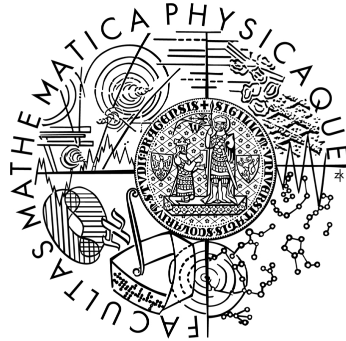
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Nikolay Stoyanov Kaleyski

Eigenvalues of symmetric interval matrices

Department of Applied Mathematics

Supervisor of the bachelor thesis:	Milan Hladík
Study programme:	Computer Science
Specialization:	General Computer Science

Prague 2014

I would like to thank, first and foremost, my supervisor, Milan Hladík, for his constant support and impeccable professionalism, as well as for introducing me to a very interesting topic that I would most probably not have discovered on my own.

I would also like to thank all the excellent teachers and lecturers from the Faculty of Mathematics and Physics who have taught me during the past three years. As much as I would like to, I am afraid that it would be unreasonable to present a complete list of their names here; suffice it to say that there hasn't been a single lecture or seminar that I haven't, to a lesser or greater degree, enjoyed.

Last but not least, I would like to thank all my family and friends for their encouragement and support during my studies. Mathematics, and computer science in general, is not an easy field of study, and sometimes even such a simple thing as a letter or a birthday greeting can be incredibly invigorating.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In .... date .... signature of the author

Název práce: Vlastní čísla symetrických intervalových matic

Autor: Nikolay Stoyanov Kaleyski

Katedra: Katedra aplikované matematiky

Vedoucí bakalářské práce: Mgr. Milan Hladík, Ph.D.

Abstrakt: Cílem práce je popsat a případně vylepšit některé algoritmy pro hledání vnitřních a vnějších odhadů hranic intervalů vlastních čísel reálných symetrických intervalových matic, převést je do verifikovaného tvaru a implementovat je v programovacím jazyce Matlab. Popsány jsou základní principy verifikace a intervalové aritmetiky, použité teoretické základy a problémy, které nastávají při verifikaci jednotlivých algoritmů, včetně možností jejich řešení. Popsány jsou experimenty ukazující některé empirické vlastnosti zkoumaných algoritmů. Praktický výsledek práce je softwarový balík funkcí pro verifikovaný výpočet odhadů množin vlastních čísel symetrických intervalových matic.

Klíčová slova: vlastní čísla intervalové matice, symetrická matice, intervalová matice, verifikace

Title: Eigenvalues of symmetric interval matrices

Author: Nikolay Stoyanov Kaleyski

Department: Department of Applied Mathematics

Supervisor: Mgr. Milan Hladík, Ph.D.

Abstract: The goal of the thesis is to describe and possibly improve some algorithms for finding inner and outer approximations of the borders of eigenvalue intervals of real symmetric interval matrices, to modify them so that they perform verified computations and to implement them in the Matlab programming language. The main principles of verification and interval arithmetic are described, as well as the used theoretical foundations and the problems which occur when attempting to verify the individual algorithms, including possibilities of overcoming them. Experiments illustrating some empirical properties of the algorithms are described. The practical result of the thesis is a software package for computing approximations of the sets of eigenvalues of symmetric interval matrices.

Keywords: eigenvalues of an interval matrix, symmetric matrix, interval matrix, verification

Contents

Introduction **3**

1 Interval arithmetic **6**
- 1.1 Real intervals . . . 6
- 1.2 Interval matrices . . . 7
- 1.3 Eigenvalues of interval matrices . . . 8
- 1.4 Outer and inner approximations . . . 10
- 1.5 Further reading . . . 10

2 Theoretical foundations **11**
- 2.1 Outer approximations . . . 11
- 2.2 Inner approximations . . . 13

3 Algorithms **16**
- 3.1 Simple bounds . . . 17
- 3.2 Direct interlacing algorithm . . . 18
- 3.3 Indirect interlacing algorithm . . . 20
- 3.4 Improving outer approximations by filtering . . . 21
  - 3.4.1 Speeding up the filtering algorithm . . . 22
- 3.5 Local improvement algorithm . . . 23
- 3.6 Vertex enumeration algorithm . . . 25
- 3.7 Submatrix vertex enumeration . . . 26

4 Implementation details **33**
- 4.1 Outer bounds . . . 35
  - 4.1.1 Simple methods . . . 35
  - 4.1.2 Methods applying the direct interlacing procedure . . . 35
  - 4.1.3 Methods applying the indirect interlacing procedure . . . 36
  - 4.1.4 Choosing between the direct and indirect method . . . 36
  - 4.1.5 Filtering methods . . . 38
- 4.2 Inner bounds . . . 38
  - 4.2.1 Methods applying the local improvement method . . . 38
  - 4.2.2 Methods applying the vertex enumeration method . . . 39
  - 4.2.3 Methods applying the direct submatrix vertex enumeration method . . . 39
- 4.3 Interface functions . . . 40

5 User guide **44**
- 5.1 Prerequisites . . . 44
- 5.2 Working with input data . . . 45
- 5.3 Computing inner and outer approximations . . . 46
- 5.4 User-defined computation modes . . . 48

6 Numerical experiments **50**
6.1 Direct and indirect interlacing methods 50
6.1.1 First experiment 50
6.1.2 Second experiment 55
6.2 Faster filtering methods 60
6.3 Sample results 63

Conclusion **66**

Introduction

As is evident by the title of the bachelor thesis itself, ”Eigenvalues of symmetric interval matrices”, the following text concerns two main areas, namely interval arithmetic and eigenvalues of matrices. Although the first chapter contains all the information that the reader might need on these two topics in order to understand the rest of text, we would still like to take a few paragraphs to describe the goals and principles of the work in a freer and more general way and to familiarise the reader with the problem at hand before we proceed with the actual theory.

Possibly the most important concept to introduce is interval arithmetic; as the name implies, this is basically the application of the ordinary arithmetic operations, such as addition, multiplication and so forth, to intervals (instead of numbers). The rules for performing such operations are quite intuitive, and are laid out in detail in Chapter 1; furthermore, the concept of working with intervals can be naturally extended to mostly any operator or function applicable to real numbers in some way, including, as we shall see, the calculation of eigenvalues and eigenvectors of matrices. Nonetheless, the easiest way to grasp the reasoning behind interval arithmetic is to understand its motivation and the reasons it is used.

In a word, intervals are used to express uncertainty. Mathematics itself is, of course, perfectly accurate and exact, and mathematical computations by their very nature cannot lead to uncertain or approximate values; the result of a certain operation does not necessarily have to always be defined, and there can be multiple solutions to a certain problem, but there can be no doubt as to the correctness of the final answer, provided that all calculations are performed correctly.

Applying mathematics to practical problems from the real world, however, introduces an element of uncertainty, as we can never say for sure what the values of our parameters and input data are. Suppose, for example, that we want to measure the length of a table, a wooden plank, or some other common object. We utilise some sort of measuring device, such as a simple ruler, and conclude that the length of the object is about 34.5 metres. The keyword here is ”about”, as the length will never be precisely 34.5 metres: if we were to take a ruler with a finer scale and, possibly, a magnifying glass to assist us in our measurement, we would come to the conclusion that the number in question is closer to, e.g., 34.87 metres. We can continue this process by using better and better measurement devices, and making more and more accurate estimates about the length of the table, but still, we would never be able to say precisely how long the table is; the best that we can do is state that its length is somewhere between 34.85 and 34.89 metres, for example. For most practical applications this should be good enough, yet any calculations that we might want to perform involving the table's length as a parameter will necessarily be uncertain as well.

As a simple example, suppose that we now want to find out how many such tables we would be able to place in a row inside a room of a given size. The obvious solution is to divide the length of the room by the length of a single table and round down, which will give us the maximal number of tables that the room can hold. Nonetheless, the length of our table is not a number, but an interval; and, therefore, our answer will be an interval as well: a room of length 314, for

example, can hold 9 tables of length 34.85, but only 8 tables of length 34.89. Now, although we don't know the "real" answer, we do know that we will be able to fit no more than 9, and no less than 8 tables inside the room in question.

Intervals are not limited to the handling of inexact measurements and can be used to express other kinds of uncertainty as well: we might have several different tables with different lengths (for example, 10, 20 and 30 metres), and might want to find out how many of them are always guaranteed to fit into the room without knowing how many of each type we have beforehand. In this case, we can express all possible lengths of any single table with the interval $[10, 30]$.

Another common use of intervals, which is somewhat similar to the first example presented (the one about inexact measurements), is providing verification for computer algorithms. The problem in this case is that, even if we know the exact values of the input parameters for a given program, in the general case we will still not be getting the exact, mathematically precise answer to our query in practice due to the rounding up and down of floating point numbers inside the computer's memory. The reason for this is that every number is represented using a finite quantity of bits, which obviously makes it impossible to represent all possible real values; essentially, numbers are rounded up or down to "fit" into the limited representation available. The same principle applies to, and can be easily illustrated using ordinary pocket calculators, where adding a very large number to a very small number will leave the large number unchanged; one can also try dividing one by a number which produces a repeating decimal (for example, three) and then multiply the result by the number again (possibly performing some meaningless operations in between, such as adding and subtracting zero); on most calculators, the result is not going to be one.

Although such small lapses in precision can seem harmless, they can quickly accumulate, e.g. during the course of more complicated arithmetic operations (multiplying such an imprecise result by a large number, for example, will increase the difference between the computed and the actual value times the number). Furthermore, for certain applications even a very small error might prove to be critical. In such cases, verification can be used, the basic principle of which is, instead of returning a single real number as a result of each computation to return an interval that is guaranteed to contain the actual result. Interval arithmetic obviously needs to be used in such verified algorithms.

The eigenvalues and eigenvectors of a square matrix should be familiar notions to anyone who has studied linear algebra, and are known to be one of the important and quite frequently used in practice characteristics of a matrix. We will remind the reader that λ is an eigenvalue of some square real matrix A if there exists a non-zero vector x such that $Ax = \lambda x$; the vector x itself is called an eigenvector of A. Furthermore, simple and well-known methods for calculating the eigenvalues of a given square matrix exist, such as finding the roots of the matrix's characteristic polynomial.

If we apply the principle of using intervals to represent uncertainty described above to matrices, we will get so called "interval matrices"; informally, one can imagine them as ordinary, real matrices in which the individual numbers (or elements) of the matrix have been replaced by intervals. It is easy to determine the result of simple operations, such as addition, multiplication with a number and multiplication with a matrix, applied to such interval matrices. Perhaps

surprisingly, however, there is no known method for finding the exact eigenvalue intervals (the eigenvalues, from single numbers, become intervals as well) of an interval matrix.

Nonetheless, certain approaches can be used to obtain some sort of estimates, or approximations, of the eigenvalue intervals of a given matrix. We can look for an outer approximation of an eigenvalue interval (which is essentially an "outer boundary", which is generally larger than the actual interval, but is guaranteed to contain it) or for an inner one (which is some interval that is generally smaller than the one being approximated; the latter is guaranteed to contain "at least" the inner approximation). Ideally, we would want to find a narrow enough outer approximation, as well as a wide enough inner approximation; while certainly not an exact solution, it can still give us a good idea of what that exact solution might possibly be.

The only question left to answer is, how could one go about computing inner and outer approximations for the eigenvalues of a given matrix; and this is precisely the subject of this bachelor thesis, which examines several methods for finding and improving such approximations. As a general rule, we must look for some number which will always be greater than the eigenvalue of any real matrix with elements inside the corresponding intervals of the interval matrix in order to establish an upper outer bound, and, in the same way, we should look for values which are always smaller to get a lower outer bound. Finding inner approximations is somewhat easier, as the eigenvalues of any matrix with its elements inside the corresponding intervals can be used for that purpose, so basically all we have to do is select one or several such matrices and examine their eigenvalues. Still, we should attempt to find some better strategy than just picking random matrices.

Once again, we remind the reader that the above paragraphs describe the principles presented in a very informal and, hopefully, intuitive way. The following text contains the precise mathematical definitions of the notions described here, such as inner and outer approximations, intervals and interval matrices, etc., as well as specific theorems and algorithms that can be used to look for approximations for the eigenvalue intervals of symmetric interval matrices and a discussion of some of the problems that arise when attempting to add verification to them.

1. Interval arithmetic

1.1 Real intervals

Definition. *Let $\underline{x}$ and $\overline{x}$ be real numbers fulfilling $\underline{x} \leq \overline{x}$. Then the set $\boldsymbol{x} = [\underline{x}, \overline{x}] = \{y \in \mathbb{R} :: \underline{x} \leq y \leq \overline{x}\}$ is called a **real interval**. The set of all real intervals is denoted as $\mathbb{IR}$. The real number $\underline{x}$, resp. $\overline{x}$ is called the **lower**, resp. **upper bound** of the interval $[\underline{x}, \overline{x}]$.*

Naturally, this definition can be adapted to any partially ordered set, specifically the set of complex numbers $\mathbb{C}$. However, since we will mostly work only with real matrices and intervals, we do not need to examine such cases in detail.

The basic arithmetic operations can be naturally extended to real intervals. Intuitively, the result of an interval operation on a pair of real intervals should encompass all possible values that can be obtained by applying the corresponding real operation to any two real numbers from the first and second interval; in other words, for an arbitrary binary operation $\circ$ on $\mathbb{R}$ and intervals $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{IR}$ we can define $\circ$ on $\mathbb{IR}$ as $\boldsymbol{x} \circ \boldsymbol{y} = \{x \circ y : x \in \boldsymbol{x}, y \in \boldsymbol{y}\}$. Specific definitions for the individual arithmetic operations are provided in the following definition.

Definition. *Let $\boldsymbol{x} = [\underline{x}, \overline{x}]$ and $\boldsymbol{y} = [\underline{y}, \overline{y}]$ be real intervals. Then the binary operations addition(+), subtraction(−), multiplication(·) and division(/) are defined on $\mathbb{IR}$ as follows:*

- $\boldsymbol{x} + \boldsymbol{y} = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$
- $\boldsymbol{x} - \boldsymbol{y} = [\underline{x} - \overline{y}, \overline{x} - \underline{y}]$
- $\boldsymbol{x} \cdot \boldsymbol{y} = [\min\{\underline{x} \cdot \underline{y}, \underline{x} \cdot \overline{y}, \overline{x} \cdot \underline{y}, \overline{x} \cdot \overline{y}\}, \max\{\underline{x} \cdot \underline{y}, \underline{x} \cdot \overline{y}, \overline{x} \cdot \overline{y}, \overline{x} \cdot \overline{y}\}]$
- $\boldsymbol{x}/\boldsymbol{y} = \boldsymbol{x} \cdot \boldsymbol{y}'$, *where* $\boldsymbol{y}' = [\frac{1}{\overline{y}}, \frac{1}{\underline{y}}]$ *and* $0 \notin \boldsymbol{y}$.

Definition. *Let $\boldsymbol{x} = [\underline{x}, \overline{x}]$ be a real interval. Then:*

- *The **width** of $\boldsymbol{x}$ is defined as* $\operatorname{width} x = \overline{x} - \underline{x}$*;*
- *The **radius** of $\boldsymbol{x}$ is defined as* $\operatorname{rad} x = \frac{1}{2} \cdot \operatorname{width} x$*;*
- *The **centre** or **midpoint** of $\boldsymbol{x}$ is defined as* $\operatorname{mid} x = \frac{1}{2} \cdot (\underline{x} + \overline{x})$
- *The **absolute value** of $\boldsymbol{x}$, also referred to as the **magnitude** of $\boldsymbol{x}$, is defined as* $|\boldsymbol{x}| = \max\{-\underline{x}, \overline{x}\}$

Note that the upper and lower bounds of an interval can be unambiguously inferred from its midpoint and radius, and vice versa. This allows for an equivalent representation of the interval $\boldsymbol{x} = [\underline{x}, \overline{x}]$ as $\boldsymbol{x} = [\operatorname{mid} x - \operatorname{rad} x, \operatorname{mid} x + \operatorname{rad} x]$.

Since intervals are essentially sets, set operations can be naturally applied to them, e.g. $\boldsymbol{x} \cap \boldsymbol{y} = \{z \in \mathbb{R} : z \in \boldsymbol{x} \ \& \ z \in \boldsymbol{y}\} = \{z \in \mathbb{R} : \underline{x} \leq z \leq \overline{x} \ \& \ \underline{y} \leq z \leq \overline{y}\} = [\max\{\underline{x}, \underline{y}\}, \min\{\overline{x}, \overline{y}\}]$ (the last equality holds as long as the intersection is non-empty). Caution needs to be taken when computing the union

of two intervals, however, as $\mathbb{IR}$ is obviously not closed with respect to the union operation; for example, the union $[1,2] \cup [3,4]$ is evidently a set of real numbers, but not a single interval. Should we need to combine two intervals in such a way, we can define an enclosure operation $\boldsymbol{x} \dot{\cup} \boldsymbol{y} = [\min\{\underline{x}, \underline{y}\}, \max\{\overline{x}, \overline{y}\}]$ which defines an interval containing both operands as subsets. Interval variants of more complex set operations, such as symmetric difference, can now be defined using intersection and enclosure.

Another import aspect of working with intervals is specifying an ordering relation. Such a relation is used, albeit implicitly, in practically all of the algorithms that we've implemented, since the results that they produce (which are a set of intervals) are sorted in ascending order. Naturally, there is more than one sensible definition which can be used for this purpose, but we will generally use the following one:

Definition. *The relation $\tilde{\leq}$ on $\mathbb{IR}$ is defined as follows:*

$$(\forall \boldsymbol{x}, \boldsymbol{y} \in \mathbb{IR})(\boldsymbol{x} \tilde{\leq} \boldsymbol{y} \leftrightarrow \operatorname{mid} x \leq \operatorname{mid} y)$$

Note that this relation is not an order in the general case, as it is not anti-symmetric: for example, according to the definition, $[1,2] \tilde{\leq} [0,3], [0,3] \tilde{\leq} [1,2]$, but $[0,3] \neq [1,2]$. However, suppose that we are working with a set of interval $S \subset \mathbb{IR}$ for which the following holds:

$$(\forall \boldsymbol{x}, \boldsymbol{y} \in S)(\operatorname{mid} \boldsymbol{x} = \operatorname{mid} \boldsymbol{y} \rightarrow \boldsymbol{x} = \boldsymbol{y})$$

Then $\tilde{\leq}$ is a correctly defined total order on S.

We will usually write only $\leq$ instead of $\tilde{\leq}$, as this is the only interval ordering that we use in the following text, and it should always be possible to determine from context whether the symbol is used for comparing numbers or intervals. Furthermore, if we state that a certain interval is smaller or larger than another, we are referring precisely to this ordering.

1.2 Interval matrices

As we mentioned in the previous section, the notion of an interval can easily be extended to any partially ordered set, including the set of real matrices $\mathbb{R}^{m \times n}$ for given dimensions $m, n \in \mathbb{N}$.

We remind the reader that an m-by-n matrix over the field of real numbers is, informally, a rectangular "table" of real numbers with m rows and n columns. The notation $\mathbb{A} \in \mathbb{R}^{m \times n}$ means that $\mathbb{A}$ is a real, m-by-n matrix. If this is the case, we will use $\mathbb{A}_{ij}$ to denote the element of the matrix located at row i and column j, for some $i \in \{1, 2, ..., m\}, j \in \{1, 2, ..., n\}$. Of course, matrices can be defined over other sets than just the real numbers as well. In any case, since these are very basic notions, we will not go into too much detail.

Considering the natural element-wise order relation on matrices, $\mathbb{A} \leq \mathbb{B} \iff (\forall i \in \{1, 2, ..., m\})(\forall j \in \{1, 2, ..., n\})(\mathbb{A}_{ij} \leq \mathbb{B}_{ij})$, for $\mathbb{A}, \mathbb{B} \in \mathbb{R}^{m \times n}$, we can define an interval matrix as follows:

Definition. *Let* $\underline{A}, \overline{A} \in \mathbb{R}^{m\times n}$ *be two real matrices of dimension* $m \times n$*, where* $m, n \in \mathbb{N}$*. We define the* ***interval matrix*** $\boldsymbol{A}$ *as the set* $\boldsymbol{A} = \{M \in \mathbb{R}^{m\times n} : \underline{A} \leq M \leq \overline{A}\}$*. The matrix* $\underline{A}$*, resp.* $\overline{A}$ *is called the* ***lower****, resp.* ***upper bound*** *of* $\boldsymbol{A}$*. The set of all interval matrices over* $\mathbb{R}^{m\times n}$ *is denoted as* $\mathbb{IR}^{m\times n}$*.*

An alternative way of looking at an interval matrix is to imagine it as an actual matrix with intervals (instead of real numbers) as elements. A given real matrix will then be a member of the interval matrix if and only if each of it's elements lies within the range defined by the interval at the corresponding position. For example, suppose that we have

$$\boldsymbol{A} = [\underline{A}, \overline{A}],$$

where

$$\underline{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \in \mathbb{R}^{2\times 2},$$

$$\overline{A} = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \in \mathbb{R}^{2\times 2},$$

and it holds that

$$(\forall i, j \in \{1, 2\})(a_{ij} \in \mathbb{R}, b_{ij} \in \mathbb{R}, a_{ij} \leq b_{ij})$$

Then the interval matrix $\boldsymbol{A}$ could also be represented as

$$\boldsymbol{A} = \begin{pmatrix} [a_{11}, b_{11}] & [a_{12}, b_{12}] \\ [a_{21}, b_{21}] & [a_{22}, b_{22}] \end{pmatrix},$$

where for a given real matrix $\mathbb{A} \in \mathbb{R}^{2\times 2}$,

$$\mathbb{A} \in \boldsymbol{A} \iff \underline{A} \leq \mathbb{A} \leq \overline{A} \iff (\forall i, j \in \{1, 2\})(a_{ij} \leq (\mathbb{A}_{ij}) \leq b_{ij})$$

Yet another different representation of an interval matrix which is used throughout the text is based on the central and radial matrices of an interval matrix, which correspond to the midpoint and radius of a general interval.

Definition. *Let* $\boldsymbol{A} = [\underline{A}, \overline{A}] \in \mathbb{IR}^{m\times n}, m, n \in \mathbb{N}$ *be an interval matrix. We define the* ***central matrix*** *of* $\boldsymbol{A}$ *as* $A_c = \frac{1}{2}\cdot(\underline{A}+\overline{A})$ *and the* ***radial matrix*** *of* $\boldsymbol{A}$ *as* $A_\delta = \frac{1}{2}(\overline{A} - \underline{A})$*. The interval matrix* $\boldsymbol{A}$ *can then be written as* $\boldsymbol{A} = [A_c - A_\delta, A_c + A_\delta]$*.*

1.3 Eigenvalues of interval matrices

As we know from linear algebra, a number $\lambda \in \mathbb{C}$ is called an eigenvalue of a matrix $A \in \mathbb{R}^{n\times n}, n \in \mathbb{N}$ if there exists a non-zero vector $v \in \mathbb{R}^n$ such that $A\cdot v = \lambda\cdot v$. A square matrix of dimension n has n (not always different) eigenvalues. Although this does not necessarily hold in the general case, for symmetric matrices the eigenvalues are real numbers. This implies that they can be naturally ordered, for example in a non-descending order, such that $\lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n$ are all eigenvalues of the matrix. These two notions can be naturally adapted to interval matrices, as is shown in the following definition. We assume that the interval matrix is symmetric, since in that case all of its eigenvalues form continuous real intervals; this is the case that we are working with in our project. Naturally, we first have to clarify what a symmetric interval matrix is:

Definition. *Let $\boldsymbol{A} \in \mathbb{IR}^{n\times n}$ be a real interval matrix. We define a* ***symmetric interval matrix*** *as the set $\boldsymbol{A}^S = \{A \in \boldsymbol{A} : A = A^T\} \subseteq \boldsymbol{A}$.*

For convenience, we may sometimes write $\boldsymbol{A}^S \in \mathbb{IR}^{n\times n}$ instead of the full $\boldsymbol{A}^S \subseteq \boldsymbol{A} \in \mathbb{IR}^{n\times n}$, indicating that $\boldsymbol{A}^S$ is a symmetric interval matrix. This should not lead to confusion, as the notation $\boldsymbol{A}^S$ indicates that the interval matrix in question is a symmetric interval matrix.

Note: From this point on, we assume that if we're talking about some symmetric interval matrix $\boldsymbol{A}^S$, then it is non-empty.

Let us take a minute to clarify the above definition, as it can be slightly counter-intuitive. A symmetric interval matrix is actually a subset of an interval matrix which only contains symmetric real matrices. As was already mentioned above, one way of looking at an interval matrix is to imagine it as an "ordinary" matrix which contains intervals instead of numbers; from this viewpoint, it might be natural to think that the symmetric interval matrix is such a matrix of intervals fulfilling the additional condition that its elements are symmetric, i.e. the interval at position (i, j) is the same as the interval at position (j, i). Formally, this means that the matrices $\underline{A}$ and $\overline{A}$ defining $\boldsymbol{A} = [\underline{A}, \overline{A}]$ are symmetric, i.e. $\underline{A}^T = \underline{A}$, $\overline{A}^T = \overline{A}$. However, such a definition is not equivalent to the one presented above! For example, consider the following matrix with interval elements:

$$\boldsymbol{A} = \begin{pmatrix} [0,5] & [1,2] \\ [1,2] & [4,6] \end{pmatrix}$$

It is obvious (actually, it follows immediately from the definition) that $\boldsymbol{A}^S \subseteq \boldsymbol{A}$, where $\boldsymbol{A}^S = \{A \in \boldsymbol{A} : A^T = A\}$. In fact, it holds that $\boldsymbol{A}^S \subsetneq \boldsymbol{A}$: for example, $A = \begin{pmatrix} 0 & 1 \\ 2 & 5 \end{pmatrix}$ is an element of $\boldsymbol{A}$, since all its elements lie within the corresponding intervals, but $A \notin \boldsymbol{A}^S$, since A is not symmetric.

On the other hand, sometimes it is useful to assume that the lower and upper bounds of an interval matrix $\boldsymbol{A} = [\underline{A}, \overline{A}]$ defining a symmetric interval matrix $\boldsymbol{A}^S$ are symmetric matrices, which we can do without loss of generality by defining a new, "symmetrized" interval matrix $\boldsymbol{A'} = [\underline{A}', \overline{A}']$, where $(\underline{A}')_{ij} = (\underline{A}')_{ji} = (\underline{A})_{ij} \cap (\overline{A}')_{ji}$ and $(\overline{A}')_{ij} = (\overline{A}')_{ji} = (\overline{A})_{ij} \cap (\overline{A})_{ji}$, in other words, by replacing each pair of intervals at symmetric positions in the matrix with their intersections. Since we're dealing with a symmetric interval matrix, all such intersections must be non-empty (otherwise $\boldsymbol{A}^S = \emptyset$).

From the above definitions it is now obvious that the eigenvalues of all matrices contained within a symmetric interval matrix are real numbers; furthermore, it is known that the i-th eigenvalues of all of these real matrices form a continuous interval, which ultimately makes the next definition possible.

Definition. *Let $\boldsymbol{A}^S \subseteq \boldsymbol{A} \in \mathbb{IR}^{n\times n}$ be a symmetric interval matrix, where $n \in \mathbb{N}$. We denote as $\Lambda(\boldsymbol{A}^S) = \{\lambda \in \mathbb{R} : (\exists A \in \boldsymbol{A}^S)(\exists x)(x \neq 0\ \&\ Ax = \lambda x)\}$ the set of all (real) eigenvalues of the matrices from $\boldsymbol{A}^S$; similarly, for $i = 1, 2, ..., n$, we denote $\Lambda_i(\boldsymbol{A}^S) = \{\lambda_i \in \mathbb{R} : (\exists A \in \boldsymbol{A}^S)($"$\lambda_i$ is the i-th eigenvalue of A"$)\}$ the set containing the i-th eigenvalue of every matrix in $\boldsymbol{A}^S$.*

Note that the definition for the eigenvalue sets of (non-symmetric) interval matrices (which we do not need) are defined in almost exactly the same way, but they can contain complex numbers and do not form intervals. We cannot define individual sets $\Lambda_i(\boldsymbol{A})$ for some interval $\boldsymbol{A}$ either, since we cannot order complex eigenvalues.

1.4 Outer and inner approximations

As Proposition 2.1. of [3] shows, the set of real eigenvalues of an interval matrix is a finite union of compact real intervals. However, in the general case (at least with our current knowledge) we cannot compute the exact bounds of these intervals; this is in contrast to the computation of eigenvalues of individual real matrices, for which simple and exact formulas exist (for example, finding the roots of the matrix's characteristic polynomial). Instead, we concentrate on finding inner and outer approximations of these bounds.

Definition. *Let $\boldsymbol{a}$ be a set. We say that the set $\boldsymbol{i}$ is an **inner approximation** of $\boldsymbol{a}$ if $\boldsymbol{i}$ is a subset of $\boldsymbol{a}$. We say that the set $\mathbf{o}$ is an **outer approximation** of $\boldsymbol{a}$ if $\boldsymbol{a}$ is a subset of $\mathbf{o}$. If $\underline{a}$ and $\overline{a}$ are respectively the smallest and largest elements of a (specifically, if $\boldsymbol{a} = [\underline{a}, \overline{a}]$ is a real interval), we say that the number $r \in \mathbb{R}$ is a **lower outer approximation**, resp. a **lower inner approximation**, resp. an **upper outer approximation**, resp. an **upper inner approximation** of $\boldsymbol{a}$, if it holds that $r \leq \underline{a}$, resp. $\underline{a} \leq r$, resp. $\overline{a} \leq r$, resp. $r \leq \overline{a}$.*

1.5 Further reading

We have presented only the necessary minimum of theory related to interval computations for the purposes of the thesis. Should the reader desire to get acquainted with the topic in more detail, we suggest referring to a specialised text on this subject, such as [1] or [6].

2. Theoretical foundations

As was mentioned in the previous section, no formulae are known for computing the exact bounds of $\Lambda(\boldsymbol{A})$ (or $\Lambda(\boldsymbol{A}^S)$, for that matter) in the general case, and therefore we have to content ourselves with finding inner and outer approximations for these values. The current section describes the main theorems used to compute such approximations and the way they are applied in the implementation. We mainly use the theoretical results of Hladík and his colleagues described in [3], [4] and [5].

2.1 Outer approximations

A simple, but quite useful theorem proven by Rohn and originally appearing as Theorem 2 in [8] allows us to establish an outer bound on $\Lambda(\boldsymbol{A})$ by performing a relatively very small amount of computations.

Theorem 1. *Let $\boldsymbol{A} = [A_c - A_\delta, A_c + A_\delta] \in \mathbb{IR}^{n\times n}, n \in \mathbb{N}$ be a square interval matrix (not necessarily symmetric). Then for each eigenvalue λ of each matrix $A \in \boldsymbol{A}$ the following inequalities hold:*

$$\lambda_{\min}(M_r) - \rho(\Delta_r) \leq Re(\lambda) \leq \lambda_{\max}(M_r) + \rho(\Delta_r)$$

$$\lambda_{\min}(M_i) - \rho(\Delta_i) \leq Im(\lambda) \leq \lambda_{\max}(M_i) + \rho(\Delta_i)$$

where

$$M_r = \tfrac{1}{2}(A_c + {A_c}^T)$$

$$\Delta_r = \tfrac{1}{2}(A_\delta + {A_\delta}^T)$$

$$M_i = \begin{pmatrix} 0 & \tfrac{1}{2}(A_c - {A_c}^T) \\ \tfrac{1}{2}({A_c}^T - A_c) & 0 \end{pmatrix}$$

$$\Delta_i = \begin{pmatrix} 0 & \Delta_r \\ \Delta_r & 0 \end{pmatrix}$$

$\lambda_{\min}(M)$ and $\lambda_{\max}(M)$ are respectively the smallest and largest eigenvalue of the real symmetric matrix M

$Re(\lambda)$ and $Im(\lambda)$ are respectively the real and imaginary component of the complex number λ

Although we've included the whole statement of the theorem for the sake of completeness, we only need the first of the two inequalities for the purpose of our work, since we're dealing with symmetric matrices, the eigenvalues of which are real numbers.

A similar theorem exists, approximating the outer bounds of the sets $\Lambda_i(\boldsymbol{A}^S)$ of a symmetric interval matrix. It is listed as equation (3.14) in Rohn's "A handbook of Results on Interval Linear Problems"([10]) as a consequence of the Wielandt-Hoffman theorem, while a complete proof is provided in [3] under Theorem 3.1.

Theorem 2. *Let $\boldsymbol{A}^S \in \mathbb{IR}^{n\times n}, n \in \mathbb{N}$, be a symmetric interval matrix. Then, for $i = 1, 2, ..., n$ and $A \in \boldsymbol{A}^S$ it holds that*

$$\lambda_i(A_c) - \rho(A_\delta) \le \lambda_i(A) \le \lambda_i(A_c) + \rho(A_\delta)$$

Evidently, the first theorem is a special case of this one if we consider only symmetric interval matrices. In any case, Theorem 2, to which we shall refer to as Rohn's theorem from this point on, is a very valuable starting point in our search for a tight outer approximation, and its results are frequently used as an input for more complicated improvement algorithms or as a basis for certain criteria (e.g. the direct vs. indirect interlacing methods). The practical utilization of the theorem involves computing all eigenvalues of the A_c and A_δ matrices in order to find the individual eigenvalues of the first matrix and the spectral radius of the second. Upper and lower bounds for the possible eigenvalues and its spectral radius can potentially be applied here if faster computation is required. Then a single arithmetic operation yields the outer approximation for each $\Lambda_i(\boldsymbol{A}^S)$.

In certain cases we need an upper bound only for the largest eigenvalue of a given interval matrix; if this is the case, the following theorem, appearing with proof as proposition 3.2 in [3], can prove to be useful.

Theorem 3. *Let $\boldsymbol{A}^S \subseteq \mathbb{IR}^{n\times n}, n \in \mathbb{N}$ be a real symmetric matrix. Then*

$$\overline{\lambda}_n(\boldsymbol{A}^S) \le \lambda_n(|\boldsymbol{A}|)$$

Another simple and very important theorem which serves as the foundation of two algorithms for approximating the eigenvalue bounds of a symmetric interval matrix is Cauchy's interlacing property.

Theorem 4. *Let $A \in \mathbb{R}^{n\times n}, n \in \mathbb{N}$ be a real symmetric matrix and denote by A_i for $i = 1, 2, ..., n$ the matrix obtained from A by removing the i-th row and column (such a submatrix is called a "principal submatrix" of A). Then it holds that*

$$\lambda_1(A) \le \lambda_2(A_i) \le \lambda_2(A) \le \lambda_3(A_i) \le ... \le \lambda_{n-1}(A_i) \le \lambda_{n-1}(A) \le \lambda_n(A_i) \le \lambda_n(A)$$

The theorem, obviously, allows us to use the eigenvalues of a principal submatrix of $\boldsymbol{A}^S$ as bounds. The direct and indirect interlacing algorithms are based on this theorem. The latter also makes use of the following theorem due to Weyl, which allows us to express bounds on the eigenvalues of a matrix $C = A + B$ using the eigenvalues of the matrices A and B:

Theorem 5. *Let $A \in \mathbb{R}^{n\times n}, B \in \mathbb{R}^{n\times n}, n \in \mathbb{N}$ be real symmetric matrices. Then, for $r, s \in \{1, 2, ..., n\}$:*

$$(r + s \le n + 1) \implies (\lambda_{r+s-1}(A + B) \ge \lambda_r(A) + \lambda_s(B))$$

$$(r + s \ge n + 1) \implies (\lambda_{r+s-n}(A + B) \le \lambda_r(A) + \lambda_s(B))$$

One of the methods we use for finding a tight outer approximation of the eigenvalue bounds of an interval matrix, namely the filtering algorithm, is based on computing some initial approximation and then iteratively improving it by "cutting off" parts of the intervals that do not contain any eigenvalues. This approach is based on the following theorem, presented with proof in [5].

Theorem 6. *Let* $\boldsymbol{A} \in \mathbb{IR}^{n\times n}, n \in \mathbb{N}$ *be a square interval matrix and* λ^0 *be a real number such that* $\lambda^0 \notin \Lambda(\boldsymbol{A})$*. Define* $\boldsymbol{M} = \boldsymbol{A} - \lambda^0 I$*. Then, for any real number* λ *fulfilling*

$$|\lambda| < \frac{1 - \frac{1}{2}\rho(|I - QM_c| + |I - QM_c|^T + |Q|M_\delta + {M_\delta}^T|Q|^T)}{\frac{1}{2}\rho(|Q| + |Q|^T)}$$

it holds that

$$(\lambda^0 + \lambda) \notin \Lambda(\boldsymbol{A})$$

where I *is the identity matrix and* $Q \neq 0$ *is an arbitrary real matrix.*

The special case for a symmetric interval matrix is listed as Corollary 2 in the original article, and follows immediately from the fact that $M_c = {M_c}^T$ and $M_\delta = {M_\delta}^T$ can be assumed for $\boldsymbol{A}$ symmetric, and from the special choice of Q to be a symmetric matrix as well.

Theorem 7. *Let* $\boldsymbol{A}^S \in \mathbb{IR}^{n\times n}, n \in \mathbb{N}$*, be a real symmetric interval matrix and* $\lambda^0 \notin \Lambda(\boldsymbol{A}^S)$*. Then, for any real number* λ *fulfilling*

$$|\lambda| < \frac{1 - \frac{1}{2}\rho(|I - QM_c| + |I - M_cQ| + |Q|M_\delta + M_\delta|Q|)}{\rho(|Q|)} \tag{2.1}$$

it holds that

$$(\lambda^0 + \lambda) \notin \Lambda(\boldsymbol{A}^S)$$

where $Q \neq 0$ *is an arbitrary real symmetric matrix.*

Let us take a moment to examine what this theorem claims more closely, as its utilization may not be immediately obvious from its statement. Suppose that we have a symmetric interval matrix $\boldsymbol{A}^S \in \mathbb{IR}^{n\times n}$ and an outer approximation $[\underline{\omega}, \overline{\omega}]$ for the set $\Lambda_i(\boldsymbol{A}^S)$ for some $i \in \{1, 2, ..., n\}$, which we want to improve. We pick some suitable symmetric Q, and take λ^0 to be $\overline{\omega}$; we also have all necessary information to compute the value of the expression on the right side of (2.1) (which represents the maximum size of the interval that we can "cut off"), which we can denote as $\lambda_{\max}$. Then the theorem states that for any real number λ fulfilling $0 < \lambda < \lambda_{\max}$, $\lambda^0 - \lambda$ and $\lambda^0 + \lambda$ cannot be eigenvalues of any matrix from $\boldsymbol{A}^S$; in other words, there is an interval with its centre at λ^0 and with radius $\lambda_{\max}$ which does not contain any eigenvalues; we can therefore reduce the approximation of the upper bound from $\overline{\omega}$ to $(\overline{\omega} - \lambda_{\max})$. In the same way, we can "tighten" the lower bound approximation, and the whole process can be repeated again. This is, in fact, most of the reasoning behind the filtering algorithm.

2.2 Inner approximations

A simple inner approximation of the set of eigenvalues of an interval matrix can, in fact, be trivially computed without the aid of any complicated theorems, which is evident once we make the observation that any pair of eigenvalues (λ_i, λ_j) of any two matrices fulfilling $\lambda_i \leq \lambda_j$ can serve as an inner approximation of $\Lambda(\boldsymbol{A})$! This follows immediately from the definition of the inner approximation and the fact that the eigenvalue sets of symmetric interval matrices form real intervals.

Observation. *Let $\boldsymbol{A}^S \in \mathbb{IR}^{n\times n}, n \in \mathbb{N}$ be a real symmetric interval matrix, $A \in \boldsymbol{A}^S, B \in \boldsymbol{A}^S$. For $i \in \{1, 2, ..., n\}$, let λ_i^a be the i-th eigenvalue of A and λ_i^b be the i-th eigenvalue of B. Define $\lambda_i^{\min} = \min\{\lambda_i^a, \lambda_i^b\}$, $\lambda_i^{\max} = \max\{\lambda_i^a, \lambda_i^b\}$. Then $[\lambda_i^{\min}, \lambda_i^{\max}]$ is an inner approximation of $\Lambda_i(\boldsymbol{A}^S)$, as well as an inner approximation of $\Lambda(\boldsymbol{A}^S)$. Furthermore, $\bigcup_{i=1}^n [\lambda_i^{\min}, \lambda_i^{\max}]$ is an inner approximation of $\Lambda(\boldsymbol{A}^S)$.*

The above fact suggests that an inner approximation of $\Lambda(\boldsymbol{A}^S)$ can be found by computing the eigenvalues of some finite subset of $\boldsymbol{A}^S$ and selecting the best values from it. Indeed, this is the foundation of the simpler inner approximation algorithms that we use, with the main difference between individual approaches being the method for selecting the finite subset of the interval matrix.

A more sophisticated approach is suggested by Theorem 1 of [4], which we present here without proof.

Theorem 8. *Let $\boldsymbol{A}^S \in \mathbb{IR}^{n\times n}, n \in \mathbb{N}$, be a real symmetric interval matrix and $\lambda \in \mathbb{R}$ be a boundary point of $\Lambda(\boldsymbol{A}^S)$. Then there exists a natural number $k \in \{1, 2, ..., n\}$ and a principal submatrix $\tilde{\boldsymbol{A}}^S \in \mathbb{IR}^{k\times k}$ of $\boldsymbol{A}^S$ such that:*

If there exists an index $j \in \{1, 2, ..., n\}$ such that $\lambda = \overline{\lambda}_j(\boldsymbol{A}^S)$, then

$$\lambda \in \{\lambda_i(\tilde{A}_c + \operatorname{diag}(z)\tilde{A}_\delta \operatorname{diag}(z)) : z \in \{\pm 1\}^k, i = 1, 2, ..., k\}$$

If there exists an index $j \in \{1, 2, ..., n\}$ such that $\lambda = \underline{\lambda}_j(\boldsymbol{A}^S)$, then

$$\lambda \in \{\lambda_i(\tilde{A}_c - \operatorname{diag}(z)\tilde{A}_\delta \operatorname{diag}(z)) : z \in \{\pm 1\}^k, i = 1, 2, ..., k\}$$

The theorem basically implies that all boundary points of $\Lambda(\boldsymbol{A}^S)$ lie within a computable, finite set of values, corresponding to a finite number of vectors z. This means that, if the upper or lower bound of a given $\Lambda_i(\boldsymbol{A}^S)$ lies on the boundary of $\Lambda(\boldsymbol{A}^S)$ (in other words, if it does not lie within any other $\Lambda_j(\boldsymbol{A}^S)$), its exact value can be found by examining all possible real matrices of the type $(\bar{A}_c) + \operatorname{diag}(z)\bar{A}_\delta \operatorname{diag}(z)$. This is the main theoretical foundation of the direct submatrix vertex enumeration.

The final theorem worth noting, due to Hertz([2]), is rather exceptional, as it actually allows us to compute two exact bounds (the upper bound for the largest eigenvalue and the lower bound for the smallest eigenvalue). The theorem's statement is the following:

Theorem 9. *Let $\boldsymbol{A}^S \in \mathbb{R}^{n\times n}, n \in \mathbb{N}$, be a real symmetric interval matrix. Then:*

$$\underline{\lambda}_1 = \min_{v\in\{\pm 1\}^n} \{\lambda_1(A_c + \operatorname{diag}(v)A_\delta \operatorname{diag}(v))$$

$$\overline{\lambda}_n = \max_{v\in\{\pm 1\}^n} \{\lambda_n(A_c + \operatorname{diag}(v)A_\delta \operatorname{diag}(v))$$

The theorem implies a simple algorithm for finding these values, which involves computing the largest (or smallest) eigenvalues of the extremal matrices described by the sign vectors v. While finding only two from a total of $2n$ bounds might seem rather limiting, the fact that we get exact bounds has multiple theoretical

and practical applications: the vertex enumeration algorithm (7), for example, examines all such extremal matrices in order to construct an inner approximation of the eigenvalue intervals, and therefore the results it finds for $\underline{\lambda}_1$ and $\overline{\lambda}_n$ are, in fact, the exact lower and upper bound of the corresponding intervals. The same can be said for the direct submatrix vertex enumeration method (Algorithm 9). Finally, Algorithm 6, while not guaranteed to find any exact bounds, is based on a similar idea as well (namely, examining only some of the extremal matrices). Furthermore, exact bounds are the best possible "approximation" (inner or outer) and knowing them allows testing the efficacy of other methods and approaches by comparing their output against this optimal result.

3. Algorithms

A note on verification

The goal of our work is to create a set of functions providing mathematically verified results. Since verification involves making certain changes to the original, unverified algorithms, and the source materials that we use do not explicitly address this issue, it makes sense to spare a few paragraphs and describe the problem in more detail. Calculations performed on computers (as well as other electronic devices) are never exact, mainly due to rounding problems. For simple arithmetic operations (addition, subtraction, etc.), this inaccuracy is usually negligible, however for more complicated computations involving large amounts of arithmetic the magnitude of error can quickly accumulate and produce highly inaccurate results. One way of coping with this problem is to use interval arithmetic and work with intervals instead of single numbers; in this way, the end result will be a range within which the actual answer is guaranteed to lie. This is precisely the principle of verification: computing mathematically correct, verified bounds for the result, which compensates for the inherent inaccuracy of electronic computations. Adding verification to an existing algorithm is usually a trivial matter, since simply substituting intervals for numbers and regular for interval arithmetic is sufficient in a lot of cases (although such a simple conversion, albeit correct, will most probably not be optimal). As we shall see, however, certain situations require special handling which may lead to significant changes in the algorithm. We will assume that we can compute the verified eigenvalues of a real matrix as a single operation, since our implementation uses an existing library for verified matrix computations which allows this.

Some general notes

The following section, as well as the articles in which the original algorithms are detailed, only describe the process of finding an approximation for the upper bounds of the eigenvalue intervals of a symmetric interval matrix $\boldsymbol{A}^S$; this is due to the fact that an approximation for the lower bounds can be obtained by running a given algorithm for the upper bounds of $-\boldsymbol{A}^S$ and inverting them afterwards.

Observation. *Let $\boldsymbol{A}^S \in \mathbb{IR}^{n\times n}, n \in \mathbb{N}$, be a real symmetric interval matrix, $\boldsymbol{B}^S = -\boldsymbol{A}^S$, $i \in \{1, 2, ..., n\}$ and let $\omega \in \mathbb{R}$ be an upper outer approximation of the set $\Lambda_i(\boldsymbol{B}^S) = [\underline{\lambda}_i, \overline{\lambda}_i]$, i.e. $\omega \geq \overline{\lambda}_i$. Then $-\omega$ is a lower outer approximation of the set $\Lambda_{n-i+1}(\boldsymbol{A}^S)$.*

Proof *According to the definition, $\lambda \in \mathbb{R}$ is an eigenvalue of some matrix $A \in \mathbb{R}^{n\times n}, n \in \mathbb{N}$, if there exists some vector $x \neq 0$ such that $Ax = \lambda x$, which is true if and only if $-Ax = -\lambda x$. Therefore each eigenvalue λ of A corresponds to an eigenvalue $-\lambda$ of $-A$, and vice versa. Since for any two real numbers a and b it's true that $a \leq b \iff -b \leq -a$, and all eigenvalues of a symmetric matrix*

are real numbers, for A symmetric we can see that $\lambda_i(A) = -\lambda_{n-i+1}(-A)$ for $i = 1, 2, ..., n$.

It is also easy to see that a matrix B is a member of $\boldsymbol{B}^S$ if and only if $-B$ is a member of $\boldsymbol{A}^S$. This statement is fairly obvious, but if a formal proof is required, it is easy to obtain, for example, by noticing that $A_c = -B_c$, $A_\delta = B_\delta$ and expressing a given real matrix A belonging to $\boldsymbol{A}^S$ as the sum of A_c and a "distance matrix" D belonging to the interval $[-A_\delta, A_\delta]$; then $-A$ can also be expressed as the sum of $B_c = -A_c$ and $-D$, which is a valid "distance matrix" for $\boldsymbol{B}^S$.

Now suppose that $\omega > \overline{\lambda}_i$, which implies that $(\forall B \in \boldsymbol{B}^S)(\lambda_i(B) < \omega)$. Since $\boldsymbol{B} = -\boldsymbol{A}$, from the above remarks we can see that for every B from $\boldsymbol{B}^S$ it holds that $-\lambda(B)$ is the (n-i+1)-th eigenvalue of $A = -B$ and $-\omega \geq -\lambda(B)$. Since this is true for every B from $\boldsymbol{B}^S$ and therefore for every A from $\boldsymbol{A}^S$, we get that $-\omega$ is an outer approximation of the lower bound of $\Lambda_{n-i+1}(\boldsymbol{B}^S)$.

A similar observation can be made for the inner approximations.

Another thing worth noting is the method by which we will rate the performance of the presented algorithms. Since the verified computation of an outer eigenvalue approximation for a real matrix is considered a basic (unit) operation as mentioned above, and it is obviously much more computationally expensive than basic arithmetic and other simple operations, we can measure the complexity of a given algorithm relative to the complexity of the verified eigenvalue computation by counting (instead of the total number of all operations) the number of times that the verified eigenvalue operation is performed. As our practical experiments show, this is a very sensible rating, as the vast majority of computation time for any of the tested algorithms is spent within the method for real matrix eigenvalue computation, while only a relatively very small portion is reserved for the further processing of its results. Note that the unit operation always computes an approximation for all eigenvalues of a given matrix (in other words, it's not possible to look for bounds for individual eigenvalues in order to save time).

3.1 Simple bounds

The simplest and fastest method we use for approximating the bounds of the eigenvalues of a symmetric matrix is essentially a direct implementation of Theorem 2. While not technically an algorithm, it is convenient to take a closer look at the complexity of this procedure, since it is used extensively as a basic component of the more complex algorithms. Note that although the library for verified computations which we use does provide a specialized procedure for computing the spectral radius of a real matrix, we do not consider it to be a unit operation and in the following description of the algorithm we compute it "manually" from the definition. This is mainly due to two reasons: on one hand, we will have a much easier time establishing the computational complexity of a given algorithm if we only have one basic unit operation instead of two. Secondly, the difference in performance between our method and the existing implementation is negligible even for very large matrices, and the latter is neither consistently faster, nor

slower than the former.

Algorithm 1: Computing bounds by Rohn's theorem

input : A symmetric interval matrix $\mathbf{A}^S \in \mathbb{IR}^{n\times n}$
output: Verified outer approximations $[\underline{\omega}_i, \overline{\omega}_i]$ of the eigenvalue interval of $\mathbf{A}^S$

1 compute verified intervals $[\underline{\lambda}_i(A_c), \overline{\lambda}_i(A_c)]$ for $i = 1, 2, ..., n$;
2 compute verified intervals $[\underline{\lambda}_i(A_\delta), \overline{\lambda}_i(A_\delta)]$ for $i = 1, 2, ..., n$;
3 find $\rho(A_\delta) = \max_{i=1,2,...,n}\{\max(|\underline{\lambda}_i(A_\delta)|, |\overline{\lambda}_i(A_\delta)|\}$;
4 **for** *i = 1 to n* **do**
5 $\quad \underline{\omega}_i \leftarrow \underline{\lambda}_i(A_c) - \rho(A_\delta)$;
6 $\quad \overline{\omega}_i \leftarrow \overline{\lambda}_i(A_c) + \rho(A_\delta)$;

As we can easily see, the verified eigenvalue computation procedure is called exactly two times, regardless of the dimension of the input matrix; naturally, the actual running time still depends on the size of the matrix, since computing the verified eigenvalue intervals is not a constant time operation.

3.2 Direct interlacing algorithm

Originally appearing as Algorithm 1 in of a symmetric interval matrix. We remind the reader that the statement of the theorem basically implies that the eigenvalue bounds of the principal submatrices of $\mathbf{A}^S$ can be used as an outer approximation for the eigenvalue bounds of $\mathbf{A}^S$. Examining all principal submatrices, however, is too computationally expensive: for a symmetric matrix of dimension $n \in \mathbb{N}$ we would need to examine and compute the eigenvalues of $\sum_{k=1}^{n-1}\binom{n}{k} = 2^n - 2$, or $O(2^n)$ matrices; the algorithm uses a heuristic which examines only $O(n)$ matrices according to a certain criterion. Although appearing as a single algorithm in the original article, the whole procedure actually consists of two more or less independent parts, which we refer to as the forward and the reverse version of the algorithm. Both operate by finding approximations of the eigenvalue sets of principal submatrices of $\mathbf{A}^S$ restricted on the set of indices I; while the former starts with the whole set $I = \{1, 2, ..., n\}$ and iteratively removes indices from it until we are left with the empty matrix, the latter begins with $I = \emptyset$ and keeps adding indices until we get the whole $\mathbf{A}^S$. For maximum precision, both versions of the algorithms should be applied and the best results taken from each; if computation time is a concern, on the other hand, we can apply, for example, only the forward version of the algorithm; numerical experiments show that it

usually performs better in the general case.

Algorithm 2: Direct interlacing algorithm, forward version

input : A symmetric interval matrix $\boldsymbol{A}^S \in \mathbb{IR}^{n\times n}$
output: Verified outer approximations $[\lambda_i^u]_{i=1}^n$ of the upper bounds of $\boldsymbol{A}^S$

1 $\boldsymbol{B}^S \leftarrow \boldsymbol{A}^S$;
2 **for** k = *1,2,...,n* **do**
3 compute a verified upper bound $\overline{\lambda}$ for the largest eigenvalue of $\boldsymbol{B}^S$;
4 $\lambda_{n-k+1}^u \leftarrow \overline{\lambda}$;
5 select an index i from $\{1, 2, ..., (n-k+1)\}$;
6 remove the i-th row and column from $\boldsymbol{B}^S$

Algorithm 3: Direct interlacing algorithm, reverse version

input : A symmetric interval matrix $\boldsymbol{A}^S \in \mathbb{IR}^{n\times n}$
output: Verified outer approximations $[\lambda_i^u]_{i=1}^n$ of the upper bounds of $\boldsymbol{A}^S$

1 $I \leftarrow \emptyset$;
2 **for** k = *1,2,...,n* **do**
3 select an index i from $\{1, 2, ..., n\} \setminus I$;
4 $I \leftarrow I \cup \{i\}$;
5 let $\boldsymbol{B}^S$ be a submatrix of $\boldsymbol{A}^S$ obtained by removing the rows and columns from $\{1, 2, ..., n\} \setminus I$;
6 compute a verified upper bound $\overline{\lambda}$ for the largest eigenvalue of $\boldsymbol{B}^S$;
7 $\lambda_i^u \leftarrow \overline{\lambda}$

Two natural questions that arise from the description of the above algorithms are: how to approximate the bounds of the i-th eigenvalue of $\boldsymbol{B}^S$ in steps 3 (of the first) and 6 (of the second algorithm), and how to select the indices i in steps 5 (of the first) and 3 (of the second algorithm). The first problem simply involves using any known method for establishing an outer approximation of the bounds; this method will be called $O(n)$ times during the computation, therefore it should provide a certain balance between speed and precision. Theorem 2 or 3 can be used here. For optimal results, both methods (which are sufficiently fast) can be applied to a matrix and the best result taken (indeed, this is how our own implementation works).

Regardless of whether we use Rohn's theorem, Theorem 3 or both, we always need to compute no more than one verified upper bound during each iteration. Since for a given input matrix of dimension n the forward and reverse loop repeat exactly n times each, the number of eigenvalue computations that need to be performed is at most kn for some constant k (which can be exactly determined provided that we know the method used to find the upper bound).

As for the selection of the index i to remove or add, any sensible heuristic can be used; the description of the algorithm in the original article contains the following two helpful suggestions:

- $i = \arg \min_{j=1,2,...,(n-k+1)} \lambda_n^u(\boldsymbol{B}_j^S)$

- $i = \arg \min_{j=1,2,...,(n-k+1)} \sum_{r,s \neq j} |\boldsymbol{B}_{r,s}|$

The first method is fairly straightforward to understand: we select the index which provides the best possible result for the currently examined index. The second one utilises the fact that the square of the Frobenius norm (defined as $||A||_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |A_{ij}|}$ of a normal matrix is equal to the sum of the squares of its eigenvalues; therefore, attempting to minimise the norm by using the second criterion (which can be computed relatively quickly) should lead to smaller eigenvalues as well.

It should be noted that the first selection method is slower, but usually performs better in practice. This is the one that is used by default in our implementation of the algorithm.

Since these are only heuristics and we do not need to use verification in the implementation of such index selector methods, we can assume that this step adds no further complexity to the algorithm (in particular, the eigenvalues of the submatrices in the first selector do not need to be computed with verification).

3.3 Indirect interlacing algorithm

The main idea behind this algorithm is to express every matrix $A \in \boldsymbol{A}^S$ as $A = A_c + A_d$, $A_d \in [-A_\delta, A_\delta]^S$, find the eigenvalues of A_c and compute bounds for the eigenvalues of $[-A_\delta, A_\delta]^S$, after which we approximate the eigenvalues of $\boldsymbol{A}^S$ using Theorem 4.

Algorithm 4: Indirect interlacing algorithm

input : A symmetric interval matrix $\boldsymbol{A}^S \in \mathbb{IR}^{n \times n}$
output: Verified outer approximations $[\lambda_i^u]_{i=1}^n$ of the upper bounds of $\boldsymbol{A}^S$

1 Compute the eigenvalues $\lambda_i(A_c)$ for $i = 1, 2, ..., n$;
2 Compute upper outer bounds $\overline{\lambda}_i$ for $\Lambda_i([-A_\delta, A_\delta]^S)$;
3 **for** $k = 1$ *to* n **do**
4 $\quad \lambda_{n-k+1}^u = \min_{i \in \{1,2,...,k\}} \{\lambda_i(A_c) + \overline{\lambda}_{k-i+1}([-A_\delta, A_\delta]^S)$

Although the actual description does not specify any one method in step 2 (and, obviously, any valid approach can be used there), the eigenvalue bounds of $[-A_\delta, A_\delta]^S$ are meant to be computed by one or both versions of the direct interlacing algorithm, and this is precisely how our implementation functions.

Since we know that the direct version of the interlacing method performs kn verified eigenvalue computations, the above algorithm will increase this count only slightly to $kn + 1 \leq kn + n = (k+1)n = k'n$ due to the verified calculation of the eigenvalues of A_c in step 1.

Also note that we don't explicitly describe that the eigenvalues of A_c are computed with verification, and that we take the upper bounds of the resulting intervals for use with the algorithm (which is the worst possible case for the outer approximation) since this is done in the same trivial way in virtually all algorithms.

A note about the interlacing algorithms

In theory, as well as in practice, both algorithms performs similarly with respect to their running times; however, the quality of the output data varies from matrix to matrix, with the direct variant performing better for some inputs, and the indirect one providing tighter approximations for others. According to [3], the direct method is better suited to matrices without, or with relatively small "gaps" between their eigenvalue intervals, while the indirect one is to be preferred in the opposite case. The experiments performed in Section (6.1) confirm this empirical observation for interval matrices with their central matrix close to zero; however, the farther the centre of an interval matrix is away from the zero matrix, the less effective the direct method becomes, even if most of its eigenvalue intervals cover each other. Nonetheless, the indirect method appears to give better results in general, and it should therefore be used if no further information is known about the matrices to which it is applied.

3.4 Improving outer approximations by filtering

The filtering algorithm described in [5] utilizes Theorem 5 in order to eliminate parts of an outer approximation that can not possibly contain any eigenvalues. Since the theorem and its implications have already been discussed in some detail in section 2.1, we will now present the actual algorithm.

Algorithm 5: Filtering algorithm

input : An interval matrix $\boldsymbol{A} \in \mathbb{IR}^{n \times n}$, an interval $\boldsymbol{a} = [\underline{a}, \overline{a}]$, a maximal number of iterations T and a precision threshold ε

output: A potentially improved interval $\mathbf{b} \subseteq \boldsymbol{a}$

```
1  b ← a ;
2  t ← 0 ;
3  λ ← ε · rad b + 1 ;
4  while λ > ε · rad b  and t < T do
5      t ← t + 1 ;
6      M ← A − b̄I ;
7      compute Q = (M_c)^{-1} ;
8      λ ← (2 − ρ(|I − QM_c| + |I − QM_c|^T + |Q|M_δ + (M_δ)^T|Q|^T)) / ρ(|Q| + |Q|^T)
9      if λ > 0 then
10         b̄ ← b̄ − λ
11     if b̄ < b̲ then
12         b ← ∅ ;
13         return ;
```

Line 8: $\lambda \leftarrow \frac{2-\rho(|I-QM_c|+|I-QM_c|^T+|Q|M_\delta+(M_\delta)^T|Q|^T)}{\rho(|Q|+|Q|^T)}$

The precision (ε) and maximal step count (T) parameters limit the number of iterations that the algorithm can go through before ending; the former keeps the procedure from wasting computation time on insignificant improvements, while the latter prevents looping. In practice, however, the algorithm usually finishes after only a small number of steps.

Another explicit condition for terminating the filtering process is the second

"if" clause, which indicates that the input interval does not contain any eigenvalues at all and has been wholly "filtered out".

The algorithm follows Theorem 7 and our previous discussion very closely. One difference is that Q, which was an arbitrary matrix in the theorem, is taken here to be the inverse of M_c. It is important to stress that we do not actually need the inverse matrix, and neither does the computation have to be exact or even accurate. A numerical computation of the inverse matrix can be used here, mainly for speed and convenience.

Note that the interval $\boldsymbol{a}$ does not necessarily have to be an outer approximation of an eigenvalue set of $\boldsymbol{A}$; however, it usually makes the most sense to filter precisely such sets. Approximations for the individual sets can be filtered separately for better results, since the algorithm does not account for gaps within the filtered set by itself.

Concerning the computational complexity, the only important step is number 8, which requires the verified computation of two spectral radii. As discussed previously in 3.1, the calculation of the spectral radius of a real matrix is computationally equivalent to finding all of its eigenvalues. A simple upper bound for the number of iterations of the loop is, of course, the number T, which means that there will be at most $2T$ verified eigenvalue computations; if we assume that T is a fixed value, then the number of iterations is bounded by a constant. This algorithm is difficult to analyse, however, since its calculation time is affected by many different factors: the nature of the filtered set, its quality as an approximation and the values of ε and T.

3.4.1 Speeding up the filtering algorithm

Practically all of the time spent by the filtering algorithm is devoted to computing the two spectral radii, which (in theory and in practice) take as much time to compute as the verified eigenvalue intervals of the corresponding matrices. Therefore, we can dramatically reduce the time required by the algorithm by substituting these exact verified computations with suitable upper approximations; it can be easily seen that such a modification will preserve the correctness of the algorithm (although the improvement we can achieve by using them will not be as good).

It is a known fact that, given a certain matrix norm $||\cdot||$, the spectral radius of a given matrix $\mathbb{A}$ cannot possibly be greater than its norm $||\mathbb{A}||$. In particular, the norms $||\mathbb{A}||_1$ (maximal column sum) and $||\mathbb{A}||_\infty$ (maximal row sum) can be computed very quickly and without having to perform any complicated operations (such as finding verified eigenvalue bounds); in fact, the time it takes to find such an approximation is practically negligible when compared to finding the verified spectral radius. This also means that we can freely utilise both norms and take the better result as an approximation.

Since there are two spectral radii which need to be computed (or approximated) in the filtering algorithm, we examine three different variations: we can compute the first spectral radius (in the numerator) and approximate the second, we can approximate the first and compute the second, or we can approximate both. Intuitively, the first two methods should have similar running times, while the last one will be very quick indeed, but will be much less accurate.

As it turns out, the first variant we mentioned (computing the spectral radius

in the numerator and approximating the one in the denominator) is indeed very useful in practice, as it reduces the accuracy of the result only slightly in the general case, but the computation time is almost two times shorter than that of the original algorithm.

The last (and least accurate variant) performs, as can be expected, quite poorly with respect to precision, but still does manage to tighten the bounds a little in some cases, and it is so fast that it can be used alongside any algorithm for outer approximations without noticeably increasing the computation time.

The third variant (approximate the spectral radius in the numerator and compute the one in the denominator) takes as much time to finish as the first, but provides results comparable to the last one; therefore, we do not use it in our implementation.

A detailed description of the experiments we conducted with the different variations of the filtering method can be found in Section 6.2.

3.5 Local improvement algorithm

All algorithms that we use for inner approximations of the eigenvalue bounds originate from [4]. The simplest of these involves examining some of the extremal matrices of $\boldsymbol{A}^S$ and potentially improving the constructed inner approximation with them; as was already mentioned, every eigenvalue of a matrix from $\boldsymbol{A}^S$ can serve as an inner approximation. Beginning with A_c, the sign vector corresponding to the eigenvector of each matrix defines the next one, until no improvement is possible and the algorithm ends.

Algorithm 6: Local improvement for inner bounds

input : A symmetric interval matrix $\boldsymbol{A}^S \in \mathbb{IR}^{n \times n}$ and an index $i \in \{1, 2, ..., n\}$

output: An upper inner approximation $\overline{\mu}_i$

1 $\overline{\mu}_i \leftarrow -\infty$;
2 $A \leftarrow A_c$;
3 compute a verified interval $[\underline{\lambda}_i, \overline{\lambda}_i]$ for the i-th eigenvalue of A and a (possibly non-verified) vector interval $v_i = [\underline{v}_i, \overline{v}_i]$ for the corresponding eigenvector ;
4 $\tilde{v} \leftarrow (0, 0, ..., 0)^T$;
5 **while** $\underline{\lambda}_i > \overline{\mu}_i$ **do**
6 $\quad \overline{\mu}_i \leftarrow \underline{\lambda}_i$;
7 $\quad s \leftarrow \operatorname{sign}(\operatorname{mid} v_i)$;
8 $\quad$ **if** $s = \tilde{v}$ **then**
9 $\quad\quad$ return ;
10 $\quad \tilde{v} \leftarrow s$;
11 $\quad D \leftarrow \operatorname{diag}(s)$;
12 $\quad A \leftarrow A_c + D A_\delta D$;
13 $\quad$ compute a verified interval $[\underline{\lambda}_i, \overline{\lambda}_i]$ for the i-th eigenvalue of A and a vector interval $[\underline{v}_i, \overline{v}_i]$ for the corresponding eigenvector ;

The algorithm above describes the computation of the inner bound for a single index i; to approximate the bounds of all eigenvalues of a given matrix, it is enough to run the procedure once for each of the indices. Obviously, the eigenvalues of a certain extremal matrix can potentially be computed several times, which would be a rather significant waste of CPU time, especially for larger matrices. This can be prevented by sacrificing a bit of memory in order to store the eigenvalues of already visited matrices; the sign vectors used for selecting extremal matrices of $\boldsymbol{A}^S$ can serve as a very effective hashing function in this case.

The eigenvalues of A_c can (and should) always be computed only once, since they are certainly used in every run of the algorithm.

As getting the same sign vector automatically implies getting the same extremal matrix (in which case we cannot possibly improve the existing bounds), it makes sense to remember the previously computed sign vector $\tilde{v}$ and compare it to the current one, prematurely ending the computation. This saves us a single verified eigenvalue computation, which, of course, does not improve the overall complexity of the algorithm, but can still provide a small boost in performance for larger matrices. This approach can be taken a step further by hashing the sign vectors of all visited matrices.

Step 6 of the algorithm is an example of a situation in which there is a noticeable difference between the logic of the verified and the non-verified version. Although in most cases it will be true that the sign vectors of $\underline{v}_i$ and $\overline{v}_i$ (and therefore all vectors in between them as well) are the same, in the general case it can happen that the lower bound for one of the interval vector's elements is negative and the upper is positive. In this particular case, this does not require any significant modification of the algorithm, as the sign vectors are nothing more than a heuristic for selecting specific real matrices from $\boldsymbol{A}^S$; essentially, we could select random extremal matrices and the result will still be a correct inner approximation (this also means that we do not have to find a verified interval for the eigenvector in step 3, and could instead use non-verified methods which will yield a single real vector; nonetheless, the actual functions that we use in our implementation compute verified intervals for both the eigenvalues and the eigenvectors of the input matrix at the same time, and performance would only suffer if we were to ignore the found eigenvector interval and run a separate non-verified method; furthermore, any real vector can be represented by an interval with the same lower and upper bound, and this is the reason that the above algorithm assumes an interval vector). Nonetheless, we need to select a particular real vector from $\boldsymbol{v_i}$, and $\mathrm{mid}\, \boldsymbol{v_i}$ is certainly no worse than any other choice; intuitively, it will also tend to be closer to the "true" sign of the eigenvector in the case of a disparity between the signs of the lower and upper bounds.

Since the algorithm potentially involves computing the eigenvalues of all extremal matrices of $\boldsymbol{A}^S$, each one corresponding to a different sign vector of the form $\{\pm 1\}^n$, we can immediately claim that there will be no more than 2^n verified computations, a figure which can be reduced to 2^{n-1} if we account for the fact that the first element of any eigenvector can be assumed to be positive due to normalisation, but which is still quite large, implying exponential complexity. In practice, the algorithm usually terminates after only a few iterations. A limit for the total number of iterations, similar to the one in the filtering algorithm, can be added to ensure a constant number of iterations in theory as well.

3.6 Vertex enumeration algorithm

The vertex enumeration algorithm is very similar to the local improvement described above in that it examines extremal matrices of $\boldsymbol{A}^S$, indexed (or "enumerated") by sign vectors. The difference is that all such matrices are examined, which will usually provide a better approximation and require more computation time. The asymptotic complexity of both algorithms is evidently the same, but vertex enumeration is slower and more accurate in practice. Furthermore, according to Theorem 9, the lower bound for the smallest eigenvalue interval and the upper bound for the largest one are exact.

Algorithm 7: Vertex enumeration for inner bounds

input : A symmetric interval matrix $\boldsymbol{A}^S \in \mathbb{IR}^{n\times n}$ and an index $i \in \{1, 2, ..., n\}$
output: An upper inner approximation $\overline{\mu}_i$

1 compute verified intervals $[\underline{\lambda}_i, \overline{\lambda}_i]$ for the eigenvalues of A_c ;
2 **for** $i = 1, 2, ..., n$ **do**
3 $\quad \overline{\mu}_i \leftarrow \underline{\lambda}_i$
4 **for** $z \in \{\pm 1\}^n, z_1 = 1$ **do**
5 $\quad A \leftarrow A_c + \mathrm{diag}(z) A_\delta \mathrm{diag}(z)$;
6 $\quad$ compute verified intervals $[\underline{\lambda}_i, \overline{\lambda}_i]$ for the eigenvalues of A ;
7 $\quad$ **for** $i = 1,2,...,n$ **do**
8 $\quad\quad \overline{\mu}_i \leftarrow \max\{\overline{\mu}_i, \underline{\lambda}_i\}$;

Note that all extremal matrices of $\boldsymbol{A}^S$ are examined for all eigenvalues, which means that all inner approximations can be computed with a single run of the algorithm (this is not true for Algorithm 6, for example, where different sets of extremal matrices are examined for different eigenvalues). Since it holds that for $(\forall z \in \{\pm 1\}^n)(\mathrm{diag}(z) A_\delta \mathrm{diag}(z) = \mathrm{diag}(-z) A_\delta \mathrm{diag}(-z))$, we can assume that the first component of z is positive (equal to 1) and therefore examine only 2^{n-1} matrices.

A common problem with both the local improvement and the vertex enumeration algorithms which arises due to the addition of verification, is that they might sometime produce improper intervals as approximations (in other words, pairs $(\underline{\mu}_i, \overline{\mu}_i)$ such that $\underline{\mu}_i > \overline{\mu}_i$). As can be seen from the descriptions of the above algorithms, we always take the lower bounds of verified eigenvalues when improving upper inner bounds so that we have a verified interval in the end. As an initial approximation we use the eigenvalues of the central matrix A_c. The initial approximation for $\overline{\mu}_i$ is therefore $\underline{\lambda}_i$, and for $\underline{\mu}_i$ it is $\overline{\lambda}_i$, which obviously does not define a correct interval. As the upper approximation is improved, it increases and, conversely, the lower approximation decreases. In the majority of cases the improvement is sufficient for the two numbers to define a valid interval; however, there is no guarantee that this will happen, and, in theory as well as in practice, the algorithms can produce improper intervals for the inner approximation, which essentially means that no inner approximation could be found. Note that this is impossible to happen in the non-verified versions of the algorithms, which begin with $\underline{\mu}_i \lambda_i(A_c) = \overline{\mu}_i$ and thus $\underline{\mu}_i$ and $\overline{\mu}_i$ always define a correct (albeit initially trivial) real interval.

One of the issues pertaining to the vertex enumeration algorithm's implementation is the second algorithm (steps 4 through 8) which requires all vectors from $\{\pm 1\}^n$ for a given n to be generated. Although hardly a difficult problem, it is still not entirely trivial, as a simple and efficient way of iteratively generating all such vectors can significantly improve the readability and effectiveness of the algorithm. A simple method for doing this is described by Rohn in section 2.2 of [9], which we use in our implementation of the vertex enumeration algorithm.

Algorithm 8: Generating all $z \in \{\pm 1\}^n$

```
1 z ← (1, 1, ..., 1)^T ;
2 v ← (0, 0, ..., 0)^T ;
3 while v ≠ (1, 1, ..., 1)^T do
4     k ← min{i : v_i = 0} ;
5     for i = 1, 2, ...(k − 1) do
6         v_i ← 0 ;
7     v_k ← 1 ;
8     z_k ← −z_k ;
9     process z ;
```

The original text describing the algorithm also contains a proof of its correctness, however its operation is rather intuitive as it is: we start with z being a vector with ones at all positions(although the procedure works for any starting vector from the set $\{\pm 1\}^n$) and the auxiliary vector v keeps track of the "inversions" which we have already processed. All inversions of the components with indices $l+1, ..., n$ are processed before inverting the component at position l; once no possible inversions remain, the algorithm terminates.

The last step of the algorithm consists of "processing" the vector v. This basically means that at this point we do what we need to with the vector; in the case of the vertex enumeration method, for example, we find the extremal matrix defined by it and proceed to make potential improvements to the existing inner approximation. In practice, the while loop of Algorithm 8 replaces the for loop on line 4 of Algorithm 7, and the "process z" template is replaced by the body of the algorithm (lines 5 through 8). Generating all vectors of a given length n with a fixed first element is as simple as generating all vectors of length $n-1$ and prepending a single 1.

3.7 Submatrix vertex enumeration

Easily the most complicated method that we use for computing inner approximations, submatrix vertex enumeration makes use of Theorem 8 and actually allows us to compute exact bounds in certain special cases. However, introducing verification to this algorithm results in some complications, as we shall see below.

We will recall that the theorem basically states that eigenvalues belonging to the boundary of $\Lambda(\boldsymbol{A}^S)$ are eigenvalues of some extremal matrix of a principal submatrix of $\boldsymbol{A}^S$. An important observation is that not all eigenvalues of such matrices are eigenvalues of $\boldsymbol{A}^S$ as well; a very simple example confirming this fact can be practically any trivial (with $A_\delta = 0$, i.e. containing only a single real matrix and therefore being practically equivalent to it) interval matrix, e.g.

$\begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix}$ with eigenvalues $\lambda_1 \approx -0.2361$ and $\lambda_2 \approx 4.2361$, but with the single eigenvalue of any of its principal submatrices being a whole number. The original algorithm describes a condition which can be used to skip over useless (not leading to eigenvalues of $\boldsymbol{A}^S$) matrices: having chosen a principal submatrix $\boldsymbol{D}^S$ of $\boldsymbol{A}^S$, we can assume that $\boldsymbol{A}^S = \begin{pmatrix} \boldsymbol{B}^S & \boldsymbol{C} \\ \boldsymbol{C}^T & \boldsymbol{D}^S \end{pmatrix}$. Then, having found some eigenvalue λ and the corresponding eigenvector y of some matrix $D \in \boldsymbol{D}^S$, we check whether there is some $C \in \boldsymbol{C}$ fulfilling $Cy = 0$, which is a necessary and sufficient condition for λ to be an eigenvalue of $\boldsymbol{A}^S$. Provided a given number-vector pair does not fail this test, further conditions are examined to decide whether λ is an element of $\Lambda_i(\boldsymbol{A}^S)$ and whether Λ_i and Λ_{i+1} can be guaranteed to be disjoint, in which case λ is proclaimed to be the exact upper bound of $\Lambda_i(\boldsymbol{A}^S)$, and the corresponding eigenvalue of some $A \in \boldsymbol{A}^S$ with D as a principal submatrix and $Cy = 0$ is used for potential improvement of the inner bound.

The original (non-verified) algorithm from [4] is presented here for reference:

Algorithm 9: Direct submatrix vertex enumeration algorithm, unverified

input : A symmetric interval matrix $\boldsymbol{A}^S \in \mathbb{IR}^{n\times n}$ and an index $p \in \{1, 2, ..., n\}$
output: An upper inner approximation $\overline{\mu}_p$

```
1  compute outer approximations [ω̲_i, ω̄_i] for the eigenvalues of A^S ;
2  compute inner approximations [μ̲_i, μ̄_i] for the eigenvalues of A^S ;
3  for each non-empty set of indices J ⊆ {1, 2, ..., n} do
4      decompose A^S = ( B^S  C ; C^T  D^S ) such that D^S is the principal
       submatrix of A^S obtained by restricting A^S to the indices from J ;
5      for z ∈ {±1}^|J|, z_1 = 1 do
6          D ← D_c + diag(z) D_δ diag(z) ;
7          for i = 1,2,...,n do
8              λ ← λ_i(D) ;
9              y ← y_i(D) ;
10             if λ > μ̄_p & λ ≤ ω̄_p & 0 ∈ C y then
11                 if p = n ∨ λ < ω̲_{p+1} then
12                     μ̄_p ← λ ;
13                 else
14                     take a C ∈ C fulfilling Cy = 0 ;
15                     A ← ( B_c  C ; C^T  D ) ;
16                     λ ← λ_p(A) ;
17                     if λ > μ̄_p then
18                         μ̄_p ← λ ;
```

Unfortunately, attempting to introduce verification to the algorithm results in some serious complications which make the computation of exact bounds in this way seemingly impossible.

In a verified version of the procedure, the computations in steps 8 and 9 should produce a real interval $\boldsymbol{\lambda} = [\underline{\lambda}, \overline{\lambda}]$ and an interval vector $\boldsymbol{y} = [\underline{y}, \overline{y}]$ instead of a single number λ and a vector y. The main problem arises from the third condition of the "if" clause in step 10, which now reads $0 \in \boldsymbol{C}\boldsymbol{y}$; since solving such an interval equation is very hard, it was considered selecting a single vector $y \in \boldsymbol{y}$, e.g. $\operatorname{mid} \boldsymbol{y}$ to use as a heuristic and solving $0 \in \boldsymbol{C}y$, but this approach would, to the best of our knowledge, not yield verified results, since we cannot guarantee that the vector y solving $0 \in \boldsymbol{C}y$ is the same as the eigenvector of D corresponding to λ. Theoretically, we could instead attempt to prove in some other way that λ is indeed an eigenvalue of $\boldsymbol{A}^S$, but this would be equivalent to providing an inner approximation with λ as the upper bound; and if our candidate λ corresponds to the value of the exact upper bound of an eigenvalue interval (and that interval is disjoint with the next one, since that is one of the conditions for applying the theorem), then we will never be able to find any eigenvalue of $\boldsymbol{A}^S$ within that interval larger than λ.

A verified version of the algorithm can be constructed by applying steps 14 through 18 for all examined eigenvalue-vector pairs, which will obviously yield correct approximations, but the possibility of computing exact bounds, which is one of the strongest points of the original algorithm, will be lost. What is left will still be a useful algorithm, and the most accurate among our inner approximation procedures, as it can be expected that it will provide us with results equal or, at least, very close to the exact bounds in the case of disjoint eigenvalue intervals.

The complexity of the algorithm is still exponential, though, and therefore it is useful to consider the branch and bound improvement described in the original article. The basic idea involves checking the solvability of the system

$$\boldsymbol{C}y = 0, (\boldsymbol{D}^S - \boldsymbol{\lambda} I)y = 0, \|y\|_\infty = 1 \tag{3.1}$$

for y; here $\boldsymbol{\lambda} = [\overline{\mu}_p, \overline{\omega}_p]$ is taken as the possible range in which a potential improvement of the inner approximation can lie. As the article shows, should the above system not have a solution for some decomposition $\boldsymbol{A}^S = \begin{pmatrix} \boldsymbol{B}^S & \boldsymbol{C} \\ \boldsymbol{C}^T & \boldsymbol{D}^S \end{pmatrix}$ of $\boldsymbol{A}^S$ for a set of indices $J \subseteq \{1, 2, ..., n\}$, we can safely not only skip the current set J, but all of its subsets as well. Provided that we have an efficient way of solving the system and proceed from larger to smaller subsets of $\{1, 2, ..., n\}$, this approach can potentially lead to a great improvement in speed.

A method for solving equations of the type $Cy = 0, C \in \boldsymbol{C}$ for C is implicitly presented in the proof of the Oettli-Prager theorem in [9]; the theorem itself basically states that, given the interval system $\boldsymbol{A}x = \boldsymbol{b}$, where $\boldsymbol{A}$ is an interval matrix and $\boldsymbol{b}$ is an interval vector, there exists a matrix $A \in \boldsymbol{A}$ and a vector $b \in \boldsymbol{b}$ such that $Ax = b$ if and only if $|A_c x - b_c| \leq A_\delta |x| + b_\delta$; in other words, it characterises so called "weak solutions" of an interval system. The proof of the theorem is constructive, being based on finding a matrix and a vector for which x solves the corresponding system, and the approach can be used in our case as

well. The following algorithm describes how to solve $Ax = 0$ for $A \in \boldsymbol{A}$.

Algorithm 10: Solving $Ax = 0$

input : An interval matrix $\boldsymbol{A} \in \mathbb{IR}^{m \times n}$ and a vector $x \in \mathbb{R}^n$
output: A real matrix $A \in \boldsymbol{A}$ such that $Ax = 0$, if it exists, or failure

```
1  if |A_c x| > A_δ|x| then
2      return failure ;
3  ξ ← (A_δ|x|) ;
4  τ ← (A_c x) ;
5  for i = 1, 2, ..., m do
6      if ξ_i > 0 then
7          ρ_i ← τ_i / ξ_i ;
8      else
9          ρ_i ← 1 ;
10 z ← sign(y) ;
11 A ← A_c − diag(ρ) A_δ diag(z) ;
```

Taking into account the above discussion, we present our verified version of the direct submatrix vertex enumeration algorithm which utilises the branch and bound improvement (Algorithm 11 on the next page).

The algorithm remains mostly the same, with the exception that we always need to solve $Cy = 0$ and construct the matrix A (guaranteed to be an element of $\boldsymbol{A}^S$) so we can ensure that the eigenvalues we find are eigenvalues of $\boldsymbol{A}^S$. As was discussed above, this not only results in a slower running time, but also means that we can no longer find exact bounds of the eigenvalue intervals. Since the eigenvalues and eigenvectors computed in steps 14-15 are never used for improving the actual approximation, they do not need to be computed with verification; the eigenvalue of A on line 20, on the other hand, needs to be verified.

The branch and bound modification introduces several new elements to the algorithm as well. Firstly, we use the set Ω to keep track of useless sets of indices, i.e. those that cannot improve the existing approximation. The specific implementation can be done in a variety of ways; in our implementation we use an array of logic values and a bijective function mapping subsets of $\{1, 2, ..., n\}$ to natural numbers (used as indices) and vice-versa. This allows us to quickly determine whether a given set is useless, but requires 2^n bits of memory; should this prove to be a problem, one could replace the array with a dynamic data structure which would only contain the maximal (with respect to inclusion) useless sets; in this case some additional computation will be required in order to determine whether a given set of indices belongs to Ω.

A different complication arises due to the fact that a certain set of indices is useless or useful only with respect to a certain eigenvalue, i.e. to a certain index p. While the simpler variant of the algorithm (without branch and bound) can be run for all eigenvalues at the same time, the version presented above requires us to either run it separately for p from 1 to n (which means we would have to compute the eigenvalues of some specific matrices at line 19 several times during successive runs of the algorithm) or use extra memory to either keep track of useless index sets for each eigenvalue individually or, alternatively, to store the

Algorithm 11: Direct submatrix vertex enumeration algorithm, verified, with branch and bound

input : A symmetric interval matrix $\boldsymbol{A}^S \in \mathbb{IR}^{n\times n}$ and an index $p \in \{1, 2, ..., n\}$

output: An upper inner approximation $\overline{\mu}_p$

1 compute outer approximations $[\underline{\omega}_i, \overline{\omega}_i]$ for the eigenvalues of $\boldsymbol{A}^S$;

2 compute inner approximations $[\underline{\mu}_i, \overline{\mu}_i]$ for the eigenvalues of $\boldsymbol{A}^S$;

3 $\Omega \leftarrow \emptyset$;

4 **for** *each non-empty set of indices* $J \subseteq \{1, 2, ..., n\}$ **do**

5 **if** $J \notin \Omega$ **then**

6 decompose $\boldsymbol{A}^S = \begin{pmatrix} \boldsymbol{B}^S & \boldsymbol{C} \\ \boldsymbol{C}^T & \boldsymbol{D}^S \end{pmatrix}$ such that $\boldsymbol{D}^S$ is the principal submatrix of $\boldsymbol{A}^S$ obtained by restricting $\boldsymbol{A}^S$ to the indices from J ;

7 attempt to prove that the system $\mathbf{C}y = 0, (\mathbf{D}^S - \lambda I)y = 0, ||y||_\infty = 1$ does not have a solution ;

8 **if** *the system is guaranteed to be unsolvable* **then**

9 $\Omega \leftarrow \Omega \cup 2^J$;

10 **else**

11 **for** $z \in \{\pm 1\}^{|J|}, z_1 = 1$ **do**

12 $D \leftarrow D_c + \text{diag}(z) D_\delta \text{diag}(z)$;

13 **for** $i = 1, 2, ..., n$ **do**

14 $\lambda \leftarrow \lambda_i(D)$;

15 $y \leftarrow y_i(D)$;

16 **if** $\lambda > \overline{\mu}_p$ & $\lambda \leq \overline{\omega}_p$ **then**

17 attempt to solve $Cy = 0$ for $C \in \mathbf{C}$;

18 **if** *a solution exists* **then**

19 $A \leftarrow \begin{pmatrix} B_c & C \\ C^T & D \end{pmatrix}$;

20 compute a verified lower bound $\underline{\lambda}_p$ for the p-th eigenvalue of A ;

21 **if** $\underline{\lambda}_p > \overline{\mu}_p$ **then**

22 $\overline{\mu}_p \leftarrow \underline{\lambda}_p$;

computed eigenvalues of already processed matrices so that they do not have to be computed again.

Checking the necessary condition for processing a set of indices at line 7 is also not explicitly described, as it is not a part of the actual algorithm. In our implementation, we utilise an existing function for testing the unsolvability of an interval system. Since we consider $||y||_\infty = 1$, we use the following approach for examining a system $\boldsymbol{A}x = 0$:

Algorithm 12: Testing unsolvability of an interval system

---

input : An interval matrix $\boldsymbol{A} \in \mathbb{IR}^{m \times n}$ and an interval vector $\boldsymbol{x} \in \mathbb{IR}^n$ representing the interval system $\boldsymbol{Ax} = 0$

output: True if $\boldsymbol{Ax} = 0$ can be guaranteed to be unsolvable, otherwise False

1 *unsolvable* $\leftarrow 1$;
2 $\xi \leftarrow 1$;
3 **while** $\xi < m$ **do**
4     $\boldsymbol{y} \leftarrow x$;
5   **for** $i < \xi$ **do**
6         $y_i \leftarrow 0$;
7     $y_\xi \leftarrow 1$;
8   **if** $\boldsymbol{Ay} = 0$ *cannot be proven to be unsolvable* **then**
9   *unsolvable* $\leftarrow 0$;
10   break ;
11    $\xi \leftarrow \xi + 1$;
12 **if** *unsolvable* $= 0$ **then**
13   return False ;
14 **else**
15   return True ;

The method presented above is rather straightforward: we replace the i-th component of the vector $\boldsymbol{x}$ with a 1, and attempt to prove unsolvability. If we fail, this means that a solution may exist; otherwise, we proceed to the next component.

Substituting zero for all components with indices less than i is actually a rather important step, as it significantly increase the speed of the algorithm; we have not performed any specific tests, but the increase in performance is definitely noticeable. Such an improvement is correct, since if a solution existed with $a_j \neq 0$ for some $j < i$, then we couldn't possibly have proven that no solution exists for j, and the loop would have terminated during an earlier iteration.

Although, as we mentioned above, the condition allowing us to compute exact bounds no longer holds for our verified version of the algorithm, an alternative, albeit weaker variant, is still plausible. Let us recall that, according to Theorem 8, the direct submatrix vertex enumeration algorithm will certainly encounter the exact bounds of disjoint eigenvalue intervals; the reason that we cannot use them

in the verified version is that we cannot assert that a given eigenvalue of a certain matrix $\mathbf{D}^S$ (from the decomposition) is also an eigenvalue of $\boldsymbol{A}^S$. Nonetheless, it might happen that only a single eigenvalue interval is found that has a non-empty intersection with a given "improvable" interval; this then necessarily means that the eigenvalue in question is the exact bound (provided, of course, that the outer approximation does not intersect its "neighbour"). Essentially, this means that the number of potential "candidates" needs to be counted for each eigenvalue interval; an algorithm based on this idea is not difficult to implement, but it will find exact bounds only rarely. The eigenvalues of D at step 14 need to be verified in this case. Such an algorithm could be implemented separately (looking only for exact bounds) or as an additional component of the method presented above.

We have created a basic implementation of this technique, but it hasn't been thoroughly tested yet as we do not consider it to be particularly perspective.

4. Implementation details

The current chapter describes the source files implementing the functionality of the algorithms described in the previous parts of the thesis, as well as their interconnections, in some detail. The source files themselves contain comments and remarks which should be sufficient to explain their stand-alone functionality; therefore, we will concentrate more on the way the individual procedures operate as part of a more complex system.

The main goal of this section is to familiarise the reader with the different computation options offered by individual implementations; input parameters that we consider to be especially important (for example, those that select different versions of the algorithm or significantly affect its behaviour) are written in bold.

Warning: The functions have been implemented and tested only on Matlab R2013b under Windows 7 and Windows 8. Hopefully, they will work without any problem for different versions of Matlab and on different operating systems, but no guarantee is given as to their functionality in this case.

A note on the programming environment

All algorithms and auxiliary procedures are implemented in the Matlab programming language, using the Intlab package ([11]) for interval computations along with the Versoft library ([7]) which provides functions for the verified computation of different values, including the eigenvalues and spectral radius of real matrices. Versoft itself requires Intlab in order to function properly, since it works with intervals.

We assume that the reader is familiar, at least on a basic level, with the Matlab programming language; if this is not the case, a multitude of tutorials can be found on-line which can serve as a quick introduction. Nonetheless, Matlab is fairly simple and intuitive, with a syntax closely resembling any modern object-oriented programming language, and it is our belief that the source files can be understood even without studying any such materials.

The most important and frequently used function from Versoft is *vereig*, which computes verified intervals for the eigenvalues and eigenvectors of a given real matrix. It is used in the form

[L,X] = vereig(A)

with a given real matrix A, and returns interval matrices L and X such that $L(i,i)$ is a verified interval for the i-th eigenvalue of A and $X(:,i)$ is a verified interval vector for the eigenvector corresponding to the i-th eigenvalue.

We also make use of *verspectrad* for verified computation of the spectral radius of a matrix.

For checking the necessary condition for skipping an index set in Algorithm 11 we utilise the functions *ilssolvable* and *ilsunsolvablefcr* by Jaroslav Horáček; unfortunately, they do not appear to have been published at the time of writing.

General notes

As was already mentioned in the chapter dealing with the algorithms we use, it is enough to know an approach for computing an approximation of the upper bounds of the eigenvalue intervals of a given matrix $\boldsymbol{A}$, since the lower bounds can be approximated by finding the upper bound approximations of $-\boldsymbol{A}$ and inverting them. As we shall very soon see, this is not just a theoretical "trick", but is used extensively in practice as well: virtually all of our implementations consist of a basic realisation of the corresponding algorithm for computation of the upper bounds (filename ends with "upper") and a wrapper method (without "upper") which simply calls the upper bounds version for a given matrix and its negation and combines the results. For example, the function which applies Algorithms 2 and 3 for an outer approximation of the eigenvalue bounds is called *eigsymencouterdirect* (see the naming conventions for more details); the whole procedure consists of calling the actual implementation *eigsymencouterdirectsingleupper* for upper bounds twice for each eigenvalue index and constructing the verified intervals. In some cases additional handling is required; unless this is the case, we will assume that the operation of the wrapper procedure is clear and will not describe it explicitly.

We've generally tried making the whole project as modular as possible, taking care to separate individual parts of algorithms into stand-alone functions, so that their functionality can be easily improved or replaced without having to modify the main method. This is the reason for the large number of auxiliary methods in the project. In addition to this, some of the procedures contain optional input parameters that allow a specific function to be used for a certain purpose instead of the default one. Nonetheless, modification of any larger algorithm should be possible, including those that do not include such parameters in their implementations.

Naming convention

All of our procedures follow a certain naming convention, and therefore we should take a minute to clarify how the names of individual functions are composed. Virtually all important procedures begin with the prefix "eigsymenc", which stands for "eigenvalues of symmetric matrices, encapsulation".

This can be followed by either "outer" or "inner" depending on the type of approximation that the given method is computing; if neither word is present in the filename, it implies that both approximations are computed or that the function in question has a more general purpose.

Next is a keyword referring to the algorithm used; for example, "direct" refers to the direct interlacing method, while "locimp" stands for "local improvement", as in the procedure of the same name.

Auxiliary functions closely tied to some method will usually share the same prefix ("eigsym", type of approximation and algorithm keyword) followed by a sort of verbal description of the specific function that is being implemented; for example, the *eigsymencouterindirectsingleupper* function computes an upper approximation using the indirect interlacing approach for a single eigenvalue. Practically all actual algorithm implementations (as opposed to wrapper methods)

are named in this way.

Certain auxiliary methods might not completely adhere to the conventions described above, especially if they are more general, with multiple uses within our project or even potential application outside of it.

4.1 Outer bounds

4.1.1 Simple methods

A direct implementation of Theorem 2 is provided in the function *eigsymencouter-rohn*. Optional input parameters allow pre-computed eigenvalues for the central and radial matrices and the spectral radius for the radial matrix to be passed to the function, which can save some computation time if it is called from within a more complex procedure which has already computed (or is planning to compute) one or more of these values as part of a different method. This option is not used in our implementation, as Rohn's theorem is always utilised as a first basic approximation before any other method is used. An additional output parameter, added solely for convenience, contains the bound for the largest eigenvalue. This is due to the possibility that some algorithms might require such information.

4.1.2 Methods applying the direct interlacing procedure

The basic implementation of the algorithm is realised as *eigsymencouterdirects-ingleupper*, which computes outer approximations for the upper bound of a single eigenvalue interval. The function contains quite a few parameters, as it allows for various modes of computation.

If we look at the original algorithms (2 and 3), we will see that the eigenvalues are meant to be computed sequentially (i.e. the first one, then the second one, then the third one, etc.) and that we require information about all previously computed eigenvalues (namely, the indices that were removed or added by the forward or reverse version of the algorithm for them) in order to compute the next one. The **indexselectionstrategy** parameter defines what happens if we call the function for a given eigenvalue and no (or only partial) information about the previously removed indices is available. If we set this to 'STEP', then the algorithm will be run (but without verification) for all missing indices in order to obtain the necessary data (this is the default as it is the fastest); 'STEPVER' is the same, but the bounds for the missing eigenvalues are computed with verification and returned as a "by-product" of the algorithm, which can potentially allow for the improvement of an existing approximation (it is assumed that the information about the previous indices is missing because the bounds for those eigenvalues were computed by the indirect method instead of the direct one); in order not to waste any time, this existing approximation can be provided as the input argument *eb*; then a verified computation will occur only for those indices where there is a real possibility of improvement. Finally, 'ALL' examines all possible combinations of indices that can be removed; this means that more matrices are examined, and therefore it is slower but might potentially yield better results. All the indices that were removed (or, equivalently, added) by previous runs of

the algorithm should be provided in the vector *indices*. Finally, **mode** is either 'FORWARD' or 'REVERSE', specifying the version of the algorithm to use.

A function computing the upper bounds of a matrix (as in steps 3 and 6 of the forward and reverse variants respectively) can be provided as **upperbound-method**. A function for adding or removing indices can be specified in **indexselectormethod**. The precise format in which these function must be written (as pertaining to input and output parameters) is described inside *eigsymencouter-directsingleupper*. The default implementations are *eigsymencouterdefaultupper-bound* and *eigsymencouterdefaultindexselector*, respectively.

Computing the upper and lower bounds of all eigenvalue intervals with the direct interlacing approach is possible via the wrapper method *eigsymencouter-direct*. An upper bound and index selection functions which are then passed to it can be explicitly specified as input parameters; aside from 'FORWARD' and 'REVERSE', the **mode** can also be 'BOTH', which runs both version of the algorithm. Since processing the two "directions" is rather similar, we have defined an auxiliary function within *eigsymencouter* which sequentially computes all eigenvalue bounds for us.

4.1.3 Methods applying the indirect interlacing procedure

The main implementing method is *eigsymencouterindirectsingleupper*, which computes the upper outer approximation for a single eigenvalue interval. As can be seen from the description of Algorithm 4, one of the steps involves computing outer bounds for a symmetric interval matrix $[-A_\delta, A_\delta]^S$, which is done with the help of the direct interlacing method described above. Generally, we expect that this matrix will not have been computed beforehand as part of some other method, and therefore we will have to call the direct method to get the needed approximation. Therefore the same optional input parameters (mode and index selector function) as in *eigsymencouterdirect* are present here as well; if provided, they are passed to the direct interlacing method call when computing the bounds of $[-A_\delta, A_\delta]^S$.

The implementation itself is straightforward, uses no auxiliary functions (save for the direct interlacing implementation), and the most complicated part is arguably a loop which iterates over a set of indices looking for the best possible value. In any case, we believe that it warrants no further discussion. The corresponding wrapper function is *eigsymencouterindirect*.

4.1.4 Choosing between the direct and indirect method

The original article describing the interlacing methods ([3]) mentions that the relative effectiveness of the two variations (direct and indirect) depends on the width of the gaps between the eigenvalue intervals of the matrix: namely, that the direct method is better in the case of narrow or non-existent gaps, and vice versa. Empirical experiments also confirm that there indeed seems to exist some sort of relationship of this sort; our attempts at investigating the nature of this relationship and the experiments performed for this purpose are described in detail in Chapter 6.

In any case, if we are able to express this relationship as a simple formula

or rule and define a criterion based on it for choosing between the direct and indirect algorithms, we can construct a fast and accurate combination of the two variants which intelligently attempts to guess which one should be applied for the computation of each eigenvalue bound. We usually refer to it as a "gap criterion", since it is based on the relative magnitude of the "gaps" (distance) between adjacent eigenvalue intervals. The basic logic of this approach is encoded in one of the computation modes of the wrapper function *eigsymencouter* and utilises an auxiliary method called *interlacingcriterium*, which receives a matrix, a previously computed outer approximation of its eigenvalues and the index of the eigenvalue we want to compute and returns either 'direct', 'indirect' or 'unknown' based on which interlacing algorithm should be used. Since we haven't yet managed to formulate an accurate enough criterion, the function in question instead attempts to make a guess about the average relative effectiveness of the two algorithms on all eigenvalue intervals of the given matrix (this approach is based on our experimental results from 6.1).

When discussing the implementations of the direct and indirect interlacing methods earlier in this chapter, we mentioned that the basic methods only compute bounds for a single eigenvalue interval; the potential application of a gap criterion is precisely the reason for this. We also discussed what options are available for obtaining the missing information about removed or added indices in the direct method (the 'STEP, 'STEPVAR' and 'ALL' strategies); while they do not play any role when computing all eigenvalues with the direct method (since no indices are "skipped"), a short comparison of these variants is in order, as the gap criterion will most likely result in missing indices due to the indirect method being preferred for certain eigenvalues intervals.

Let us assume that we have computed the first k eigenvalues of a symmetric interval matrix $\boldsymbol{A}^S$ with the direct interlacing method, and therefore know the principal submatrix $\tilde{\boldsymbol{A}}^S$ of $\boldsymbol{A}^S$ of dimension $(n-k)$ which the algorithm has produced. Furthermore, suppose that the $(k+1)$ eigenvalue was computed with the indirect algorithm and that now we want to compute the $(k+2)-nd$ eigenvalue with the direct algorithm again. There are, evidently, two possible ways to proceed: either select an index from $\tilde{\boldsymbol{A}}^S$ using the index selector function provided and then repeat the same process for the principal submatrix $\tilde{\boldsymbol{B}}^S$ which would have resulted from the previous step; or simply select the best matrix (with the same criterion used in the index selector) of dimension $(n-(k+3))$ of $\tilde{\boldsymbol{A}}^S$, essentially "skipping" a step. However, this approach is actually more difficult to compute: in the first case(where we removed one index from $\tilde{\boldsymbol{A}}^S$ and then another from the resulting matrix) we would need to examine $\binom{n-k}{1}+\binom{n-k-1}{1} = n-k+n-k-1 = 2(n-k)-1$ matrices. In the second variant(where we skip a matrix and remove two indices at once) we would have to inspect $\binom{n-k}{2} = \frac{(n-k)(n-k-1)}{2} = \frac{n^2+k^2-2nk+n+k}{2}$, which obviously grows much faster than the aforementioned $2(n-k)-1$. It is larger for any k for matrices of dimension at least 6; on the other hand, the difference in performance will not be so noticeable for smaller matrices, and therefore, assuming that we're optimising the algorithm for speed, we should always run the index selector on the matrices we've skipped; this has the added advantage of finding a better approximation in case the prediction of the gap criterion function was

incorrect. Unfortunately, it also implies that these eigenvalues will have to be computed twice, once with both methods. As long as the criterion selects the indirect one for all eigenvalues from a certain point onward, the direct method will not be called for these eigenvalues. Furthermore, the indirect approach will never be used on eigenvalues for which the direct one was chosen by the criterion. Therefore, it is safe to assume that even with this rather unfortunate obstacle accounted for, the resulting function will still be significantly faster than always running both methods, and it will almost certainly be equally or more accurate than calling only one.

One final note: in the case that the gap criterion function cannot decide which version of the direct interlacing method to use, we have decided to go with the indirect one, as our numerical experiments show that it is usually more precise and it also gives a higher probability of not having to run the direct method on a sequence of indices.

4.1.5 Filtering methods

The *eigsymencouterfiltersingleupper* function is a straightforward implementation of the filtering algorithm (Algorithm 5) which follows its description more or less to the letter, and so we believe that no detailed elaborations are necessary. The **p** and **m** input parameters define the required precision and maximal number of iterations (the same as in the algorithm's description).

Worth noting, however, are the last two input parameters, **sr1** and **sr2**; these are references to functions that compute an upper bound for the spectral radius of a given real matrix. By default, we use *verspectrad* which is a part of the Versoft package ([7]) and computes the actual spectral radius, with verification. As we pointed out in the discussion on Algorithm 5, however, using quicker (albeit less accurate) approximations can lead to dramatic decreases in computation time (see 6.2 for our experimental results); for this purpose, *eigsymencfiltersumnorm*, which computes the maximal column and maximal row sum, can be explicitly referred to in one or both of the last two parameters. A reference to a function implementing a possible better approximation can be placed here as well.

The wrapper function, *eigsymencouterfilter*, which filters all outer approximations from both sides, simply applies the former procedure twice for each eigenvalue and returns the results. It takes the same input arguments for the purpose of passing them to *eigsymencouterfiltersingleupper*.

4.2 Inner bounds

4.2.1 Methods applying the local improvement method

The *eigsymencinnerlocimpsingleupper* function contains the basic implementation of the local improvement procedure (Algorithm 6). The implementation contains no particularly interesting moments, and the description of the algorithm should be sufficient for understanding how it functions. As with the filtering function described above, only the upper bound for a single eigenvalue is computed. The matrix to examine and the index of the eigenvalue are passed to the function as

the first two input parameters. Optionally, precomputed eigenvalues and eigenvectors of the central matrix of the input matrix can be given as the third and fourth input parameter respectively. The first output value is the inner approximation computed by the matrix; the second output value is basically the upper bound of the verified interval which defined the final (in every step of the algorithm, a verified interval is computed for an eigenvalue of some extremal matrix, and the lower bound of that interval serves as a potential improvement of the approximation; the second output value is simply the upper bound of this interval). This second output may be required by other, more complex methods, such as the direct submatrix vertex enumeration procedure. More details can be found in its description further on.

eigsymencinnerlocimp is the corresponding wrapper method, which runs the above function twice for all eigenvalues to compute upper and lower bounds. As was already discussed, it is possible (due to verification) that these bounds will not define correct intervals; for this reason, the function returns the lower and the upper bounds as separate output parameters (namely, the first and second), followed by the lower and upper (as third and fourth output parameters) limits of the corresponding verified intervals (as described in the above paragraph).

4.2.2 Methods applying the vertex enumeration method

The basic implementation of Algorithm 7 can be found in *eigsymencinnervertensingleupper*, which takes an interval symmetric matrix and an index as input. Precomputed eigenvalue intervals for the central matrix can be passed to the function in the third, optional, argument. The only remarkable thing about the implementation is the use of the auxiliary class *vectoriterator* for generating all possible vectors from $\{\pm 1\}^n$ by Algorithm 8. The wrapper function is *eigsymencinnerverten*.

Otherwise, the majority of the remarks pertaining to the local improvement algorithm and its implementing functions also applies to vertex enumeration as well: the results may not necessarily be correct intervals, and so the functions has two output parameters corresponding to the lower and the upper bounds respectively.

4.2.3 Methods applying the direct submatrix vertex enumeration method

As was discussed in the description of the direct submatrix vertex enumeration algorithm(9), significant changes need to be made in order to have it perform verified computations. More importantly, the tests that we performed rather heavily imply that using the branch and bound version of the algorithm is, in fact, undesirable, as it takes more time to verify the condition for skipping a set of indices than we gain by actually skipping it (and all of its subset as well). On the other hand, the original (unverified) algorithm required such a strong condition due to the fact that one of its main strengths was finding exact bounds for the eigenvalue intervals in some specific cases; since this property of the algorithm is all but lost after applying verification to it (see the discussion of the algorithm for details), we can replace this necessary condition by a faster heuristic, possibly skipping some

useful intervals but ultimately attaining a higher speed and still offering a better accuracy than the local improvement and vertex enumeration methods. Developing a well balanced criterion for this purpose might prove to be an interesting question to research, but for our implementation we have chosen a rather simple one: instead of trying to guarantee that the system in the necessary condition does not have a solution in order to skip an index set, we instead process it only if the system is guaranteed to be solvable. According to our experimental data (see 6.3) this virtually always produces the same results as the version without branch and bound, but the computation time is shorter (although, admittedly, not by much). Such a heuristic is implemented in *negativecheckcondition*; we still keep our implementation of Algorithm 12 as *defaultcheckcondition*.

On the other hand, there is still a small possibility that exact eigenvalue bounds might be computed; if this is the case, we cannot afford to skip even a single potentially useful index set. If this is the case, it is better (with respect to computation time) to ignore the necessary condition and simply process all index sets without branch and bound; this could, of course, be achieved by implementing a trivial condition check which always fails, but the data structures we use for the branch and bound may occupy a significant amount of memory for larger matrices; therefore, it is useful to have a separate implementation of the algorithm without these data structures as well.

Due to this, we have two basic implementing functions. The first, *eigsymencinnerdsveupperbb* allows us to use branch and bound (for example, with the heuristic described above) at the cost of some memory; aside from precomputed inner and outer approximations of the eigenvalue intervals, its parameters include **ev**, a logical value which specifies whether to look for exact bounds or not (which requires more verified computations and leads to a slower algorithm) and **checkcondition** which is a reference to a function for accepting or rejecting index sets (the default is *defaultcheckcondition*).

The second basic function is *eigsymencinnerdsveupper*, which takes the same parameters as the first one sans the function for accepting or rejecting index sets.

Both methods function similarly (the only difference being, of course, due to the branch and bound and its related data structures); improving inner approximations is done according to Algorithm 11 in both cases. Exact bounds are determined by counting the number of possible candidates for each eigenvalue interval; we should note that this is done more to illustrate the idea that was discussed in the algorithm's description, and hasn't been thoroughly tested yet. In any cases, it doesn't seem, and neither can it be expected to find exact bounds except in certain very specific cases.

The wrapper functions are *eigsymencinnerdsve* and *eigsymencinnerdsvebb*. Their implementations are trivial.

4.3 Interface functions

A number of functions do not provide any actual functionality by themselves, but rather organise and combine the procedures described in the previous sections into more complex, and more accessible computational mechanisms. We refer to such functions as "interface functions" since a hypothetical user of our implementation is expected to work with them and not with the underlying methods directly.

The user will most probably need to find both inner and the outer approximations of the eigenvalues of a given symmetric interval matrix, and it is to be expected that our package will contain a single function which he will be able to call and which will compute and organise all results for him. This function is called *eigsymenc* in our implementation, since it does, indeed, provide a complete encapsulation of all the eigenvalue intervals of symmetric interval matrices. In essence, this is but a wrapper method: the actual "work" is done by *eigsymencinner* and *eigsymencouter*, which provide a complete calculation of the inner and outer approximations of the eigenvalue intervals, respectively. These functions can also be called directly by the user by giving them the interval matrix of interest.

On several occasions we have mentioned that the speed of a given algorithm or approach can be increased at the cost of accuracy, and vice-versa (which is, of course, a basic fact and not surprising in itself). However, a question that remains open is how to resolve this dilemma, i.e. when to choose precision over efficiency and when to sacrifice accuracy for the sake of better computation times. Since this is impossible to say, however, without knowing the specifics of the context for which the calculations are performed, it is best to leave this choice to the user, who will be, no doubt, better informed than us. On the other hand, our program should be accessible and should not require understanding of its internals in order to use it.

For this purpose, the interface functions use a common convention of five levels for specifying the desired accuracy of the computation: these are *FASTEST, FASTER, EFFECTIVE, TIGHTER and TIGHTEST*, and their intuitive meaning should be obvious. One of these values can be supplied as an optional input argument to any of the interface methods described above. If no such argument is specified, *EFFECTIVE* is assumed as default, and this is the mode that we expect will be used by the user most frequently and for the majority of situations.

The interface functions are programmed in such a way that modifying the existing modes or even adding new ones should be very simple, and within the capabilities of the average user in the case that the provided five modes are not flexible enough for some specific application. Both *eigsymencouter* and *eigsymencinner* contain a main "switch" statement which calls different functions according to the specified computation mode and then selects the best results from them. Introducing a new mode is as simple as adding a new branch to this "switch" statement; reading the documentation of the individual functions should be enough to understand their input and output parameters, as well as their basic functionality, without examining the code in detail.

For outer approximations, the fastest mode uses only Rohn's theorem (Theorem 2) to provide a quick result; in fact, since this is so fast, we use it as an initial approximation for all modes.

The *FASTER* mode selects one of the direct and indirect interlacing methods according to our heuristic developed in the course of the experiment in Section 6.1; this only makes sense for matrices that tend to be centred around 0, but the function *eigsymencoutermethodselector*, which is called to choose the method, can be modified to use a more accurate heuristic or to correspond to specific data sets (for example, it could be made to always choose the direct method if the

user knows it is always preferable for the specific matrices he is working with); however, instead of modifying the actual method, we recommend defining a new function in the same format and calling it from *eigsymencouter*; multiple criteria can be defined in this way, and even used to create new user-defined computation modes for specific classes of matrices.

Finally, we apply the fastest (and least accurate) filtering method, as described in the corresponding experiment (Section 6.2); it runs very fast (especially compared to the computation time required by the interlacing algorithm) so we don't lose too much performance even if it doesn't improve the found approximation. We only use the forward direction of the direct algorithm, as it almost always works better according to our experimental data.

The *EFFECTIVE* mode attempts to select either the direct or indirect interlacing method for individual eigenvalues based on the outer approximation computed so far; heuristics based, for example, on the experiments described in 6.1, can be used here; the function which performs the method selection is *interlacingcriterium*; in the current version it simply calls the method selection function used in the *FASTER* mode; in the same way as described above, more complex criteria can be defined and incorporated into *eigsymencouter*. The method uses the "EI" version of the filtering algorithm as described in Section 6.2; it works roughly twice as fast as the original filtering method, but usually produces very good results.

The *TIGHTER* mode calls both the forward direct and indirect interlacing algorithms and performs "EI" filtering.

The most accurate mode, *TIGHTEST*, additionally uses the reverse direction of the direct interlacing method as well, and performs the most accurate (but slowest) filtering (referred to as "EE" in the experiments). In other words, here we utilise the most precise versions of all methods at our disposal.

The wrapper method for computing inner approximations, *eigsymencinner*, functions in a similar way. Worth noting is that it returns six values representing the found approximations instead of just one as with the outer approximation. The first two output parameters represent the lower and upper approximations found; these are returned separately, as it is possible that they define an improper interval (this occurs for very "thin" interval matrices due to verification); if this happens, we know that the methods used have failed to find a verified inner approximation.

The third and fourth output arguments are logic vectors that indicate, respectively, which lower and upper bounds are exact; this is indicated by a value of 1 in the corresponding element of the vector.

The last two output arguments are the outer "limits" of the lower, resp. upper approximations which are known to be exact; the upper inner approximation of a given eigenvalue interval is essentially the highest value that is guaranteed to lie within that interval; the actual value may, of course, be larger. In the case that an exact bound is computed, the *largest* value that it can take is listed in the corresponding position of these last two vectors; in this way, taking the upper inner approximation of an interval as a lower bound and the "limit" given in the corresponding position of the last output vector as an upper bound will yield a verified interval inside which the exact bound must lie. The same situation

applies analogically for the lower approximations. For results that are not known to be exact (according to the third and fourth output arguments) the values in these last two vectors should be ignored.

Since some of the inner approximation algorithms required an outer approximation of the eigenvalue bounds as well, this can be provided as a third, optional argument, to *eigsymencinner*; otherwise, it is computed with *eigsymencouter* according to the specified computation mode.

The last input argument indicates whether we want to search for exact bounds or not; this concerns only the Direct submatrix vertex enumeration algorithm, but the probability of getting any results is very small.

Finally, *eigsymenc* is the most general wrapper function, which calls *eigsymencouter* and *eigsymencinner* with a given computation mode and returns the results in a single matrix for convenience. It also replaces any improper intervals with "NaN" (not a number) values.

5. User guide

This section explains in some detail how to install and run the software implementations of the previously described algorithms. A basic familiarity with the Matlab programming language and environment is assumed.

5.1 Prerequisites

Our functions make use of the Intlab package ([11]) for working with interval data, which must be correctly installed and set up in order for them to function correctly. The installation process is very simple and involves adding Intlab's directory (the directory to which Intlab's files were extracted) to the list of paths in Matlab; the exact way of doing this varies slightly between the different versions of Matlab, but in general an option named "Set Path" must be used. The command "addpath" might also be used as an alternative. Matlab's documentation can be consulted for more information in case of doubt.

It is also necessary to edit the "startup.m" file (which should be located in Intlab's main directory) with a text editor and replace the line "cd c:\intlab" to refer to the actual location of Intlab's directory on the user's hard drive.

After the two steps described above are completed, Intlab will start and initialise automatically every time Matlab is run.

Another library which we utilise is Rohn's Versoft package ([7]) which contains functions performing differed verified operations. More specifically, we use the function *vereig*, which computes verified intervals for the eigenvalues and eigenvectors of a given real matrix. Sometimes we also make use of *verspectrad* for computing spectral radii of matrices. Aside from downloading and extracting the library in question to a suitable directory, the only installation required is adding the path of that directory to Matlab's list of paths, which is done in the same way as described above for Intlab.

For the branch and bound version of the direct submatrix vertex enumeration algorithm we also utilise the functions *ilsunsolvablefcr* and *ilssolvable* developed by Jaroslav Horáček; these do not appear to be published at the time of writing, and should be included in our software package.

Our own software package is installed in the same way as Versoft: the source files are copied to the user's hard drive, and the directory containing them is added to the list of paths in Matlab.

Note that functions from a certain directory can be called even if that directory is not in the list of paths, provided that the directory in question is Matlab's current working directory. However, this is not particularly convenient.

5.2 Working with input data

The Intlab package is used for representing and working with interval data; for a reminder of the notions used, the reader can refer to the first chapter of the thesis.

Constructing an interval in Intlab can be done in two ways: either by specifying the interval's lower and upper bound (infimum and supremum), or by giving its centre and radius. This is accomplished by using the command *infsup* and *midrad*, which are both quite intuitive to use. As an example, suppose that we want to create a representation of the real interval $[3, 5]$ and save it to a variable called "a". The following two commands both accomplish this goal:

a = infsup(3,5);

a = midrad(4,1);

Constructing an interval matrix is done in much the same way, except that the arguments supplied to *infsup* or *midrad* are real matrices instead of numbers. For example, the command

A = infsup([0 1; 2 3], [4 5; 6 7]);

places a representation of the interval matrix $\begin{pmatrix} [0,4] & [1,5] \\ [2,6] & [3,7] \end{pmatrix}$ into the variable "A". Vectors, as special cases of matrices, are constructed in the same way as well.

Should we need to determine the lower bound, upper bound, centre and radius of a given interval, we can accomplish this with the commands *inf*, *sup*, *mid* and *rad* respectively. Calling

inf(a)

will return 4, the lower bound, for the real interval defined by the previous few commands.

The basic arithmetic operations, such as addition, subtraction, multiplication, etc. can be applied to interval values and their results correspond to the definitions given in Chapter 1. For example, if we first define some intervals and assign them to variables as follows

a = infsup(3,5); b = infsup(-1,2); c = infsup(-3,-2);

the output of the commands

a + b
a + c
a * b

a / c
a / b

will be the intervals $[2, 7], [0, 3], [-5, 10], [-2.5, -1]$ and $(-\infty, +\infty)$ respectively; note that the last operation involves division by an interval containing zero, and is therefore undefined according to the background presented in Chapter 1.

5.3 Computing inner and outer approximations

Now that we've described how to represent and work with interval matrices, we can demonstrate how to use the functions we've developed in order to find inner and outer approximations. In the following examples, assume that the variable "A" contains a representation of a symmetric interval matrix.

The simplest way to use our software package is to call

eigsymenc(A)

This utilises a selection of algorithms with a reasonable ratio of computation time and accuracy to find both an inner and an outer approximation. The output is displayed in the following format:

```
[  -23.8792,    2.5518] [  -21.7612,   -2.7955]
[  -18.6798,    7.6840] [  -10.3685,    1.7928]
[  -11.3274,   11.1878] [   -3.8650,    5.0872]
[   -6.1519,   18.4198] [    3.4518,   11.0428]
[   -0.4759,   25.2010] [    5.3998,   22.8773]
```

This is essentially a matrix with two columns, containing intervals; the first contains the computed outer approximation and the second contains the inner one. The eigenvalue intervals are sorted in ascending order. In the case that a verified inner approximation could not be computed for a given value, the corresponding element of the second column will contain a "NaN" (not a number) value.

Usually we will want to save the result to a variable; this can be done, for example by typing

```
results = eigsymenc(A)
```

which will save the matrix produced by the function to a variable called "results". We can access individual elements of any matrix in Matlab by specifying the indices of the row and column that we are interested in. For example, to get the outer approximation for the second smallest eigenvalue interval, we can use

```
results(2,1)
```

Although the default algorithms should work well in most cases, we can specify a so called "computation mode" as a second input parameter to *eigsymenc*.

The available options are *FASTEST*, *FASTER*, *EFFECTIVE*, *TIGHTER* and *TIGHTEST*. The computation mode basically tells our program which algorithms to use for the computation, as some are more precise, but slower than others. The *EFFECTIVE* mode is used by default (i.e. if we do not specify a computation mode), and if we find that the results are unsatisfactory, or the speed of computation is too slow, we can select a different mode. As an example, we can get the fastest possible approximations by calling

```
eigsymenc(A,'FASTEST')
```

Perhaps surprisingly, *eigsymenc* is a simple wrapper method which obtains its approximations by calling two more specific functions, *eigsymencouter* and *eigsymencinner*. It might happen that we want to find only an outer or only an inner approximation for some given matrix, or that we want to apply different computation modes for the outer and inner approximation. In this case we can circumvent *eigsymenc* and call the two functions mentioned above ourselves.

The first, *eigsymencouter*, has the same interface as *eigsymenc*: the first parameter is the matrix that we want to examine, and the second one (which is optional) is the desired computation mode. The output is an interval vector which corresponds to the first column of the matrix returned by *eigsymenc*.

On the other hand, *eigsymencinner* is a little more complicated. It has four input and six output parameters and takes the form

```
[ l, u, el, eu, ll, ul ] = eigsymencinner( iA, mode, iouter, e )
```

The first two input arguments are the same as those of the previous two functions; the third is an outer approximation of the matrix's eigenvalue intervals, and the fourth is a logical value (true or false) indicating whether we want to try and compute exact bounds. This only works with the *TIGHTEST* computation mode, and quite rarely at that, so *eigsymenc* assumes that we always set e to false and doesn't display any results concerning exact bounds. Only the first input parameter is required. If no outer approximation is supplied, *eigsymencinner* will compute one by calling *eigsymencouter* with the specified computation mode; if no value of e is given, it is assume to be false.

The first two output arguments are real vectors containing the computed approximations for the lower and upper bounds, respectively. These are given separately and not as a single interval vector due to the possibility that the inner approximation algorithms will produce improper intervals as a result; in this case *eigsymenc* outputted an "NaN" value. The third and fourth output parameters are logical vectors (containing true and false values) that indicate which lower, resp. upper bounds are exact. If a given upper bound is known to be exact, the value in the corresponding field of the u vector is the lower and the value in the corresponding field of the lu vector is the upper bound of the verified interval inside which the bound must lie. Analogically, for an exact lower bound, its verified interval is defined by the corresponding elements of the ll and l vectors.

5.4 User-defined computation modes

It is worth noting that *eigsymencouter* and *eigsymencinner* are themselves wrapper methods as well; the actual "work" is done by functions implementing different variations of the algorithms described in Chapter 3. The user is not required to understand the functionality and reasoning behind these implementations in order to be able to use our package (although one can refer to Chapter 4 and the documentations inside the individual functions to gather more insight), but the internal workings of *eigsymencinner* and *eigsymencouter* are very simple, and one particular benefit that the user can gain by studying them is the ability to define custom computation modes.

The motivation for this is more or less obvious: first of all, our classification of the different combinations of algorithms into five computation modes does not necessarily have to correspond to the user's notions, or to the requirements of a specific application; what we consider to be "effective" might, in fact, prove to be too slow, or not accurate enough, depending on the needs and resources available to the user. Furthermore, we have organised our functions into computation modes based on their performance for randomly generated matrices. It is entirely possible that the user might be working with a specific class of matrices, for which our observations are inaccurate or, conversely, unneeded; according to our experimental data, for example, in the general case the forward version of the direct interlacing algorithm almost always performs better than the reverse direction. Now suppose that a certain project requires computing the eigenvalue bounds of some very specific class of matrices, for which the forward direction is, on the contrary, ineffective. Instead of having to resort to one of the slower modes which run the algorithm in both directions, the user could define a new mode specifically for these matrices that uses only the reverse direction and saves time. Naturally, this requires some basic understanding of how *eigsymencinner* and *eigsymencouter* work.

Both wrapper functions contain a single "switch" statement which calls individual algorithm implementations according to the mode specified. In order to add a new mode, all that the user has to do is add a new case to this statement, for example between the last existing mode and the "otherwise" keyword:

```
...
case 'TIGHTEST'
   ...

case 'NEWMODE '
   ienc = eigsymencouterdirect(iA, 'reverse');

otherwise
   ...
```

Of course, this requires more detailed knowledge of the functions implementing

the individual algorithms. Although we will not go into details here, reading the documentation inside the source files along with Chapter 4 might prove useful in this respect.

6. Numerical experiments

6.1 Direct and indirect interlacing methods

As the original authors themselves mention in [3], the direct interlacing method appears to be generally more effective for matrices with narrower (or non-existing) gaps between their eigenvalue intervals, while the indirect approach is usually better in the opposite case (when the spaces between the eigenvalue intervals are larger). While no specific criterion for differentiating between "narrow" and "wide" gaps for this purpose is presented in the article, finding one would be very useful to us, since it could be used to construct a very effective (with respect to computation time) implementation of the discussed algorithms, as it would, for a given matrix, pick only one of the two variants (either indirect, or direct), yet still produce better outer approximations than if only one of the two algorithms was used for all matrices. Naturally, the more accurate the criterion used, the better the results will be in the general case.

A lot of experiments were performed with randomly generated matrices in attempts to gather information about the relationship between the spaces between the eigenvalue intervals of a matrix and the relative effectiveness of the direct and indirect interlacing methods. This section presents some of the more eloquent results.

6.1.1 First experiment

Our first experiment was an attempt to find a criterion for choosing between the direct and indirect interlacing methods for a specific eigenvalue of a given symmetric interval matrix. As was mentioned above, the direct method usually performs well for matrices with eigenvalue intervals that either overlap, or have narrow "gaps" between them, while the indirect version generally works better in the opposite case. Obviously, the first thing that we need to look for is a way to express the size of a "gap" more accurately, since words like "narrow" and "wide" do not mean anything in a mathematical context unless they are supported by exact definitions.

In order to characterise the magnitude of a "gap" for some symmetric interval matrix $\boldsymbol{A}^S \subseteq \mathbb{IR}^{n\times n}, n \in \mathbb{N}$ with eigenvalue intervals $\Lambda_1 = [\underline{\lambda}_1, \overline{\lambda}_1] \leq \Lambda_2 = [\underline{\lambda}_2, \overline{\lambda}_2] \leq ... \leq \Lambda_n = [\underline{\lambda}_n, \overline{\lambda}_n]$, we define, for $i \in \{1, 2, ..., n-1\}$ the gap coefficient ξ_i describing the gap between the i-th and the $(i+1)$-st eigenvalue interval as:

$$\xi_i = \frac{\underline{\lambda}_{i+1} - \overline{\lambda}_i}{||\boldsymbol{A}^S||},$$

where $||\boldsymbol{A}^S||$ is the application of some matrix norm to the symmetric interval matrix $\boldsymbol{A}^S$. In the experiment we used $||\boldsymbol{A}^S||_1$ and $||\boldsymbol{A}^S||_\infty$, where $|\boldsymbol{A}_{ij}|$ denotes the magnitude of the interval at row i and column j of the interval matrix; since we're working with a symmetric interval matrix, we can assume that $A_{ij} = A_{ji}$ for $i, j \in \{1, 2, ..., n\}$.

Note that in the definition above, the gap coefficient can be negative as well; in this way, it not only denotes whether or not two adjacent intervals overlap,

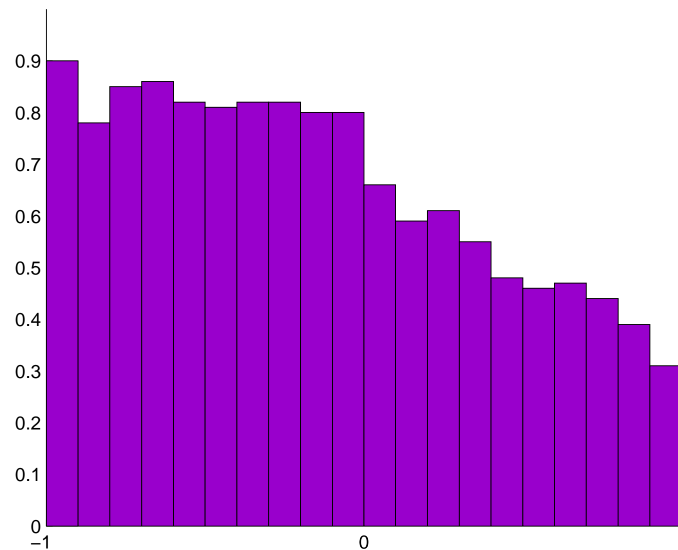
but also describes how much they overlap.

Another thing worth mentioning here is that, while the above definition refers to the exact bounds of the matrix's eigenvalue intervals, we are never going to know them in practice (otherwise there would be no point in running either the direct or indirect algorithm at all), but we can simply use some already existing outer approximation as an estimate; for example, Rohn's theorem is a very good candidate for such an initial approximation, as it's very fast to compute and in a lot of cases is quite accurate.

As a general rule, the direct and indirect method almost always produce the same result for the supremum of the largest eigenvalue (and therefore the infimum of the smallest eigenvalue), while their outputs for the rest of the eigenvalues are usually much more different and correspond to the empirical observations described above.

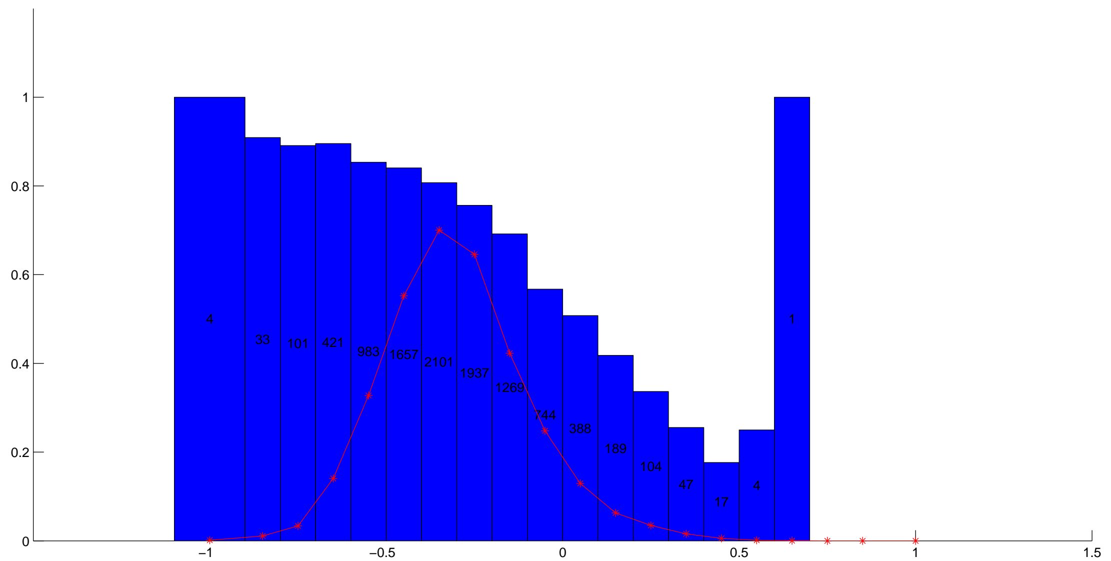
In order to gather information about the relation between the examined gap coefficient and the results of the direct and indirect method, we generate a large number of random symmetric interval matrices with values in the range $[-10, 10]$, and assign each of the matrices to a given interval (the number of intervals and the edges of each interval are variables) according to the magnitude of it's gap coefficient in order to get a histogram of the generated matrices; furthermore, for each matrix we compute its eigenvalue bounds using both the direct and the indirect interlacing methods, and count how many suprema are approximated more sharply with the direct interlacing method. This way, for every interval in the histogram, we know what percent of the matrices are better approximated with the direct method, which basically means that we know what is the approximate probability of the direct method of providing a better (or at least equal) approximation. For matrices of higher dimensions, separate histograms are constructed for each individual eigenvalue/gap coefficient pair.

The first phase of the experiment examined matrices of dimension 2, since they only have one eigenvalue interval and therefore we do not have to take any "interference" between the individual gaps or intervals into account. The first few tests (which we do not present in detail here) lead us to the conclusion that the interval between -1 and 1 (as values of the gap coefficient) is particularly interesting and should be studied in more detail, as it contains the point for which the direct and indirect methods perform equally well on average. The following graph was obtained by generating 10000 random matrices and using a histogram with intervals of magnitude 0.1:



From the above diagram, we can see that the length of the intersection of two adjacent intervals appears to be irrelevant, as long as the intervals do overlap; in this case, there is an estimated probability of about 0.8 that the direct method will perform better; as the gaps increase, however, its effectiveness gradually decreases.

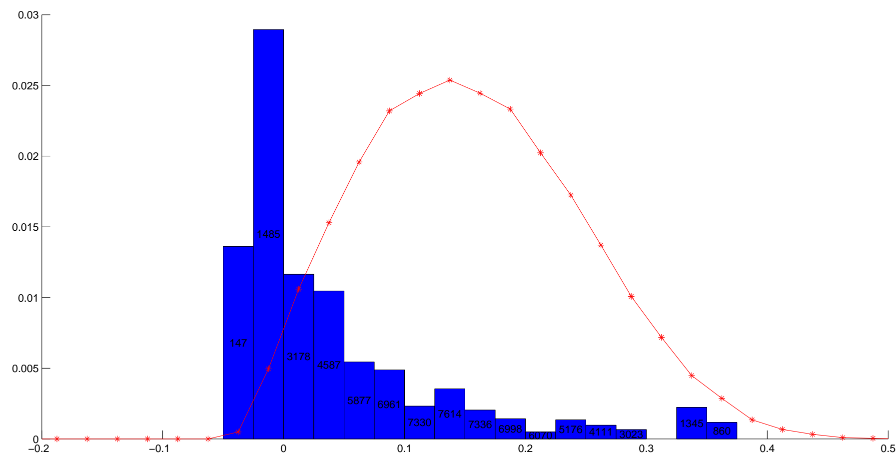
Performing a similar experiment for matrices of higher dimensions yields comparable results: the histograms for the individual eigenvalues do not differ too much from each other, and generally look similar. There is one major difference, though, and that is the fact that the measured relative effectiveness of the direct method now decreases for negative values of the gap coefficient as well (i.e. it now matters how much the intervals overlap). The graph obtained for the second eigenvalue (first gap) of matrices of dimension 3 is given below as an example. The dotted line describes the distribution of the generated matrices, while the numbers in the individual columns of the histogram correspond to the total number of generated matrices that belong to the given column (i.e. the number of matrices with a gap coefficient in the corresponding range).

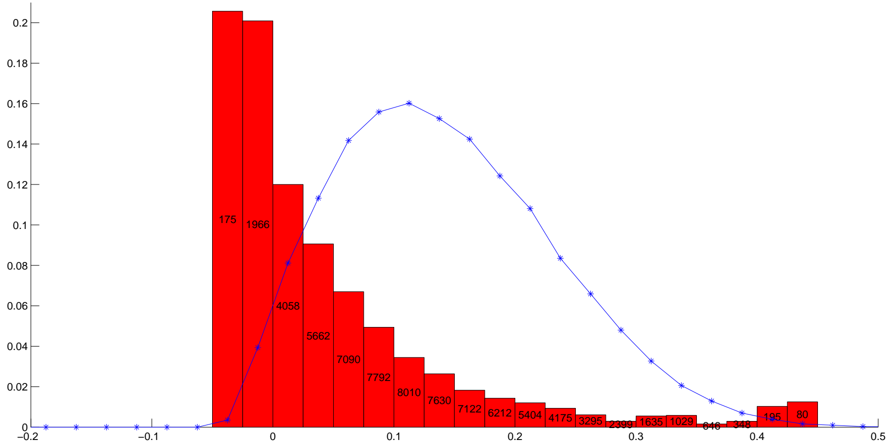


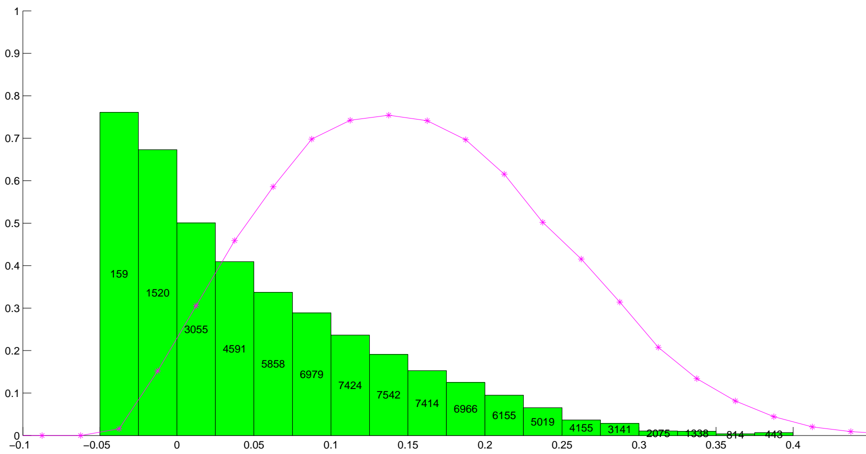
It appears that the point (the value of the gap coefficient) at which the direct method starts being more effective shifts to the left as the matrix dimension increases. Nonetheless, the distribution of the matrices produced by our matrix generating function implies that our data is less accurate for very small and very large gap coefficients; it would therefore be necessary to examine these "extremal" intervals more carefully before formulating a general hypothesis.

The new generation method that we use is based on the idea of "expanding" the elements of a randomly generated interval matrix with very "thin" intervals by multiplying them by a random number from a specified interval with a given probability.

Using an expansion multiplier interval of $[0.05, 0.1]$ and a probability of 0.5, and performing the same experiment as above for matrices of dimension 4, we were able to produce much more interesting results. The plots for the second, third and fourth eigenvalues of such matrices are reproduced below.







While the plot for the fourth eigenvalue appears to agree with the observations from the previous tests, the other two graphs suffer from significant "deformations", not only in their shape, but also in the overall magnitude of the direct algorithm's effectiveness: the highest measured success rate (in the leftmost interval of the histogram) for the third eigenvalue is a mere 20 percent, while for the second eigenvalue it is practically zero.

Repeating the same experiment with slightly different values of the matrix generation function's parameters (specifically, increasing the amount by which the initial "thin" intervals can be expanded), we notice that the results are similar to the diagrams depicted above, yet the observed "deformations" are not as severe: the overall shape of the individual histograms is closer to the one obtained by the very first experiments, while the maximum success rate of the direct method for the second eigenvalue is 0.1, as opposed to the 0.03 from the previous experiment. The direct algorithm's performance for the third and fourth eigenvalues is also slightly better.

This leads us to the conclusion that the choice between the direct and indirect interlacing method for a given symmetric interval matrix cannot be made by approximating the relative magnitude of its interval "gaps" alone, but other factors need to be taken into account as well.

If no accurate criterion can be developed, and if the user has the capability of generating a large amount of matrices of the specific type that he is going to work with, we could suggest implementing a dynamic data structure which will automatically construct a histogram as the ones we described above and update it after every computation that involves both the direct and the indirect interlacing methods; this could then be used to approximate their relative effectiveness for further computations.

6.1.2 Second experiment

Our second experiment attempted to establish a connection between the magnitude of the radial matrix A_δ of a given symmetric interval matrix $\boldsymbol{A}^S$ and the effectiveness of the direct versus the indirect interlacing method. This is a certain modification of the problem described above, as we are not directly examining the gaps between the individual eigenvalue intervals; on the other hand, an increase in the magnitude of the radial matrix naturally leads to an "expansion" of the eigenvalue intervals of the interval matrix in question, which in turn implies "smaller" gaps between them. The benefit is that the experiment is simpler, since we do not need to worry about the sizes of the individual gaps, but can instead take a single number (the maximal element of the interval matrix's magnitude) as a characteristic for a given $\boldsymbol{A}^S$. As we shall see from the results below, this assumption does indeed produce satisfying results.

In order to generate random interval symmetric matrices, we first generate two real matrices: the central, and the radial matrix. The elements of the first are taken to be in a certain range (e.g. between -10 and 10), while the values of the latter lie between zero and a given number r. Although Matlab does not include a function for generating symmetric matrices, we can simply ignore the values in the lower-left half of a generated (non-symmetric) matrix and copy the numbers from the upper-right half into their respective positions; effectively, we generate the upper-right half and then "mirror" it to obtain a full matrix. Such a method should preserve the random distribution of the generated matrices. Having generated the central and radial matrix, we then simply combine them into the final, symmetric interval matrix.

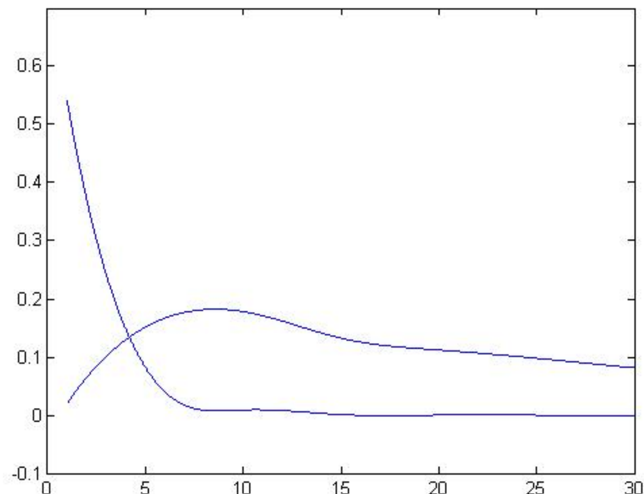
The experiment was conducted by generating matrices of given dimensions (between 3 and 10) and with given magnitudes of the radial matrices (1, 5, 10, 15, 20, 25 and 30), computing an outer approximation of their eigenvalues in four different ways (using the direct and indirect method, each one utilising only the forward or only the reverse direction of the direct approach) and then calculating a table of quotients (for each combination of a dimension and a magnitude) obtained by dividing the length of the outer approximation from each of the four methods by the length of the "optimal" approximation (the combination of the best results from all methods). Basically, the lower the quotient, the better the performance of the method in question (if the quotient is 0, then the length of the approximation matches that of the optimal one). Obviously, we've made the assumption that the quality of a given approximation depends not only the amount that it "cuts off", but on the relative size of that amount with respect to the size of the whole approximation.

Having computed the coefficients for all generated matrices with a given di-

mension and magnitude of the radius, we can take their mean, minimum and maximum values and use them as a representation for all matrices with the given characteristics, although in practice we only use the mean values. As a useful by-product of the experiment, we can measure the average times for computing the outer approximation as well. Having computed such "representative values" for all radius magnitudes of matrices of a given dimension, we use cubic spline interpolation in order to approximate the functions mapping the magnitude of the radial matrix to the quotient for one of the four methods. We then graph the computed approximations.

The experimental data shows that the forward direction of the direct interlacing method almost always produces better results than the reverse direction, while their running time is essentially the same; therefore, we only examine the forward direction of the direct method in our experiment. The effectiveness of the two directions of the indirect approach, on the other hand, differ between individual cases (although usually not by much), so we use their average as a general measure of the indirect method's accuracy.

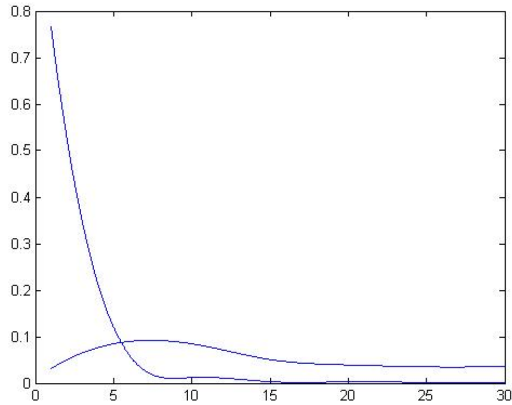
The following image is a graph we obtained for matrices of dimension 3:

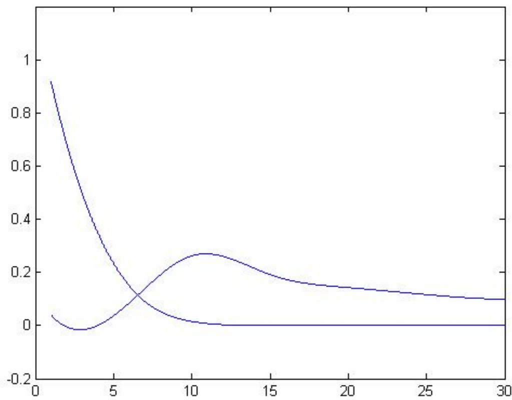


The decreasing function (resembling e^{-x}) is the one mapping radius magnitude to quotients for the direct forward method, while the second one (which initially increases and then begins decreasing) is the same function, but for the indirect method. What we see generally agrees with the observations in the original article: the direct method performs very poorly for matrices with smaller radii (and, therefore, with larger gaps between their eigenvalue intervals), but quickly improves in performance as the radius increases. The indirect method, on the other hand, performs very well for matrices with a small radius, and its performance does not change too gradually for larger radii, remaining more or less constant and relatively satisfying as well. This immediately leads us to the conclusion that the indirect method should always be preferred in the case that no additional information is available about a given matrix.

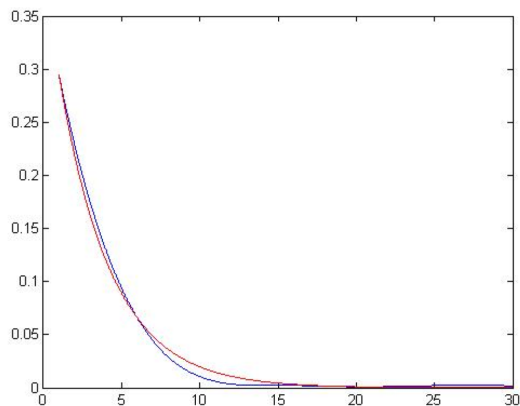
The intersection of the two graphs (which in this case is for a magnitude of about 4.2) is the point at which the direct method begins to be more effective. Performing the experiment for matrices of higher dimensions (but with the same parameters for generating the central and radial matrices), we can see that the

magnitude corresponding to this point shifts to the right; it is (approximately) 4.5 for matrices of dimension 4 and 5, 5.5 for matrices of dimension 5, 5.8 for dimension 6, 6 for dimension 8, 6.2 for dimension 9 and 6.5 for dimension 10. Otherwise, the overall "shape" of the combined graph is always the same; the graphs for dimensions 6 and 10 are presented below for comparison.





Notice that the graph of the direct method's effectiveness closely resembles the exponential function; indeed, it can be rather accurately approximated by plotting ae^{bx} for some $a, b \in \mathbb{R}$; the following diagram illustrates this method for matrices of dimension 5, where we chose the numbers a and b such that the exponential would pass through the first two known points of the approximated function:



This implies that if we can estimate how the values a and b change with

respect to the matrix's dimension and other parameters, we can develop a simple heuristic which will choose the direct method in the case that the constructed exponential yields a value that is not greater than a certain threshold, and select the indirect approach otherwise. However, our attempts to predict the changes of the a and b parameters did not prove to be successful, as these numbers do not change in a consistent way (for example, increasing the range of the radial matrix might cause one of them to increase in one case, and to decrease in another). Therefore, we take a different approach, based on trying to predict the point of the intersection between the direct and indirect method's effectiveness function.

Table 6.1 presents the results obtained for matrices of different dimensions and different ranges for the values of the central and radial matrix. Here the columns (0, 2.5, etc.) describe the "centre" and the rows (10, 20, etc.) describe the "range" used to generate the central matrices in each experiment. For example, to generate a central matrix with a centre at 2.5 and a range of 10 is to select its values from the interval $[-2.5, 7.5]$. The numbers in the individual cells represent the values of r (the maximal magnitude of the radial matrix) for which the graphs of the effectiveness function of the direct and the indirect methods intersect.

Note that we only shift the "centre" to the right in the experiment, as previous tests showed that if we are to move it the same distance to the left, e.g. to -2.5 instead of 2.5, we get the same results (in other words, the output of the experiment depends only on the amount and not the direction of the displacement). This is actually rather obvious, as a negative shift would theoretically produce the same matrices as the positive one but with opposite signs.

From the table, we can see that increasing the dimension of the matrix shifts the intersection point for matrices centred at zero slightly to the left, although not by much. Increasing the range from which the values of the central matrix are picked, on the other hand, shifts the intersection point to the right and has a much more pronounced effect; intuitively, this is because a given magnitude of the radial matrix is relatively smaller with respect to a central matrix, the values of which are farther apart.

If we examine the rate at which the coordinate of the intersection point increases for broader "ranges" (centred at 0) of the central matrix, we will notice that if we double the range, the coordinate roughly doubles as well. For example, if we look at the results for matrices of dimension 10 and divide the coordinate of each row by that of the previous one, e.g. divide 2.5 for range 20 by 1.6 for range 10, we will get the quotients 1.5625, 1.92, 1.9375, 1.9882; these are all numbers that are very close to two. Similar values can be obtained from the data concerning the other two dimensions. Still, these quotients are somewhat inconsistent; furthermore, if we try to find quotients for rows that are farther apart (e.g. for ranges 160 and 20), we usually get a number which does not correspond to the intuitive hypothesis that the quotient should be two to the power of the rows in-between; if we divide (for dimension 10) 18.5 by 2.5, for example, we get 7.4, which is close enough to 8, but is still somewhat inaccurate; furthermore, the distance between the actual quotient and the power of 2 will obviously be larger, the bigger the number of rows that we "skip".

On the other hand, imagine that the biggest possible distance between two elements of a central matrix is $2a$ for $a \in \mathbb{R}$, i.e. it is generated with values in the

range $[-a, a]$. If the radial matrix has a maximal magnitude of r for $r \in \mathbb{R}, r \geq 0$, then we can reduce this "gap" by at most r from each side, leaving us with a gap of length $2a - 2r$. Doubling the range of values of the central matrix, i.e. going from $[-a, a]$ to $[-2a, 2a]$, effectively makes our radial matrix twice less effective at bridging the gap, and therefore we need to increase the radius from r to r' such that the reduced gap is twice as large (in other words, the same size as before relative to the matrix's range) in order for it to have the same effect. In other words, we want to find an r' such that $\frac{4a-2r'}{2a-2r} = 2 \iff \frac{2a-r'}{a-r} = 2$. The solution is obviously $r' = 2r$; however, if we look at the quotients $\frac{2a-r'}{a-r}$ obtained from the experiment data, we will see that they are actually closer to 2 than the ones we get by simply dividing the r' by r; for dimension 10 we get 2.08, 2.01, 2.009, 2.001 (compare with the directly computed quotients above). Using the same approach for more "distant" rows, we get more accurate results as well: $\frac{160-18.5}{20-2.5} = 8.09$, which is evidently better than the 7.4 computed above. If we now know that the direct and indirect effectiveness functions intersect at 1.6 for range 10, and want to know their intersection for range 80, we can do it using the above observations by multiplying $(10 - 1.6) = 8.4$ by $(2.01)^3$ and subtracting the result from 80; this yields 11.78, which is a more accurate result than $1.6 \times 8 = 12.8$, at least according to our data. We use 2.01 instead of 2, as the quotients we got above are somewhat larger than 2 as well (but appear to tend towards it with increasing range). If we want to compute the intersection point for a range which is not an exact power of 2 using this method, we can simply use a logarithm with base 2, e.g. 15 can be represented as $2^{\log_2(15)}$, so we will multiply by $(2.01)^{\log_2(15)}$.

Predicting what will happen with the intersection point as a result of shifting the central matrix range to the right, on the other hand, is more complicated, as the experimental data does not indicate as clear a dependency; using cubic spline interpolation and examining the resulting graphs leads us to the conclusion that the functions mapping the magnitude of the shift to the intersection point could possibly resemble an exponential function. There seems to be some initial "hesitation" for the first shift, but after that doubling the magnitude of the displacement roughly doubles the intersection radius as well; the amount by which we multiply for further shifts gradually decreases. We could attempt to model a function of the type $f(x) = 1 + e^{a-bx}$ or similar (with an initial value of 2 and tending towards 1), giving the amount by which we have to multiple the intersection magnitude at x in order to get the one at $2x$. Finding the approximate value for a shift of magnitude $2^n x$ given the value for a shift of magnitude x would then involve finding the product of the values of the function for all arguments $x, 2x, 4x, ..., 2^n x$; if we ignore the constant in the function's definition, we can compute this product by exponentiating the sum of $(a - bx)$ for all arguments x; this can be done in constant time by using the well-known formula for the sum of an arithmetic series.

Within the scope of our current project, however, we will content ourselves with the heuristic we have found for matrices centred at zero. A more detailed analysis of the behaviour of the functions described above might prove to be a very interesting and productive topic for further research.

Dimension 3	0	2.5	5	7.5	10	12.5	15
10	2.1	2.1	4.2	6.6	11	13.9	16.15
20	3	3.5	3.9	5.4	7.5	12.4	13.6
40	6.1	6	5.7	6	7.4	8.2	12
80	12						
160	24.8						

Dimension 5	0	2.5	5	7.5	10	12.5	15
5	0	2.5	6.6	11.5	14.9	19.2	23.3
10	1.8	2.3	5	8.34	13.4	17.5	21.6
20	2.9	3	4	5.4	9.5	15.1	18
40	5.6						
80	11.5						
160	22.35						

Dimension 10	0	2.5	5	7.5	10	12.5	15
10	1.6	2.2	6.3	12.5	17.3	23	27.5
20	2.5	2.7	4.3	7	14.2	19.1	24
40	4.8	4.8	5	6	7.5	11.2	18
80	9.3						
160	18.5						

Table 6.1: Direct vs indirect algorithm

In order to judge whether the heuristic developed above have any practical value, we conducted several simple tests in which we allowed it to choose between the direct and indirect interlacing method. For radial matrices with a magnitude within a certain range (between 1 and 100), the results were very good: our method was able to identify the better approach in about 75 percent of all cases, which varied between 61 and 86 percent for different classes of matrices (individual eigenvalues and their lower and upper bounds were considered separate cases) and was able to minimise the total precision lost in the majority of cases as well. For larger magnitudes, the results were less satisfactory, which is without doubt due to the inexact data and approximations we used for constructing the criterion. Gathering additional data (for example, for very large and very small magnitudes) and "fine-tuning" the method could make it very valuable for performing quick and efficient computations.

6.2 Faster filtering methods

As was discussed in the description of Algorithm 5 (the filtering algorithm), an upper bound for the spectral radius of a matrix can be used in place of its exact computation in order to speed up the implementation; naturally, this results in a loss of precision, but might be the better option if time is more important than accuracy for a certain application. The main question that needs to be answered is, how much time we gain and how much precision we lose when using such approximations. The experiment described in this section is meant to provide an

0.01	EE	EI	IE	II
2	1.0189	0.0188	1.0064	1.0064
3	1.0255	1.0253	1.0086	1.0085
4	1.0136	1.0134	1.0024	1.0023
5	1.0066	1.0065	1.0006	1.0005
10	1.0001	1.0000	1	1
25	1	1	1	1

0.1	EE	EI	IE	II
2	1.0266	1.0265	1.0101	1.0100
3	1.0256	1.0253	1.0072	1.0071
4	1.0112	1.0111	1.0014	1.0013
5	1.0058	1.0057	1.0004	1.0003
10	1.0000	1.0000	1	1
25	1	1	1	1

1	EE	EI	IE	II
2	1.0198	1.0196	1.0065	1.0064
3	1.0226	1.0224	1.0064	1.0063
4	1.0118	1.0116	1.0018	1.0017
5	1.0075	1.0074	1.0008	1.0007
10	1.0000	1.0000	1	1
25	1	1	1	1

Table 6.2: Efficacy of filtering methods

answer to precisely this question.

We compared the performance of four different variants of the filtering algorithm (one which computes both spectral radii exactly, one which uses approximations for both, and two which approximate either the first or the second spectral radius while computing the other one from the definition) on matrices of varying dimensions and with different magnitudes of their radial matrices. For each combination of dimension and magnitude, we generated 250 random matrices, applied the direct and indirect interlacing methods on them to obtain an initial outer approximations, and then attempted to improve them with the four different versions of the filtering algorithm described above, evaluating the quality of the results by finding the quotient of the initial and filtered approximation's magnitudes (the larger the quotient, the more the interval was reduced).

Table 6.2 contains the quotients we obtained from our experiments. Here "EE", "EI", "IE" and "II" stand for the four different methods described above, "E" and "I" signifying, respectively, exact and inexact computation of the two spectral radii in the filtering algorithm; for example "EI" is the method where we use an exact computation for the spectral radius in the numerator and an approximation for the denominator. Our experiment uses the better of $||A||_1$ and $||A||_\infty$ as an approximation for the spectral radius. The three parts of the table present the results for three possible maximal magnitudes of the radial matrix, and the columns in each of these parts represent the size of the examined matrices.

0.01	EE	EI	IE	II
2	0.0651	0.0370	0.0290	0.0028
3	0.1007	0.0557	0.0440	0.0028
4	0.1265	0.0671	0.0571	0.0025
5	0.1579	0.0818	0.0743	0.0024
10	0.4166	0.2101	0.2091	0.0025
25	2.1332	1.0239	1.1182	0.0027

0.1	EE	EI	IE	II
2	0.0672	0.0378	0.0296	0.0029
3	0.0996	0.0547	0.0415	0.0026
4	0.1228	0.0652	0.0558	0.0025
5	0.1572	0.0815	0.0740	0.0024
10	0.4275	0.2157	0.2146	0.0025
25	2.1323	1.0276	1.1048	0.0027

1	EE	EI	IE	II
2	0.0750	0.0430	0.0328	0.0033
3	0.0987	0.0538	0.0426	0.0027
4	0.1241	0.0652	0.0569	0.0025
5	0.1604	0.0833	0.0748	0.0025
10	0.4275	0.2157	0.2146	0.0025
25	2.1136	1.0224	1.0956	0.0027

Table 6.3: Speed of filtering methods

A value of 1.000 indicates that the quotient in question is very close to but not equal to 1 (in other words, there was some minimal improvement done by the algorithm), whereas a value of 1 means that either no improvement was made at all, or it was so insignificant that it could no be stored in the computer's memory. Once again, we used the mean value of the quotient for all matrices in a given class and for all the eigenvalue intervals of those matrices.

Table 6.3 is structured in much the same way, and contains the average running times of the algorithms.

We can immediately see that the method which approximates the spectral radius in the denominator but computes the first one exactly gives results comparable to the original method (which computes both radii precisely), but works almost twice as fast (which was to be expected, considering that we avoid one verified computation at the cost of finding the sums of the rows and columns of the matrix). Therefore, it can safely be used as a quicker substitute for almost all practical applications. Substituting both spectral radius computations by estimates yields the worst results of all, but the time required for the computation is negligible, and some improvement can occur, so it could be used at practically no additional cost if time is of the essence. Indeed, the only variant which appears to have no useful application in the general case is "IE", i.e. approximating the spectral radius in the numerator and computing the one in the denominator; this version of the algorithm takes roughly as much time as "EI", but produced results only slightly better than the least precise "II".

6.3 Sample results

In order to illustrate the relative computation time and accuracy of the modes provided by the wrapper methods for outer and for inner approximations, we conducted several simple experiments which involved generating random matrices of given dimensions and measuring their mean time and performance; precision was measured relatively, by finding the quotient of the length of the interval produces by a given mode and the one obtained from using the *TIGHTEST* computation mode. The experiment was run for 100 matrices of each dimension, the values of the central matrices where taken between -10 and 10, and the magnitude of the radial matrices was generated randomly.

Table 6.4 lists the results for the outer approximations; the first part contains the average quotients (lower is better), the second one the average times. The modes *FASTEST* and *EFFECTIVE* give very similar results due to the fact that they both utilise the same interlacing method in the current implementation (the difference is in the filtering used). The *TIGHTER* mode produces much sharper approximations for a few of the tested matrices, but the average quotient is the same due to the fact that this happened quite rarely; in these cases the approximations were equal to the one obtained by the *TIGHTEST* mode. In practice, this means that one should only use the last two methods if computation time is not a concern, or if accuracy is crucial.

Accuracy	Fastest	Faster	Effective	Tighter	Test
3	1.1933	1.0020	1.0000	1.0000	1.000
5	1.2312	1.0094	1.0032	1.0022	1.000
10	1.2742	1.0025	1.0024	1.0023	1.000

Time	Fastest	Faster	Effective	Tighter	Test
3	0.0333	0.2481	0.4637	0.8486	1.2053
5	0.0664	0.6499	1.2098	2.1501	3.1190
10	0.1988	3.1436	6.4416	9.9883	14.7495

Table 6.4: Comparison of the outer approximation modes

The same experiment was conducted for *eigsymencinner.m*, with the difference that the quotient was inverted (in other words, the magnitude of the approximation produced by *TIGHTEST* was divided by the magnitude of each method's approximation in order to preserve the convention that lower quotients correspond to better results). Additionally, since there is a chance that the methods might fail to find an inner approximation, we also count the number of such failures for each method. Quotients were obviously not computed in these cases. Computation times always include finding all required data, such as outer approximations.

Table 6.5 illustrates the result of our experiment; note that the last part lists the total number of failures (out of a total of 100 examined matrices) instead of an arithmetic mean. Due to the exponential complexity of the direct submatrix vertex enumeration algorithm, computing its results for dimension 10 required so much computation time that we weren't able to complete the experiment for it; therefore we only provide data for the three fastest computation modes.

We should note that introducing a maximal number of steps to the vertex enumeration algorithm (and, possibly, some sort of method for randomly selecting different extremal matrices) might make it faster for practical applications while still more accurate than the local improvement method.

Accuracy	Fastest	Faster	Effective	Tighter	Test
3	1.0640	1.0640	1.0251	1.0000	1.0000
5	1.2425	1.2425	1.0370	1.0000	1.0000
10	?	?	?	?	?

Time	Fastest	Faster	Effective	Tighter	Test
3	0.1952	0.1989	0.4228	1.6161	2.0197
5	0.0675	0.9952	5.5656	22.9795	26.1576
10	4.1444	4.2011	990.6110	?	?

Failures	Fastest	Faster	Effective	Tighter	Test
3	1	1	1	1	1
5	6	0	0	0	0
10	2	0	0	0	0

Table 6.5: Comparison of the inner approximation modes

Conclusion

We have introduced the main principles of interval arithmetic and verification, and have presented a collection of theorems and algorithms that can be used to find approximations for the eigenvalue intervals of symmetric interval matrices. We saw that for most of the presented approaches a verified version could be constructed quite easily, but in some cases (namely, the computation of exact bounds using the Direct submatrix vertex enumeration algorithm for inner bounds) this conversion can be quite problematic.

The main result is, of course, the actual software implementation of the verified algorithms in the Matlab programming language, which can be used to compute verified approximations for the individual eigenvalue intervals, or the eigenvalue set as a whole, of any real symmetric interval matrix. In some degenerate cases, an inner approximation may not be found, but this is not due to a problem in the algorithms, but rather because of the nature of verification itself.

There is a lot of room for further work and a lot of potential improvements can be made to the project. Additional methods and modifications of the existing ones that might provide either faster or more precise computations can, of course, always be added to the implementation. Aside from that, a more careful study of the dependency between the characteristics of a given interval matrix and the relative efficiency of the direct versus the indirect interlacing methods (and, possibly, other methods as well) and the development of a more accurate criterion with which to select between the two methods can also be very useful, as it will increase the accuracy of the faster modes of computation. Finally, the direct submatrix vertex enumeration algorithm might be worthy of a deeper theoretical analysis for the purpose of finding a way to add verification to it while leaving the exact bound check intact; indeed, one of the most important features of the algorithm was its capability of finding (or confirming) exact bounds in certain cases; unfortunately, this promises to be a rather difficult task.

We will conclude by remarking that, as mentioned above, all the functions from our software library were written with flexibility in mind, so that a potential user might easily replace a "piece" of the program's logic with something else (for example, a better direct versus indirect criterion might be found, or a different index selector might be desired for the interlacing methods, either in general, or for working with a specific class of matrices) without modifying (or even having to understand) the rest of the program. This is especially relevant for the wrapper methods for outer and inner approximations, since the user can "fine-tune" the accuracy versus speed ratio of the different computation modes if the existing settings do not match his requirements for some reason. On the other hand, no changes are necessary, and the functions can be used immediately after installation.

Bibliography

[1] Götz Alefeld and Jürgen Herzberger. *Introduction to Interval Computations*. Computer Science and Applied Mathematics. Academic Press, New York, 1983.

[2] D. Hertz. The extreme eigenvalues and stability of real symmetric interval matrices. *Automatic Control, IEEE Transactions on*, 37(4):532–535, April 1992.

[3] Milan Hladík, David Daney, and Elias Tsigaridas. Bounds on real eigenvalues and singular values of interval matrices. *SIAM J. Matrix Anal. Appl.*, 31(4):2116–2129, 2010.

[4] Milan Hladík, David Daney, and Elias P. Tsigaridas. Characterizing and approximating eigenvalue sets of symmetric interval matrices. *Comput. Math. Appl.*, 62(8):3152–3163, 2011.

[5] Milan Hladík, David Daney, and Elias P. Tsigaridas. A filtering method for the interval eigenvalue problem. *Appl. Math. Comput.*, 217(12):5236–5242, 2011.

[6] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. SIAM, Philadelphia, PA, 2009.

[7] J. Rohn. Versoft: Guide. `http://www.nsc.ru/interval/Programing/versoft/guide.html`.

[8] J. Rohn. Bounds on eigenvalues of interval matrices. *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik*, 78(S3):1049–1050, 1998.

[9] J. Rohn. Solvability of systems of interval linear equations and inequalities. In *Linear Optimization Problems with Inexact Data*, pages 35–77. Springer US, 2006.

[10] Jiří Rohn. A handbook of results on interval linear problems. 2005.

[11] S.M. Rump. INTLAB - INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, Dordrecht, 1999. `http://www.ti3.tuhh.de/rump/`.