석 사 학 위 논 문
Master's Thesis

검증된 실수 연산

Verified Real Computation

2017

박 세 원 (朴 世 原 Park, Sewon)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

석 사 학 위 논 문

검증된 실수 연산

2017

박 세 원

한 국 과 학 기 술 원

전산학부

검증된 실수 연산

박 세 원

위 논문은 한국과학기술원 석사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2017년 6월 9일

심사위원장 Martin Ziegler (인)

심 사 위 원 류 석 영 (인)

심 사 위 원 Helmut Alt (인)

Verified Real Computation

Sewon Park

Advisor: Martin Ziegler

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Daejeon, Korea
May 31, 2017

Approved by

Martin Ziegler
Professor of School of Computing

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

MCS
20154408

박세원. 검증된 실수 연산. 전산학부 . 2017년. 41+iii 쪽. 지도교수: 지글러마틴. (영문 논문)
Sewon Park. Verified Real Computation. School of Computing . 2017. 41+iii pages. Advisor: Martin Ziegler. (Text in English)

초 록

근삿값의 무한수열으로서의 실수를 사용하는 연산의 경우 대수적 계산 모형에서의 연산 또는 자연수 유리수 등과 같은 이산적인 대상에 대한 연산과 다른 특성을 갖는다. 실수의 크기 비교는 계산 불가능한 문제임이 알려져 있으며 실수 연산의 계산 복잡도 또한 출력 오차에 영향을 받기에 대수적 계산 모형에서의 또는 이산적 대상에 대한 계산 이론을 직접 대입하기 어려운 점이 있다. 프로그램 검증은 종료 조건을 포함한 프로그램의 입출력 관계에 대한 명제를 증명하는 것이며 간단한 명령형 프로그램의 경우 몇 가지의 추론 규칙을 통하여 진행할 수 있다. 이 논문에서는 기존에 존재하던 프로그램 검증의 추론 규칙을 실수 연산으로 확장한다. 또한 몇 가지 실수 연산 프로그램을 예시로 확장된 추론 규칙을 사용하여 검증한다.

중복된 고윳값을 가질 수 있는 행렬의 경우 튜링 머신에서 계산 불가능하며 추가로 서로 다른 고윳값의 개수가 주어질 경우 계산 가능하다는 사실이 알려져 있다. 이 논문에서는 서로 다른 고윳값의 개수가 주어진 실수 대칭 행렬 또는 복소수 에르미트행렬의 대각화 문제에 대해서 다룬다. 특성 다항식의 해를 찾는 여러 알고리즘들을 분석하며 고윳값을 근사하는 방식에 대해 새로운 접근 방법을 제안한다. 또한 위의 알고리즘들을 실제 구현하며 실행 및 분석한다.

핵 심 낱 말 계산 해석학, 실수 연산, 프로그램 검증, 행렬 대각화

Abstract

Computing with infinite data such as real numbers cannot be performed exactly on a Turing machine. Instead, real numbers are treated as an infinite sequence of approximations. Computing a real function means that a Turing machine converts approximations to the argument into approximations of the value, according to Computable Analysis. [Brattka&Hertling'98] have suggested an equivalent but more practical model of computation over the reals: the feasible real-RAM. Based on this fundamental idea, the `iRRAM` library for `C++` has been developed as an imperative programming language which supports abstract datatypes with exact and multivalued semantics.

In this dissertation, we extend Floyd-Hoare logic to formally verify correctness of such algorithms. A set of inference rules is suggested as a systematic framework for program verification.

We also consider the diagonalization of complex self–adjoint matrices with a particular emphasis on the degenerate cases, which in general are ill-posed and in fact provably uncomputable. Employing a Recursive Analysis, we have designed, implemented and evaluated several reliable algorithms on `iRRAM` which can compute *some* orthonormal basis of eigenvectors, in the sense of output approximation, up to any desired precision - provided that, in addition to approximations to the $d \cdot (d+1)/2$ matrix' entries, its number of *distinct* eigenvalues k is known/provided.

Keywords Computable Analysis, Exact Real Computation, Formal Verification, Matrix Diagonalization

Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1 Computable Analysis

Computing with real numbers or other continuous mathematical objects is often misunderstood. In computing, to represent real numbers, a system of fixed precision floating point numbers such as `double` is commonly used. However, such numeric systems cannot exactly represent the mathematical behaviors of real numbers. For example, arithmetic operations, such as $\{+, -, \times, \div\}$, introduce rounding errors with the floating–point arithmetic. Hence, the floating–point real numbers fail to form a field.

Instead, Computable Analysis considers with a continuous object as a sequence of finite approximations [1, 2]. For example, a real number x is considered as a sequence $(q_i)_{i\in\mathbb{N}} \subseteq \mathbb{Q}_2$ of dyadic rational numbers such that $|x - q_i| \leq 2^{-i}$ [3]. This representation makes addition of two real numbers as $(x + y)_i = (x)_{i+1} + (y)_{i+1}$ precisely equal to the one in mathematics.

However, not all real numbers are computable via a Turing machine. There are real numbers for which a Turing machine does not exist to compute the dyadic approximations.

The formal definition of a real number that is computable is as follows:

Definition 1. *A real number x is computable if a Turing machine exists that can compute $q_i \in \mathbb{Q}_2$ such that $|x - q_i| \leq 2^{-i}$ for any input $i \in \mathbb{N}$.*

To generalize the notion of computability, the formal definition of a computable function can be defined as follows:

Definition 2. *A real function $f : \mathbb{R} \to \mathbb{R}$ is computable if a Turing machine exists that can convert any sequence $(p_i)_{i\in\mathbb{N}} \subseteq \mathbb{Q}_2$ where $|x - p_i| \leq 2^{-i}$, into a sequence $(q_j)_{j\in\mathbb{N}} \subseteq \mathbb{Q}_2$ such that $|f(x) - q_j| \leq 2^{-j}$.*

It can be understood that a function is computable if an output of the function is computable up to any desired precision, where an input of the function is accessible up to an arbitrary precision during the computation. With this notion of computability, it is easy to verify that the field operations of real numbers are computable; moreover, transcendental numbers and functions such as $e, \sin x, \cos x$ are also computable [4]. Here, we introduce a fundamental theorem for the computability theory of real numbers:

Theorem 3. *A computable function is continuous* [2, Theorem 4.3.1].

The theorem implies that any discontinuous functions are not computable. A direct consequence of the theorem is a little surprising:

Corollary 4. *The inequality test of real numbers is not computable;*

$$f(x, y) = \begin{cases} 1 & \text{if } x > y, \\ 0 & \text{otherwise.} \end{cases}$$

is not continuous. This is obvious from how we defined a real number. One can say that two real numbers x, y, whose representations are sequences $(x_i), (y_i)$, differ if some indices i, j exist, such that

$|x_i - y_j| > 2^{-i} + 2^{-j}$. *Hence, distinguishing two real numbers is the process of finding such indices* i, j. *However, if the two real numbers coincide, then no such indices exist. Therefore, distinguishing two real numbers never halts if the two real numbers are identical.*

One who is familiar with recursive analysis would recognize that the equality test is a well-known equivalent of the co-Halting problem [2, Exercise 4.2.9]. What the theorem and the corollary say is that it is indistinguishable whether two real numbers are arbitrarily close to each other, or they are identical to each other. Note that the function becomes computable with the premise that $x \neq y$; the function becomes continuous when $x = y$ is removed from the domain. Such a function is called *partial* and is denoted by $f :\subseteq (\mathbb{R} \times \mathbb{R}) \to \{0, 1\}$. Another way to make the above function continuous is to modify the function's behavior. If one does not expect the function to exactly distinguish two real numbers but is willing to tolerate some error in the result, the function can be modified to be computable. Then,

$$f(x, y; \epsilon) = \begin{cases} 1 & \text{if } x > y - \epsilon, \\ 0 & \text{if } x < y + \epsilon. \end{cases}$$

The "function" is computable when $\epsilon > 0$; a Turing machine exists that can evaluate $x > y - \epsilon$ and $x < y + \epsilon$ simultaneously. Since at least one of the two conditions holds for all x, y, at least one of the evaluations will terminate eventually. Then, the Turing machine output will be either 1 or 0 depending on which one has terminated, without waiting for the other one to terminate.

An important fact to be observed here is the behavior of the function. Note that the function is multivalued; for $|x - y| < \epsilon$, $f(x, y; \epsilon)$ is defined to be both 0 and 1. This nondeterministic (in this case it has nothing to do with the nondeterministic Turing machine) and multivalued feature is an essential and unavoidable property of the real computation [5, 6]. Such a function, with the index $\epsilon > 0$, is called a *multivalued* and *total* function, and is denoted by $f : (\mathbb{R} \times \mathbb{R}) \rightrightarrows \{0, 1\}$. In computable analysis, the representation of a real number as a sequence of approximations makes the operations of the real numbers exact. That is, it follows the behaviors of the operations of the actual real numbers. However, with the representation, trivial functions such as an inequality test becomes uncomputable. The two perspectives for making such functions computable, by restricting the function domain, or modifying the function to be multivalued, are the key ideas for designing algorithms in real computation.

1.2 Exact Real Computation and `iRRAM`

Seeing a real number as an infinite sequence of approximations makes the real numbers' operations exact. Also in engineering perspective, it makes no risk of rounding or truncate errors of a numeric program; a program would output a correct approximation up to any desired precision. However, this representation, regarding a number as an infinite sequence, is unnecessarily complicated for one to build a program using the representation.

On the other hand, there exists a model of computation which regards a real number as a single entity and considers operations of real numbers, including in/-equality test, as unit cost computable: BSS-machine / algebraic unit-cost model [7]. The intuitive algebraic model of computation is simple but unrealistic; continuous objects, which contain infinite data, cannot be manipulated in unit time or exactly compared on a Turing machine [8].

Combining the strengths of each model, a model of imperative computation over real numbers has been suggested [9]: Exact Real Computation which deals a real number as a single entity, where the

computation of sequences is only done in the backend. `iRRAM`, an implemented version of Exact Real Computation, is a library of Object-Oriented and imperative programming language `C++` which offers an abstract datatype `REAL` [10]; an instance of the type `REAL` is the entity mentioned above. In this dissertation, notations provided by `iRRAM` such as `REAL` are used to describe programs. However, as only the general ideas of imperative programming for Exact Real Computation are used, the description of the program can be easily deployed in any other imperative programming language that implements Exact Real Computation.

As `REAL` is a datatype that implements the representation of real numbers introduced in the previous subchapter, an inequality test of two `REAL` variables is *partially* defined; the test does not terminate when the two real numbers, that the variables represent, coincide. Therefore,

$$x, y : \texttt{REAL}\ ,\ x > y = \begin{cases} \texttt{True} & : x > y, \\ \texttt{False} & : x < y, \\ \downarrow & : x = y. \end{cases}$$

Here, $\downarrow$ means that the program diverges, i.e., *freeze*s. This behavior of a test requires extended boolean set $\{\text{True}, \text{False}, \downarrow\}$; the datatype of the test is called `LAZY_BOOLEAN` which corresponds to the three-point generalized Sierpinski space $\{\downarrow, 0, 1\}$ with topology $\big\{\emptyset, \{0\}, \{1\}, \{0, 1\}, \{\downarrow, 0, 1\}\big\}$, where 1 corresponds to True and 0 corresponds to False.

The discussion we had in the previous section is still valid; a program with the expression $x > y$ can have the computable variants; one is to restrict the domain of the program and the other is to modify the program to be multivalued and total. Claiming a program to be correct with a restricted domain can be done by specifying a requirement of the program. Of course, the correctness of such specification needs to be proved; this perspective is mainly covered in chapter 3. On the other hand, a multivalued test can be devised with a similar concept of Dijkstra's nondeterministic guarded command [11]. Recall that in the previous subchapter, the function f became computable with the simultaneous evaluations of the two partially defined inequalities with a promise that at least one of the two inequalities holds.

Generalization of the example is to make a program evaluate several `LAZY_BOOLEAN` expressions with a promise that at least one of them holds to be `True`, then proceed a computation according to an inequality that holds. `iRRAM` has its own implementation of such multivalued test, `choose`:

$$\texttt{choose}(x_1 > y_1, x_2 > y_2, \cdots, x_n > y_n) = \begin{cases} j & :\ x_j > y_j, \\ \downarrow & :\ x_i \leq y_i \text{ for } i = 1 \cdots n. \end{cases}$$

Such multivalued test lacks of extensionality; any j that satisfies $x_j > y_j$ can be returned. See that the indexed function $f(x, y; \epsilon)$ in the previous subchapter can be expressed as follow:

```
1  int f(REAL x, REAL y, REAL epsilon)
2  {
3    if (choose(x > y - epsilon, x < y + epsilon) =1) return 1;
4    return 0;
5  }
```

One further noticeable observation is the correspondence between a function and a real number. Suppose there exists a function $f : \mathbb{N} \to \mathbb{R}$ that approximates a real number x such that $|f(n) - x| \leq 2^{-n}$. Then, the function is, by definition, the real number x. In mathematics, it is expressed in $x = \lim_{n\to\infty} f(n)$. In `iRRAM` , there is an operation `limit` which receives such f and returns an instance of the type `REAL` . If the function receives multiple inputs other than the output precision n, the `limit` operation can be applied by binding the other inputs.

1.3 Formal Verification

Formal verification is a process to prove that a program is correct. Opposed to a program written in a declarative programming language, which has a transparent and explicit specification, when a program is written in an imperative programming language, it is hard to reason its correct specification. It is because the program is only written by how a computation should proceed. Moreover, usage of mutable variables is a major reason of an imperative program being hard to reason.

Tony Hoare and Robert Floyd have attempted to specify a *simple* imperative program, which manipulates mutable states, in terms of logical propositions [12, 13]. Let C denote a program. Then, a specification of C is written by $\{P\}\ C\ \{Q\}$, where P and Q are logical propositions which describe possible states of mutable variables; let a program manipulates variables x, y, then a proposition P, where x, y are free variables of P, defines possible states of x, y. The specification is read as if the proposition P holds before the program is executed, the proposition R holds after the execution, when the program terminates. From this perspective, P is called precondition of the program p whereas R is called the postcondition of the program. Such triple that describes a specification of a program is called a Hoare triple.

A process of proving correctness of such specification is defined as verification. The condition "when the program terminates" is crucial here. Hence, a specification with unproved or disproved termination is called *partially* correct. When a specification is proved to be correct including a condition of a termination, the specification is called *totally* correct and the corresponding Hoare triple is denoted as $[P]\ p\ [R]$.

Whereas an intuitive proof of a specification is possible, a set of inference rules were introduced. So called Hoare logic is a formal system consists of such inference rules. Hence, verification is a sequence of specifications induced by the inference rules.

The logic only considers a program with integer and boolean expressions:

$$\langle \texttt{integer} \rangle \ni e = \begin{cases} 0, -1, 1, -2, 2, \cdots & , \\ v & : \text{ integer variable}, \\ e_1\ op\ e_2 & : \ e_1, e_2 \in \langle \texttt{integer} \rangle,\ op \in \{+, -, \times\}, \end{cases}$$

$$\langle \texttt{boolean} \rangle \ni e = \begin{cases} \text{True}, \text{False}, & \\ e_1\ op\ e_2 & : \ e_1, e_2 \in \langle \texttt{integer} \rangle,\ op \in \{>, \geq, <, \leq, =, \neq\}, \\ e_1\ op\ e_2 & : \ e_1, e_2 \in \langle \texttt{boolean} \rangle,\ op \in \{\wedge, \vee\}, \end{cases}$$

The expressions and their operations are considered to be exactly those of $\mathbb{Z}$ and $\mathbb{B} = \{\text{True}, \text{False}\}$. Hence, there exists a natural mapping between an expression in a program and an expression in $\mathbb{Z}$ and $\mathbb{B}$.

A program is simple in sense that it is composed of commands with only the 4 constructors:

$$\langle \texttt{command} \rangle \ni C = \begin{cases} x := e, \\ C_1; C_2, \\ \textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2, \\ \textbf{while}\ b\ \textbf{do}\ C_1, \end{cases}$$

for x an integer variable, $e \in \langle \texttt{integer} \rangle$, $b \in \langle \texttt{boolean} \rangle$ and $C_1, C_2 \in \langle \texttt{command} \rangle$. For those constructors of a command, the inference rules are defined as follow:

Assignment **(AS)**:

$$\frac{}{[P[e/x]]\ x := e\ [P]}$$

Sequential Composition **(SQ)**:

$$\frac{[P]\ C_1\ [Q] \quad [Q]\ C_2\ [R]}{[P]\ C_1; C_2\ [R]}$$

Consequence **(CONS)**:

$$\frac{P \to P'\ [P']\ C\ [Q']\ Q' \to Q}{[P]\ C\ [Q]}$$

Conditional **(COND)**:

$$\frac{[P \wedge b]\ C_1\ [R] \quad [P \wedge \neg b]\ C_2\ [R]}{[P]\ \ \textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ [R]}$$

While **(WHILE)**:

$$\frac{\forall z \exists n \in \mathbb{N}^+ : [I \wedge b \wedge V = z]\ C\ [I \wedge V = z - n], \qquad \exists n_0 \in \mathbb{Z} : I \wedge V \leq n_0 \to \neg b}{[I]\ \textbf{while}\ b\ \textbf{do}\ C[I \wedge \neg b]}$$

The other rules are easy to understand but the inference rule of the while command needs some explanation. Without ensuring a termination, the inference rule of the while command of partially correct specification is as follow:

$$\frac{\{I \wedge b\}\ C\ \{I\}}{\{I\}\ \textbf{while}\ b\ \textbf{do}\ C\{I \wedge \neg b\}}$$

The proposition I is called *loop invariant*, which does not get affected inside the while loop. For such I, if the loop terminates eventually, I should hold as it does not alter inside the while loop and $\neg b$ should also hold as the loop should have been escaped. The problem is how to ensure a termination. A value V is called *loop variant*, which is a value that strictly decreases inside the while loop. Though it can be more generalized, in this dissertation, the loop variant is forced to decrease by some positive natural numbers. If the loop's condition b has a lower bound according to V, that is, if there exists an integer n_0 such that $V \leq n_0$ implies $\neg b$, the loop is guaranteed to terminate.

Hence, formal verification of a program is a sequence of specifications induced by the of above 5 inference rules. However, see that some rules, **(SQ)** and **(WHILE)** contains implications. Though other specifications can be built directly by the inference rules, an implication needs to be proved. Therefore, a formal verification is a sequence of specifications and proved or provable implications; the implications are called *Theorem* in many theorem prover, such as `Coq`. The theorems in a verification will be labelled as **Tn** where n iterates through $1, 2, \cdots$.

Other than the inference rules of the constructors of commands, the following inference rules can be introduced:

Conjunction Rule **(CONJ)**:

$$\frac{[P]\ \mathtt{C}\ [Q] \qquad [P']\ \mathtt{C}\ [Q']}{[P \wedge P']\ \mathtt{C}\ [Q \wedge Q']}$$

Constant Rule **(CONST)**:

$$\frac{[P]\ \mathtt{C}\ [Q] \qquad \text{free variables of } R \text{ do not get assigned in } \mathtt{C}}{[P \wedge R]\ \mathtt{C}\ [Q \wedge R]}$$

The above rules are not necessary to prove correctness; however, the above two rules make a verification process simpler by splitting a specification into independent properties.

1.4 Degenerate Matrix Diagonalization

Matrix diagonalization is a well-studied problem in linear algebra with its fluent applications in quantum physics. In this dissertation, we consider Hermitian matrices; a Hermitian matrix is a complex square matrix which is self adjoint. With a complex matrix A, we denote its conjugate by $A^\dagger$. A Hermitian matrix is a complex matrix H such that $H = H^\dagger$ and a unitary matrix is a complex matrix U such that $UU^\dagger = U^\dagger U = I$ where I is an identity matrix.

It is well-known that a Hermitian matrix is unitarily diagonalizable with real eigenvalues; there exists a unitary matrix U and a diagonal matrix D with real entries such that $H = UDU^\dagger$. A computational problem of matrix diagonalization is the problem to compute such unitary and diagonal matrices.

The problem is composed as follow: first, obtain the distinct eigenvalues of a given matrix H, then compute corresponding eigenvectors. Eigenvalues are roots of the characteristic polynomial of a matrix, $\chi_H := \det(H - z \cdot I)$. Hence, obtaining the eigenvalues corresponds to (i) obtaining the characteristic polynomial of a given matrix and (ii) finding roots of the polynomial. Eigenvectors are bases of an eigenspace; for an eigenvalue λ, the corresponding eigenspace is the kernel of the operator $H - \lambda \cdot I$. Hence eigenvectors can be obtained by solving the linear system $(H - \lambda \cdot I) \cdot x = 0$.

A matrix being diagonalizable, the dimensionality of the eigenspace corresponding to an eigenvalue is exactly the multiplicity of the eigenvalue as a root of the characteristic polynomial. Hence, the dimensionality of an eigenspace can be determined by finding the multiplicity of the corresponding eigenvalue.

After having a set of eigenvectors corresponding to an eigenvalue, Gram-Schmidt process can be applied to obtain the corresponding orthonormal set of eigenvectors. A matrix that has the orthonormal set of eigenvectors as its columns is a unitary matrix U.

Chapter 2. Overview, Previous and Related Work

There have been various studies to formally verify numeric programs that uses floating point arithmetics: a numerical program that uses datatype of `float` or `double`. Moreover, an effort to automize a verification process with an actual `C` program of numeric computation was made [14]; from a specification annotated `C` source code, using a `C` static analyzer, such as `frama-c` and an intermediate platform, such as `why`, verification conditions can be obtained. The verification conditions are later passed to a theorem prover, such as `coq`, and interactively proving the conditions yields correctness of the program.

Formal verification of an imperative program over continuous datatypes, such as `REAL` , was introduced in [15]. The work had verified `iRRAM` programs in `C++` by translating them into specification annotated `C` programs, then feeding them into the `C` verification framework. However, the perspective lacks of total correctness verification; as it was seen in chapter 1.2, proving termination of `REAL` datatype is a complicated problem. The present dissertation takes a formal approach: we extend Hoare triples to the specification of multivalued computation [16]; and devise a formal proof system to derive total correctness proofs – tentatively manually, later automatically.

Matrix diagonalization, introduced in chapter 1.4, is a well-studied problem with efficient methods for the non-degenerate case: the problem is generally uncomputable; however, it becomes computable when the number of distinct eigenvalues is provided [17]; moreover, such *discrete enrichment* is *optimal* [18]. We turn this computability result into an actual algorithm, implement and evaluate it in iRRAM: the arguably first rigorous solution to the matrix diagonalization problem covering all degenerate cases.

A square matrix being possibly degenerate, the characteristic polynomial possibly has repeated roots; hence simple root finding methods cannot be applied. Therefore, clustering the roots of the characteristic polynomial is needed. In [19], a simple $\mathbb{C}$ subdivision algorithm is suggested which uses Pellet predicate [20]; the predicate was also suggested in [21]. With Gräffe's iteration [22] and modified Newton method [23, 24] the algorithm was improved in [25]. Later, the algorithm and its analysis were improved in [26]. The benchmark algorithm of ours is the latest algorithm in [26].

In chapter 3, a framework of systematic verification of real computation is introduced, improving the last work of [16]. in subchapter 3.1, the semantics of `REAL` (see chapter 1.2), `LAZY_BOOLEAN` and commands of program using the `iRRAM` datatypes are specified; moreover, with specified commands and datatypes, a simple program with an invariant context is defined. With the specified simple programs, we extend the inference rules of Hoare logic to verify the simple programs in subchapter 3.2. In chapter 3.2.1, we extend the definition of program to a function, that receives and returns values; a rule of verification for such programs is suggested in terms of the inference rules introduced in 3.2. As proofs of concept of our suggested framework, in chapter 3.3, programs of root finding and solving linear system are described and verified formally.

In chapter 4, we review and devise algorithms of matrix diagonalization with possibly degenerate real/complex self–adjoint matrices. Their correctness is not proved under the framework; however, their correctness is proved with regard to the semantics defined in chapter 3. In chapter 4.1.2, we improved the part of approximating a real root with a promise that all roots of a predicate are located in the real axis. Moreover, in chapter 4.1.3, we improved a predicate that tells the number of roots contained in an open interval (see [20, 25]). The variant algorithms are implemented and evaluated in chapter 5.

Chapter 3. Verification in Real Computation

In this chapter, a framework of formal verification to prove a correctness of a program in real computation is introduced. Formal verification of a simple imperative program with only integer variables, integer expressions and boolean expressions was recalled in chapter 1.3. As it was seen in chapter 1.2, a real computation has a few different behaviors. Therefore, in order to build a framework of verification in real computation, the semantics of `REAL` and its operations are needed to be specified.

The semantics of `REAL` are formally specified in the following subchapter. With the specified semantics, the Hoare logic introduced in chapter 1.3 is extended for the use of the datatype `REAL` in subchapter 3.2, with an example of a formal verification of a program of approximating the maximum value within two real numbers. In the following subchapter 3.3, semantics of advanced datatypes `POLYNOMIAL` and `MATRIX` are introduced. Moreover, programs with the types are also introduced: root finding program and a Gaussian elimination program.

3.1 Exact Operations and Multivalued Tests

3.1.1 Semantics of Data types

It is important to distinguish expressions in programs and values in logical statements. Though $\langle$`integer`$\rangle$ was considered to be identical to $\mathbb{Z}$ in the previous chapter, at this moment, it is needed to see the formal correspondence between a datatype of a program and a mathematical object that the datatype is describing. An expression of the datatype `integer` describes an expression in $\mathbb{Z}$; a *domain* of the type `integer` is $\mathcal{D}($`integer`$) = \mathbb{Z}_{\downarrow} := \mathbb{Z} \cup \{\downarrow\}$. Similarly, operations (`+, -, *`) defined over the datatype `integer` are to describe the operations $(+, -, \times)$ of $\mathbb{Z}$. How expressions in a program and expressions in the domain correspond is defined by a mapping so-called *semantic function*.

Let $\langle$`integer`$\rangle$ denote a set of all expressions of `integer`. Expressions are constructed by constants, $0, 1, -1, 2, \cdots$, variables and inductively by binary operations; that is, for $e \in \langle$`integer`$\rangle$, e is either a constant $e = n$ for any $n \in \mathbb{Z}$, a variable $e =$ x : `integer`, or an inductively constructed expression such as $e = e_1$ *op* e_2, where $e_1, e_2 \in \langle$`integer`$\rangle$ and *op* is one of `+,-,*`.

When an expression is a variable, the value is defined by a value that the variable is assigned to be; such mapping is called *state*, σ : var $\to \mathbb{Z}$. Let Σ be a set of all states, the semantic function is $[\![\cdot]\!]_{\langle\texttt{integer}\rangle} : \langle$`integer`$\rangle \to \Sigma \to \mathbb{Z}_{\downarrow}$. That is, $[\![\cdot]\!]_{\langle\texttt{integer}\rangle}$ maps a `integer` expression to a value in $\mathbb{Z}$. The following defines how the `integer` expressions describe $\mathbb{Z}$:

$$\begin{aligned} \forall n, \sigma,\ [\![n]\!]_{\langle\texttt{integer}\rangle}\sigma &= n \in \mathbb{Z}, \\ [\![v]\!]_{\langle\texttt{integer}\rangle}\sigma &= \sigma v \in \mathbb{Z}, \\ [\![e_1\ op\ e_2]\!]_{\langle\texttt{integer}\rangle}\sigma &= [\![e_1]\!]_{\langle\texttt{integer}\rangle}\sigma\ [\![op]\!]\ [\![e_2]\!]_{\langle\texttt{integer}\rangle}\sigma \in \mathbb{Z}_{\downarrow}. \end{aligned}$$

The above semantic function suggests that the operations of `integer` are exactly those of $\mathbb{Z}$.

However, this is not always true. As it was mentioned in chapter 1.1, a set of expressions in a datatype `double` fails to form a field; that is, the operations are not exact, due to a rounding error.

With the corresponding semantic function, $[\![\cdot]\!]_{\langle \texttt{double} \rangle} : \langle \texttt{double} \rangle \to \Sigma \to \mathbb{R}$ where $\Sigma := \text{var} \to \mathbb{R}$,

$$[\![e_1 + e_2]\!]_{\langle \texttt{double} \rangle}\sigma \neq [\![e_1]\!]_{\langle \texttt{double} \rangle}\sigma + [\![e_2]\!]_{\langle \texttt{double} \rangle}\sigma \in \mathbb{R}.$$

The difference of behaviors of the two datatypes can be interpreted that the operations of `integer` are *exact* but `double`'s are not. The notion of *exact* can be defined as follow:

Definition 5. *Consider an arbitrary datatype* `Arb` *and a set of all expressions of the type* $\langle \texttt{Arb} \rangle$. *Suppose* $\mathcal{D}(\texttt{Arb}) = \mathbb{A}_\downarrow$. *Let* $\langle \texttt{C} \rangle$ *be a set of all constants of the datatype and* ρ *be atomic semantic function such that* $\rho : \langle \texttt{C} \rangle \to \mathbb{A}$, *possibly not injective or surjective. With* $\Sigma_{\langle \texttt{Arb} \rangle}$ *being a set of all states* $\sigma : var \to \mathbb{A}$, *the semantic function* $[\![\cdot]\!]_{\langle \texttt{Arb} \rangle} : \langle \texttt{Arb} \rangle \to \Sigma_{\langle \texttt{Arb} \rangle} \to \mathbb{A}_\downarrow$ *becomes*

$$\forall c \in \langle \texttt{C} \rangle, \sigma \in \Sigma_{\langle \texttt{Arb} \rangle},\ [\![c]\!]_{\langle \texttt{Arb} \rangle}\sigma = \rho(c),$$
$$[\![v]\!]_{\langle \texttt{Arb} \rangle}\sigma = \sigma(v).$$

Let $op_{\langle \texttt{Arb} \rangle}$ *be a* n*–ary operator in* $\langle \texttt{Arb} \rangle$ *that describes a* n*–ary operator* $op_{\mathbb{A}}$ *in* $\mathbb{A}$: $[\![op_{\langle \texttt{Arb} \rangle}]\!] = op_{\mathbb{A}}$. *The operator is* exact *if*

$$[\![op_{\langle \texttt{Arb} \rangle}(e_1, e_2, \cdots, e_n)]\!]_{\langle \texttt{Arb} \rangle}\sigma = [\![op_{\langle \texttt{Arb} \rangle}]\!]([\![e_1]\!]_{\langle \texttt{Arb} \rangle}\sigma, [\![e_2]\!]_{\langle \texttt{Arb} \rangle}\sigma, \cdots, [\![e_n]\!]_{\langle \texttt{Arb} \rangle}\sigma),$$

for all σ *and* e_j.

Let arbitrary datatypes $\texttt{Arb}_1, \texttt{Arb}_2, \cdots \texttt{Arb}_n, \texttt{Arb}_{n+1}$ *describe sets* $\mathbb{A}_{1\downarrow}, \cdots \mathbb{A}_{n+1\downarrow}$. *An operator op :* $\langle \texttt{Arb}_1 \rangle \times \langle \texttt{Arb}_2 \rangle \times \cdots \times \langle \texttt{Arb}_n \rangle \to \langle \texttt{Arb}_n \rangle$ *where* $[\![op]\!] : \mathbb{A}_1 \times \cdots \times \mathbb{A}_n \to \mathbb{A}_{n+1}$ *is exact if*

$$[\![op_{\langle \texttt{Arb} \rangle}(e_1, e_2, \cdots, e_n)]\!]_{\langle \texttt{Arb}_{n+1} \rangle}\sigma = [\![op]\!]([\![e_1]\!]_{\langle \texttt{Arb}_1 \rangle}\sigma, [\![e_2]\!]_{\langle \texttt{Arb}_2 \rangle}\sigma, \cdots, [\![e_n]\!]_{\langle \texttt{Arb}_n \rangle}\sigma),$$

for all $\sigma \in var \to (\bigcup \mathbb{A}_i)_\downarrow$ *and* $e_i \in \langle \texttt{Arb}_i \rangle$ *for all* i.

Operations of `Arb` being exact, when an expression is composed of only the exact operations, semantics of the operations are simply rewriting of the expression with operations and variables in $\mathbb{A}$: $e = x\ op_1\ (3\ op_1\ y) \mapsto e' = x[\![op_1]\!]([\![3]\!][\![op_1]\!]y)$.

From the facts that were discovered in chapter 1.2, the following can be claimed:

Theorem 6. *From the definition of* `REAL` *in chapter 1.2, binary operations* `+,-,*,/, power, maximum` *and unary operations* `abs, cos, sin` *are exact with regard to the definition 5.*

However, an operation being exact does not imply that the operation is totally defined; e.g., $[\![x/y]\!]_{\langle \texttt{REAL} \rangle}\sigma = [\![x]\!]_{\langle \texttt{REAL} \rangle}\sigma \div [\![y]\!]_{\langle \texttt{REAL} \rangle}\sigma = \downarrow$ if $\sigma(y) = 0$. Hence, an expression composed of partial operations should also be considered.

Recall that an inequality test of `REAL` does not have the datatype of `boolean` but does have the datatype of `LAZY_BOOLEAN`, which corresponds to the generalized Sierpinski space, $\mathcal{S}$. Hence, the semantic function $[\![\cdot]\!]_{\langle \texttt{LAZY\_BOOLEAN} \rangle} : \langle \texttt{LAZY\_BOOLEAN} \rangle \to \Sigma \to \mathcal{S}$ of the datatype `LAZY_BOOLEAN` should be also defined, where Σ is a set of states $\sigma : \text{var} \to \mathbb{R}$.

Definition 7. *Let* $\langle \texttt{LAZY\_BOOLEAN} \rangle$ *be a set of all expressions of* `LAZY_BOOLEAN` *datatype. Let* $e_1,\ e_2 \in \langle \texttt{REAL} \rangle$. *Then,*

$$[\![e_1 > e_2]\!]_{\langle \texttt{LAZY\_BOOLEAN} \rangle}\sigma = \begin{cases} 1 & : [\![e_1]\!]_{\langle \texttt{REAL} \rangle}\sigma > [\![e_2]\!]_{\langle \texttt{REAL} \rangle}\sigma, \\ 0 & : [\![e_1]\!]_{\langle \texttt{REAL} \rangle}\sigma < [\![e_2]\!]_{\langle \texttt{REAL} \rangle}\sigma, \\ \downarrow & : [\![e_1]\!]_{\langle \texttt{REAL} \rangle}\sigma = [\![e_2]\!]_{\langle \texttt{REAL} \rangle}\sigma \text{ or } [\![e_1]\!]_{\langle \texttt{REAL} \rangle}\sigma = \downarrow \vee [\![e_2]\!]_{\langle \texttt{REAL} \rangle}\sigma = \downarrow . \end{cases}$$

Let $e_1, \cdots e_n \in \langle$*LAZY_BOOLEAN*$\rangle$. *Then,*

$$[\![\ \textit{choose}(e_1, \cdots e_n) = 1]\!]_{\langle \textit{LAZY\_BOOLEAN} \rangle}\sigma = \begin{cases} 1 & : [\![e_1]\!]_{\langle \textit{LAZY\_BOOLEAN} \rangle}\sigma = 1, \\ 0 & : [\![e_j]\!]_{\langle \textit{LAZY\_BOOLEAN} \rangle}\sigma = 1 \textit{ for } i \neq 1, \\ \downarrow & : [\![e_j]\!]_{\langle \textit{LAZY\_BOOLEAN} \rangle}\sigma \textit{ for all } j. \end{cases}$$

See that the semantic of `choose` is a multivalued function.

3.1.2 Commands and Contexts

With the semantics of datatypes `LAZY_BOOLEAN` and `REAL` specified, also the semantics of commands needs to be specified. First, it needs to be declared that the datatype `LAZY_BOOLEAN` does not have a variable; an expression of the datatype `LAZY_BOOLEAN` only appears in a conditional statement.

So far, variables that were considered in chapter 1.3 had only a datatype of `integer`; there was no need to track variable's type. However, in a program that we are willing to verify, variables with different datatypes are used, such as `integer`, `REAL` , `COMPLEX`.

With such a situation, it is natural to define a command that declars a variable with its type. Consider a program that uses variables $v_1, \cdots v_n$ of types $\texttt{Arb}_1, \cdots, \texttt{Arb}_n$. Whereas value assignments of the variables are defined by a state $\sigma : \langle \text{var} \rangle \to \bigcup_i \mathcal{D}(\texttt{Arb}_i)$, type assignments are done by *context* $\pi : \langle \text{var} \rangle \to \bigcup_i \{\texttt{Arb}_i\}$. Hence, the semantic of the command of declaration is to alter the context of a program.

However, including such command induces existence of local variables and exceptions caused by manipulating uninitialized variables. This induced situation is over-complicated to solve problems that we intend to verify. Hence, we simplify a program to have a fixed invariant context; variables and their types never change during the program. A *context invariant program* is a sequence of commands, and variables and their types initially defined. Below is what a context invariant program looks like:

```
1 x : integer , y : integer , z : REAL
2 ═══════
3 C
```

The above part is the invariant context $\pi = \{x, y \mapsto \texttt{integer}, z \mapsto \texttt{REAL}\}$ of the program, whereas the below part is a command of the program. Let a notation $\pi|\texttt{C}$ denotes the above program. Note that the command `C` should only contain variables that are in the domain of the function π.

Now, consider assertions, predicates that define relations between variables. The predicates should contain some information of the types of the variables; e.g., $P = x \in \mathbb{R} \wedge y \in \mathbb{R} \wedge x < y$. However, with a context invariant program, there is no need to write explicitly the types of the variables; for any assertion in a program $\pi|\texttt{C}$, the type declaring part is $\Pi = \wedge_{v \in \text{dom}(\pi)} v \in \mathcal{D}(\pi(v))$.

Whereas the header of a program defines the invariant context, the body is a command. Let $\langle$`command`$\rangle$ denote a set of all commands. A command is constructed inductively as follow:

$$\langle \texttt{command} \rangle \ni \texttt{C} = \begin{cases} x := e \ : \ x \in \text{dom}(\pi),\ e \in \langle \pi(x) \rangle, \\ C_1; C_2 \ : \ C_1, C_2 \in \langle \texttt{command} \rangle, \\ \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \ : \ b \in \langle \texttt{boolean} \rangle,\ C_1, C_2 \in \langle \texttt{command} \rangle, \\ \textbf{while } b \textbf{ do } C_1 \ : \ b \in \langle \texttt{boolean} \rangle,\ C_1 \in \langle \texttt{command} \rangle, \\ \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \ : \ b \in \langle \texttt{LAZY\_BOOLEAN} \rangle,\ C_1, C_2 \in \langle \texttt{command} \rangle, \\ \textbf{while } b \textbf{ do } C_1 \ : \ b \in \langle \texttt{LAZY\_BOOLEAN} \rangle,\ C_1 \in \langle \texttt{command} \rangle. \end{cases}$$

See that every variable that is used in the command is predefined in the invariant context.

3.2 Extended Hoare Logic

In this subchapter, we define inference rules corresponding to the commands' constructors. Whereas a state defines specific assignments of variables, a proposition of pre/postcondition defines a set of possible states. For a proposition P, if P allows a state σ, it is written as $\sigma \models P$; e.g., $\{x \mapsto 4 \in \mathbb{R}, y \mapsto 5 \in \mathbb{R}\} \models (x, y) \in \mathbb{R} \times \mathbb{R} \wedge x < y$. Moreover, let us define a set of all states that satisfy a predicate P as $\Sigma(P) := \{\sigma \in \Sigma \ : \ \sigma \models P\}$.

As partial operations, such as `/` of `REAL` , cannot be omitted from our consideration, expressions that can lead to undefined behavior needs some particular care; assignment such as `z:=x/y` should avoid a state $\sigma(y) = 0$. Generally, for any partially defined expression e, any precondition P of totally correct specification should have $\forall \sigma \in \Sigma, \sigma \in \Sigma(P) \rightarrow [\![e]\!]\sigma \neq \downarrow$. Therefore, the inference rule of assignment should be defined as follow:

Assignment **(AS)**:

$$\frac{\sigma \in \Sigma(P[e/x]) \rightarrow [\![e]\!]\sigma \neq \downarrow}{[P[e/x]] \ x := e \ [P]}$$

See that a command **if** b **then** $C1$ **else** C_2's semantic is not defined for states σ such that $[\![b]\!]_{\langle \texttt{LAZY\_BOOLEAN} \rangle}\sigma \ni \downarrow$. Therefore, we have the following rule:

$$\frac{\left[P \wedge S_1\right] \ C_1 \ \left[Q\right] \qquad \left[P \wedge S_0\right] \ C_2 \ \left[Q\right] \qquad \sigma \in \Sigma(S_i) \leftrightarrow [\![b]\!]_{\langle \texttt{LAZY\_BOOLEAN} \rangle}\sigma \ni i \text{ for } i \in \{1, 0, \downarrow\}}{\left[P \wedge \neg S_\downarrow\right] \quad \textbf{if } b \textbf{ then } C1 \textbf{ else } C_2 \quad \left[Q\right]}$$

The above general inference rule can be specialized when b is in form of $x > y$ or in form of $\texttt{choose}(x_1 > y_1, x_2 > y_2) = 1$:

Conditional with Lazy Inequality **(CLI)**:

$$\frac{\left[P \wedge x > y\right] \ C_1 \ \left[Q\right] \qquad \left[P \wedge x < y\right] \ C_2 \ \left[Q\right]}{\left[P \wedge x \neq y\right] \quad \textbf{if } x > y \textbf{ then } C_1 \textbf{ else } C_2 \quad \left[Q\right]}$$

and *Conditional with Lazy Choose* **(CLC)**:

$$\frac{\left[P \wedge x_1 > y_1\right] \ C_1 \ \left[Q\right] \qquad \left[P \wedge x_2 > y_2\right] \ C_2 \ \left[Q\right]}{\left[P \wedge (x_1 > y_1 \vee x_2 > y_2)\right] \quad \textbf{if } \texttt{choose}(x_1 > y_1, x_2 > y_2) = 1 \textbf{ then } C_1 \textbf{ else } C_2 \quad \left[Q\right]}$$

Similarly, a while command with `LAZY_BOOLEAN` needs a requirement that $[\![b]\!]_{\langle \texttt{LAZY\_BOOLEAN} \rangle}\sigma \not\ni \downarrow$ whenever the condition is needed to be evaluated. Recall the inference rule **(WHILE)**. As a loop invariant is a proposition that is required to hold for every execution of the loop, if the loop invariant ensures $[\![b]\!]_{\langle \texttt{LAZY\_BOOLEAN} \rangle}\sigma \not\ni \downarrow$, then the termination would follow. Hence, the inference rule of such while loop would be as follow:

$$\frac{\begin{array}{l}\sigma \in \Sigma(S_i) \leftrightarrow [\![b]\!]_{\langle \texttt{LAZY\_BOOLEAN} \rangle}\sigma \ni i \text{ for } i \in \{1, 0, \downarrow\} \\ \forall z \in \mathbb{Z} \exists n \in \mathbb{N}^+ : \ \left[I \wedge S_1 \wedge V = z\right] \ C \ \left[I \wedge V = z - n\right], \quad \exists n_0 \in \mathbb{Z} : I \wedge V \leq n_0 \rightarrow S_0 \wedge \neg S_1, \quad I \rightarrow \neg S_\downarrow\end{array}}{\left[I\right] \ \textbf{while } b \textbf{ do } C \ \left[I \wedge S_0\right]}$$

This general inference rule for a while loop also can be specialized:

While with Lazy Inequality **(WLI)**:

$$\frac{\forall z \in \mathbb{Z} \exists n \in \mathbb{N}^+ : \left[I \wedge x > y \wedge V = z\right]\ C\ \left[I \wedge V = z - n\right], \quad \exists n_0 \in \mathbb{Z} : I \wedge V \leq n_0 \rightarrow y < x, \quad I \rightarrow x \neq y}{\left[I\right]\ \textbf{while}\ x > y\ \textbf{do}\ C\ \left[I \wedge y < x\right]}$$

While with Lazy Choose **(WLC)**:

$$\frac{\forall z \in \mathbb{Z} \exists n \in \mathbb{N}^+ : \left[I \wedge x_1 > y_1 \wedge V = z\right]\ C\ \left[I \wedge V = z - n\right], \quad \exists n_0 \in \mathbb{Z} : I \wedge V \leq n_0 \rightarrow x_1 \leq y_1 \quad I \rightarrow (x_1 > y_1 \vee x_2 > y_2)}{\left[I\right]\ \textbf{while}\ \texttt{choose}(x_1 > y_1, x_2 > y_2) = 1\ \textbf{do}\ C\ \left[I \wedge x_2 > y_2\right]}$$

The termination condition is as it was defined in chapter 1.3. However, when we consider a program that finds an existing element of certain value in a list or an array, having a loop variant as the remaining length to the element in the list or an array, the condition $\exists n_0 \in \mathbb{Z}, I \wedge V \leq n_0 \rightarrow S_0$ cannot be satisfied. However, it is obvious that the condition of termination, such that the element will be reached eventually, is satisfied; with an integer loop variant which decreases by 1 in every step, having a promise that $\exists n_0 \in \mathbb{Z}\ :\ ([V = n_0 \rightarrow S_0] \wedge [R \rightarrow V \geq n_0])$, the loop variant will reach n_0 eventually provided that R holds initially. Therefore,

$$\frac{\begin{array}{l}\sigma \in \Sigma(S_i) \leftrightarrow [\![b]\!]_{\langle \texttt{LAZY\_BOOLEAN} \rangle}\sigma \ni i \text{ for } i \in \{1, 0, \downarrow\} \\ \forall z \in \mathbb{Z}\ \left[I \wedge S_1 \wedge V = z\right]\ C\ \left[I \wedge V = z - 1\right] \\ \exists n_0 \in \mathbb{Z}\ :\ (I \wedge R \rightarrow V \geq n_0) \wedge (I \wedge V = n_0 \rightarrow S_0 \wedge \neg S_1), \quad I \rightarrow \neg S_\downarrow\end{array}}{\left[I \wedge R\right]\ \textbf{while}\ b\ \textbf{do}\ C\ \left[I \wedge S_0\right]}$$

The two specialized inference rules induced form this inference rule can be considered similarly:

While with Lazy Inequality and Natural Variant **(WLIN)**:

$$\frac{\begin{array}{l}\forall z \in \mathbb{Z}\ \left[I \wedge x > y \wedge V = z\right]\ C\ \left[I \wedge V = z - 1\right] \\ \exists N_0 \in \mathbb{Z}\ :\ (I \wedge R \rightarrow V \geq n_0) \wedge (I \wedge V = n_0 \rightarrow x < y), \quad I \rightarrow x \neq y\end{array}}{\left[I \wedge R\right]\ \textbf{while}\ x > y\ \textbf{do}\ C\ \left[I \wedge y < x\right]}$$

While with Lazy Choose and Natural Variant **(WLCN)**:

$$\frac{\begin{array}{l}\forall z \in \mathbb{Z}\ \left[I \wedge x_1 > y_1 \wedge V = z\right]\ C\ \left[I \wedge V = z - 1\right] \\ \exists N_0 \in \mathbb{Z}\ :\ (I \wedge R \rightarrow V \geq n_0) \wedge (I \wedge V = n_0 \rightarrow x_1 \leq y_1), \quad I \rightarrow x_1 > y_1 \vee x_2 > y_2\end{array}}{\left[I \wedge R\right]\ \textbf{while}\ \texttt{choose}(x_1 > y_1, x_2 > y_2) = 1\ \textbf{do}\ C\ \left[I \wedge x_2 > y_2\right]}$$

3.2.1 Formal Verification of a Program as an Operator

So far, we have discovered the inference rules for proving correctness of a context invariant program $\pi|\mathtt{C}$; that is, a sequence of commands with variables and types defined initially. Verification of such program is to find how the relations between values of the variables change due to the program, with a proof of termination.

For an example, consider the following program:

```
1  x: REAL, y: REAL, r: REAL
2  ==========
3  C::=
4      if (x>y) then r := x;
5      else r := y;
```

As we have seen in chapter 1.2, the program is well-defined with a promise that $x \neq y$. The fully correct specification is as follow:

$$[x \neq y]\ \pi = \{x, y, r \mapsto \texttt{REAL}\ \}|\texttt{C}\ [r = \max(x, y)].$$

The verification of the above specification would be a sequence of specifications as follow:

0.	$\Pi = x, y, r \in \mathbb{R}$	**(CONTEXT)**
1.	$\big[x > y\big]\ r := x\ \big[x > y \wedge r = x\big]$	**(AS)**
2.	$x > y \wedge r = x \rightarrow r = \max(x, y)$	**(T1)**
3.	$\big[x > y\big]\ r := x\ \big[r = \max(x, y)\big]$	**(CONS, 1,2)**
4.	$\big[x < y\big]\ r := y\ \big[x < y \wedge r = y\big]$	**(AS)**
5.	$x < y \wedge r = y \rightarrow r = \max(x, y)$	**(T2)**
6.	$\big[x < y\big]\ r := x\ \big[r = \max(x, y)\big]$	**(CONS, 4,5)**
7.	$\big[x \neq y\big]$ **if** $x > y$ **then** $r := x$ **else** $r := y\ \big[r = \max(x, y)\big]$	**(CLI, 3,6)**

See that the theorems **(T1)** and **(T2)** are trivial as they coincide with the definition of max.

Therefore, we can have the proved specification of the program $\pi|\texttt{C}$:

$$\big[x \neq y\big]\ \pi|\texttt{C}\ \big[r = \max(x, y)\big]$$

The specification means that given a context π such that three variables x, y, r have the datatype `REAL` , if $x \neq y$, then $r = \max(x, y)$ after the termination of the program.

Though we have suggested a framework of proving such context invariant program, what we commonly expect a program to be is a function: a program that receives and returns some values. Suppose a program of a function named `prog` that receives input $x_1, \cdots x_n$ of types $\texttt{Arb}_1, \cdots, \texttt{Arb}_n$, uses local variables $y_1, \cdots, y_m$ of types $\texttt{Arb}'_1, \cdots, \texttt{Arb}'_n$ and returns a variable $v \in \{x_1, \cdots, y_m\}$ of a type `Arb`, after executing some command `C`. Then, the *function–program* is expressed as follow:

$$\texttt{Arb prog}(\texttt{Arb}_1\ x_1, \cdots, \texttt{Arb}_n\ x_n)\{\texttt{Arb}'_1\ y_1; \cdots \texttt{Arb}'_m\ y_m; \texttt{C};\ \textbf{return}\ v;\}$$

where `C` is a command that has been defined in the previous chapter; the definition lacks of variable declaration; therefore, without using any global variables, the command `C` has an invariant context $\pi = \{x_1 \mapsto \texttt{Arb}_1, \cdots, y_m \mapsto \texttt{Arb}_m\}$. Therefore, a behavior of the body part of the program, with respect to relations between the variables, can be proved by verifying the context invariant program $\pi|\texttt{C}$.

Specification of a function–program is expressed as a Hoare triple $\big[P\big]\ \texttt{Arb prog}(\texttt{Arb}_1\ x_1, \cdots, \texttt{Arb}_n\ x_n)\ \big[Q\big]$, where P is a proposition with free variables $x_1 \in \mathcal{D}(\texttt{Arb}), \cdots x_n \in \mathcal{D}(\texttt{Arb})$ and Q is a proposition of free variables $x_1 \in \mathcal{D}(\texttt{Arb}), \cdots, x_n \in \mathcal{D}(\texttt{Arb}), \gamma \in \mathcal{D}(\texttt{Arb})$; P defines possible states of the inputs and Q defines possible return values γ in terms of those inputs. Similar to a context invariant program, as the context parts of P and Q follow from input variables' and output variable's type, they are omitted when we explicitly write P and Q.

If a specification of an operation is proved, its semantic can be formulated as

$$[\![\texttt{prog}(x_1, \cdots, x_n)]\!]_{\langle \texttt{Arb} \rangle}\sigma = \begin{cases} \alpha\ :\ \sigma \cup \{\gamma \mapsto \alpha\} \in \Sigma(Q) & \text{if } \sigma \in \Sigma(P), \\ \downarrow & \text{otherwise.} \end{cases}$$

where $\sigma \cup \{\gamma \mapsto \alpha\}$ denotes a state whose domain is $\mathrm{dom}(\sigma) \cup \{\gamma\}$ with $\sigma \cup \{\gamma \mapsto \alpha\}(x) = \sigma(x)$ if $x \in \mathrm{dom}(\sigma)$ and $\sigma \cup \{\gamma \mapsto \alpha\}(x) = \alpha$ if $x = \gamma$. Therefore, the `prog` acts as an operation in an expression of `Arb`; the program appearing as an expression in an assignment command can be formulated; e.g., `z:=max(x,y)`.

Now, we need to see verification of a function–program (from now on, it is called just *program*). Let the command part of the program $\pi|\mathtt{C}$. Verification of the context invariant part can be done by the inference rules introduced in the last subchapter. Let $\left[P\right]\ \pi|\mathtt{C}\ \left[Q\right]$, where P and Q are propositions of only input and output variables; if a local variable appears in the proposition P, it is a wrong verification (we assume that a declaration is followed by initialization with an arbitrary value).

Notice that the assertion Q does not give any meaningful relation between input and output values; if input variables change in the program, Q only gives a relation between an output value and the changed input variables. A program being capsulized as a function, a relation between the output variable and input variables with their initial values is needed. See that this can be done by copying values of the input variables:

Observation 8. *With a context* $\pi = \{x_1 \mapsto \mathtt{Arb}_1, x_n \mapsto \mathtt{Arb}_n, \cdots, y_1 \mapsto \mathtt{Arb}'_1, y_m \mapsto \mathtt{Arb}'_m\}$*, if a context invariant program* $\pi \cup \{x'_1 \mapsto \mathtt{Arb}_1, \cdots x'_n \mapsto \mathtt{Arb}_n\}|x'_1 := x_1; \cdots x'_n := x_n; \mathtt{C} =:: \pi'|\mathtt{C}'$ *has a verified specification*

$$\left[P\right]\ \pi'|\mathtt{C}'\ \left[Q\right]$$

where $y_1, \cdots, y_m$ *does not appear in* P*, the program*

$$\pi(v)\ \mathtt{prog}(\pi(x_1)\ x_1, \cdots \pi(x_n)\ x_n)\{\pi(y_1)\ y_1; \cdots\ \pi(y_m)\ y_m;\ \mathtt{C};\ \mathbf{return}\ v;\}$$

satisfies the following specification:

$$\left[P\right]\ \mathtt{prog}(\pi(x_1)\ x_1, \cdots \pi(x_n)\ x_n)\ \left[Q|_{x'_1,\cdots,x'_n,v}[x_1/x'_1, \cdots, x_n/x'_n, \gamma/v]\right]$$

Now, see the following program:

```
1  REAL max(REAL x, REAL y)
2  {
3    REAL r, x', y';
4    x' := x; y' := y;
5    if (x>y) r := x;
6    r := y;
7    return r;
8  }
```

With `C` ::= line 5–6 and $\mathtt{C}_1$ ::= line 4–6, recall that $\left[x \neq y\right]\ \pi|\mathtt{C}\ \left[r = \max(x, y)\right]$, where $\pi = \{x, y, r, x', y' \mapsto \mathtt{REAL}\ \}$; as the variables x', y' do not appear in the command, it is identical to the one we have proved. With an assumption that $x' = x \wedge y' = y$, as $x', y'x, y$ do not get assigned to anything, we have $\left[x \neq y \wedge x' = x \wedge y' = y\right]\ \pi|\mathtt{C}\ \left[r = \max(x, y) \wedge x' = x \wedge y' = y\right]$ from **(CONST)**. From the rule **(AS)**, we have $\left[x \neq y\right]\ \pi|\mathtt{x':=x;\ y':=\ y}\ \left[x \neq y \wedge x' = x \wedge y' = y\right]$. From $r = \max(x, y) \wedge x' = x \wedge y' = y \rightarrow r = \max(x', y')$, we have $\left[x \neq y\right]\ \pi|\mathtt{C}_1\ \left[r = \max(x', y')\right]$. Since $x \neq y|_{x,y} = x \neq y$ and $r = \max(x', y')|_{x',y',r}[x/x', y/y', \gamma/r] \equiv \gamma = \max(x, y)$, we have

$$\left[x \neq y\right]\ \mathtt{REAL\ max(REAL}\ x, \mathtt{REAL}\ y)\{\mathtt{REAL}\ r; \mathtt{C};\ \ \mathbf{return}\ r;\}\ \left[\gamma = \max(x, y)\right]\ .$$

As it was discussed earlier, `max` can be used as an operation of $\langle\mathtt{REAL}\rangle$, the semantic of the operation should be able to be defined as well, according to the proved specification. So called modularization,

consider the program `max` as a binary operation, a constructor of a `REAL` expresesion. Then, the proved specification yields the semantics of the expression:

$$[\![\texttt{max}(e_1, e_2)]\!]_{\langle \texttt{REAL} \rangle}\sigma = \begin{cases} \max([\![e_1]\!]_{\langle \texttt{REAL} \rangle}\sigma, [\![e_2]\!]_{\langle \texttt{REAL} \rangle}\sigma) \in \mathbb{R} & : [\![e_1]\!]_{\langle \texttt{REAL} \rangle}\sigma \neq [\![e_2]\!]_{\langle \texttt{REAL} \rangle}\sigma \\ \downarrow & : \text{otherwise} \end{cases}$$

The above example of verification suggests that not every input variables should be copied; if an input variable x_i does not get assigned in the program, by **(AS)** and **(CONST)**, $x_i = x_i'$ appears in the postcondition. Hence, for input variables that do not get assigned in the program, we do not have to copy them in the beginning.

Consider the following program which is expected to compute an approximation of the maximum value of two real numbers; given two real numbers x, y and an integer n, we expect the program to compute a real number r such that $|r - \max(x, y)| < 2^{-n}$. Hence, the intended specification of such program `max_approx` would be as follow: $\left[\text{True}\right]$ `REAL max_approx(REAL` x`, REAL` y`, integer` n`)` $\left[|\gamma - \max(x, y)| < 2^{-n}\right]$. Suppose the below program has such specification.

```
1  REAL max_approx(REAL x, REAL y, integer n)
2  {
3    REAL epsilon, r;
4    epsilon := power(2,-n);
5    if (choose(abs(x-y) < epsilon, abs(x-y) > epsilon / 2) == 1) then r = x;
6    else if (x>y) then r := x;
7         else r := y;
8    return r;
9  }
```

Let $\texttt{C}_1$ ::= line 4 and $\texttt{C}_2$ ::= line 5–7. With ϵ denoting the variable `epsilon`, the invariant context of both commands are $\pi = \{x, y, r, \epsilon \mapsto \texttt{REAL}\ , n \mapsto \texttt{integer}\}$. With the inference rule of assignment, the specification can be obtained directly:

$$\left[\text{True}\right]\ \pi|\texttt{C}_1\ \left[\epsilon = 2^{-n}\right]$$

The verification for $\texttt{C}_2$ is a sequence of specifications induced by the inference rules and theorems as follow:

0. $\Pi = x, y, r, \epsilon \in \mathbb{R} \wedge n \in \mathbb{Z}$ **(CONTEXT)**
1. $\left[\epsilon > 0 \wedge x > y\right]\ r := x\ \left[\epsilon > 0 \wedge x > y \wedge r = x\right]$ **(AS)**
2. $x > y \wedge r = x \rightarrow r = \max(x, y)$ **(T1)**
3. $\left[\epsilon > 0 \wedge x > y\right]\ r := x\ \left[\epsilon > 0 \wedge r = \max(x, y)\right]$ **(CONS, 1,2)**
4. $\left[\epsilon > 0 \wedge x < y\right]\ r := y\ \left[\epsilon > 0 \wedge x < y \wedge r = y\right]$ **(AS)**
5. $x < y \wedge r = y \rightarrow r = \max(x, y)$ **(T2)**
6. $\left[\epsilon > 0 \wedge x < y\right]\ r := x\ \left[\epsilon > 0 \wedge r = \max(x, y)\right]$ **(CONS, 4,5)**
7. $\left[x \neq y\right]$ **if** $x > y$ **then** $r := x$ **else** $r := y\ \left[r = \max(x, y)\right]$ **(CLI, 3,6)**
8. $\epsilon > 0 \wedge r = \max(x, y) \rightarrow \epsilon > 0 \wedge |r - \max(x, y)| < \epsilon$ **(T3)**
9. $\epsilon > 0 \wedge |x - y| > \dfrac{\epsilon}{2} \rightarrow x \neq y$ **(T4)**
10. $\left[\epsilon > 0 \wedge |x - y| > \dfrac{\epsilon}{2}\right]$ **if** $x > y$ **then** $r := x$ **else** $r := y\ \left[|r - \max(x, y)| < \epsilon\right]$ **(CONSEQ, 7,8,9)**
11. $\left[\epsilon > 0 \wedge |x - y| < \epsilon\right]\ r := x\ \left[e > 0 \wedge |x - y| < e \wedge r = x\right]$ **(AS)**

12. $\epsilon > 0 \wedge |x - y| < \epsilon \wedge r = x \rightarrow |r - \max(x, y)| < \epsilon$ **(T5)**

13. $\left[\epsilon > 0 \wedge |x - y| < \epsilon\right]\ r := x\ \left[\,|r - \max(x, y)| < \epsilon\right]$ **(CONS, 11,12)**

14. $\left[\epsilon > 0 \wedge (|x - y| < \epsilon \vee |x - y| > \frac{\epsilon}{2})\right]\ \texttt{max\_approx}\ \left[\,|r - \max(x, y)| < \epsilon\right]$ **(CLI, 10,13)**

15. $\epsilon > 0 \rightarrow \epsilon > 0 \wedge (|x - y| < \epsilon \vee |x - y| > \frac{\epsilon}{2})$ **(T6)**

16. $\left[\epsilon > 0\right]\ \mathtt{C}_2\ \left[\,|r - \max(x, y)| < \epsilon\right]$ **(CONS, 15,16)**

The theorems can be proved trivially as well. As ϵ and n do not get assigned, assuming $\epsilon = 2^{-n}$, from **(CONST)**, we have

$$\left[2^{-n} > 0\right]\ \pi|\mathtt{C}_2\ \left[\,|r - \max(x, y)| < 2^{-n}\right].$$

Moreover, since $\Pi \rightarrow 2^{-n} > 0$, we have

$$\left[\text{True}\right]\ \pi|\mathtt{C}_1; \mathtt{C}_2\ \left[\,|r - \max(x, y)| < 2^{-n}\right].$$

Therefore, we can conclude the following specification:

$$\begin{gathered}\left[\text{True}\right] \\ \texttt{REAL max\_approx}(\texttt{REAL } x, \texttt{REAL } y) \\ \left[\,|\gamma - \max(x, y)| < 2^{-n}\right].\end{gathered}$$

As it was mentioned in chapter 1.2 and the previous chapter, for `maximum(REAL x, REAL y)` := `limit(max_approx, x, y)`, we have $[\![\texttt{maximum(REAL } e_1, \texttt{REAL } e_2)]\!]\sigma = \max([\![e_1]\!]\sigma, [\![e_2]\!]\sigma)$, totally defined.

3.3 Examples

In this subchapter, verification of two programs are given which use more advanced datatypes; `POLYNOMIAL` and `MATRIX` are used respectively, in the programs `trisection` and `Gaussian_elimination`.

With a datatype `REAL` , a datatype of `POLYNOMIAL` can be introduced as a list of `REAL` . However, how the datatype is actually implemented is not an interesting question; as long as operations of a datatype that we are interested in are exact, the actual implementation does not matter.

The only operation of the datatype `POLYNOMIAL` we are interested in is an evaluation of a polynomial. An operation `REAL eval(POLYNOMIAL ,REAL )` implemented in a natural way is exact as it can be implemented by a composition of arithmetic operations of `REAL` . Hence, we define a datatype `POLYNOMIAL` , whose domain is $\mathbb{R}[x]_{\downarrow}$, which has an exact evaluation operation: $[\![\texttt{eval}(p, x)]\!]\sigma = [\![p]\!]\sigma([\![x]\!]\sigma)$.

A datatype `COMPLEX`, where $\mathcal{D}(\texttt{COMPLEX}) = \mathbb{C}_{\downarrow}$, can be implemented as a tuple of `REAL` . Similarly, implementing arithmetic operations of $\mathbb{C}$ naturally makes the operations exact. Moreover, we define an operator `REAL abs(COMPLEX`c`)` that returns the magnitude of $[\![c]\!]\sigma$.

`MATRIX` is a datatype whose domain is a set $\bigcup_i \{\mathbb{C}^{i\times i}\}_{\downarrow}$; that is, an instance of `MATRIX` describes any finite dimensional complex matrix. We can implement the datatype as d^2 list of `COMPLEX` and row/column operations naturally, where d is the dimension of an instance of the type. Moreover, let `MATRIX submatrix(MATRIX M, integer d, integer j)` returns a square sub matrix of M with its top left and bottom right elements are $M(j, j)$ and $M(d-1, d-1)$, where $d = \dim(M)$. One more operation we would define is an element access; let `COMPLEX element(MATRIX M, integer j)` returns $M(\lfloor \frac{j}{d} \rfloor, j - \lfloor \frac{j}{d} \rfloor)$.

3.3.1 Trisection

We recall a famous theorem in fundamental analysis, intermediate value theorem (IVT); given a continuous function $f : [a, b] \to \mathbb{R}$, if $f(a) < 0$ and $f(b) > 0$ then f has a root in $[a, b]$. We can also restrict the theorem for a real polynomial as a real polynomial is a continuous function. Let us consider a program that refines an interval that contains a unique and odd root of a polynomial; that is, given an interval $[a, b]$, a polynomial p and a positive real number ϵ with a promise that $p(a) < 0 \wedge p(b) > 0$ and p has a single root in the interval, the objective is to refine the interval $[a, b]$ to width less than ϵ with the same properties. With the expected behavior, the specification of such program `trisection` should be

$$\begin{aligned}&\big[p(a) < 0 \wedge p(b) > 0 \wedge \exists!z, (p(z) = 0 \wedge a < z < b) \wedge a < b \wedge \epsilon > 0\big]\\ &\texttt{REAL trisection}(\texttt{REAL } a, \texttt{REAL } b, \texttt{REAL } \epsilon, \texttt{POLYNOMIAL } p)\\ &\big[a \leq \gamma \leq b \wedge \exists!z, p(z) = 0 \wedge \gamma - \epsilon < z < \gamma + \epsilon\big]\ .\end{aligned}$$

The output γ can considered as ϵ–approximation of the root in the initial interval $[a, b]$. Now, we are going to verify that the below program satisfies the specification:

```
1  REAL trisection(REAL a, REAL b, REAL epsilon, POLYNOMIAL p)
2  {
3    while(choose(b-a > epsilon/2, b-a < epsilon) == 1)
4    {
5      if(choose(eval(p,(a+2*b)/3) > 0, eval(p,(2*a+b)/3) < 0) == 1)
6        b = (a+2*b)/3;
7      else
8        a = (2*a+b)/3;
9    }
10   return a;
11 }
```

Let $\texttt{C}_1$ ::= line 5–8 and $\texttt{C}_2$ ::= line 3–9. Then, we need to verify $\pi|\texttt{a}' := \texttt{a};\ \texttt{b}' := \texttt{b};\ \texttt{C}_2;$, where $\pi = \{a, b, a', b', \epsilon \mapsto \texttt{REAL}\ ,\ p \mapsto \texttt{POLYNOMIAL}\ \}$.

0. $\Pi = a, b, a', b', \epsilon \in \mathbb{R} \wedge p \in \mathbb{R}[x]$ **(CONTEXT)**

1. $\big[p(a) < 0 \wedge p(b) > 0 \wedge \exists!z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b' \wedge p(\frac{a+2b}{3}) > 0\big]$

   $b := (a + 2 * b)/3$ **(AS)**

   $\big[p(a) < 0 \wedge p(\frac{3b-a}{2}) > 0 \wedge \exists!z, (p(z) = 0 \wedge a < z < \frac{3b-a}{2}) \wedge a' \leq a < \frac{3b-a}{2} \leq b' \wedge p(b) > 0\big]$

2. $p(a) < 0 \wedge p(\frac{3b-a}{2}) > 0 \wedge \exists!z, (p(z) = 0 \wedge a < z < \frac{3b-a}{2}) \wedge a' \leq a < \frac{3b-a}{2} \leq b' \wedge p(b) > 0$

   $\to p(a) < 0 \wedge p(b) > 0 \wedge \exists!z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b'$ **(T1)**

3. $\big[p(a) < 0 \wedge p(b) > 0 \wedge \exists!z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b' \wedge p(\frac{a+2b}{3}) > 0\big]$

   $b := (a + 2 * b)/3$ **(CONS,1,2)**

   $\big[p(a) < 0 \wedge p(b) > 0 \wedge \exists!z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b'\big]$

4. $\big[p(a) < 0 \wedge p(b) > 0 \wedge \exists!z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b' \wedge p(\frac{2a+b}{3}) < 0\big]$

   $a := (2 * a + b)/3$ **(AS)**

   $\big[p(\frac{3a-b}{2}) < 0 \wedge p(b) > 0 \wedge \exists!z, (p(z) = 0 \wedge a < z < \frac{3a-b}{2}) \wedge a' \leq a < \frac{3a-b}{2} \leq b' \wedge p(a) < 0\big]$

5. $p(\frac{3a-b}{2}) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < \frac{3a-b}{2}) \wedge a' \leq a < \frac{3a-b}{2} \leq b' \wedge p(a) < 0$
$\rightarrow p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b'$ **(T2)**

6. $\left[p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b' \wedge p(\frac{2a+b}{3}) < 0\right]$
$a := (2 * a + b)/3$ **(CONS,4,5)**
$\left[p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b'\right]$

7. $\left[p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b' \wedge (p(\frac{a+2b}{3}) < 0 \vee p(\frac{2a+b}{3}) < 0)\right]$
$\mathtt{C}_1$ **(CLC,4,5)**
$\left[p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b'\right]$

8. $p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b'$ **(T3)**
$\rightarrow p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b' \wedge (p(\frac{a+2b}{3}) < 0 \vee p(\frac{2a+b}{3}) < 0)$

9. $\left[p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b'\right]$
$\mathtt{C}_1$ **(CONS, 7,8)**
$\left[p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b'\right]$

10. $\left[p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b' \wedge \epsilon > 0\right]$
$\mathtt{C}_1$ **(CONST, 9)**
$\left[p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b' \wedge \epsilon > 0\right]$

See that the theorems **T1** and **T2** are direct consequences of Theorem **UIVT** : $\forall (f \in C(\mathbb{R}))(x\ y\ r \in \mathbb{R}), (x < y) \rightarrow (f(x) < 0) \rightarrow (0 < f(y)) \rightarrow (\exists! z \in \mathbb{R}, (x \leq z) \wedge (z \leq y) \wedge f(z) = 0) \rightarrow (x < r) \rightarrow (r < y) \rightarrow (f(r) > 0) \rightarrow \exists! q \in \mathbb{R}, (x \leq q) \wedge (q \leq r) \wedge f(q) = 0$, as $a' \leq a < \frac{3b-a}{2} \leq b' \rightarrow a' \leq a < b \leq b'$ and $a' \leq a < \frac{3a-b}{2} \leq b' \rightarrow a' \leq a < b \leq b'$ can be done by rewriting. The proof of the theorem can be done simply by using axioms of $\mathbb{R}$ and eliminating existential quantifiers.

The theorem **T3** is a direct consequence of Theorem _ : $\forall (f \in C(\mathbb{R}))(x\ y\ x'\ y' \in \mathbb{R}), (x < y) \rightarrow (f(x) < 0) \rightarrow (0 < f(y)) \rightarrow (\exists! z \in \mathbb{R}, (x \leq z) \wedge (z \leq y) \wedge f(z) = 0) \rightarrow (x < x') \rightarrow (x < y') \rightarrow (x' < y) \rightarrow (y' < y) \rightarrow (x' < y') \rightarrow (f(x') < 0) \vee (0 < f(y'))$. The theorem can be proved easily with an axiom of excluded middle.

The above verification proves that $I := p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b' \wedge \epsilon > 0$ is a loop invariant of the command $\mathtt{C}_2$. See that $I \rightarrow \epsilon > 0 \rightarrow b - a > \frac{\epsilon}{2} \vee b - a < \epsilon$ is trivial.

Now, we need to find the loop variant to ensure termination of the loop. Let $V := \lceil \log_{\frac{3}{2}} b - a \rceil$. Then, $\left[\lceil \log_{\frac{3}{2}} b - a \rceil = z\right]\ \mathtt{C}_1\ \left[\lceil \log_{\frac{3}{2}} b - a \rceil = z - 1\right]$ for any z. Also, $\lceil \log_{\frac{3}{2}} b - a \rceil < \log_{\frac{3}{2}} \frac{\epsilon}{2} \rightarrow b - a < \frac{\epsilon}{2}$. Therefore, with the rule **(CLC)**, we have the following specification:

$$\left[p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b' \wedge \epsilon > 0\right]$$
$$\pi | \mathtt{C}_2$$
$$\left[p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b' \wedge \epsilon > 0 \wedge b - a < \epsilon\right]$$

As $\left[p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a < b \wedge \epsilon > 0\right]$ `a':=a; b':=b;` $\left[p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a < b \wedge \epsilon > 0 \wedge a = a' \wedge b = b'\right]$ and $a < b \wedge a = a' \wedge b = b' \rightarrow a' \leq a < b \leq b'$,

we derive the following specification:

$$\begin{aligned}
&\big[p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a < b \wedge \epsilon > 0\big] \\
&\pi|\texttt{a':=a'; b':=b;}\ \mathtt{C}_2 \\
&\big[p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b' \wedge b - a < \epsilon' \wedge \epsilon' > 0\big]
\end{aligned}$$

Moreover, $\exists! z, (p(z) = 0 \wedge a < z < b) \wedge a' \leq a < b \leq b' \wedge b - a < \epsilon \wedge \epsilon > 0 \rightarrow a' \leq a \leq b' \wedge \exists! z, (p(z) = 0 \wedge a - \epsilon < z < a + \epsilon)$. Therefore, we have following specification for the context invariant program:

$$\begin{aligned}
&\big[p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a < b \wedge \epsilon > 0\big] \\
&\pi|\texttt{a':=a'; b':=b;}\ \mathtt{C}_2 \\
&\big[a' \leq a \leq b' \wedge \exists! z, (p(z) = 0 \wedge a - \epsilon < z < a + \epsilon)\big]
\end{aligned}$$

Hence, we complete the verification of the program `trisection`:

$$\begin{aligned}
&\big[p(a) < 0 \wedge p(b) > 0 \wedge \exists! z, (p(z) = 0 \wedge a < z < b) \wedge a < b \wedge \epsilon > 0\big] \\
&\texttt{REAL trisection(REAL}\ a, \texttt{REAL}\ b, \texttt{REAL}\ \epsilon, \texttt{POLYNOMIAL}\ p) \\
&\big[a \leq \gamma \leq b \wedge \exists! z, p(z) = 0 \wedge \gamma - \epsilon < z < \gamma + \epsilon\big]\ .
\end{aligned}$$

3.3.2 Gaussian Elimination

Consider the problem of solving the homogeneous linear system; that is solving the matrix equation $A \cdot x = b$:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}.$$

Observe that row operations on A such as switching rows, scaling rows by nontrivial factor and subtracting one row to another can be done by acting the same row operations on b. Moreover, switching columns of A can be done by switching corresponding rows of x.

For a given rank, $k := \text{rank}(A)$, Gaussian elimination method produces matrices A' and b' where there exists a constructive row operations R and a permutation π such that $RA\Pi = A', Rb = b'$ so that the following linear system holds, where Π is the column permutation matrix corresponding to π.

$$\begin{pmatrix} a'_{11} & a'_{12} & \cdots & a'_{1k} & a'_{1(k+1)} & \cdots & a'_{1n} \\ 0 & a'_{22} & \cdots & a'_{2k} & a'_{2(k+1)} & \cdots & a'_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a'_{kk} & a'_{k(k+1)} & \cdots & a'_{kn} \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \end{pmatrix} \cdot \begin{pmatrix} x_{\pi(1)} \\ x_{\pi(2)} \\ \vdots \\ x_{\pi(n)} \end{pmatrix} = \begin{pmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{pmatrix}.$$

The method proceeds as follow: for each $i = 1, 2, \cdots k$, find a nonzero element among the $(n - i + 1) \times (n - i + 1)$ sized sub-matrix with the top-left element (i, i). Say such element is located at (j, k). Then, permute the kth column and i column. Moreover switch the jth row and ith row so that the

element is now located at (i, i). By subtracting all other rows located below by scaling to make elements located at $(i+1, i), \cdots, (n, i)$ be zero.

Recording all row operations and column permutations yields the solution. See that applying the step for $i = 1, 2, \cdots k$ promises that there always exists nonzero element in the sub matrix; otherwise, the matrix's rank will not be k. See the algebraic algorithm described in algorithm 1.

Algorithm 1. *Algebraic Algorithm of Gaussian Elimination*

1: ***procedure*** GAUSSIAN_ELIMINATION$(n,\ k : \mathbb{N},\ A : \mathbb{C}^{n\times n}, b : \mathbb{C}^n)$
2: $\quad i, j, l, p_i, p_j : \mathbb{N};\ \Pi := \text{identity} : \mathbb{N}^{n\times n};$
3: $\quad$ ***for*** $i := 0 \to k-1$ ***do***
4: $\quad\quad (p_i, p_j) := \text{choose\_pivot}(\text{submatrix}(A, i), n-i)$
5: $\quad\quad A := \text{switchColumn}(\mathrm{A}, \mathrm{p_j}, \mathrm{i})$
6: $\quad\quad \Pi := \Pi \cdot \text{permute}(\mathrm{n}, \mathrm{p_j}, \mathrm{i})$
7: $\quad\quad A := \text{switchRow}(\mathrm{A}, \mathrm{p_i}, \mathrm{k})$
8: $\quad\quad b := \text{switchRow}(\mathrm{b}, \mathrm{p_i}, \mathrm{k})$
9: $\quad\quad$ ***for*** $l := i+1 \to n-1$ ***do***
10: $\quad\quad\quad b(l) := b(l) - (A(l,i)/A(i,i)) \cdot b(i)$
11: $\quad\quad\quad$ ***for*** $m := i \to n-1$ ***do***
12: $\quad\quad\quad\quad A(l,m) := A(l,m) - (A(l,i)/A(i,i)) \cdot A(i,m)$
13: $\quad\quad\quad$ ***end for***
14: $\quad\quad$ ***end for***
15: $\quad$ ***end for***
16: $\quad$ ***return*** A, b, Π
17: ***end procedure***

The operator choose_pivot returns an index (p_i, p_j) such that the corresponding matrix element $A(p_i, p_j)$'s absolute value is positive with $i \leq p_i, p_j \leq n$. The behaviors of the operators swith_column/row are defined in natural sense. $\text{permute}(\mathrm{n}, \mathrm{p_j}, \mathrm{i})$ generates a $n \times n$ permutation matrix that switches p_jth and ith columns, acting on the right–side.

This algebraic computation is known to be correct in many literatures of Linear Algebra. Commonly, choose_pivot is expected to return an index of an element whose absolute value is the maximum; however, it is for numerical stability; if we consider an algebraic computation, where arithmetics can be done exactly in unit time, it does not matter.

With the algebraic computation, assuming an equality test can be established totally, the rank k is actually not needed; testing $A(i, i) = 0$ at the line 9 will detect the singularity of the matrix $\mathbb{C}$; moreover, i in the state is the rank of the matrix A. However, with the realizable computational model, the rank k is necessarily required [17] — and this enrichment is optimal [18].

Now, consider an `iRRAM` program with the types replaced with `integer, REAL, MATRIX`. The operations `switchColumn`, `switchRow` are simply reassigning elements of a `MATRIX` and `permute` is creating an integer matrix. Hence, assuming the operations can be done exactly, correctness of a computation without any `LAZY_BOOLEAN` expression follows exactly from the corresponding algebraic computation.

However, locating a nonzero element requires inequality test of real numbers; hence, `choose_pivot`'s behavior needs verification. First, notice that locating a maximum element is not computable. Hence, we expect `choose_pivot` to locate at least a half maximum element among the sub matrix, for stability.

With the nonzero maximum, the found element won't have zero value. Hence the requirement of

the correctness of choose_pivot follows; `Gaussian_elimination` is correct as long as choose_pivot locates a nonzero element.

A program of choosing a pivot element can be split into two parts: obtaining maximum of the absolute values and locating an element that yields at least half of the maximum value. The program `abs_max` is supposed to receive an instance of `MATRIX` and an integer of the dimension of the matrix and return the maximum of the absolute values of the matrix elements.

```
1  REAL abs_max(MATRIX m, integer d)
2  {
3    integer j;  REAL r;
4    j:=1; r:=abs(element(m,0));
5    while (j<d*d)
6    {
7      r := maximum(r, abs(element(m,j)));
8      j = j + 1;
9    }
10   return r;
11 }
```

Let `C` ::= line 4–9, $\mathtt{C}_1$ ::= line 7–8 and $\mathtt{C}_2$ ::= line 5–9. The invariant context is $\pi = \{m \mapsto \mathtt{MATRIX}, d, j \mapsto \mathtt{integer}, r \mapsto \mathtt{REAL}\}$. The below series of specifications yields a loop invariant.

0. $\Pi = m \in \bigcup_n \{\mathbb{C}^{n\times n}\} \wedge d, j \in \mathbb{Z} \wedge r \in \mathbb{R}$ **(CONTEXT)**

1. $\left[j < d^2 \wedge 1 \le j \wedge d = \dim(m) \wedge \max(r, \left|m(\lfloor\frac{j}{d}\rfloor, j - \lfloor\frac{j}{d}\rfloor)\right|) = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j}{d}\rfloor, j - \lfloor\frac{j}{d}\rfloor)\right|\}\right]$

   `r := maximum(r, abs(element(m,j)))`

   $\left[j < d^2 \wedge 1 \le j \wedge d = \dim(m) \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j}{d}\rfloor, j - \lfloor\frac{j}{d}\rfloor)\right|\}\right]$ **(AS)**

2. $j < d^2 \wedge 1 \le j \wedge d = \dim(m) \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j-1}{d}\rfloor, j-1 - \lfloor\frac{j-1}{d}\rfloor)\right|\}$

   $\rightarrow 1 \le j \wedge j < d^2 \wedge \max(r, \left|m(\lfloor\frac{j}{d}\rfloor, j - \lfloor\frac{j}{d}\rfloor)\right|) = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j}{d}\rfloor, j - \lfloor\frac{j}{d}\rfloor)\right|\}$ **(T1)**

3. $\left[j < d^2 \wedge 1 \le j \wedge d = \dim(m) \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j-1}{d}\rfloor, j-1 - \lfloor\frac{j-1}{d}\rfloor)\right|\}\right]$

   `r := maximum(r, abs(element(m,j)))`

   $\left[j < d^2 \wedge 1 \le j \wedge d = \dim(m) \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j}{d}\rfloor, j - \lfloor\frac{j}{d}\rfloor)\right|\}\right]$ **(CONS,1,2)**

4. $\left[j < d^2 \wedge 1 \le j \wedge d = \dim(m) \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j}{d}\rfloor, j - \lfloor\frac{j}{d}\rfloor)\right|\}\right]$

   `j := j+1`

   $\left[j < d^2 + 1 \wedge 2 \le j \wedge d = \dim(d) \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j-1}{d}\rfloor, j-1 - \lfloor\frac{j-1}{d}\rfloor)\right|\}\right]$ **(AS)**

5. $j < d^2 + 1 \wedge 2 \le j \wedge d = \dim(m) \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j-1}{d}\rfloor, j-1 - \lfloor\frac{j-1}{d}\rfloor)\right|\}$

   $\rightarrow 1 \le j \wedge d = \dim(m) \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j-1}{d}\rfloor, j-1 - \lfloor\frac{j-1}{d}\rfloor)\right|\}$ **(T2)**

6. $\left[j < d^2 \wedge 1 \le j \wedge d = \dim(m) \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j}{d}\rfloor, j - \lfloor\frac{j}{d}\rfloor)\right|\}\right]$

   `j := j+1`

$\left[1 \le j \wedge d = \dim(m) \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j-1}{d}\rfloor, j-1-\lfloor\frac{j-1}{d}\rfloor)\right|\}\right]$ **(CONS,4,5)**

7. $\left[j < d^2 \wedge 1 \le j \wedge d = \dim(m) \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j-1}{d}\rfloor, j-1-\lfloor\frac{j-1}{d}\rfloor)\right|\}\right]$

   $\mathtt{C}_1$

   $\left[1 \le j \wedge d = \dim(m) \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j-1}{d}\rfloor, j-1-\lfloor\frac{j-1}{d}\rfloor)\right|\}\right]$ **(SEQ,3,6)**

T1 follows by the definition of max and **T2** is a direct consequence of a conjunction elimination. See that $I := 1 \le j \wedge d = \dim(m) \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j-1}{d}\rfloor, j-1-\lfloor\frac{j-1}{d}\rfloor)\right|\}$ is the loop invariant of $\mathtt{C}_2$. Let the loop variant $V := d^2 - j$. Then, V decreases by 1, $V = 0 \to j \ge d^2$ and $V > 0 \leftrightarrow j < d^2$. Therefore, we have the following:

$$\left[1 \le j < d^2 \wedge d = \dim(m) \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j-1}{d}\rfloor, j-1-\lfloor\frac{j-1}{d}\rfloor)\right|\}\right]$$

$\mathtt{C}_2$

$$\left[1 \le j \wedge d = \dim(m) \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{d-1}{d}\rfloor, d-1-\lfloor\frac{d-1}{d}\rfloor)\right|\}\right]$$

Applying the assignment rule on commands `j:=1; r:=abs(element(m,0))` yields

$$\left[d = \dim(m)\right] \texttt{ j:=1; r:=abs(element(m,0))} \left[d = \dim(m) \wedge j = 1 \wedge r = |(m(0,0)|\right]$$

As $j = 1 \wedge r = |(m(0,0)| \to 1 \le j < d^2 \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{j-1}{d}\rfloor, j-1-\lfloor\frac{j-1}{d}\rfloor)\right|\}$, we have the following specification:

$$\left[d = \dim(m)\right] \texttt{ j:=1; r:=abs(element(m,0)); } \mathtt{C}_2 \left[1 \le j \wedge r = \mathbf{max}\{|m(0,0)|, \cdots, \left|m(\lfloor\frac{d-1}{d}\rfloor, d-1-\lfloor\frac{d-1}{d}\rfloor)\right|\}\right]$$

Therefore, we can yield the following:

$$\left[d = \dim(m)\right] \texttt{ REAL abs\_max(MATRIX m, integer d) } \left[\gamma = \mathbf{max}\,|m|\right]$$

Now, a program that uses `abs_max` as its subroutine can be devised. The program `choose_pivot` receives a sub matrix and an integer of the dimension of the matrix. The program is expected to output an index of some element whose magnitude exceeds half of the maximum magnitude of the elements of the matrix.

```
1  integer choose_pivot(MATRIX m, integer d)
2  {
3    REAL r;
4    integer j;
5
6    r := abs_max(m,d);
7    j := 0;
8    while(choose(abs(element(m,j)) < r, abs(element(m,j))>r/2) == 1)
9      j := j + 1;
10
11   return j;
12 }
```

Let $\mathtt{C}_1$ ::= line 6–7 and $\mathtt{C}_2$ ::= line 8–9. The invariant context of the commands is $\pi = \{m \mapsto \mathtt{MATRIX}, d, j \mapsto \mathtt{integer}, r \mapsto \mathtt{REAL}\}$. Recall the specification of `abs_max`. As the variable `m` does not get altered, having $m \neq 0$ implies the following specification:

$$\left[d = \dim(m) \wedge m \neq 0\right] \texttt{ r := abs\_max(m,d) } \left[d = \dim(m) \wedge m \neq 0 \wedge r = \mathbf{max}(|m|)\right]$$

As $d = \dim(m) \wedge m \neq 0 \wedge r = \mathbf{max}(|m|) \rightarrow d = \dim(m) \wedge r = \mathbf{max}(|m|) \wedge r > 0$, we have

$$\left[d = \dim(m) \wedge m \neq 0\right] \texttt{ r := abs\_max(m,d); j:=0; } \left[d = \dim(m) \wedge r = \mathbf{max}(|m|) \wedge r > 0 \wedge j = 0\right]$$

It is simple to verify $I := r > 0$ is a loop invariant; r does not vary throughout the loop. Moreover, see that $r > 0 \rightarrow r' > \frac{r}{2} \vee r' < r$ for any real number r'; the safety condition is satisfied.

Let $V := d^2 - j$. See that $\left[I \wedge V = n\right]$ `j:= j+1` $\left[I \wedge V = n - 1\right]$ for any n. As $r = \max(|m|) \rightarrow \exists 0 \leq i < d^2 \ : \ r = \left|m(\lfloor \frac{i}{d} \rfloor, i - \lfloor \frac{i}{d} \rfloor)\right| \rightarrow \exists 0 \leq i < d^2 \ : \ \left|m(\lfloor \frac{i}{d} \rfloor, i - \lfloor \frac{i}{d} \rfloor)\right| \geq r \rightarrow \exists 0 < n_0 \leq d^2 \ : \ d^2 - j = n_0 \rightarrow \left|m(\lfloor \frac{i}{d} \rfloor, i - \lfloor \frac{i}{d} \rfloor)\right| \geq r \wedge j = 0 \rightarrow d^2 - j \geq n_0$. Therefore, with $R := j = 0$, by **(WLCN)**, we have the following:

$$\left[r > 0 \wedge j = 0\right] \ \mathtt{C}_2 \ \left[r > 0 \wedge \left|m(\lfloor \frac{j}{d} \rfloor, j - \lfloor \frac{j}{d} \rfloor)\right| > \frac{r}{2}\right]$$

As r does not get assigned in $\mathtt{C}_2$, by **(CONST)**, we derive the following specification:

$$\left[r = \mathbf{max}(|m|) \wedge r > 0 \wedge j = 0\right] \ \mathtt{C}_2 \ \left[r = \mathbf{max}(|m|) \wedge r > 0 \wedge \left|m(\lfloor \frac{j}{d} \rfloor, j - \lfloor \frac{j}{d} \rfloor)\right| > \frac{r}{2}\right]$$

Since $d = \dim(m) \wedge r = \mathbf{max}(|m|) \wedge r > 0 \wedge j = 0 \rightarrow r = \mathbf{max}(|m|) \wedge r > 0 \wedge j = 0$, by **(CONS)** and **(SQ)**, we have

$$\left[m \neq 0 \wedge d = \dim(m)\right]$$
`integer choose_pivot(MATRIX m, integer d)`
$$\left[\left|m(\lfloor \frac{\gamma}{d} \rfloor, \gamma - \lfloor \frac{\gamma}{d} \rfloor)\right| > \frac{\mathbf{max}(|m|)}{2} > 0\right]$$

Hence, the original `choose_pivot`, in algorithm 1, can be recovered by $(p_i, p_j) := (\lfloor \frac{\gamma}{d} \rfloor, \gamma - \lfloor \frac{\gamma}{d} \rfloor)$.

3.4 Formal Verification in Coq

Verification of a program is a sequence of specifications induced by the inference rules and theorems (implications). As the specifications in the sequence are direct consequence of the inference rules, they are done systematically. However, the theorems, which are subject to be proved, need a proof system, if we want to avoid intuitive proof of correctness.

Coq is a theorem prover that performs a formal proof system (url: http://coq.inria.fr/). Coq is based on the calculus of inductive construction. Hence, a real number which is basically uncountable and not inductive is not a simple object in Coq. Though there were efforts to build constructive real numbers in Coq, such as C-CORN [27], a constructive real number is not what we need; we do not want to compute real numbers in Coq. Instead, Coq provides a library **Reals** which declares real numbers axiomatized; for examples, such as distributive laws and inequality transitivity are axiomatized.

Using this axiomatized type, the theorems that we used for verifying programs in the previous chapter can be proved. For example, the theorem used to prove **(T1)** in the program `trisection` is expressed as follow in Coq:

```
1 Theorem UIVT : forall (f: R->R) (x y r: R) , continuity f -> (x < y)%R -> (f x <0)%R ->
2 (0 < f y)%R -> (exists! z : R, (x<=z)%R /\ (z<=y)%R /\ f z = 0%R) -> (x < r )%R ->
3 (r< y)%R -> (f r >0)%R -> exists! q : R, (x<=q)%R /\ (q<=r)%R /\ f q = 0%R.
```

The theorem **UIVT** can be proved using Coq's `tactics`. However, as it was mentioned, the theorem **(T3)** requires an axiom of excluded middle which Coq does not allow to use by default. However, adding the classic axiom makes the proof possible. Similarly, adding required classic axioms or trivial axioms of real numbers makes the proofs of the theorems used in the previous chapter possible.

Chapter 4. Matrix Diagonalization

In the previous chapter 3, we introduced a framework of formally verifying a program of real computation. In this chapter, we introduce and review algorithms that solve the problem of degenerate matrix diagonalization, which can be used in real computation: totally correct with real numbers and their semantics defined in 1.2. In chapter 4.1.1, we review a simple subdivision algorithm that isolates all distinct roots of a polynomial [19, 26]. In chapter 4.1.2, we suggest a new root isolating method combining `trisection` (see chapter 3.3.1), Gräffe iteration and Newton iteration [22, 23, 25]. In chapter 4.1.3, we improve the root containment predicate introduced in [20, 19] using a promise that all roots are real numbers.

Matrix diagonalization is a computational problem, given a Hermitian matrix H, to compute a unitary matrix U and a diagonal matrix D such that $H = UDU^\dagger$. It can be done by computing the eigenvalues and corresponding eigenvectors of H. It is well known that any Hermitian matrix is unitarily diagonalizable with real eigenvalues. Generally, the problem of matrix diagonalization is not computable; however, it becomes computable when the number of distinct eigenvalues k is given [17]; moreover, it was shown that such discrete enrichment k is optimal [18]. Therefore, the problem can be modified to be computable as follow:

Problem 9. *Given a Hermitian matrix $H \in \mathbb{C}^{d\times d}$ and an integer k with a promise that H has exactly k distinct eigenvalues, compute a unitary matrix U and a diagonal matrix D such that $H = UDU^\dagger$.*

For an operator of a finite dimensional Hilbert space, it is a well-known fact that the eigenvalues coincide with the spectrum of the operator, $\sigma(H) = \{z : H - zI \text{ is not invertible}\}$. Therefore, eigenvalues are precisely the roots of $\det(H - zI)$. It is easy to verify that the determinant is a polynomial of z with degree d, namely, characteristic polynomial.

The *Faddeev-LeVerrier Algorithm* produces the characteristic polynomial's d coefficients using $\mathcal{O}(d^4)$ arithmetic operations by recursively applying the following linear recurrence involving the matrix's powers' traces:

Lemma 10 (Faddeev-LeVerrier Algorithm [28])**.** *For a complex square matrix $A \in \mathbb{C}^{d\times d}$ with characteristic polynomial $\chi_A(z) = z^d + p_1 z^{d-1} + \cdots + p_d \in \mathbb{C}[z]$, it holds*

$$Tr(T^k) \;+\; p_1 \cdot Tr(A^{k-1}) \;+\; \cdots \;+\; p_{k-1} \cdot Tr(A) \;+\; k \cdot p_k \;=\; 0, \qquad 1 \le k \le d. \tag{4.1}$$

Observe that a characteristic polynomial is monic; a Hermitian matrix having real eigenvalues implies that the characteristic polynomial is a real polynomial. Therefore, once the coefficients of the characteristic polynomial are obtained, methods to find/approximate roots of a real polynomial can be applied. The precise description is introduced in the section 4.1.

Suppose λ_i is an eigenvalue of H with multiplicity μ_i. Eigenvectors corresponding to λ_i are vectors in $\{v : \mathbb{C}^n \;:\; Hv = \lambda v\}$. Hence, obtaining independent set of eigenvectors can be done by obtaining bases of the kernel $H - \lambda_i I$. With H being diagonalizable, the kernel has the dimension of μ_i. Hence, applying the program `Gaussian_elimination` of chapter 3.3.2 with $k := d - \mu_i$ yields μ_i independent eigenvectors. Finally, applying the famous Gram-Schmidt algorithm yields orthonormal eigenvectors of the eigenspace.

4.1 Root Finding

In this section, we consider a computational problem of finding the distinct roots of a real polynomial with a promise that all roots are real. The problem is formulated as follow:

Problem 11. *Given the coefficients and the degree of a real polynomial p with a promise of the number of distinct roots k, find (approximate up to any desired precision) all distinct roots of the p with each multiplicity.*

See that the problem can be separated into two sequential subproblems:

Problem 12 (2.1)**.** *Given the coefficients and the degree of a real polynomial p with a promise that p's roots are real and it has precisely k distinct roots, isolate all distinct roots of p; that is, obtain k pairs of open intervals and integers, $((c_i - r_i, c_i + r_i), \mu_i)$ with $c_i, r_i \in \mathbb{Q}_2$ for $i = 1, \cdots, d$, such that each $(c_i - r_i, c_i + r_i)$ and $(c_i - 3 \cdot r_i, c_i + 3 \cdot r_i)$ contains a single root of p with multiplicity μ_i.*

Problem 13 (2.2)**.** *Given the coefficients, the degree of a real polynomial p and a pair of an open interval and an integer $(m - r, m + r), \mu$ with a promise that p's roots are real and $(m - r, m + r)$ and $(m - 3 \cdot r, m + 3 \cdot r)$ both contains a single root of p with multiplicity μ, approximate the root up to any desired precision.*

Suppose there exists a program `root_approx(integer n, POLYNOMIAL p)` which returns all distinct roots of the polynomial p up to precision 2^{-n}. Then, the `limit` operator can make the approximations into `REAL` instances which are exactly roots of the polynomial p.

4.1.1 Root Isolation

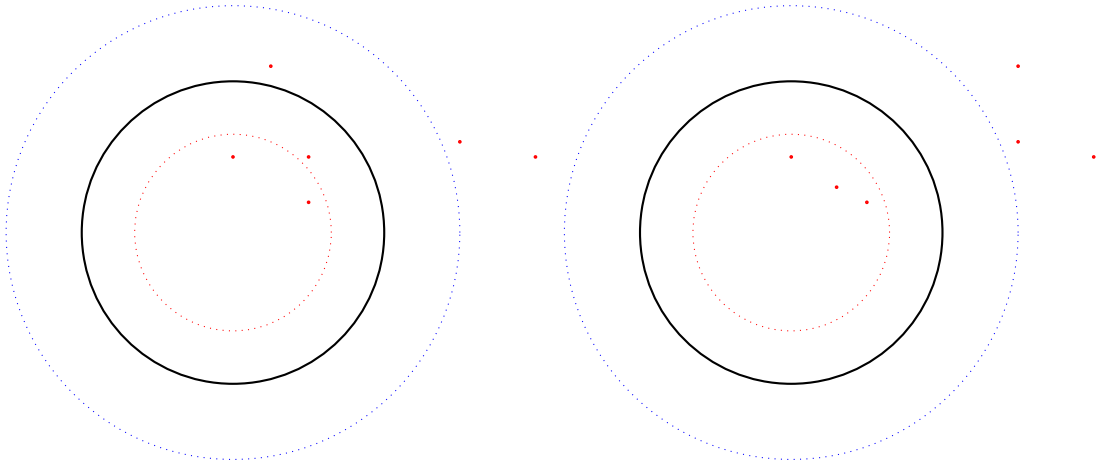
Isolating roots requires a computable predicate that ensures a root containment of a subset of $\mathbb{C}$; given an arbitrary interval, finding the number of roots contained in the interval is not a continuous problem; for example, considering $x^2 + \epsilon$, the number of roots in $(-1, 1)$ is not continuous with respect to the ϵ.

Before dealing with the existence of such predicate, let us assume that there exists such predicate; how such computable predicate is implemented is a different problem. Suppose there exists a computable predicate that receives an open disc on the complex plane with an integer j and tells whether the polynomial contains *precisely* j roots in the disc.

Observe that the problem is essentially discontinuous with regard to the radius of the disc. Therefore, a multivalued variant should be considered, with a notation $D(m, r) := \{z \in \mathbb{C} \ : \ |m - z| < r\}$ denoting an open disc and $r' \in \mathbb{R}$, $r' \cdot D(m, r) := D(m, r' \cdot r)$:

Definition 14. *If $T^{(\ell)}(p, m, r, j) =$ `True`, then $D(m, r)$ contains exactly j number of roots of p counted with multiplicity. If $\ell \cdot D(m, r)$ and $\overline{\ell^{-1} \cdot D(m, r)}$ both contain exactly j number of roots counted with multiplicity, then $T^{(\ell)}(p, m, r, j)$ holds. For $j = 0$, $T^{(\ell)}(p, m, r, 0)$ holds if $\ell \cdot D(m, r)$ contains no root.*

The multivalued behavior of the predicate $T^{(\ell)}$ is illustrated in figure 4.1: The outer dotted line is a disc of $\ell \cdot D_{1,2}$ and the inner dotted line is a disc of $\ell^{-1} \cdot D_{1,2}$. Observe that $T^{(\ell)}(p, m_1, r_1, 3)$ can be either `True` or `False`; though the disc contains exactly 3 roots, as $\ell \cdot D_1 \cap \overline{\ell^{-1} \cdot D_1}^c$ contains roots, the result of the predicate is not determined. However, as $\ell \cdot D_2$ and $\overline{\ell \cdot D_2}$ contains both the three roots, $T^{(\ell)}(p, m_2, r_2, 3)$ is determined to be `True`.



(a) $D_1 = D(m_1, r_1)$ (b) $D_2 = D(m_2, r_2)$

Figure 4.1: Multivalued semantic of the predicate $T^{(\ell)}$

For simplicity, if a disc $D := D(m, r)$ is defined explicitly, the predicate would be expressed as $T^{(\ell)}(p, D, j)$ instead of $T^{(\ell)}(p, m, r, j)$.

The predicate can be applied to devise a multivalued function that shows how many number of roots that a disc contains, simply by applying the predicate for all j: $T^{(\ell)}_*(p, m, r) : \{-1, 0, 1, \cdots d\}$ is a multivalued function such that $T^{(\ell)}_*(p, m, r) = j$ implies $D(m, r)$ contains exactly j number of roots. If $\ell \cdot D(m, r)$ and $\overline{\ell^{-1} \cdot D(m, r)}$ both contain exactly j number of roots, then $T^{(\ell)}(p, m, r) = j$. $T^{(\ell)}(p, m, r) = -1$ implies that $T^{(\ell)}(p, m, r, j)$ has failed for all j.

With the predicate, a simple subdivision algorithm can be naturally considered. For any given polynomial, there exists some computable upper bounds of the magnitude of roots of the polynomial. Here, two of them are introduced:

1. (Rouché Bound): For a polynomial $p(x) := a_d x^d + \cdots + a_1 x + a_0$, all roots of the polynomial are contained in a disc of radius $\max(1, \frac{\sum_{i=0}^{d-1} |a_i|}{|a_d|})$, centered at the origin.
2. (Bound by $T^{(\ell)}$): For $D_i := D(m, 2^i)$, compute the predicate $T^{(\ell)}(p, D, d)$ with increasing i. By the definition of the predicate's behavior, this yields a disc that contains all roots of p with at most 2ℓ times larger radius compared to the least upper bound.

Hence, the subdivision algorithm, starting with a root bound as above, subdivides a subset on the complex plane with testing for a root containment. Although the predicate is defined over a *disc*, a disc is not a natural entity to run an iterative subdivision. Hence, a *box* is used as an entity of the subdivision, where a box $B(m, w) := \{z \in \mathbb{C} : |\mathrm{Re}(z - m)|, |\mathrm{Im}(z - m)| \leq \frac{w}{2}\}$ is a closed square box of width w centered at m and a subdivision of the box is a set of boxes $\{B(m - \frac{r}{2}(1 + i), \frac{r}{2}), B(m - \frac{r}{2}(-1 + i), \frac{r}{2}), B(m - \frac{r}{2}(-1 - i), \frac{r}{2}), B(m - \frac{r}{2}(1 - i), \frac{r}{2})\}$.

During each step of a box subdivision, each box is transformed into a disc. If the function ensures that the disc contains no root, the box is discarded. However, if the function cannot exclude the disc, the the box is subdivided, until all k disjoint boxes that contains some root of the polynomial are found.

The disc transformation should consider two criteria: (1) there should be no box that get discarded wrongly, that is, the disc transformation should contain the box, (2) the subdivision should eventually terminate, that is, the disc D's inner boundary $\overline{\ell^{-1} \cdot D}$ should contain the box. Then, the disc transformation can be defined as follow:

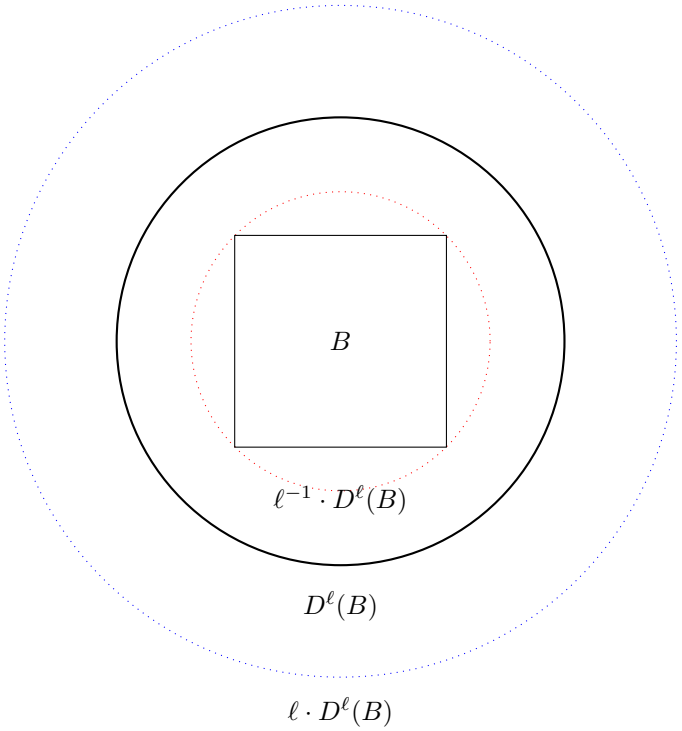


Figure 4.2: (ℓ) disc transformation of a box.

Definition 15. *Given a box $B(m, w)$, we define an indexed disc transformation $D^{(\ell)}(B(m,r)) := D(m, \frac{\sqrt{2}\ell w}{2})$.*

See that in the definition, $D^{(\ell)}(B(m,r))$ is the smallest disc such that $\ell^{-1} \cdot D^{(\ell)}(B(m,r))$ contains $B(m,r)$. This can be seen in figure 4.2. Moreover, the fact leads the subdivision to reach a condition to separate the roots with the predicate holds eventually, as it can be seen in figure 4.3; with the definition of the disc transformation, the inner dotted line contains the box. Hence, the box subdivision leads to the sufficient condition for the predicate.

Now, let us consider a subdivision tree, a tree of boxes where a child boxes are generated by a subdivision of the parent box. Consider a polynomial p which has the minimum root spacing ϵ and a subdivision tree with its root box of the width W_0. Then, the following lemma can be obtained:

Lemma 16. *1. $T^{(\ell)}$ holds for each distinct root at depth at least $\log_2 \frac{1}{\epsilon} + 2\log_2 \ell + \log_2 W_0 + 1$,*

2. For at depth d, there are at most $k\left(\sqrt{2}\ell^2 + 2\right)^2$ nodes that the exclusion test does not hold.

Proof.

- For a B that contain a root z, the predicate holds if $\ell \cdot D^{(\ell)}(B)$ contains z only. That is, if $\ell^2 \frac{r_d}{\sqrt{2}} - w_d \leq \epsilon$, where w_d denotes the width of a box in the depth d. This implies that if $d \geq \lceil \log_2 \frac{1}{\epsilon} + \log_2 W_0 + 2\log_2 \ell \rceil$, the predicate holds.
- For a depth d, we count the number of boxes that intersect with any $\ell \cdot D^{(\ell)}(B)$. See that for a root in a box, the other boxes that does not intersect with the $\ell \cdot D^{(\ell)}(B)$ the exclusion test does not get affected by the box. The number of such boxes are at most $\left(\lceil\sqrt{2}\ell^2\rceil + 1\right)^2$. As there are k roots, the number of such boxes are bounded by $k\left(\lceil\sqrt{2}\ell^2\rceil + 1\right)^2$.

□

With the lemma, we can introduce the following naive subdivision algorithm:

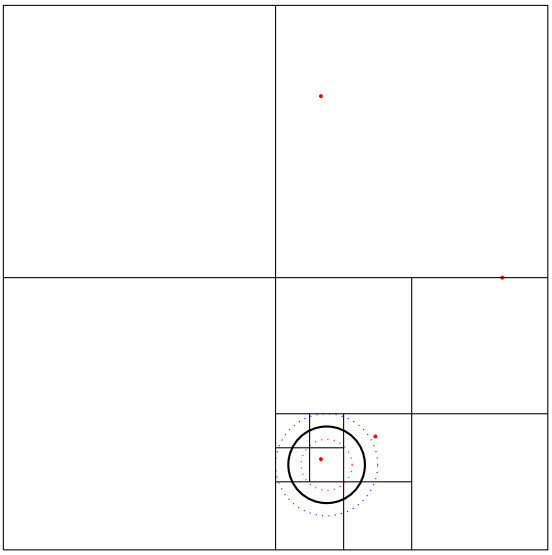


Figure 4.3: Box subdivision toward a root isolation

Algorithm 2. *Construct a subdivision tree $\mathcal{T}$: expand a node $B(m,w)$ (that means to subdivide the box) if $T^{(\ell)}(p, D^\ell(B), 0)$ does not hold. If $T^{(\ell)}(p, D^\ell(B), j) \wedge T^{(\ell)}(p, 3 \cdot D^l(B), j)$ holds for some j, insert $D^\ell(B)$ into a solution queue. The construction is done when there are k disjoint discs in the solution queue. Return the disjoint discs.*

See that we test for both $T^{(\ell)}(p, D^\ell(B), j) \wedge T^{(\ell)}(p, 3 \cdot D^l(B), j)$. With lemma 16, we can immediately prove the correctness of the algorithm:

Theorem 17. *Algorithm 2 is correct.*

Proof.

- The termination follows from the previous lemma.
- Disjoint discs in the solution queue contains distinct roots of p and each of them separates roots. Therefore, when the construction terminates, when there exists k disjoint discs in the solution queue, they are the discs separate all distinct root of the polynomial .

□

4.1.2 Root approximation

We are given a polynomial p with a promise that all the roots of p are real. Moreover, $(c-r, c+r)$ and $(c-3\cdot r, c+3\cdot r)$ contain a single root of multiplicity μ. For any desired $\epsilon > 0$, we need to compute an interval $(c'-\epsilon, c'+\epsilon) \subseteq (c-r, c+r)$ that contains the root.

One of the fastest method that is known is Newton iteration that can be applied recursively and yields quadratic convergence:

Lemma 18 (Newton iteration [25, Lemma 6])**.** *Suppose $B(m,w)$, $D^{(4/3)}(B(m,w))$ and $3 \cdot D^{(4/3)}(B)$ contains exactly k roots. With $N := 4$ initially, choose $x := m+w$. If $f'(x) \neq 0$, compute $x' := x - k\frac{p(x)}{p'(x)}$. Let $N := N^2$ and return $B(x', \frac{w}{4N})$ if the box contains all roots of $B(m,w)$. `Newton` iterates the process. If $D^{(4/3)}(B)$ and $2^{22} \cdot d^2 \cdot D^{(4/3)}(B)$ contains the same roots, Newton step succeeds.*

Recall that we already introduced a root refinement method which is verified to be correct: `trisection` in chapter 3.3.1; however the method only yields linear convergence. Hence, the most natural strategy

is to apply the linear convergence method until the sufficient condition for the quadratic convergence is established.

However, the precondition of `trisection` requires (i) the input function to be a strictly increasing function and (ii) the interval contains only one single root of the function. Notice that this precondition cannot always be satisfied; suppose μ is an even integer, then the precondition (i) is not satisfied. Hence, to feed our problem into `trisection`, we need an intermediate process to match the condition.

Suppose μ is an even integer. Then, we can always take the derivative to make the multiplicity of the root odd. However unintended root of the first derivative may appear in the interval, which violates the precondition (ii). There exists a relation between the spacing between two adjacent roots and the spacing between each root to the closest root of the first derivative. Hence, it is expected that separating far enough the two adjacent roots will yield the appropriate precondition.

There exists a method that squares roots of the polynomial.

Lemma 19 (Dandelin-Lobachesky-Gräffe [22])**.** *For a polynomial $p = a_d x^d + \cdots a_1 x + a_0$, let*

$$p_e(x) := a_{2\lfloor \frac{d}{2} \rfloor} x^{\lfloor \frac{d}{2} \rfloor} + a_{2\lfloor \frac{d}{2} \rfloor - 2} x^{\lfloor \frac{d}{2} \rfloor - 1} + \cdots + a_2 x + a_0,$$
$$p_o(x) := a_{2\lfloor \frac{d-1}{2} \rfloor + 1} x^{\lfloor \frac{d-1}{2} \rfloor} + a_{2\lfloor \frac{d-1}{2} \rfloor - 1} x^{\lfloor \frac{d-1}{2} \rfloor - 1} + \cdots + a_3 x + a_1.$$

Define a first iteration as $G_1(p)(x) := (-1)^d (p_e(x)^2 - x \cdot p_o(x)^2)$ and the Nth iteration recursively: $G_N(p)(x) := G_1(G_{N-1}(p))(x)$ for $N > 1$. If $p(x) = \prod_{i=1}^d (x - x_i)$, then $G_N(p)(x) = \prod_{i=1}^d (x - x_i^{2^N})$.

The iteration of squaring roots of a polynomial can be used to appropriately separate two adjacent roots:

Lemma 20. *Suppose $(c-r, c+r)$ and $(c-\rho\cdot r, c+\rho\cdot r)$ contain precisely one root of p with multiplicity μ. Then, $G_N(p_{c,r})'$ contains $\mu - 1$ roots in $[0, 1)$ and also in $[0, \rho')$, where $N := \max(1, \lceil \log_2 \log_\rho(\rho' d/\mu) \rceil)$.*

Proof. Since each Gräffe iteration squares the location of the roots, $q(x) = (x - x_0^M)^\mu \cdot (x - x_{\mu+1}^M) \cdots (x - x_d^M)$ for even $M = 2^N$. Therefore $q'(x^*) = 0 \neq q(x^*)$ implies

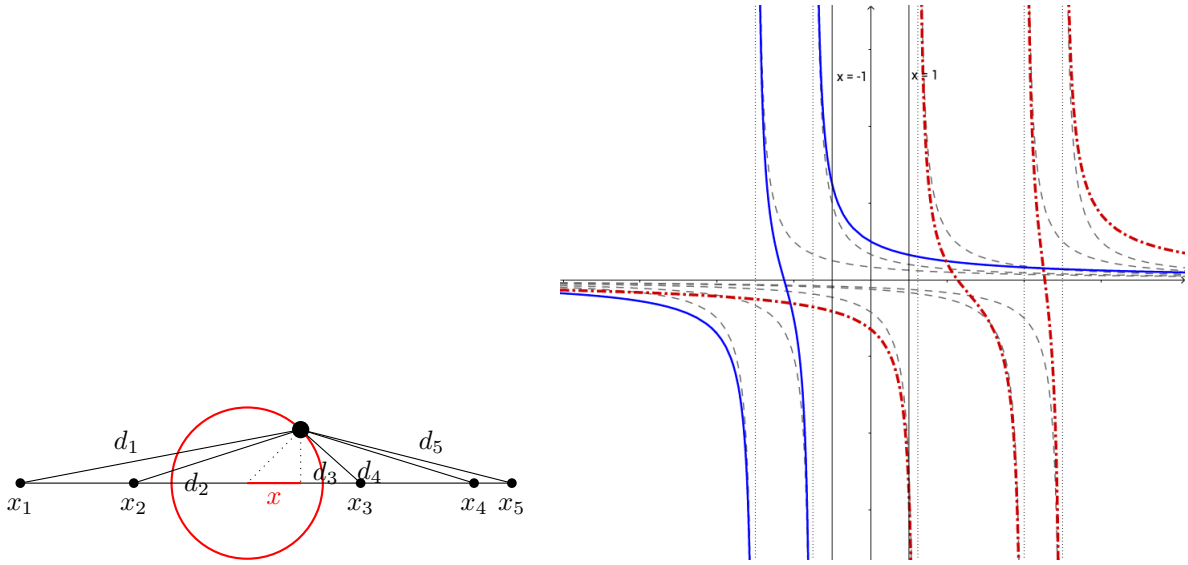
$$0 \;=\; \frac{q'(x^*)}{q(x^*)} \;=\; \frac{\mu}{x^* - x_0^M} \;+\; \sum\nolimits_{j=\mu+1}^{d} \frac{1}{x^* - x_j^M} \;\geq\; \frac{\mu}{x^*} \;+\; \frac{d-\mu}{x^* - \rho^M}$$

for $x_0 < x^* < x_{\mu+1}$, and thus $x^* \geq \frac{\mu}{d} \cdot \rho^M \geq \rho'$ by choice of M, which is $N := \max(1, \lceil \log_2 \log_\rho(\rho' d/\mu) \rceil)$. □

Recall that $G_N(p)$ is a polynomial that squares roots of p N times. Therefore, if z is a root of $G_N(p_{m,r})$, one of $m \pm r \cdot z^{\frac{1}{2^N}}$ is the root of p. By the above lemma, $G_N(p_{m,r})'$ satisfies the precondition of `trisection`. Hence, applying `trisection` until we reach the sufficient condition for the `Newton`, then applying `Newton` yields the root of $G_N(p_{m,r})$.

Once we have computed the root z of $G_N(p_{m,r})$, we can compute $z_+ := m + r \cdot z^{\frac{1}{2^N}}$ and $z_- := m - r \cdot z^{\frac{1}{2^N}}$. See that deciding whether z_+ or z_- is a root of p is not computable as $z_+ = z_-$ may happen when $z = 0$. However, as we dealt the problem of max, an approximation of the actual root can be computed as follow:

Algorithm 3. *Compute $t :=$ `choose`$(|z_+ - z_-| < \epsilon, |z_+ - z_-| > \frac{\epsilon}{2})$. If $t = 1$, then both z_+ and z_- are ϵ–approximation of the root. Hence, return z_+. If $t = 2$, then we know $z_+ \neq z_-$. Therefore, compute $t' :=$ `choose`$(|p(z_+)| > 0, |p(z_+)| > 0)$. If $t' = 1$, then return z_+. Otherwise return z_-.*



(a) $h(x) := \prod d_i^2 = \prod(x_i^2 - 2xx_i + 1)$ (b) h_l: the thick solid blue line, $-h_r$: the thick dotted red line

Figure 4.4: Proof of lemma 23.

4.1.3 Root Containment Predicate

For the previous subchapters, existence of the predicate $T^{(\ell)}$ was assumed. In this subchapter, realizations of the predicate is introduced.

A polynomial being an entire holomorphic function, the Roucé's theorem can be considered. The Roucé's theorem offers a relation between roots of two holomorphic functions:

Theorem 21 (Symmetric Rouché's Theorem [29])**.** *Let $f, g \in H(\Omega)$ and $K \subset \Omega$ be a bounded region with continuous boundary. If $|f - g| < |f| + |g|$ on ∂K, then f and g have the same number of roots in K.*

Setting a region to be a unit disc that has its center at the origin, the Rouché's theorem can be applied as follow:

Lemma 22 (Pellet's Test [20])**.** *A polynomial $p = \sum_{i=0}^{d} a_i x^i$ has k roots in a unit disc centered at origin counted with multiplicities if $|a_k| > \sum_{i \neq k} |a_i|$.*

Proof. Let $f := p$ and $g := a_k x^k$. Then, on the unit ring, $|f - g| \leq \sum_{i \neq k} |a_i| < |a_k| \leq |g| + |f|$. Therefore, p has exactly k roots counted with multiplicities. □

Pellet's test is a predicate induced from the lemma such that

$$C(a_d x^d + \cdots a_1 x + a_0, k) := |a_k| > \sum_{i \neq k} |a_i|$$

See that if the predicate $C(p, k)$ holds, then the polynomial has exactly k roots in the unit disc centered at the origin, counted with multiplicity.

Pellet's test can be strengthened when it is promised that all roots of a polynomial are located on the real axis. The point is to find out the minimum value of $|p|$ on the unit ring centered at the origin.

Lemma 23. *Let $\{x_i\}_{i=1,\cdots,d}$ be arbitrary d points on the real line. The product of the distances from each point to a point on a ring centered at the origin is minimized at one of the two intersecting points; i.e., a function $f(x,y) := \prod_{i=1}^{d} \sqrt{(x-x_i)^2+y^2}$ subject to $x^2+y^2=1$ is minimized when $y=0$ and $x=-1$ or 1.*

Proof. Let $d_i := \sqrt{(x-x_i)^2+y^2}$. Then, $d_i^2 = x_i^2 - 2xx_i + 1$. Let $h(x) := \prod_{i=1}^{d} d_i^2 = \prod_{i=1}^{d} \left(x_i^2 - 2xx_i + 1\right)$; see figure 4.4 (a). See that if $x_i = \pm 1$ for some i, f is minimized to 0 when $x = \pm 1$. If $x_i = 0$ for some i, then $d_i = 1$ for any x. Hence, we may exclude the cases of $x_i = -1, 0$, or 1. Then, $h(x) > 0$ for $x \in [-1,1]$. Therefore, $h'(x)$ shares its roots with $h'(x)/h(x)$. See that

$$\frac{h'(x)}{h(x)} = \sum_{i=1}^{d} \frac{-2x_i}{x_i^2 - 2xx_i + 1} = \sum_{i=1}^{d} \frac{1}{x - (\frac{x_i}{2} + \frac{1}{2x_i})}. \tag{4.2}$$

As $x_i \neq 0$, the above expression is well-defined. Moreover, it is easy to verify that $\left|\frac{x_i}{2} + \frac{1}{2x_i}\right| > 1$.

Let J be a set of indices such that $\frac{x_i}{2} + \frac{1}{2x_i} < -1$. Define $h_l(x) := \sum_{i\in J} \frac{1}{x-(\frac{x_i}{2}+\frac{1}{2x_i})}$ and $h_r(x) := -\sum_{i\notin J} \frac{1}{x-(\frac{x_i}{2}+\frac{1}{2x_i})}$. See that h_l is strictly decreasing in $[-1,1]$ and h_r is strictly increasing in $[-1,1]$. Hence, $h_l(x) = h_r(x)$ has at most one root in $[-1,1]$ which implies that $h'(x)$ has at most one root in $[-1,1]$; see figure 4.4 (b)

Suppose $h'(x)$ does not have any root in $(-1,1)$. Then, clearly, minimum and maximum of $h(x)$ is attained at the two endpoints. Otherwise, if $h'(x') = 0$ for some $x' \in (-1,1)$, by the strictly increasing/decreasing behavior of h_l and h_r, $h'(x) > 0$ for $-1 \leq x < x'$ and $h'(x) < 0$ for $1 \geq x > x'$. Therefore $h(x')$ is the maximum value of $h(x)$ in $[-1,1]$ and either $h(-1)$ or $h(1)$ is the minimum value of $h(x)$ in $[-1,1]$. □

Lemma 23 can be applied into the proof of Pellet's test as follow:

Lemma 24. *Suppose $p = \sum_{i=0}^{d} a_i x^i$ has real roots only. The polynomial p has exactly k roots in $(-1,1)$ counted with multiplicities if $|a_k| + \min(|p(-1)|, |p(1)|) > \sum_{i\neq k} |a_i|$.*

Proof. With the previous lemma, we have that $\left|p(e^{i\theta})\right|$ attains its minimum at either $\theta = 0$ or π. Applying this inequality into the proof of lemma 22 yields the result. □

Let the $C_{\mathbb{R}}$ be a predicate induced by lemma 24, real Pellet's test, such that

$$C_{\mathbb{R}}(p := a_d x^d + \cdots + a_1 x + a_0, k) := |a_k| + \min(|p(-1)|, |p(1)|) > \sum_{i\neq k} |a_i|.$$

where p is required to have only real roots. If the predicate $C_{\mathbb{R}}(p := a_d x^d + \cdots + a_1 x + a_0, k)$ holds, then the unit disc centered at the origin contains k roots of p, counted with multiplicity.

One critical point that needs to be considered is that the predicate $C_{\mathbb{R}}$ (nor C) is not computable; for the case $|a_k| = \sum_{i\neq k} |a_i| + \min(|p(-1)|, |p(1)|)$, a program that implements the predicate with the datatype `REAL` will freeze. Therefore, a computable variant of the predicate is necessary:

$$\tilde{C}(p,k) = \begin{cases} \text{True} & : |a_k| > \sum_{i\neq k} |a_i| + \min(|p(-1)|, |p(1)|), \\ \text{False} & : |a_k| < \frac{3}{2} \sum_{i\neq k} |a_i| \end{cases}$$

$$\tilde{C}_{\mathbb{R}}(p,k) = \begin{cases} \text{True} & : |a_k| > \sum_{i\neq k} |a_i| + \min(|p(-1)|, |p(1)|), \\ \text{False} & : |a_k| < \frac{3}{2} \sum_{i\neq k} |a_i| + \min(|p(-1)|, |p(1)|). \end{cases}$$

This multivalued functions are totally defined, if p is not a zero function. As we are not considering a constant function, the predicates $\tilde{C}, \tilde{C}_{\mathbb{R}}$ are totally defined.

Observing the one–way implication of the predicate, C.Yap et al., has discovered the sufficient condition for the predicate:

Theorem 25. [26, Theorem 2] *If $D(0, \frac{1}{11d})$ and $D(0, 18 \cdot d^3)$ both contain k roots of p counted with multiplicity, then $|a_k| > \frac{3}{2}\sum_{i \neq k} |a_i|$. Therefore, $\tilde{C}(p, k) = True$.*

This condition can be directly applied to the predicate $\tilde{C}_{\mathbb{R}}$:

Corollary 26. *If $D(-\frac{1}{11d}, \frac{1}{11d})$ and $D(-18 \cdot d^3, 18 \cdot d^3)$ both contain k roots of p counted with multiplicity, then $|a_k| > \frac{3}{2}\sum_{i \neq k} |a_i|$. Therefore, $\tilde{C}_{\mathbb{R}}(p, k) = True$.*

It is obvious that if the predicate $\tilde{C}$ holds, then $\tilde{C}_{\mathbb{R}}$ holds. However, the order of improvement is not discovered yet. See that the predicate $\tilde{C}$ requires $O(d^4)$ of separation between contained roots and excluded roots.

Recall Gräffe iteration introduced in lemma 19. The iteration squares roots of a polynomial. If the roots inside the unit disc centered at the origin are considered, applying Gräffe iteration will attract those roots to the origin and repel the roots outside the disc. Hence, applying Gräffe iteration sufficient number of times will yield the sufficient condition of the predicate mentioned in theorem 25:

Theorem 27. [25, Lemma 3] *Let $N := \lceil \log(1 + \log n) \rceil + 5$. If $\tilde{C}(G_N(p), k) = True$, then p has k roots in $D(0, 1)$. If $D(0, \frac{2\sqrt{2}}{3})$ and $D(0, \frac{4}{3})$ both contain k roots of p counted with multiplicity, then $\tilde{C}(G_N(p), k) = True$.*

This theorem can directly applied to the predicate $\tilde{C}_{\mathbb{R}}$:

Corollary 28. *Let $N := \lceil \log(1 + \log n) \rceil + 5$ and p be a polynomial whose roots are real numbers. If $\tilde{C}_{\mathbb{R}}(G_N(p), k) = True$, then p has k roots in $(-1, 1)$. If $(-\frac{2\sqrt{2}}{3}, \frac{2\sqrt{2}}{3})$ and $D(-\frac{4}{3}, \frac{4}{3})$ both contain k roots of p counted with multiplicity, then $\tilde{C}_{\mathbb{R}}(G_N(p), k) = True$.*

Moreover, applying the iteration sufficient times, the separation can be achieved to be any fixed $\ell > 1$; this is the reason why the predicate in chapter 4.1.1 was parameterized into an arbitrary real number ℓ.

Now, suppose one wants to check a root containment of an arbitrary disc $D(m, r)$. Then, it is always possible to translate and delay the the polynomial $p \mapsto p_{m,r}$ so that the interested disc becomes the unit disc centered at the origin. Hence, $\tilde{C}(p, m, r, k) := \tilde{C}(p_{m,r})$ and same for $\tilde{C}_{\mathbb{R}}$.

Chapter 5. Experimental Results

5.1 Evaluation

In this chapter, we offer experimental results restricting the problem to be real symmetric matrix diagonalization and with $\ell = \frac{4}{3}$ (from the previous chapter, as in [25]). Observe that with the possible combinations of algorithms for the subproblems, there can be several algorithms for the problem of matrix diagonalization. The problem of root finding has been separated into two subproblems: root isolating and root approximating. Though it seems inefficient, with the root isolating problem, it is possible to use the algorithm to approximate the roots; after isolating all roots by algorithm 2, the algorithm then can be modified to run until the discs have radius less than any positive ϵ.

Let algorithm (i) be an algorithm induced from the recursive relation in lemma 10. The algorithm (i) receives a Hermitian matrix and returns a real polynomial which is the characteristic polynomial of the matrix. Let algorithm (ii) be algorithm 2. Let algorithm (iii) be an algorithm of Gräffe iteration + `trisection` + algorithm 3 (introduced in chapter 4.1.2). Let algorithm (iv) be an algorithm induced from lemma 18. Let algorithm (v) be the Gaussian elimination algorithm whose precise description was introduced in chapter 3.3.2.

Moreover, let algorithm (vi) be the root isolating algorithm introduced in [26]. The algorithm basically approximates roots by using *component* subdivision. A component is a set of adjacent boxes on the complex plane. The precise description can be found in [25, 26]. The algorithms are known near-optimal, with the number of boxes that the algorithm produces bounded by $O(k \log\log \frac{W_0}{\epsilon} + k \log d)$ [26, Lemma 9]; it uses the algorithm (iv) within the process of component subdivision with checking correctness of the result; recall that the algorithm (ii) produces $O(k \cdot \log \frac{W_0}{\epsilon} + k)$. The tradeoff of those $\log\log$ factor of the root spacing will be shown in chapter 5.

Now, the algorithms that we are interested in are defined as follow:

- Algorithm (a) is composed of algorithms (i) + (ii) + (v)
- Algorithm (b) is composed of algorithms (i) + (ii) + (iii) + (v)
- Algorithm (c) is composed of algorithms (i) + (vi) + (v)
- Algorithm (d) is composed of algorithms (i) + (ii) + (iii) + (iv) + (v)

The four algorithms (a)–(d) are implemented in `C++` with `iRRAM` library (see chapter 1.2) in its version from December 27 (2016) available at https://github.com/fbrausse/iRRAM and in turn building on `MPFR` and `GMP`.

Experiments were conducted on several identical virtual machines each running 64-bit Ubuntu 16.04 on Intel Xeon E5-2630L two–core CPUs at 2.40GHz (4800 bogomips) with 15MB of cache and 2GB main memory. Performance measurements were based on CPU time consumption.

Though, it is impossible to check the exact correctness of output value in real computation (see chapter 1.2), we did verify empirically the correctness of our implementations on 50×50 highly degenerate matrices as follow:

- With six distinct random eigenvalues $\lambda_1, \cdots, \lambda_6$, generate a diagonal matrix D with 25-fold λ_1, 12-fold λ_2, 6-fold λ_3, 4-fold λ_4, 2-fold λ_5, and single λ_6.

- With a random orthogonal matrix O, generated according to [30], apply a orthogonal transformation $H := ODO^{-1}$.
- Run our implementation to diagonalize the random matrix H.
- Test the multiplicities of each eigenvalue that the algorithm found with the initially set multiplicities. Moreover, with the returned eigenvectors x_j, test for $\max_i \|A \cdot x_j - \lambda_j \vec{x}_j\| \neq 0$; the test should not terminate.

This has been confirmed for the slowest Algorithm (a) up to output precision 2^{-180}, for Algorithms (b+c) up to 2^{-300}, and for Algorithm (d) up to 2^{-1600}.

Regarding empirical efficiency evaluation, we are mainly interested in the running time of the four algorithms and their dependence on three natural parameters:

(i) output precision n, i.e. absolute error bound 2^{-n} for both eigenvalues and eigenvectors

(ii) matrix dimension d

(iii) eigenvalue distribution, such as (but not limited to) their minimal separation ε.

For the latter purpose we deterministically generate two families of non-degenerate symmetric random matrices with increasingly ill-conditioned eigenvector problems as follows: deterministically set up diagonal matrices with (I) equally spaced eigenvalues $\mathrm{diag}\big(0, \frac{1}{d}, \frac{2}{d}, \ldots, \frac{d-1}{d}\big)$, and (II) exponentially accumulating $\mathrm{diag}\big(1, \frac{1}{2}, \frac{1}{4}, \ldots, 2^{-d+1}\big)$; then apply to each a random orthogonal transformation; and have that recovered using our algorithms. The corresponding runtime measurements are depicted in Figure 5.2a).

As third family (III) we consider *Gaussian Orthogonal Ensemble* matrices, that is, $(A + A^+)/\sqrt{2}$ where $A \in \mathbb{R}^{d \times d}$ has independently standard normally distributed entries; see Figure 5.2b). These are known to be non-degenerate almost surely [31].

5.2 Interpretation

Qualitatively speaking, our measurements confirm the practical relevance claimed in [26, §1.3]: Using their Algorithm (c) we were able to diagonalize matrices up to size 200×200 — rigorously up to guaranteed absolute output error 2^{-1000}. They also exhibit the gradual asymptotic performance improvement when moving from Algorithm (a) via (b) to (c); see Figure 5.2.

However 'our new' Algorithm (b) does outperform Algorithm (c) on medium-sized instances. This can be explained by the overhead of each Newton iteration whose quadratic convergence kicks in and pays off only for very high precision; notice for instance the 'constant' factor 2^{20} in [25, Lemma 6]. This has led us to consider, implement, and evaluate the hybrid Algorithm (d); which indeed can be seen to improve over all previous ones.

Algorithm (a) on the other hand becomes infeasible from dimension 40 on; see Figure 5.1.

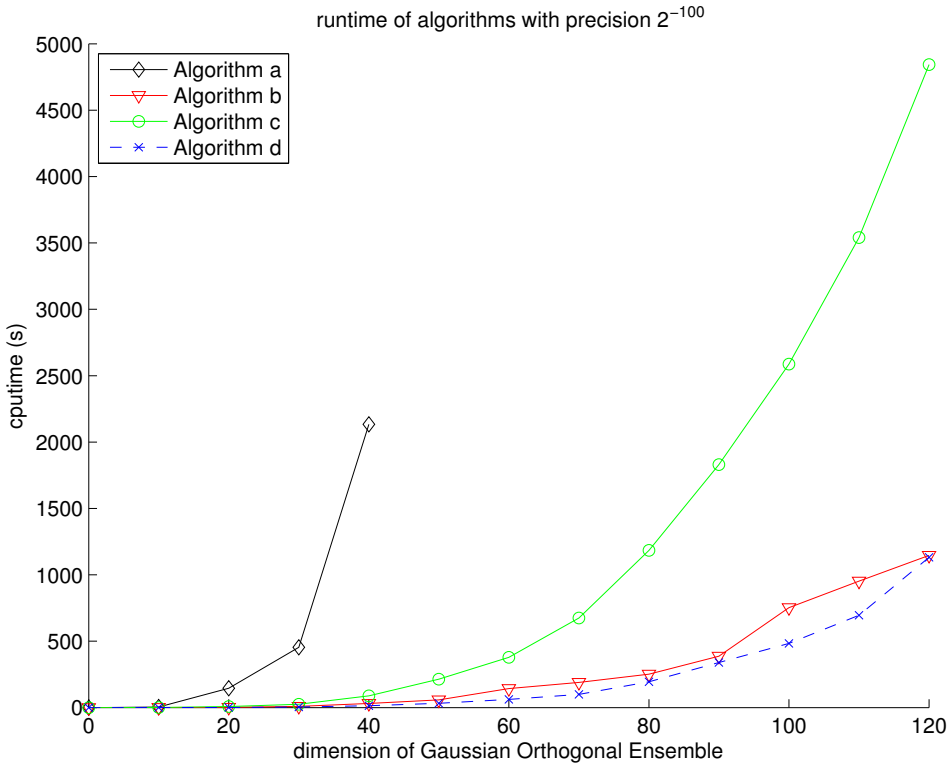


Figure 5.1: Runtime in dependence on dimension for fixed precision.

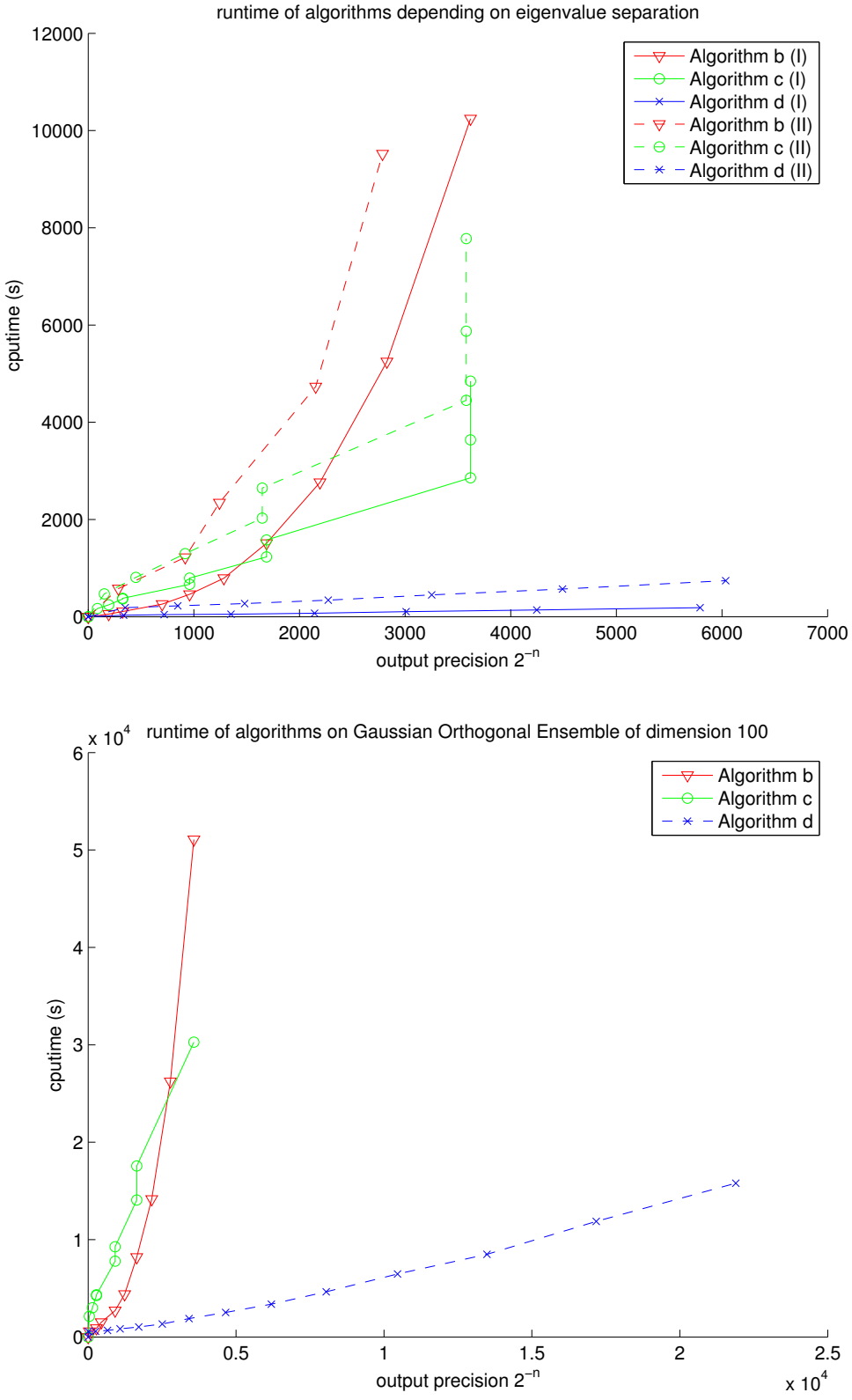


Figure 5.2: Runtime measurements of Algorithms (b), (c), and (d) on benchmark matrices (I), (II), and (III) of dimension 100 with increasing precision.

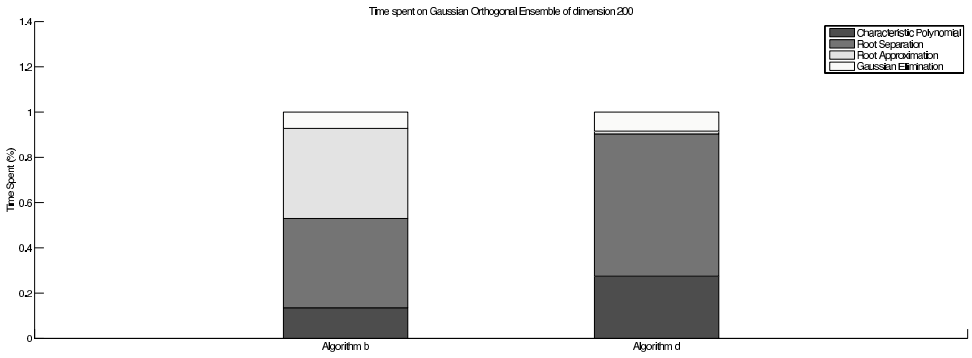


Figure 5.3: Fraction of runtime spent in different parts of Algorithms (b) and (d).

Chapter 6. Concluding Remark

We have proposed a framework for formal verification in real computation. We have extended Hoare logic by generalizing inference rules to include partial operations and `LAZY_BOOLEAN` conditional expressions. Moreover, we have proved correctness of some simple programs in real computation on the proposed framework.

Soundness of the proof system follows inductively; any program that can be proved on the framework can be considered to be correct. However, we haven't consider a memory restriction (hardware problem); our framework considers a computational environment to have enough– that is unlimited – memory. For a real computation, memory safety is another main concern; a `REAL` instance containing sufficient precision often leads to huge memory consumption.

One remaining concern is to show how powerful the framework is. Two perspectives for evaluating a prove system are soundness and (expressive) completeness. The question of how expressive the framework is should be answered in the future.

Thus far, the work of formal verification has been done manually; after applying inference rules, theorems that are used in the verification are proven in `Coq` or manually. The theorems of real numbers solved in `Coq` showed some possibility of the concept of using a proof assistant such as `Coq` for verification in real computation. However, parsing verification conditions – so called theorems – to a theorem prover automatically is a significant issue.

In this light, our future work is to develop such a tool for `iRRAM` ; with a specification annotated program, a tool that sends verification conditions to a theorem prover is needed. In order to develop such a tool, the tightest specification for each command should be specified; a *weakest precondition calculus* can be considered for this.

We have suggested and implemented algorithms of Hermitian and real symmetric matrix diagonalization, using existing algorithms and also by devising new algorithms. The problem we dealt with was a matrix diagonalization of degenerate matrices with the number of distinct eigenvalues provided: a computability-theoretically necessary and sufficient condition.

Our experimental analyses demonstrate their practical feasibility when perusing a recent combination of three approaches to total polynomial root finding due to Sagraloff, Sharma, Yap et al. The analyses further exhibit an intermediate range of precision where a total variant of trisection as a fourth, additional approach further improves the overall performance.

While established and highly-optimized libraries such as `MatLab` easily outperform us and produce correct results in many cases, we guarantee total correctness — and thus support Computer-Assisted Proofs [32] and Experimental Mathematics [33]:

Random Matrix Theory predicts that the normalized eigenvalues $\lambda/\sqrt{d}$ of a Gaussian Orthogonal Ensemble are asymptotically distributed according to the *Wigner semicircular density* $\sqrt{4-|\lambda|^2}/2\pi$ [34, Theorem 2.4.2]; our algorithms allow us to explore the rate of convergence, cmp. Figure 6.1.

While the present Algorithms (a)–(d) are specifically tailored for the case of Hermitian matrices, future work will generalize to the normal case with complex eigenvalues.

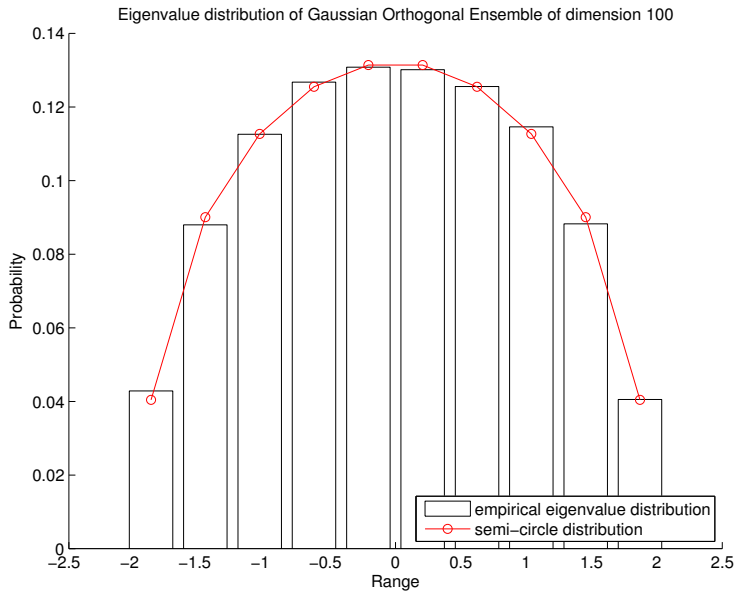


Figure 6.1: Predicted and Empirical Eigenvalue Distribution of Gaussian Orthogonal Ensemble.

Bibliography

[1] M. B. Pour-El and J. I. Richards, *Computability in analysis and physics*, vol. 1. Cambridge University Press, 2017.

[2] K. Weihrauch, *Computable Analysis: An Introduction*. Springer, 2000.

[3] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London mathematical society*, vol. 2, no. 1, pp. 230–265, 1937.

[4] V. Brattka, "The emperor's new recursiveness: The epigraph of the exponential function in two models of computability.," *Words, languages & combinatorics*, vol. 3, pp. 63–72, 2000.

[5] H. Luckhardt, "A fundamental effect in computations on real numbers," *Theoretical Computer Science*, vol. 5, no. 3, pp. 321–324, 1977.

[6] A. Pauly and M. Ziegler, "Relative computability and uniform continuity of relations," *arXiv preprint arXiv:1105.3050*, 2011.

[7] L. Blum, F. Cucker, M. Shub, and S. Smale, *Complexity and real computation*. Springer, 1998.

[8] P. Boldi and S. Vigna, "Equality is a jump," *Theoretical computer science*, vol. 219, no. 1-2, pp. 49–64, 1999.

[9] V. Brattka and P. Hertling, "Feasible real random access machines," *Journal of Complexity*, vol. 14, no. 4, pp. 490–526, 1998.

[10] N. T. Müller, "The irram: Exact arithmetic in c++," in *Computability and Complexity in Analysis*, pp. 222–252, Springer, 2001.

[11] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[12] R. W. Floyd, "Assigning meanings to programs," *Mathematical aspects of computer science*, vol. 19, no. 19-32, p. 1, 1967.

[13] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[14] S. Boldo and J.-C. Filliâtre, "Formal verification of floating-point programs," in *Computer Arithmetic, 2007. ARITH'07. 18th IEEE Symposium on*, pp. 187–194, IEEE, 2007.

[15] N. T. Müller and C. Uhrhan, "Some steps into verification of exact real arithmetic," in *NASA Formal Methods Symposium*, pp. 168–173, Springer, 2012.

[16] N. Müller, S. Park, N. Preining, and M. Ziegler, "On formal verification in imperative multivalued programming over continuous data types," *arXiv preprint arXiv:1608.05787*, 2016.

[17] M. Ziegler and V. Brattka, "Computability in linear algebra," *Theoretical Computer Science*, vol. 326, no. 1-3, pp. 187–211, 2004.

[18] M. Ziegler, "Real computation with least discrete advice: A complexity theory of nonuniform computability with applications to effective linear algebra," *Annals of Pure and Applied Logic*, vol. 163, no. 8, pp. 1108–1139, 2012.

[19] C. Yap, M. Sagraloff, and V. Sharma, "Analytic root clustering: A complete algorithm using soft zero tests," in *Conference on Computability in Europe*, pp. 434–444, Springer, 2013.

[20] Q. I. Rahman and G. Schmeisser, *Analytic theory of polynomials*. No. 26, Oxford University Press, 2002.

[21] S. M. Rump, "Ten methods to bound multiple roots of polynomials," *Journal of Computational and Applied Mathematics*, vol. 156, no. 2, pp. 403–432, 2003.

[22] A. S. Householder, "Dandelin, lobacevskii, or gräffe," *The American Mathematical Monthly*, vol. 66, no. 6, pp. 464–466, 1959.

[23] E. Schröder, "Über unendlich viele algorithmen zur auflösung der gleichungen," *Mathematische Annalen*, vol. 2, no. 2, pp. 317–365, 1870.

[24] M. Sagraloff, "When newton meets descartes: A simple and fast algorithm to isolate the real roots of a polynomial," in *Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation*, pp. 297–304, ACM, 2012.

[25] R. Becker, M. Sagraloff, V. Sharma, and C. Yap, "A simple near-optimal subdivision algorithm for complex root isolation based on the pellet test and newton iteration," *arXiv preprint arXiv:1509.06231*, 2016.

[26] R. Becker, M. Sagraloff, V. Sharma, J. Xu, and C. Yap, "Complexity analysis of root clustering for a complex polynomial," in *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, pp. 71–78, ACM, 2016.

[27] R. Krebbers and B. Spitters, "Computer certified efficient exact reals in coq," in *International Conference on Intelligent Computer Mathematics*, pp. 90–106, Springer, 2011.

[28] R. R. Silva, "The trace formulas yield the inverse metric formula," *Journal of Mathematical Physics*, vol. 39, no. 11, pp. 6206–6213, 1998.

[29] T. Estermann, *Complex numbers and functions*. University of London, Athlone Press, 1962.

[30] G. W. Stewart, "The efficient generation of random orthogonal matrices with an application to condition estimators," *SIAM Journal on Numerical Analysis*, vol. 17, no. 3, pp. 403–409, 1980.

[31] T. Tao and V. Vu, "Random matrices have simple spectrum," *arXiv preprint arXiv:1412.1438*, 2014.

[32] S. M. Rump, "Computer-assisted proof i," *Bulletin of the Japan Society for Industrial and Applied Mathematics*, vol. 14, no. 3, pp. 214–223, 2004.

[33] J. Borwein and K. Devlin, "The computer as crucible: An introduction to experimental mathematics," *The Australian Mathematical Society*, p. 208, 2009.

[34] T. Tao, *Topics in random matrix theory*, vol. 132. American Mathematical Society Providence, RI, 2012.