowed before and it still would produce the following formula:

$$x \in m - \frac{F}{F'} = m - \frac{[0,0]}{F'} = [m,m]$$

This is wrong for functions with more than one root. If $0 \notin F'$ there is no issue, as f has at most one root, which would be found in one step. However, if the function has more roots, all except one are lost.

An example would be the function $g : \mathbb{R} \to \mathbb{R}, x \mapsto (x-1)(x-2) = x^2 - 3x + 2$ with $g'(x) = 2x - 3$, $\mathbb{Y} = [0,4]$ and the roots $x_1 = 1$ and $x_2 = 2$. The result after one step is $[2,2]$, which obviously does not contain 1.

Therefore allowing the calculation with $F = [0,0]$ can lead to a wrong result. But what if $0 \in F$, if $F \neq [0,0]$? There are, similar to (2.6), three cases: Using g and $\mathbb{Y}$ from above all calculations with $G \doteq [-1,0]$, $G \doteq [0,1]$ and $G \doteq [-1,1]$ are resulting in $\mathbb{Y} \cap (-\infty, \infty) = \mathbb{Y}$.

In theory, all those results are correct, as they contain every root. But it is neither helpful nor is an algorithm needed to know that all roots of a function are in $(-\infty, \infty)$. Therefore in the implementation, this case has to be treated differently and we do so by bisecting the interval. Speaking of the implementation, it is time to complete the method.

We have shown, that every root of f in the starting interval $\mathbb{X}$ is also in $m - F/F'$. To narrow down the interval, it is intersected with the starting interval. This is particularly helpful if the enclosure F' contains 0 and $m - F/F'$ is an unbounded interval.

Using the previous steps we can sum our current state as follows:

$$x \in \mathbb{X} \wedge f(x) = 0 \wedge F' \neq [0,0] \wedge (F \neq [0,0] \vee 0 \notin F') \Longrightarrow x \in \mathbb{X} \cap \left(m - \frac{F}{F'} \right) \tag{2.9}$$

The next and almost final step is the definition of one Newton step, the recursive formula of the interval Newton method with starting interval X_0. First we look at the cases, if 0 is not in F' or either the lower or upper bound of F', so we do not have to split the interval:

$$X_{k+1} = X_k \cap \left(\mathrm{m}(X_k) - \frac{F(\mathrm{m}(X_k))}{F'(X_k)} \right) \tag{2.10}$$

However, if we have to split the interval, we get two results:

$$X_{\underline{k+1}} = X_k \cap \left(\mathrm{m}(X_k) - \frac{F(\mathrm{m}(X_k))}{\left[\underline{F'(X_k)}, 0\right]} \right) \text{ and } X_{\overline{k+1}} = X_k \cap \left(\mathrm{m}(X_k) - \frac{F(\mathrm{m}(X_k))}{\left[0, \overline{F'(X_k)}\right]} \right)$$

The method will be applied to both these intervals, therefore creating a recursion tree. The final step is to show the convergence of the method because implementing a non-converging algorithm is not the goal of this thesis.

One trivial case is as previously mentioned with $F(\mathrm{m}(X_i)) = [0,0]$ and $0 \notin F'(X_i)$, as the algorithm terminates after one step:

$$X_{k+1} = X_k \cap \left\{ \mathrm{m}(X_k) + \frac{F(\mathrm{m}(X_k))}{F'(X_k)} \right\} = X_k \cap [\mathrm{m}(X_k), \mathrm{m}(X_k)] = [\mathrm{m}(X_k), \mathrm{m}(X_k)]$$

Looking at the non-trivial case, we assume that $0 \notin F(\mathrm{m}(X_k))$ and $F'(\mathrm{m}(X_k)) \neq [0,0]$. Also, we assume, that we do not split the interval and therefore have one result. Then X_{k+1} does not contain $\mathrm{m}(X_k)$. It follows, that $\mathrm{w}(X_{k+1}) < 2 \cdot \mathrm{w}(X_k)$ with $\mathrm{w}(X) = \overline{X} - \underline{X}$ being the width of an interval. This shows, that in each step the width of the interval is more than halved and recursive Newton steps result in a nested sequence of intervals.

If $\underline{F'} < 0 < \overline{F'}$ we have to split the interval following (2). Analogous to the proof without splitting we know, that $X_{\underline{k+1}} < 2 \cdot \mathrm{w}(X_k)$ and $X_{\overline{k+1}} < 2 \cdot \mathrm{w}(X_k)$. It follows, that $\mathrm{w}(X_{\underline{k+1}}) + \mathrm{w}(X_{\overline{k+1}}) < \mathrm{w}(X_k)$. Therefore we have shown that even in the split case we have a converging method.

3 Formalisation of the Interval Newton Method

3.1 Dependencies

Isabelle/HOL

Isabelle/HOL is a logic in the generic proof assistant Isabelle, where HOL stands for "higher-order logic". There are other built-in logics, however, in this thesis, we only use HOL, which is also the most used one.

Proof assistants are primarily used to help verify implemented code. To make life easier for the user, there are already many proven theorems, which serve as a basis for upcoming proofs. These are organised in different theories, together with helpful lemmas, definitions and functions.

In this thesis, we verify that our implementation of the interval Newton method does not oversee any root. To achieve this, we use different already available theories, the most important ones I will briefly introduce.

Interval Arithmetic

As its name already suggests, the interval Newton method heavily relies on interval arithmetic and intervals in general. One option is the `Interval` theory, which supports many operations on intervals, despite some flaws. The biggest one is the missing support for empty, open, and unbounded intervals. As we can emulate both empty and open intervals we use this datatype.

Floating-Point Numbers

Floating-point numbers (in the future referred to as floats) in Isabelle/HOL are numbers in the infinite set $\{m \cdot 2^e : m, e \in \mathbb{Z}\}$. This also mirrors the constructor, which takes two natural numbers as mantissa and exponent. In contrary to standards like "IEEE 754", neither the mantissa nor the exponent is bounded by a maximum number of bits.

However, in practice, one usually limits the number of bits used in a computation to a given precision for performance reasons.

Interval Float and Approximation Bounds

Both the `Interval_Float` theory and the `Approximation_Bounds` theory based on it extend the `Interval` theory with many helpful operations and lemmas on float intervals. Important for this thesis are primarily the operations with an additional precision argument which guarantee that both endpoints of the resulting interval have at most the desired precision.

Approximation

The `Approximation` package is probably the single most important dependency of the implementation, as it makes the implementation way easier to use. It provides not only the "floatarith" datatype but many helpful functions supporting it.

This type is used to model functions recursively and supports many different operations:

```
datatype floatarith
  = Add floatarith floatarith | Minus floatarith [...]
  | Var nat | Num float
```

The constructor "Var" uses the index of a list to represent a variable. In our case, we only use univariate functions and therefore the index of the variable is always 0.

The result of a function is calculated using `interpret_floatarith`. This function takes the floatarith and a list containing the values of the variable:

```
fun interpret_floatarith :: "floatarith ⇒ real list ⇒ real" where
"interpret_floatarith (Add a b) vs = (interpret_floatarith a vs)
  + (interpret_floatarith b vs)" |
"interpret_floatarith (Minus a) vs = - (interpret_floatarith a vs)" |
[...]
"interpret_floatarith (Num f) vs = f"
"interpret_floatarith (Var n) vs = vs ! n"
```

However, we do not want to evaluate the result of a function for a given input but instead for all values of an interval. This can be achieved with the `approx` function, which approximates the enclosure of a function for a given interval. With this, we do not have to manually calculate bounding intervals for the method, which makes it more usable, as hinted before.

3.2 Implementation

One small note before we begin: The complete code needed for the implementation of the interval Newton method is available via zenodo [3].

Before going into details it is important to clarify the goals of the implementation. The idea is to find and approximate all roots of a function in a given float interval. In addition, two natural numbers, an epsilon, and a precision argument are needed. The epsilon works as a termination condition regarding the width of the intervals while the precision is necessary for all operations on floats. The function is given as a floatarith based on the `Approximation` package.

This implementation of the interval Newton method consists of three main parts:

`newton_step_float` calculates the result of a Newton step using a precision argument, the current interval, its midpoint and the given enclosures F and F'.

`newton_step_floatarith` is based on `newton_step_float` and calculates the enclosures using the function in floatarith form, a precision argument, and the interval. If the floatarith is not supported, the function aborts.

`newton_floatarith` is the recursive function based on `newton_step_floatarith`. In addition to floatarith, precision and interval also an epsilon value is needed as condition for termination.

Design Choices

The main design choice behind the method was selecting the most suitable type to return the resulting intervals in the different parts. After each Newton step, there are three different scenarios: The result is either an empty interval, one interval, or two intervals. As the interval type doesn't support empty intervals, there has to be a different solution. This can either be achieved with a "sum type" wrapped in an "option" or with a list of float intervals.

In my opinion, a list is in this case more suitable, especially, because it avoids unwieldy case distinctions. In addition, the three cases can be represented conveniently by an empty list or a list containing one or two intervals.

Because several of the operations can fail, the two parts using floatariths are written in the option monad and the final result is a "float interval list option" to reflect this

possibility of failure. Also, it is possible that the recursive method does not terminate. In this case the result would also be "None".

3.2.1 Newton Step – Mathematics

In theory, one Newton step could be handled in a single function definition, but it is easier to split the mathematical part working with enclosures from the part, which uses floatariths to calculate both F and F'.

The Newton step on floats is the backbone of the implementation and as previously mentioned is responsible for (almost) all calculations according to (2.10). It takes a natural number as precision argument `prec` for all operations on floats, the three float intervals `X`, `F` and `F'` and a point `m` in `X`. At the moment the exact definition of `m` is irrelevant, as long as it lies in `X`. This is guaranteed from the calculation in `newton_step_floatarith` and therefore not checked here, as `newton_step_float` is not meant to be used on its own. Now let us look at the Newton step on floats in a mathematical form:

$$
\mathrm{N}(X,F,F',m) = \begin{cases}
F' = [0,0] & \to \begin{cases} 0 \in F & \to X \\ \text{else} & \to \emptyset \end{cases} \\
0 \notin F' & \to X \cap \left(m - \dfrac{F}{F'} \right) \\
\overline{F'} = 0 & \to \begin{cases} \overline{F} < 0 & \to X \cap \left(\infty, m - \overline{F} / \underline{F'}\right] \\ 0 < \underline{F} & \to X \cap \left[m - \underline{F} / \underline{F'}, \infty\right) \\ \underline{F} < 0 < \overline{F} & \to [\underline{X}, m] \,;\, [m, \overline{X}] \end{cases} \\
\underline{F'} = 0 & \to \begin{cases} \overline{F} < 0 & \to X \cap \left[m - \overline{F} / \overline{F'}, \infty\right) \\ 0 < \underline{F} & \to X \cap \left(\infty, m - \underline{F} / \overline{F'}\right] \\ \underline{F} < 0 < \overline{F} & \to [\underline{X}, m] \,;\, [m, \overline{X}] \end{cases} \\
\underline{F'} < 0 < \overline{F'} & \to \begin{cases} \overline{F} < 0 & \to X \cap \left(\infty, m - \overline{F} / \underline{F'}\right] \,;\, X \cap \left[m - \overline{F} / \overline{F'}, \infty\right) \\ 0 < \underline{F} & \to X \cap \left(\infty, m - \underline{F} / \overline{F'}\right] \,;\, X \cap \left[m - \underline{F} / \underline{F'}, \infty\right) \\ \underline{F} < 0 < \overline{F} & \to [\underline{X}, m] \,;\, [m, \overline{X}] \end{cases}
\end{cases}
$$

After seeing the mathematical idea, hopefully, the full code becomes more readable:

```
definition newton_step_float:: "nat ⇒ float interval ⇒ float interval ⇒
    float interval ⇒ float ⇒ float interval list" where
"newton_step_float prec X F F' m = (
  if lower F' = 0 ∧ upper F' = 0 then (
    if 0 ∈ set_of F then [X] else []
  ) else if 0 < lower F' ∧ upper F' < 0 then (
    float_interval_intersection X (
      plus_float_interval prec (interval_of m) (
        - mult_float_interval prec F (inverse_float_interval' prec F')
      )
    )
  ) else if upper F' = 0 then (
    if upper F < 0 then (
      float_interval_intersection_unbounded_lower X (
        float_plus_up prec m (- float_divl prec (upper F) (lower F'))
      )
    ) else if 0 < lower F then (
      float_interval_intersection_unbounded_upper X (
        float_plus_down prec m (- float_divr prec (lower F) (lower F'))
      )
    ) else split_float_interval X m
  ) else if lower F' = 0 then (
    if upper F < 0 then (
      float_interval_intersection_unbounded_upper X (
        float_plus_down prec m (- float_divr prec (upper F) (upper F'))
      )
    ) else if 0 < lower F then (
      float_interval_intersection_unbounded_lower X (
        float_plus_up prec m (- float_divl prec (lower F) (upper F'))
      )
    ) else split_float_interval X m
  ) else (
    if upper F < 0 then (
      float_interval_intersection_unbounded_lower X (
        float_plus_up prec m (- float_divl prec (upper F) (lower F'))
      ) @ float_interval_intersection_unbounded_upper X (
        float_plus_down prec m (- float_divr prec (upper F) (upper F'))
```

```
      )
    ) else if 0 < lower F then (
      float_interval_intersection_unbounded_lower X (
        float_plus_up prec m (- float_divl prec (lower F) (upper F'))
      ) @ float_interval_intersection_unbounded_upper X (
        float_plus_down prec m (- float_divr prec (lower F) (lower F'))
      )
    ) else split_interval X m
  )
)"
```

In the code there occur some new functions, mostly for interval intersections with a list as result:

`float_interval_intersection ivl ivl'`	Returns a list containing the intersection of two intervals; if the intervals are disjoint, an empty list is returned.
`float_interval_intersection _unbounded_lower ivl upper`	Like above, but this takes an interval and the upper bound of a half-open interval.
`float_interval_intersection _unbounded_upper ivl lower`	Like above, but this takes an interval and the lower bound of a half-open interval.
`inverse_float_interval' prec ivl`	This calculates the inverse of an interval, assuming $0 \notin$`ivl`.
`split_float_interval ivl m`	This splits an interval at a point and returns a list with the two intervals (or one if the point is not in the interval)

Table 3.1: Definitions on Intervals

One more word regarding `inverse_float_interval'`: This function is almost the same as `inverse_float_interval` of the `Interval_Float` theory, but has one big difference. If we use this function inside of the Newton step, we know, that the interval `F'` in question does not contain 0. As this is not guaranteed in the original function the result is wrapped in an "option", in order to return "None" if the interval contains zero. To avoid this unnecessary case distinction I defined `inverse_float_interval'`, which is of course not intended to be used on it is own.

3.2.2 Newton Step – Processing

The second part of the Newton step mainly calculates `F` and `F'` using the function `f`. But let us have a look at the code first:

```
definition newton_step_floatarith:: "floatarith ⇒ nat ⇒ float interval
    ⇒ float interval list option" where
"newton_step_floatarith f prec ivl = do {
  let m = float_down prec (mid ivl);
  let f' = DERIV_floatarith 0 f;
  F ← approx prec f [Some (interval_of m)];
  F' ← approx prec f' [Some ivl];
  if m ≤ lower ivl then Some [round_interval prec ivl]
  else if isDERIV_approx prec 0 f [Some ivl] then
    Some (map (round_interval prec) (newton_step_float prec ivl F F' m))
  else None
}"
```

The first step is the calculation of the midpoint `m` of the interval. In the first version of this function, the corresponding line was just "**let** `m = mid ivl`". The issue here is the definition of the midpoint: `mid i = (lower i + upper i) * Float 1 (-1)"`. With this definition, it is possible that the precision of `m` is different from `prec`, which led to unwanted results. Therefore the midpoint is rounded down to the desired precision of the calculation. This also could lead to a numerical issue if the rounded down midpoint is less than or equal to the lower bound of the interval, which is checked four lines later.

In the next line, the derivative `f'` of `f` is calculated using the `Approximation` package.

Following this, both the enclosures `F` of `f` over `[m,m]` and `F'` of `f'` over `ivl` are approximated. If either of the returned float interval options by `approx` equals "None", `newton_step_floatarith` also results into "None".

The first if-statement checks, if the midpoint is a valid input. If `m` is less than the lower bound the method is invalid or at least unproven. If the midpoint is the same as the lower bound, this can create serious issues. For example, this occurred for the function x^2, when the midpoint of $\left[0, 1.449 \times 10^{-9}\right]$ was the lower bound and therefore 0. This led to the split $[0,0]\,;\left[0, 1.449 \times 10^{-9}\right]$ and a non-terminating method. In addition to that, if `m` equals the lower bound, the width of the interval is very small. This also means that the approximation is accurate enough to justify terminating.

The else-if-statement checks if `f` is differentiable and thus the validity of `f'`. If it is valid, the preparation is done and the intervals are used by `newton_step_float`. In the last step, the resulting intervals get rounded to ensure that they have exactly the desired precision. And if the derivative is not valid, of course "None" is returned.

Despite it following from the implementation it is important to state that the result "None" does not mean that there is no root in the interval, but rather that there was an issue during runtime, most likely because the function was not supported. If there are no roots, the desired result would be "Some []".

3.2.3 Recursive Newton Method

Although the single Newton step is the most important part of the implementation, it is obviously not enough. In order to have a working algorithm we need a recursive wrapper function:

```
partial_function (option) newton_floatarith:: "nat ⇒ floatarith ⇒ nat ⇒
    float interval ⇒ float interval list option" where
"newton_floatarith eps f prec ivl = (
  if width ivl < Float 1 (-eps) then Some [ivl] else do {
    ivls ← newton_step_floatarith f prec ivl;
    if ivls = [ivl] then Some [ivl]
    else do {
      ivls' ← mapM (newton_floatarith eps f prec) ivls;
      Some (concat ivls')
    }
  }
)"
```

The first thing to notice is the signature of the recursive function: In contrast to both Newton steps, this is a partial function with "option" as mode argument instead of a definition. While we have already shown convergence without rounding in the mathematical background, we cannot prove termination for all inputs. In a partial function this is not needed, as non-termination is modelled with "None". [4, p. 263].

In the first step the epsilon value is used to check if the width of the interval is smaller than $2^{-\epsilon}$. The ϵ is used as the first condition of termination and represents the desired minimum accuracy of the approximation. Of course, it is possible that there are two roots in an interval, if either the epsilon-value is set too low (and therefore the accepted

width is too wide) or the function simply has two (or even more) roots extremely close to each other. If the width is small enough, the interval is accurate enough and returned, otherwise we proceed to the main body of the function.

If the interval is not exact enough, the result of `newton_step_floatarith` is calculated in the first step. As before, if it is "None", this will also be the result of the whole recursive function. But in theory, it is extremely unlikely if not even impossible, that `newton_step_floatarith` returns "None" in a later recursion, if there is no issue in the starting interval. This is the case as the possible result "None" mostly depends on the function `f`, which does not change.

Following this, it is checked whether the Newton step modified the interval at all. If this was not the case, there is no further refinement possible and therefore this is also the final result. Mathematically speaking this is also the only valid termination requirement, however, in practice this can easily result in an infinite loop.

If the output is not the same as the input, the recursive part follows. The new defined function mapM is a modified mapping function:

```
fun mapM:: "('a ⇒ 'b option) ⇒ 'a list ⇒ 'b list option" where
"mapM f [] = Some []" |
"mapM f (x#xs) = do {
  y ← f x;
  ys ← mapM f xs;
  Some (y # ys)
}"
```

With `mapM` `newton_floatarith` is recursively called on the new intervals resulting from the Newton step. If the function returns "None" on either element of the list, `mapM` is also "None". In our case however, as mentioned before, `newton_step_floatarith` returning "None" for only one interval and therefore discarding information from others is negligible.

The final step after the recursive call is the concatenation of all lists of float intervals, thus creating the final output, a list of intervals.

3.3 Proof

Besides the implementation, correctness is also an important part of this thesis. While the proofs of the three functions differ, the corresponding lemmas are all relatively

similar. We assume, that x is a root, so that $f(x) = 0$, and also that x is in the starting interval. With these assumptions, we show that x is in the resulting list. It is important to note that, even though the computation takes place entirely on floats, x is a real number!

3.3.1 Newton Step – Mathematics

Similarly to the implementation section, both parts of the Newton step are treated separately. Also, because this part unsurprisingly relies heavily on the mathematical background of the method, there will be many references to the corresponding chapter. Also, most of the lemmas are shown without proof, as those are not too interesting.

The single most important lemma is the mean value theorem (2.1), as it is the mathematical foundation of the method. This lemma is a modified version of `mvt_very_simple` from the Derivative theory:

```
lemma mvt_very_simple':
  fixes f :: "real ⇒ real"
  assumes "l ≤ u"
    and "⋀x. ⟦l ≤ x; x ≤ u⟧ ⟹
      (f has_field_derivative f' x) (at x within {l..u})"
  shows "∃s ∈ {l..u}. f u - f l = f' s * (u - l)"
```

The following lemmas are all wrapped in a locale and proven with real-number values. "A locale is a functor that maps parameters (including implicit type parameters) and a specification to a list of declarations." [4, S. 101] In our case, this means that the following fixes and assumptions are valid for all of the following lemmas until we explicitly change the locale. For now we are working on real numbers and real intervals, as both floats and float intervals can easily be converted to real numbers and real intervals:

```
locale newton_step_locale =
  fixes f f' :: "real ⇒ real"
  fixes X F' :: "real interval"
  fixes m :: real
  assumes f': "x ∈ set_of X ⟹
    (f has_field_derivative f' x)(at x within set_of X)"
  assumes f'_enclosure: "x ∈ set_of X ⟹ f' x ∈ set_of F'"
  assumes m: "m ∈ set_of X"
```

We have the function f with the derivative f' on the interval $\mathbb{X}$ in which we search for roots. Also we have the enclosure F' of f' on $\mathbb{X}$ and a point m in $\mathbb{X}$. It is important to

note, that right now we do not know that m is the midpoint. Later, to use the following lemmas, we interpret the fixed parameters and prove the assumptions.

The assumptions of the locale along with some not too interesting steps are used for the following lemma, which is a slightly transformed version of (2.2):

```
lemma f_transformed:
  assumes "x ∈ set_of X" "y ∈ set_of X" "x < y"
  shows "∃s ∈ {x..y}. f x = f y + f' s * (x - y)"
```

The next step is both setting x to a root of f and the division with the derivative, where the latter is only possible, if $f(x) \neq f(y)$ (2.4):

```
lemma f_transformed_with_unequal_values:
  assumes "x ∈ set_of X" "f x = 0" "y ∈ set_of X" "x < y" "f x ≠ f y"
  shows "∃s ∈ {x..y}. x = y - f y / f' s"
```

There is also an analogous lemma `f_transformed_with_unequal_values'`, where the fourth assumption is flipped to `"y < x"` and therefore the set is `{y..x}`, everything else (including the proof) is the same. With this transformation the next step is replacing `f' s` with the enclosure of the derivative and extending the calculation onto intervals:

```
lemma enclosure_not_zero:
  assumes "x ∈ set_of X" "f x = 0" "0 ∉ set_of F'"
  shows "x ∈ set_of (interval_of m - interval_of (f m)
    * inverse_interval' F')"
```

The important part here is the third assumption, `"0 ∉ set_of F'"`, as this calculation would not be valid otherwise. The reason is the division by F', which leads to the multiplication by the inverse of F'. Looking at the extended interval arithmetic (2.6), we have to prove something similar for all three cases, where $0 \in F' \neq [0, 0]$. Because those are half-open intervals, these lemmas do not show that the root is in the interval rather lower than the upper bound or higher than the lower bound. However, because the sign of $f(m)$ is unclear, we have two lemmas for every case, one with the assumption `f m < 0` and one with `0 < f m`. Following the design of the function to be proven a version with $f(m) = 0$ is not needed.

For all following lemmas the first two assumptions are the same with `"x ∈ set_of X"` and `"f x = 0"`, the latter two differ. Because of that, all six lemmas proving the bounds and their differences can easily be shown in a table:

	lower F' < 0 ∧ upper F' = 0	lower F' < 0 ∧ 0 < upper F'	lower F' = 0 ∧ 0 < upper F'
f m < 0	x ≤ m - f m / lower F'	⟵ ∨ ⟶	x ≥ m - f m / upper F'
0 < f m	x ≥ m - f m / lower F'	⟵ ∨ ⟶	x ≤ m - f m / upper F'

Table 3.2: Bounds, if the Enclosure contains 0

The final step for the auxiliary lemmas is replacing $f(m)$ with an enclosure F, analogous to (2.7). Therefore we first have to extend our locale:

```
locale newton_step_locale' = newton_step_locale +
  fixes F :: "real interval"
  assumes f_enclosure: "f m ∈ set_of F"
```

Using both the `f_enclosure` assumption and the `enclosure_not_zero` lemma we have proven earlier we get for $0 \notin F'$:

```
lemma f'enclosure_not_zero:
  assumes "x ∈ set_of X" "f x = 0" "0 ∉ set_of F'"
  shows "x ∈ set_of (interval_of m - F * inverse_interval' F')"
```

Of course we also need an updated table for the half open intervals. Important here is the difference when the upper or the lower bound of F is needed, as this makes quite a difference:

	lower F' < 0 ∧ upper F' = 0	lower F' < 0 ∧ 0 < upper F'	lower F' = 0 ∧ 0 < upper F'
upper F < 0	x ≤ m - upper F / lower F'	⟵ ∨ ⟶	x ≥ m - upper F / upper F'
0 < lower F	x ≥ m - lower F / lower F'	⟵ ∨ ⟶	x ≤ m - lower F / upper F'

Table 3.3: Refined Bounds, if the Enclosure contains 0

As before, if $0 \in F$, we go into a different case and therefore do not have to show anything at this point.

Before we can start the proof of `newton_step_float`, we need a new locale using floats.

```
locale newton_step_locale_float =
  fixes f f' :: "real ⇒ real"
  fixes X F F' :: "float interval"
  fixes m :: "float"
  assumes f': "x ∈ set_of (real_interval X) ⟹
    (f has_field_derivative f' x)(at x within set_of (real_interval X))"
  assumes f'_enclosure: "x ∈ set_of (real_interval X) ⟹
    f' x ∈ set_of (real_interval F')"
  assumes m: "m ∈ set_of X"
  assumes f_enclosure: "f m ∈ set_of (real_interval F)"
```

As mentioned before, the floats and float intervals of this locale can easily be interpreted as `newton_step_locale'`:

```
interpretation newton_step_locale' f f' "real_interval X"
  "real_interval F'" "real_of_float m" "real_interval F"
```

With this we can use all the above lemmas and are finally able to show the correctness of the mathematical backbone `newton_step_float`:

```
lemma newton_step_float_correctness:
  fixes x :: real
  assumes "x ∈ set_of (real_interval X)" "f x = 0"
  shows "x ∈ real_set_of_float_interval_list (
    newton_step_float prec X F F' m)"
```

One small note: The function `real_set_of_float_interval_list` is the union over all float intervals of a list, but every interval is converted with `real_interval`, thus creating a set of real numbers.

As the lemma is the most important one, I'll go into more details of the proof. There are five different cases to consider:

- `"lower F' = 0" "upper F' = 0"`
- `"0 < lower F' ∨ upper F' < 0"`
- `"lower F' < 0" "upper F' = 0"`
- `"lower F' = 0" "upper F' > 0"`
- `"lower F' < 0" "0 < upper F'"`

Enclosure is Zero

The first case is primarily another case distinction if $0 \in F$. If this holds, the result is the input interval and contains every root.

If this is false it is easy in theory, however in practice a little bit more complicated. The first step is proving the following lemma:

```
lemma deriv_is_zero:
  fixes f :: "real ⇒ real"
  assumes "l ≤ u"
    and "⋀x. ⟦l ≤ x; x ≤ u⟧ ⟹
      (f has_field_derivative (λx. 0) x) (at x within {l..u})"
  shows "∀x ∈ {l..u}. ∀x' ∈ {l..u}. f x = f x'"
```

With `deriv_is_zero` we know that $f(x) = f(m)$. In the next step, we have $f(m) \neq 0$, as $f(m) \in F$, but $0 \notin F$. This however leads to a contradiction with the assumption, that $f(x) = 0$.

Zero is not in the Enclosure

Despite this being the easiest part to show when proving the algorithm for real numbers, it was probably the most complicated on floats. For every operation, we needed to show, that this operation is monotonic. This included among other things the difference,

```
lemma diff_mono_real_interval:
  assumes "a ≤ real_interval a'" "b ≤ real_interval b'"
  shows "a - b ≤ real_interval (a' - b')"
```

the multiplication,

```
lemma mult_float_interval_mono:
  assumes "a ≤ real_interval a'" "b ≤ real_interval b'"
  shows "a * b ≤ real_interval (mult_float_interval prec a' b')"
```

and the division, respectively the inversion.

```
lemma inverse_float_interval'_mono:
  assumes "a ≤ real_interval a'" "0 ∉ set_of a'"
  shows "inverse_interval' a ≤ real_interval (inverse_float_interval'
     prec a')"
```

With mainly those three lemmas we can show, that:

```
"set_of (interval_of m - real_interval F * inverse_interval' (
real_interval F')) ⊆ set_of (real_interval (interval_of m -
mult_float_interval prec F (inverse_float_interval' prec F')))
```

With this, `f'enclosure_not_zero` and an auxiliary lemma on interval intersections we can show that the root is still in this interval.

Enclosure contains Zero

The two cases if either the lower or the upper bound of the enclosure is zero are almost identical and the case if zero is inbetween those two bounds is only a mixture and therefore similar as well. Because of that we only look at the proof for the case `"lower F' < 0" "upper F' = 0"`:

Before starting the proof, we have to distinguish between three more cases, as we do in the function we have to prove:

- `"0 < lower F"`
- `"upper F < 0"`
- `"0" ∈ set_of F`

Let us look at the first case: Following the table of bounds we've shown earlier we know, that $x \leq m - \overline{F}/\underline{F'}$ (3.3). With this we can show, that:

```
"x ≤ float_plus_up prec m (- float_divl prec (upper F) (lower F'))"
```

To prove the current goal, we only need one more lemma on the half open interval intersection:

```
lemma in_both_float_intervals_in_ul_intersection':
  assumes "x ∈ set_of (real_interval ivl)" "x ≤ u"
  shows "x ∈ real_set_of_float_interval_list (
     float_interval_intersection_unbounded_lower ivl u)"
```

Using this and the definition of `newton_step_float` we can show, that the root x is still in the interval.

While the second case can be shown analogously, the third case is proven differently. As we only split the interval into two intervals, it is trivial to show that no element is lost.

3.3.2 Newton Step – Processing

After showing, that roots are retained in the mathematical part, we of course have to show the same for `newton_step_floatarith`. Therefore we have to prove the following lemma:

```
lemma newton_step_floatarith_correctness:
  fixes x :: real
  assumes "newton_step_floatarith f prec ivl = Some ivls"
    "interpret_floatarith f [x] = 0" "x ∈ set_of (real_interval ivl)"
  shows "x ∈ real_set_of_float_interval_list ivls"
```

Following the implementation there are two case distinctions to be made, if `m` $\leq$ `lower ivl` or not. If yes the resulting interval list equals `[ivl]`, so the root is obviously still included.

If not we use the first assumption and the monad syntax to compute the derivative, obtain the intervals and show that the derivative is correct:

- `"f' = DERIV_floatarith 0 f"`
- `"approx prec f' [Some ivl] = Some F'"`
- `"approx prec f [Some (interval_of m)] = Some F"`
- `"isDERIV_approx prec 0 f [Some ivl]"`

This case is proven easily using the interpretation as `newton_step_locale_float`, so that we can use the already proven lemma `newton_step_float_correctness`. However, the interpretation is not as trivial as it seems at first glance.

The first problem is defining the functions `f f' :: "real ⇒ real"` from the two floatarith expressions. This can be achieved with the following two lambda-functions: `"λx. interpret_floatarith f [x]"` $\stackrel{!}{=} f_i$, `"λx. interpret_floatarith f' [x]"` $\stackrel{!}{=} f_i'$.

In order to prove this interpretation, we have to show four goals:

- f_i' being the derivative of f_i on `ivl`
- the correctness of `F'`, the enclosure of f_i'
- the correctness of `F`, the enclosure of f_i

- `m` being in the interval `ivl`

Most of those goals can be shown using lemmas of the `Approximation` package. Now we can use, as previously indicated, `newton_step_float_correctness` and have:

```
"x ∈ real_set_of_float_interval_list (newton_step_float prec ivl F F' m)"
```

The last step is to show, that the root is also in the rounded list of intervals, which we get after mapping. With one auxiliary lemma based on "`in_round_intervalI`" we show that we do not lose any elements after rounding. That this holds is no big surprise, as the lower bound is truncated down and the upper bound truncated up.

3.3.3 Recursive Newton Method

Now what is left to show is the correctness of the recursive `newton_floatarith` function. We do so with the following lemma:

```
lemma newton_floatarith_correctness:
  fixes x :: real
  assumes "newton_floatarith eps f prec ivl = Some ivls"
    "interpret_floatarith f [x] = 0" "x ∈ set_of (real_interval ivl)"
  shows "x ∈ real_set_of_float_interval_list ivls"
```

As the function is recursive, it is almost natural to prove it with induction. We start with a case distinction, whether the function terminates or not. If this is the case, `ivls` equals `[ivl]`, and `"x ∈ real_set_of_float_interval_list ivls"` directly follows from the induction hypothesis.

The more difficult part is certainly the recursive case:

```
(rec) ivls' ivlss' where
  "newton_step_floatarith f prec ivl = Some ivls'" "ivls = concat ivlss'"
  "mapM (newton_floatarith eps f prec) ivls' = Some ivlss'"
```

Using both the premises and the previously shown correctness lemma of the Newton step we can show, that `ivls'` contains the fixed root. Because of that there exists an `i` so that `"x ∈ set_of (real_interval (ivls' ! i))"` and `"i < length ivls'"`.

In the next step we show that the result of the recursive function with the i-th interval is "Some float interval": `"newton_floatarith eps f prec (ivls' ! i)= Some xivls"`. With the premises and the hypothesis we can show, that the root is contained in `xivls`.

In the next step we rewrite `ivlss'` and show, that `xivls` $\in$ `set ivlss'`. Obviously `xivls` is in the concatenation of `ivlss'`, and because the root is in `xivls`, it is also in the concatenation and therefore in the result of the Newton function.

An important thing to note: While we have shown, that we do not lose any roots applying this method, we cannot guarantee that there are no false positives. What I mean by that is the possibility of intervals in the result list that do not contain a root.

4 Examples

After all the theory it is finally time to test the implementation of the interval Newton method and evaluate the results. If not stated otherwise, the default parameters are 30 for both epsilon and the precision argument. Testing showed setting both of these parameters to the same value delivers the best results. Also, the interval, in which we search for roots, is $[-10, 10]$ by default.

However, not only the quality of the results will be examined, but also the needed amount of iterations to get those results. The amount of iterations is counted as the height of the recursion tree: the longest sequence of nested intervals leading into a result interval.

There are some intervals with a pretty small width so that both endpoints are the same after rounding, for example, $[-2.0000000019, -1.9999999991]$. These intervals are marked as $[-2.00 \precsim -2.00]$, this is for sure not an optimal solution, however the best I came up with while retaining readability in tables.

Before heading into the examples we look at a concrete evaluation for $f(x) = x$, the identity function with the single root 0:

```
definition f where "f = floatarith.Var 0"
definition I where "I = Interval''(-10,10)"
value "newton_floatarith 30 f 30 I"
  = "Some [Interval (Float 0 (- 1), Float 0 0)]"
```

The result $\left[0 \cdot 2^{-1}, 0 \cdot 2^{0}\right]$ equals $[0, 0]$, our desired result. As a small reminder, if we set I to for example $[-10, -5]$, the result is `"Some []"` instead of `"None"`.

4.1 Basic Cases

First, we look at some basic functions to see, if the method is working as intended and to get some first results and check their accuracy.

We start with approximating simple roots of polynomials. Of course, there are more convenient methods to determine the roots of these functions, especially in the following examples: As all functions are factorised, it is trivial to identify the roots.

Function	Result	Iterations
$f_1(x) = 0$	$[-10, 10]$	1
$f_2(x) = 1$	$\emptyset$	1
$f_3(x) = x$	$[0, 0]$	2
$f_4(x) = x - 1$	$[1, 1]$	2
$f_5(x) = (x-1)(x+2)$	$[-2, -2]\,;[1, 1]$	10
$f_6(x) = (x-1)(x+2)(x-3)$	$[-2.000 \precsim -2.000]\,;[1, 1]\,;[3, 3]$	10
$f_{6'}(x) = x^3 - 2x^2 - 5x + 6$	$[-2.000 \precsim -2.000]\,;[1, 1]\,;[3.000 \precsim 3.000]$	10
$f_{6_o}(x) = xxx - 2xx - 5x + 6$	$[-2, -2]\,;[1, 1]\,,[3, 3]$	11

Table 4.1: Polynomial Cases

Except for the different variations f_6 all results are not only correct but as accurate as possible. The first root of f_6 and both the first and third roots of $f_{6'}$ are approximated accurately, but not optimally, which would be the exact intervals $[-2, -2]$ and $[3, 3]$. The first interval, approximating the root -2, is equal for both functions:

$$\left[-2 - 2^{-29}, -2 + 2^{-30}\right] \approx [-2.0000000019, -1.9999999991]$$

The approximation of the root 3 of $f_{6'}$ has a pretty similar accuracy:

$$\left[3 - 3 * 2^{-29}, 3 + 2^{-28}\right] \approx [2.9999999944, 3.0000000037]$$

What is remarkable on the other hand is the result of f_{6_o}, an optimised version of f_6. This is a really important observation: Rewriting a function can lead to a different and hopefully better result. Now we know for sure that the method can give us the best possible result for f_6 after rewriting. But is it possible, to get the best possible result without changing the function?

In order to change the result without changing the function, we have to change the parameters. Setting both the epsilon and the precision to 50 gives us the following intervals after 11 iterations:

$$\left[-2-2^{-49},-2+2^{-50}\right];[1,1];[3,3]$$

This is a more accurate approximation, but still not the optimal result $[-2,-2]$ we achieved with f_{6_o}. The result of f_6 after 12 iterations and setting both parameters to 79 is optimal:

$$[-2,-2];[1,1];[3,3]$$

However, if we set both epsilon and the precision to for example 86, after 12 iterations we get a worse result than before:

$$\left[-2-2^{-85},-2+2^{-86}\right];[1,1];[3,3]$$

And if we go back to 79 for both epsilon and precision, but take these parameters for $f_{6'}$, the method terminates after 11 iterations with the following result:

$$[-2,-2];[1,1];\left[3-2^{-78},3+2^{-77}\right]$$

This example shows, how much the final result is not only depending on the parameters but also how the function is defined. But why is that? The reason for this is hidden in the `Approximation` package, and to be more concrete in the `DERIV_floatarith` and the `approx` function. All variations of f_6 have the derivative $3x^2-4x-5$ but in different forms. The correct bounding interval F' would be $\left[-6\frac{1}{3},335\right]$, the enclosure of the midpoint is $F=[6,6]$. However, looking at the calculated derivatives and more importantly on the approximated enclosures F' we see quite some differences:

Function	Derivative	F'	F
f_6	$f_6' = (x-1)(x+2)+(2x+1)(x-3)$	$[-610,590]$	$[6,6]$
$f_{6'}$	$f_{6'}' = 3x^2-4x-5$	$[-45,335]$	$[6,6]$
f_{6_o}	$f_{6_o}' = 3xx-4x-5$	$[-645,635]$	$[6,6]$

Table 4.2: Effects of Rewriting Functions

This is rather surprising, as the most accurate approximation of F' delivers the worst result and vice versa. Of course, this is only the first step and not what I want to focus on. More important is the possible effect rewriting a function can have, as the width of the approximation of the same function differs between 380 and 1280, which is more than threefold.

Probably more interesting than finding trivial roots of factorised polynomials is the approximation of irrational numbers, for example $\sqrt{2}$. This is, because there are no natural numbers so that $(n/m)^2 = 2$. Therefore the result of an equation like $x^2 - 2 = 0$ can only be approximated, if just writing $\sqrt{2}$ is not an option. This equation not only has an irrational number as a result, but is in the perfect form of what the interval Newton method is about. Replacing 0 with $f_9(x)$ we have a function, with the method we can approximate the roots of f_9 and thus approximate $\sqrt{2}$. In the next examples we look at functions with square roots as roots, some of them are irrational, some of them not:

Function	Result	Iterations
$f_7(x) = xx$	$\left[-4.887 \times 10^{-10}, 0\right] ; \left[0, 1.449 \times 10^{-9}\right]$	27
$f_8(x) = xx - 1$	$[-1, -1] ; [1, 1]$	10
$f_9(x) = xx - 2$	$[-1.414 \precsim -1.414] ; [1.414 \precsim 1.414]$	9
$f_{10}(x) = xx - 3$	$[-1.732 \precsim -1.732] ; [1.732 \precsim 1.732]$	9
$f_{11}(x) = xx - 4$	$[-2, -2] ; [2, 2]$	8
$f_{12}(x) = xx - 5$	$[-2.236 \precsim -2.236] ; [2.236 \precsim 2.236]$	9

Table 4.3: Square Roots

Before we talk more about the roots, it is notable, that f_7 needs more than twice the iterations as all other functions, we looked at so far. The reason for this is the multiple root 0. Multiple roots are a known numerical difficulty and a quote by Goedecker sums this up very well:

> If roots of high multiplicity exist, [any] method [for finding roots has] to be used with caution. [1, P. 1063]

Now let us have a closer look on the approximation of the square roots. Therefore we look at the approximations of the positive roots of f_9, f_{10} and f_{12}, the approximations of $\sqrt{2}$, $\sqrt{3}$ and $\sqrt{5}$:

$$\sqrt{2} \in [1.41421356145292520523071289 0625, \quad 1.41421356238424777984619140625] =: I_9$$
$$\sqrt{3} \in [1.73205080721527338027954101 5625, \quad 1.73205080814659595489501953125] =: I_{10}$$
$$\sqrt{5} \in [2.23606797680258750915527343 75, \quad 2.23606797866523265838623046875] =: I_{12}$$

All three intervals are rather small and therefore the approximations accurate. Let us have a closer look at the width of these intervals:

$$\begin{aligned} \mathrm{w}(I_9) &= 9.313 \times 10^{-10} \\ \mathrm{w}(I_{10}) &= 9.313 \times 10^{-10} \\ \mathrm{w}(I_{12}) &= 1.863 \times 10^{-9} \end{aligned}$$

The obvious question is if we can decrease the width in order to increase the accuracy. And yes, setting both the epsilon and the precision argument to higher values leads to better approximations. In the next table only the widths of the intervals, approximating the respective square roots, are listed. Also, the starting interval is set to $[0, 5]$.

Precision/ Epsilon	Accuracy $\sqrt{2}$	Accuracy $\sqrt{3}$	Accuracy $\sqrt{5}$	Iterations
30	9.313×10^{-10}	9.313×10^{-10}	1.863×10^{-9}	7, 7, 7
40	9.095×10^{-13}	9.095×10^{-13}	1.819×10^{-12}	7, 7, 8
50	8.882×10^{-16}	8.882×10^{-16}	1.776×10^{-15}	7, 8, 8
100	7.889×10^{-31}	7.889×10^{-31}	1.578×10^{-30}	8, 9, 9
500	3.055×10^{-151}	3.055×10^{-151}	6.110×10^{-151}	11, 11, 11

Table 4.4: Accuracy of Root Approximations

There are two interesting observations to be made: The approximations of $\sqrt{2}$ and $\sqrt{3}$ have the same interval width for these arguments and, more importantly, the method does not need that many more iterations for way better quality. One more extreme example: The approximation of $\sqrt{2}$ with setting both the precision and epsilon arguments to 10000 the implementation does need 15 iterations, with 100000 it ends after 19 iterations.

4.2 Advanced Cases

For the more advanced cases, we increase both the epsilon and the precision value and set them to 50.

We start with a function with a root of the high multiplicity 50, namely $f_{13} = x^{50}$. After 56 iterations we get a good result in the interval $[-10, 10]$, if we take into account, that roots of these functions are often rather difficult to calculate:

$$\left[-8.882 \times 10^{-16}, 0\right] ; \left[0, 8.882 \times 10^{-16}\right]$$

In the next step, we once again approximate an irrational number, namely π. The interval, in which we search for roots of $f_{14} = \sin(x)$, is $[2, 4]$. After 7 iterations we get the following result:

$$[3.142 \precsim 3.142] =: I_{14} \quad \Longrightarrow \quad \mathrm{w}(I_{14}) = 7.105 \times 10^{-15}$$

There is one more "famous" irrational number left, which is e. In the interval $[1, 3]$ we search for the root of $f_{15} = \ln(x) - 1$. The implementation terminates after 6 iterations with the following result:

$$[2.718 \precsim 2.718] =: I_{15} \quad \Longrightarrow \quad \mathrm{w}(I_{15}) = 5.329 \times 10^{-15}$$

As the natural logarithm is the inverse function of e^x, the same holds for the Lambert W-function and xe^x. Using the interval Newton method we can approximate $\mathrm{W}(z)$ by searching the roots of $f(x) = xe^x - z$. For example we approximate $\mathrm{W}(23)$, a root of the function $f_{16}(x) = xe^x - 23$, in $[1, 3]$. After 11 iterations we get the following result:

$$[2.302 \precsim 2.302] =: I_{16} \quad \Longrightarrow \quad \mathrm{w}(I_{16}) = 5.329 \times 10^{-15}$$

4.3 Corner Cases

Now we take a look at some corner cases with epsilon and the precision being 50:

Function	Interval	Result	Iterations
$f_{17} = x^2$	$\left[2^{-30}, 10\right]$	$\emptyset$	24
$f_{17} = x^2$	$\left[2^{-50}, 10\right]$	$\left[8.882 \times 10^{-16}, 1.896 \times 10^{-15}\right]$	38
$f_{18} = x^2 - 2$	$\left[2^{-50}, 10\right]$	$\emptyset$	3
$f_{18} = x^2 - 2$	$\left[2^{-100000}, 10\right]$	$\emptyset$	3
$f_{19} = \sin(x + 1)$	$\left[-1 + 2^{-30}, 2\right]$	$\emptyset$	3
$f_{19} = \sin(x + 1)$	$\left[-1 + 2^{-50}, 2\right]$	$\left[-1 + 2^{-50}, -1 + 15 \cdot 2^{-51}\right]$	6

Table 4.5: Roots Outside of Starting Interval

We start with functions possessing roots that are not included in the interval by a small margin. These examples show, that our bounds often have to come very close to the root to get false positives. This is a good time to recall that while we have proven, that we do not lose any root applying the method, we cannot prevent intervals without a root in our result. In the examples we get these wrong approximations for f_{17} and f_{19} using the small distance of 2^{-50}.

In the following examples we take a look at functions where one of the bounds of the starting interval is a root. Also, I added the result if the root is inside the starting interval for direct comparison, the corresponding rows are highlighted:

Function	Interval	Result	Iterations
$f_{20} = x + 3$	$[-10, -3]$	$[-3, -3]$	2
$f_{20} = x + 3$	$[-3, 10]$	$[-3, -3]$	2
$f_{20} = x + 3$	$[-10, 10]$	$[-3, -3]$	2
$f_{21} = x^2 - 9$	$[-10, -3]$	$\left[-3 - 1.776 \times 10^{-15}, -3\right]$	7
$f_{21} = x^2 - 9$	$[-3, 0]$	$\left[-3, -3 + 1.776 \times 10^{-15}\right]$	7
$f_{21} = x^2 - 9$	$[0, 3]$	$\left[3 - 1.776 \times 10^{-15}, 3\right]$	7
$f_{21} = x^2 - 9$	$[3, 10]$	$\left[3, 3 + 1.776 \times 10^{-15}\right]$	7
$f_{21} = x^2 - 9$	$[-10, 10]$	$[-3, -3]\,;[3, 3]$	10
$f_{22} = \sin(x)$	$[-2, 0]$	$\left[-2.119 \times 10^{-15}, 0\right]$	6
$f_{22} = \sin(x)$	$[0, 2]$	$\left[0, 5.354 \times 10^{-15}\right]$	6
$f_{22} = \sin(x)$	$[-2, 2]$	$\left[-2.119 \times 10^{-15}, 0\right] ; \left[0, 5.354 \times 10^{-15}\right]$	7

Table 4.6: Roots at Bounds of Starting Interval

The result for each of these examples behaves a little different: For f_{20}, the interval does not have any difference, the result is the same and so is the number of iterations.

In the example of f_{21} we can see that the quality of the approximation of the root decreases if the root is at the interval bound. Of course, the offset is low; however, if the root is not equal to a bound, the approximation would be exact. This distance can be small, the result of the method for the interval $\left[-3 - 2^{-50}, 3 + 2^{-50}\right]$ is the desired interval $[-3, -3]$ and $[3, 3]$.

In the third example, we approximate the only rational root of $\sin(x)$, which is 0. Here we cannot achieve a better result with enlarging the interval, instead, the opposite happens: In the first iteration we split the starting interval into the two intervals $[-2, 0]$ and $[0, 2]$. This also explains why the result is simply the concatenation of the previous results and takes one more iteration to terminate. To prohibit this split, we can use the starting interval $[-1, 2]$ and get the result $\left[-2.335 \times 10^{-15}, 2.671 \times 10^{-15}\right]$. Now the root does not appear twice in the result, however, it is still not the optimal result $[0, 0]$, which I could not find in testing.

Lastly we look at two functions without a root, despite getting infinitely close to it:

Function	Interval	Result	Iterations
$f_{23} = x^{-1}$	$[-1000000, -500000]$	$\emptyset$	1
$f_{23} = x^{-1}$	$[500000, 1000000]$	$\emptyset$	1
$f_{24} = e^x$	$[-1000000, -500000]$	$\emptyset$	20

Table 4.7: Functions Converging to Zero Without Roots

For both functions, the method does result in an empty list, our desired result.

5 Conclusion

With approximately 1000 lines of code, I have successfully formalised the interval Newton method using the proof assistant Isabelle/HOL. With this implementation, it is possible to approximate the roots of many types of functions, supported by the floatarith datatype.

The main part of this thesis is the proof that we do not lose any roots using the method, so this was an expected outcome during testing. A positive surprise, however, was the quality of our results. Except from functions with roots of high multiplicity or irrational roots we often got perfect results. And speaking of irrational roots, we have also shown that we can use this method to calculate good approximations of square roots and π, for example, in a reasonable time.

Additionally, using this method, in theory, it would be possible to implement the Lambert W-function in Isabelle/HOL. Maybe the floatarith datatype could be extended, as of right now both the exponential function and the natural logarithm are supported, however not the Lambert W-function.

Future Work

For a possible work in the future, the next step would be the support of more than just one variable for functions, which would enable more than just one dimension.

Right now, the quality of our implementation is heavily dependent on the quality of the approximations of enclosures. Recalling the effect, what simple rewriting can have on the calculation of enclosures (4.2), it could be possible to increase the quality of our results by getting better approximations.

Another possibility to improve our results is some sort of post-processing. Right now we have our results as soon as the recursive function terminates. There certainly is some unused potential to filter out intervals containing no root and to merge overlapping intervals containing the same root.

List of Tables

Bibliography

[1] S. Goedecker. 'Remark on Algorithms to Find Roots of Polynomials'. In: *SIAM Journal on Scientific Computing* 15.5 (1994), pp. 1059–1063. ISSN: 1064-8275. DOI: 10.1137/0915064.

[2] R. E. Moore, R. B. Kearfott and M. J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, 2009. ISBN: 978-0-89871-669-6. DOI: 10.1137/1.9780898717716.

[3] D. Seidl. *Formalisation of the Interval Newton Method in Isabelle/HOL*. 2021. DOI: 10.5281/zenodo.4757264.

[4] M. Wenzel. 'The Isabelle/Isar Reference Manual'. In: (2021).