Interval Analysis in Matlab

Gareth I. Hargreaves

December 2002

MIMS EPrint: **2009.1**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

ISSN 1749-9097

Interval Analysis in Matlab

Gareth I. Hargreaves[*]

December 18, 2002

Abstract

The introduction of fast and efficient software for interval arithmetic, such as the MATLAB toolbox INTLAB, has resulted in the increased popularity of the use of interval analysis. We give an introduction to interval arithmetic and explain how it is implemented in the toolbox INTLAB. A tutorial is provided for those who wish to learn how to use INTLAB.

We then focus on the interval versions of some important problems in numerical analysis. A variety of techniques for solving interval linear systems of equations are discussed, and these are then tested to compare timings and accuracy. We consider univariate and multivariate interval nonlinear systems and describe algorithms that enclose all the roots.

Finally, we give an application of interval analysis. Interval arithmetic is used to take account of rounding errors in the computation of Viswanath's constant, the rate at which a random Fibonacci sequence increases.

1 Introduction

The concept of interval analysis is to compute with intervals of real numbers in place of real numbers. While floating point arithmetic is affected by rounding errors, and can produce inaccurate results, interval arithmetic has the advantage of giving rigorous bounds for the exact solution. An application is when some parameters are not known exactly but are known to lie within a certain interval; algorithms may be implemented using interval arithmetic with uncertain parameters as intervals to produce an interval that bounds all possible results.

If the lower and upper bounds of the interval can be rounded down and rounded up respectively then finite precision calculations can be performed using intervals, to give an enclosure of the exact solution. Although it is not difficult to implement existing algorithms using intervals in place of real numbers, the result may be of no use if the interval obtained is too wide. If this is the case, other algorithms must be considered or new ones developed in order to make the interval result as narrow as possible.

The idea of bounding rounding errors using intervals was introduced by several people in the 1950's. However, interval analysis is said to have begun with a book on the subject

[*] Department of Mathematics, University of Manchester, Manchester, M13 9PL, England (hargreaves@ma.man.ac.uk, http://www.ma.man.ac.uk/~hargreaves/).

by Moore [12] in 1966. Since then, thousands of articles have appeared and numerous books published on the subject.

Interval algorithms may be used in most areas of numerical analysis, and are used in many applications such as engineering problems and computer aided design. Another application is in computer assisted proofs. Several conjectures have recently been proven using interval analysis, perhaps most famously Kepler's conjecture [4], which remained unsolved for nearly 400 years.

The ability to alter the rounding mode on modern computers has allowed a variety of software to be produced for handling interval arithmetic, but only recently has it been possible to exploit high-performance computers. A specification for Basic Linear Algebra Subroutines is defined which covers linear algebra algorithms such scalar products, matrix–vector and matrix multiplication. These algorithms are collected in level 1, 2 and 3 BLAS. Computer manufacturers implement these routines so that BLAS are fast on their particular machines, which allows for fast portable codes to be written. Rump [17] showed that by expressing intervals by the midpoint and radius, interval arithmetic can be implemented entirely using BLAS. This gives a fast and efficient way of performing interval calculations, in particular, vector and matrix operations, on most computers. Rump used his findings to produce the MATLAB toolbox INTLAB, which will be used throughout this report.

The early sections of this report serve as an introduction to interval analysis and INTLAB. The properties of intervals and interval arithmetic are discussed with details of how they are implemented in INTLAB. A tutorial to INTLAB is given in Section 4 which shows how to use the various routines provided. Sections 5, 6 and 7 deal with solving interval linear systems of equations and interval nonlinear equations. Various methods are described, including the INTLAB functions for such problems, which are then tested. Finally an application of interval analysis is given: interval arithmetic is used to take account of rounding errors in the calculation of Viswanath's constant.

2 Interval Arithmetic

2.1 Notation

Intervals will be represented by boldface, with the brackets "$[\cdot]$" used for intervals defined by an upper bound and a lower bound. Underscores will be used to denote lower bounds of intervals and overscores will denote upper bounds. For intervals defined by a midpoint and a radius the brackets "$< \cdot >$" will be used.

2.2 Real Interval Arithmetic

A real interval $\boldsymbol{x}$ is a nonempty set of real numbers

$$\boldsymbol{x} = [\underline{x}, \overline{x}] = \{x \in \mathbb{R} : \underline{x} \leq x \leq \overline{x}\},$$

where $\underline{x}$ is called the *infimum* and $\overline{x}$ is called the *supremum*. The set of all intervals over $\mathbb{R}$ is denoted by $\mathbb{IR}$ where

$$\mathbb{IR} = \{[\underline{x}, \overline{x}] : \underline{x}, \overline{x} \in \mathbb{R}, \underline{x} \leq \overline{x}\}.$$

The *midpoint* of $\boldsymbol{x}$,

$$\operatorname{mid}(\boldsymbol{x}) = \check{x} = \frac{1}{2}(\underline{x} + \overline{x})$$

and the *radius* of $\boldsymbol{x}$,

$$\operatorname{rad}(\boldsymbol{x}) = \frac{1}{2}(\overline{x} - \underline{x}),$$

may also be used to define an interval $\boldsymbol{x} \in \mathbb{IR}$. An interval with midpoint a and radius r will be denoted by $<a, r>$. If an interval has zero radius it is called a *point interval* or *thin interval*, and contains a single point represented by

$$[x, x] \equiv x.$$

A *thick interval* has a radius greater than zero.

The *absolute value* or the *magnitude* of an interval $\boldsymbol{x}$ is defined as

$$|\boldsymbol{x}| = \operatorname{mag}(\boldsymbol{x}) = \max\{|\tilde{x}| : \tilde{x} \in \boldsymbol{x}\},$$

and the *mignitude* of $\boldsymbol{x}$ is defined as

$$\operatorname{mig}(\boldsymbol{x}) = \min\{|\tilde{x}| : \tilde{x} \in \boldsymbol{x}\}.$$

These can both be calculated using the end points of $\boldsymbol{x}$ by

$$\begin{aligned} \operatorname{mag}(\boldsymbol{x}) &= \max\{|\underline{x}|, |\overline{x}|\}), \\ \operatorname{mig}(\boldsymbol{x}) &= \begin{cases} \min(|\underline{x}|, |\overline{x}|) & \text{if } 0 \notin \boldsymbol{x} \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

An interval $\boldsymbol{x}$ is a *subset* of an interval $\boldsymbol{y}$, denoted by $\boldsymbol{x} \subseteq \boldsymbol{y}$, if and only if $\underline{y} \leq \underline{x}$ and $\overline{y} \geq \overline{x}$. The relation $\boldsymbol{x} < \boldsymbol{y}$ means that $\overline{x} < \underline{y}$, and other inequalities are defined in a similar way.

Interval arithmetic operations are defined on $\mathbb{IR}$ such that the interval result encloses all possible real results. Given $\boldsymbol{x} = [\underline{x}, \overline{x}]$ and $\boldsymbol{y} = [\underline{y}, \overline{y}]$, the four elementary operations are defined by

$$\boldsymbol{x} \operatorname{op} \boldsymbol{y} = \{x \operatorname{op} y : x \in \boldsymbol{x}, y \in \boldsymbol{y}\} \quad \text{for op} \in \{+, -, \times, \div\}. \tag{1}$$

Although (1) defines the elementary operations mathematically, they are implemented with

$$\begin{aligned} \boldsymbol{x} + \boldsymbol{y} &= [\underline{x} + \underline{y}, \overline{x} + \overline{y}], \\ \boldsymbol{x} - \boldsymbol{y} &= [\underline{x} - \overline{y}, \overline{x} - \underline{y}], \\ \boldsymbol{x} \times \boldsymbol{y} &= [\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}], \\ 1/\boldsymbol{x} &= [1/\overline{x}, 1/\underline{x}] \quad \text{if } \underline{x} > 0 \text{ or } \overline{x} < 0, \\ \boldsymbol{x} \div \boldsymbol{y} &= \boldsymbol{x} \times 1/\boldsymbol{y}. \end{aligned} \tag{2}$$

For the elementary interval operations, division by an interval containing zero is not defined. It is often useful to remove this restriction to give what is called *extended interval arithmetic*, which will be used in later sections. Extended interval arithmetic must satisfy (1), which leads to the following rules. If $\boldsymbol{x} = [\underline{x}, \overline{x}]$ and $\boldsymbol{y} = [\underline{y}, \overline{y}]$ with $\underline{y} \leq 0 \leq \overline{y}$ and $\underline{y} < \overline{y}$, then the rules for division are as follows:

$$\boldsymbol{x}/\boldsymbol{y} = \begin{cases} [\overline{x}/\underline{y}, \infty] & \text{if } \overline{x} \leq 0 \text{ and } \overline{y} = 0, \\ [-\infty, \overline{x}/\overline{y}] \cup [\overline{x}/\underline{y}, \infty] & \text{if } \overline{x} \leq 0 \text{ and } \underline{y} < 0 < \overline{y}, \\ [-\infty, \overline{x}/\overline{y}] & \text{if } \overline{x} \leq 0 \text{ and } \underline{y} = 0, \\ [-\infty, \infty] & \text{if } \underline{x} < 0 < \overline{x}, \\ [-\infty, \underline{x}/\underline{y}] & \text{if } \underline{x} \geq 0 \text{ and } \overline{y} = 0, \\ [-\infty, \underline{x}/\underline{y}] \cup [\underline{x}/\overline{y}, \infty] & \text{if } \underline{x} \geq 0 \text{ and } \underline{y} < 0 < \overline{y}, \\ [\underline{x}/\overline{y}, \infty] & \text{if } \underline{x} \geq 0 \text{ and } \underline{y} = 0. \end{cases} \tag{3}$$

The addition and subtraction of infinite or semi-infinite intervals are then defined by the following:

$$\begin{aligned} [\underline{x}, \overline{x}] + [-\infty, \overline{y}] &= [-\infty, \overline{x} + \overline{y}], \\ [\underline{x}, \overline{x}] + [\underline{y}, \infty] &= [\underline{x} + \underline{y}, \infty], \\ [\underline{x}, \overline{x}] + [-\infty, \infty] &= [-\infty, \infty], \\ [\underline{x}, \overline{x}] - [-\infty, \infty] &= [-\infty, \infty], \\ [\underline{x}, \overline{x}] - [-\infty, \overline{y}] &= [\underline{x} - \overline{y}, \infty], \\ [\underline{x}, \overline{x}] - [\underline{y}, \infty] &= [-\infty, \overline{x} - \underline{y}]. \end{aligned}$$

For further rules for extended interval arithmetic see [5].

For addition and multiplication the associative and commutative laws hold. However

$$\boldsymbol{x}(\boldsymbol{y} + \boldsymbol{z}) \neq \boldsymbol{x}\boldsymbol{y} + \boldsymbol{x}\boldsymbol{z},$$

except in special cases, therefore the distributive law does not hold. Instead there is the sub-distributive law

$$\boldsymbol{x}(\boldsymbol{y} + \boldsymbol{z}) \subseteq \boldsymbol{x}\boldsymbol{y} + \boldsymbol{x}\boldsymbol{z}. \tag{4}$$

Another example of a rule valid in real arithmetic that does not hold in $\mathbb{IR}$ is that for thick intervals,

$$\boldsymbol{x} - \boldsymbol{x} \neq 0.$$

As an example consider $\boldsymbol{x} = [2, 3]$, which gives $\boldsymbol{x} - \boldsymbol{x} = [-1, 1] \neq 0$. This is because an interval containing the difference between all possible results of two independent numbers lying within $\boldsymbol{x}$ is calculated, rather than the difference between two identical numbers.

An important result is the inclusion property, often labelled as the fundamental theorem of interval analysis.

Theorem 2.1. *If the function $f(\boldsymbol{z_1}, \ldots, \boldsymbol{z_n})$ is an expression with a finite number of intervals $\boldsymbol{z_1}, \ldots, \boldsymbol{z_n} \in \mathbb{IR}$ and interval operations $(+.-, \times, \div)$, and if*

$$\boldsymbol{x_1} \subseteq \boldsymbol{z_1}, \ldots, \boldsymbol{x_n} \subseteq \boldsymbol{z_n},$$

then

$$f(\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}) \subseteq f(\boldsymbol{z_1}, \ldots, \boldsymbol{z_n}).$$

Proof. From the definition of the interval arithmetic operations (1) it follows that, if $\boldsymbol{v} \subseteq \boldsymbol{w}$ and $\boldsymbol{x} \subseteq \boldsymbol{y}$ then

$$\begin{aligned} \boldsymbol{v} + \boldsymbol{x} &\subseteq \boldsymbol{w} + \boldsymbol{y} \\ \boldsymbol{v} - \boldsymbol{x} &\subseteq \boldsymbol{w} - \boldsymbol{y} \\ \boldsymbol{v}\boldsymbol{x} &\subseteq \boldsymbol{w}\boldsymbol{y} \\ \boldsymbol{v}/\boldsymbol{x} &\subseteq \boldsymbol{w}/\boldsymbol{y}. \end{aligned}$$

Combining this with the inclusion relation,

$$\boldsymbol{w} \subseteq \boldsymbol{x} \text{ and } \boldsymbol{x} \subseteq \boldsymbol{y} \implies \boldsymbol{w} \subseteq \boldsymbol{y},$$

and an inductive argument, the result follows. □

2.3 Interval Vectors and Matrices

An *interval vector* is defined to be a vector with interval components and the space of all n dimensional interval vectors is denoted by $\mathbb{IR}^n$. Similarly an *interval matrix* is a matrix with interval components and the space of all $m \times n$ matrices is denoted by $\mathbb{IR}^{m\times n}$. A point matrix or point vector has components all with zero radius, otherwise it is said to be a thick matrix or thick vector.

Arithmetic operations on interval vectors and matrices are carried out according to the operations on $\mathbb{IR}$ in the same way that real vector and matrices operations are carried out according to real operations. The comparison of interval vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{IR}^n$ is componentwise. For example $\boldsymbol{x} > \boldsymbol{y}$ means that $\boldsymbol{x}_i > \boldsymbol{y}_i$ for all i. The infinity norm of a vector $\boldsymbol{y} \in \mathbb{IR}^n$ is given by

$$\|\boldsymbol{y}\|_\infty = \max\{|\boldsymbol{y}_i| : i = 1, \ldots, n\}$$

We define two types of matrices of importance in Section 5. An interval matrix $\boldsymbol{A}$ is called an M-matrix if and only if $\boldsymbol{A}_{ij} \leq 0$ for all $i \neq j$ and $\boldsymbol{A}u > 0$ for some positive vector $u \in \mathbb{R}^n$. If the comparison matrix $\langle \boldsymbol{A} \rangle$, where

$$\begin{aligned} \langle \boldsymbol{A} \rangle_{ii} &= \min\{|\alpha| : \alpha \in \boldsymbol{A}_{ii}\}, \\ \langle \boldsymbol{A} \rangle_{ik} &= -\max\{|\alpha| : \alpha \in \boldsymbol{A}_{ik}\}, \end{aligned}$$

is an M-matrix then $\boldsymbol{A}$ is said to be an H-matrix.

2.4 Complex Interval Arithmetic

Intervals of complex numbers may be of two types. A rectangular complex interval is defined by two real intervals $\boldsymbol{x}$ and $\boldsymbol{y}$,

$$\boldsymbol{z} = \boldsymbol{x} + i\boldsymbol{y} = \{x + iy : x \in \boldsymbol{x}, y \in \boldsymbol{y}\},$$

and produces a rectangle of complex numbers in the complex plane with sides parallel to the coordinate axes. Complex interval operations are defined in terms of the real intervals $\boldsymbol{x} \in \mathbb{R}$ and $\boldsymbol{y} \in \mathbb{R}$ in the same way that complex operations on $z = x + iy$ are

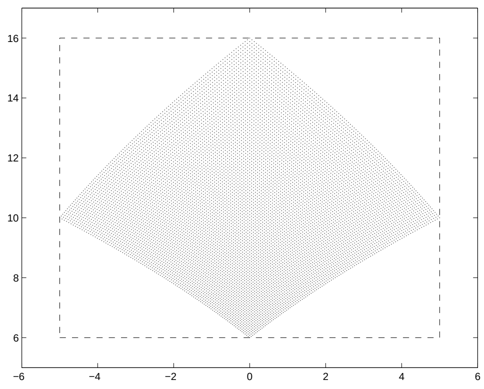


Figure 1: Example of complex interval multiplication.

defined in terms of x and y. For example, multiplication of two complex interval numbers $\boldsymbol{z_1} = \boldsymbol{x_1} + i\boldsymbol{y_1}$ and $\boldsymbol{z_2} = \boldsymbol{x_2} + i\boldsymbol{y_2}$ is defined by

$$\boldsymbol{z_1} \times \boldsymbol{z_2} = \boldsymbol{x_1}\boldsymbol{x_2} - \boldsymbol{y_1}\boldsymbol{y_2} + i(\boldsymbol{x_1}\boldsymbol{y_2} + \boldsymbol{x_2}\boldsymbol{y_1}), \tag{5}$$

where multiplication of the real intervals is as given by (2).

A problem with multiplication of rectangular complex intervals is that $\boldsymbol{x_1}\boldsymbol{x_2} - \boldsymbol{y_1}\boldsymbol{y_2} + i(\boldsymbol{x_1}\boldsymbol{y_2} + \boldsymbol{x_2}\boldsymbol{y_1})$ produces a rectangle in the complex plane, whereas the actual range is not this shape. Therefore an overestimation of the exact range of $\boldsymbol{z_1}\boldsymbol{z_2}$ is calculated. For example, if $\boldsymbol{z_1} = [1,2] + i[1,2]$ and $\boldsymbol{z_2} = [3,4] + i[3,4]$ then multiplication as defined by (5) gives the rectangle

$$\boldsymbol{z_1} \times \boldsymbol{z_2} = [-5,5] + i[6,16]. \tag{6}$$

However, $\{pq : p \in \boldsymbol{z_1}, q \in \boldsymbol{z_2}\}$, shown by the shaded region in Figure 1, is not rectangular but lies within the area given by (6), displayed with dashed lines.

An alternative interval is a circular complex interval $<a, r>$ which is a closed circular disk of radius r and center a:

$$\boldsymbol{z} = <a, r> = \{z \in \mathbb{C} : |z - a| \le r\}.$$

There are several possible definitions for multiplication of circular complex intervals which result in an overestimation that is less than when rectangular complex intervals are used. A simple definition is that introduced by Gargantini and Henrici [3]; for two complex intervals $\boldsymbol{z_1} = <a_1, r_1>$ and $\boldsymbol{z_2} = <a_2, r_2>$,

$$\boldsymbol{z_1} \times \boldsymbol{z_2} = <a_1 a_2, |a_1| r_2 + |a_2| r_1 + r_1 r_2>.$$

It can be shown that

$$\{pq : p \in \boldsymbol{z_1}, q \in \boldsymbol{z_2}\boldsymbol{\}} \subseteq \boldsymbol{z_1} \times \boldsymbol{z_2},$$

since

$$\begin{aligned} |pq - a_1 a_2| &= |a_1(q - a_2) + a_2(p - a_1) + (p - a_1)(q - a_2)| \\ &\le |a_1||q - a_2| + |a_2||p - a_1| + |p - a_1||q - a_2| \\ &\le |a_1| r_2 + |a_2| r_1 + r_1 r_2. \end{aligned}$$

2.5 Outward Rounding

The interval $\boldsymbol{x} = [\underline{x}, \overline{x}]$ may not be representable on a machine if $\underline{x}$ and $\overline{x}$ are not machine numbers. On a machine $\underline{x}$ and $\overline{x}$ must be rounded and the default is usually rounding to the nearest representable number. A rounded interval in this way, $\tilde{\boldsymbol{x}}$, may not bound the original interval $\boldsymbol{x}$.

In order that $\boldsymbol{x} \subseteq \tilde{\boldsymbol{x}}$, $\underline{x}$ must be rounded downward and $\overline{x}$ must be rounded upward, which is called *outward rounding*. The ubiquitous IEEE standard for floating point arithmetic [9] has four rounding modes, nearest, round down, round up and round towards zero, thus making interval arithmetic possible on essentially all current computers.

The MATLAB toolbox INTLAB uses the routine `setround` to change the rounding mode of the processor between nearest, round up and round down. This routine has been used to create functions for input, output and arithmetic of intervals. An introduction to INTLAB is given in the next section.

3 Introduction to INTLAB

Wherever MATLAB and IEEE 754 arithmetic is available INTLAB is able to be used. The INTLAB toolbox by Siegfried Rump is freely available from

`http://www.ti3.tu-harburg.de/~rump/intlab/index.html`

with information on how to install and use it. Here version 4 is used. All routines in INTLAB are MATLAB M-files, except for one routine, `setround`, for changing the rounding mode, which allows for portability and high speed on various systems.

3.1 Interval Input and Output

Real intervals in INTLAB are stored by the infimum and supremum, whereas complex intervals are stored by the midpoint and radius. However, this is not seen by the user. Intervals may be entered using either representation. For example the interval $\boldsymbol{x} = [1, 1]$ is entered using infimum and supremum as

```
>> x = infsup(-1,1);
```

but the same interval could be entered using the midpoint and radius as

```
>> x = midrad(0,1);
```

Since complex intervals are stored as a circular region using the midpoint and radius it is more accurate to input such intervals in this way. For example the circular region with midpoint at $1 + i$ and radius 1 is entered using

```
>> y = midrad(1+i,1);
```

If a rectangular region is entered using the infimum and supremum then the region is stored with an overestimation as the smallest circular region enclosing it. The infimum is entered as the bottom left point of the region and the supremum is the top right point. The region with a infimum of $1 + i$ and a supremum of $2 + 2i$ is entered as

```
>> z = infsup(1+i,2+2i);
```

however it is stored by the midpoint and radius notation as

```
>> midrad(z)
intval z =
<1.50000000000000 + 1.50000000000000i, 0.70710678118655>
```

Interval vectors and matrices are entered in a similar way with the arguments being vectors or matrices of the required size.

The function intval provides another way to enter an interval variable or can be used to change a variable to interval type. This function gives an interval with verified bounds of a real or complex value and is a vital part of verification algorithms. However, care has to be taken when using it. It is widely known that the value 0.1 cannot be expressed in binary floating point arithmetic so intval may be used to give a rigorous bound. Using

```
>> x = intval(0.1);
```

the variable x will not necessarily contain an interval including 0.1, since 0.1 is converted to binary format before being passed to intval. The result is a thin interval with

```
>> rad(x)
ans =
0
```

Rigorous bounds can be obtained using intval with a string argument such as

```
>> x = intval('0.1')
intval x =
[0.09999999999999, 0.10000000000001]
```

which uses an INTLAB verified conversion to binary. It can be seen that x contains 0.1 since the radius is nonzero.

```
>> rad(x)
ans =
1.387778780781446e-017
```

Using a string argument with more than one value will produce an interval column vector with components that are intervals that enclose the values entered. The structure may be altered to a matrix using reshape or a column vector using the transpose command “’”.

The infimum, supremum, midpoint and radius of a interval $\boldsymbol{x}$ can be obtained with the commands inf(x), sup(x), mid(x), rad(x) respectively. The default output is to display uncertainties by “_”. This means that an enclosure of the interval is given by the midpoint as the digits displayed and the radius as 1 unit of the last displayed figure. For example, the interval $\boldsymbol{y} = <1, 0.1>$ entered in midpoint/radius notation gives

```
>> y = midrad(1,0.1)
intval y =
1.0_____________
```

This output is the same as the interval that was entered. If the interval is changed to $\boldsymbol{y} = <1, 0.2>$ then an overestimation is given.

```
>> y = midrad(1,0.2)
intval y =
1.______________
```

This suggests that the interval is in the interior of $[0, 2]$ although it is stored as $[0.8, 1.2]$.

The default may be changed, or the commands infsup(x) and midrad(x) can be used to give the output in infimum/supremum or midpoint/radius notation. The output is as the notation from Section 2.1, for example

```
>> x = infsup(0,1); infsup(x), midrad(x)
intval x =
[0.00000000000000, 1.00000000000000]
intval x =
<0.50000000000000, 0.50000000000001>
```

and interval output is verified to give rigorous bounds.

3.2 INTLAB Arithmetic

INTLAB enables basic operations to be performed on real and complex interval scalars, vectors and matrices. These operations are entered similar to real and complex arithmetic in MATLAB. For example, if the matrix $\boldsymbol{A}$ is entered, then `A^2` performs $\boldsymbol{A}*\boldsymbol{A}$ in interval arithmetic, whereas `A.^2` results in each component of $\boldsymbol{A}$ being squared using interval arithmetic.

Standard functions such as trigonometric and exponential functions are available and used in the usual MATLAB way.

3.3 Matrix Multiplication

Matrix multiplication is often the most time consuming part of an algorithm so it is important to have fast interval matrix multiplication in order to obtain efficient implementations of interval algorithms. INTLAB distinguishes between three cases for real and complex matrices here; we will consider only real matrices.

If the matrices A and B are both point matrices, then a verified bound, $\boldsymbol{C} \in \mathbb{IR}^{n\times n}$, can be obtained as the solution to $A*B$. The rounding mode is set down and the infimum of the solution is calculated by $\underline{C} = A * B$. The rounding mode is then set up to give the supremum by $\overline{C} = A * B$. This gives a verified bound on the multiplication of two point matrices in $4n^3$ flops, where a *flop* is defined to be a floating point operation.

The second case is where one of the matrices is an interval matrix, say $\boldsymbol{B} \in \mathbb{IR}^{n\times n}$, and the other matrix is a point matrix. Three algorithms are presented in [18] of which the quickest uses the midpoint and radius of $\boldsymbol{B}$ and is the method used in INTLAB. The matrices $C_1 = A \times \text{mid}(\boldsymbol{B})$ and $C_2 = A \times \text{mid}(\boldsymbol{B})$ are calculated with the rounding mode set down and up respectively. The solution $\boldsymbol{C} = A * \boldsymbol{B}$ is then calculated using upward rounding with the midpoint matrix as

$$\text{mid}(\boldsymbol{C}) = C_1 + \frac{1}{2}(C_2 - C_1),$$

and the radius matrix given by

$$\text{rad}(\boldsymbol{C}) = \text{mid}(\boldsymbol{C}) - C_1 + \text{abs}(A) \times \text{rad}(\boldsymbol{B}),$$

where $\text{abs}(A)$ is the matrix of magnitudes of the components of A. The solution $\boldsymbol{C}$ is then converted so that it is stored by the infimum and supremum. This gives a bound on the solution by performing three point matrix multiplications, which requires $6n^3$ flops.

Finally, the two matrices may be both thick interval matrices. An algorithm similar to that described for verified point matrix multiplication is available which requires $16n^3$ flops. This produces a sharp bound on the solution but a quicker approach may be used when allowing a certain overestimation of the result. An alternative algorithm available in INTLAB calculates a bound for $\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$ by representing the matrices by the midpoint and radius. The matrices C_1 and C_2 are calculated by multiplying the midpoints of $\boldsymbol{A}$

and $\boldsymbol{B}$, with the rounding mode set down for C_1 and up for C_2. An outer estimate of the solution $\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$ is then calculated with upward rounding with the midpoint as

$$\operatorname{mid}(\boldsymbol{C}) = C_1 + \frac{1}{2}(C_2 - C_1),$$

and the radius given by

$$\operatorname{rad}(\boldsymbol{C}) = \operatorname{mid}(\boldsymbol{C}) - C_1 + \operatorname{rad}(\boldsymbol{A})(\operatorname{abs}(\operatorname{mid}(\boldsymbol{B})) + \operatorname{rad}(\boldsymbol{B})) + \operatorname{abs}(\operatorname{mid}(\boldsymbol{A})) \times \operatorname{rad}(\boldsymbol{B}).$$

This gives an overestimation of the solution in $8n^3$ flops. If the intervals of the matrices are narrow then the overestimation will be small, and for any size of interval the overestimation is limited by a factor of 1.5. The default is to use the fast interval matrix multiplication, but this can be changed so that the slow but sharp method is used.

3.4 Automatic Differentiation

Derivatives are important in numerical algorithms; for example in Section 7 derivatives are required for finding solutions of nonlinear equations. Symbolic differentiation is difficult for complicated expressions and packages such as Maple and Mathematica can be used to find derivatives, but these may produce large expressions which are time consuming to evaluate. An alternative often used is to obtain derivatives using finite differences. This overcomes the problem of finding an expression for the derivative, but truncation and rounding errors are introduced which can lead to inaccurate results.

An alternative is to use automatic differentiation, in which derivatives are computed using the chain rule for composite functions. This method produces results accurate except for rounding errors. However, if a variable is of interval type then interval arithmetic can be used to calculate an enclosure of the true derivative. INTLAB has an implementation of *forward mode* automatic differentiation, so called due to the forward substitution required to find the derivatives.

Suppose that $f : \mathbb{R}^m \to \mathbb{R}^n$ is a function given by an expression in which only elementary operations occur, such as $+, -, \times, /, \sin, \cos,$ etc. The expression for $f(x),\ x = (x_1, \dots, x_m)$ can be decomposed into a list of equations representing the function,

$$\begin{aligned} t_i(x) &= g_i(x) = x_i & i &= 1, \dots, m, \\ t_i(x) &= g_i(t_1(x), \dots, t_{i-1}(x)) & i &= m+1, \dots, l, \end{aligned} \tag{7}$$

where t_i and g_i are scalar functions and only one elementary operation occurs in each $g(i)$.

The functions g_i are all differentiable since the derivatives of the elementary operations are known. Using the chain rule on (7) gives

$$\frac{\partial t_i}{\partial t_j} = \delta_{ij} + \sum_{k=j}^{i-1} \frac{\partial g_i}{\partial t_k} \frac{\partial t_k}{\partial t_j} \quad \text{where } \delta_{ij} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

Writing $t = (t_1, \dots, t_l)$ and $g = (g_1, \dots, g_l)$, (7) can be written as $t = g(t)$. Equation (8) can be written as

$$Dt = I + DgDt,$$

where

$$Dg = \left\{ \frac{\partial g_i}{\partial dt_j} \right\} = \begin{pmatrix} 0 & \cdots & & \\ \frac{\partial g_2}{\partial t_1} & 0 & \cdots & \\ \frac{\partial g_3}{\partial t_1} & \frac{\partial g_3}{\partial t_2} & 0 & \cdots \\ \vdots & \ddots & \ddots & \ddots \end{pmatrix},$$

and

$$Dt = \left\{ \frac{\partial t_i}{\partial t_j} \right\} = \begin{pmatrix} 1 & 0 & \cdots & \\ \frac{\partial t_2}{\partial t_1} & 1 & \ddots & \\ \frac{\partial t_3}{\partial t_1} & \frac{\partial t_3}{\partial t_2} & 1 & \ddots \\ \vdots & \ddots & \ddots & \ddots \end{pmatrix}.$$

This implies that $(I - Dg)Dt = I$, which can be solved for any of the columns in Dt using forward substitution. To find all the partial derivatives of f, the first m columns of Dt can be computed.

An interval scalar, vector or matrix $\boldsymbol{x}$ is initialised to be of gradient type by the following command.

```
>> x = gradientinit(x)
```

If an expression involving a variable of gradient type is computed then the value and gradient are stored in the substructures `.x` and `.dx` respectively. For example, if an M-file is created as

```
function y = f(x)
y = 2*x^2+sin(x);
```

the thin interval $\boldsymbol{x} = [1, 1]$ can be initialised to be of gradient type

```
>> x = intval(1);
>> x = gradientinit(x);
```

The values $f(1)$ and $f'(1)$ can be evaluated by

```
>> y = f(x)
intval gradient value y.x =
[2.84147098480789, 2.84147098480790]
intval gradient derivative(s) y.dx =
[4.54030230586813, 4.54030230586815]
```

which gives intervals containing $f(1)$ and $f'(1)$. Entering `y.x` gives $f(\boldsymbol{x})$ and `y.dx` gives the partial derivatives of f.

4 INTLAB Tutorial

Here we give a brief tutorial to using INTLAB. The user should work through this tutorial by typing what appears after the command prompt ">>". It is assumed that the user is familiar with MATLAB and has INTLAB correctly installed. To start INTLAB type:

```
>> startintlab
```

This adds the INTLAB directories to the MATLAB search path and initialises the required global variables.

4.1 Input and Output

There are four ways to enter a real interval. The first is the function `intval` which allows the input of a floating point matrix. The following creates a 2 × 2 matrix of point intervals.

```
>> A = intval([0.1,2;3,4])
intval A =
   0.1000   2.0000
   3.0000   4.0000
```

The matrix will not necessarily contain an interval enclosing 0.1, since 0.1 is converted to binary format before `intval` is called. Using a string argument overcomes this problem. This example creates a vector with interval components that encloses the values entered.

```
>> A = intval('0.1 2 3 4')
intval A =
    0.1000
    2.0000
    3.0000
    4.0000
```

The first component now encloses 0.1. This can be seen since the radius is nonzero.

```
>> rad(A(1,1))
ans =
  1.3878e-017
```

Notice that `A` is a column vector. All output using a string argument produces a vector of this form. It may be changed to a 2 × 2 matrix by:

```
>> A = reshape(A,2,2)
intval A =
    0.1000    3.0000
    2.0000    4.0000
```

The third alternative is to give an interval by its midpoint and radius.

```
>> B = midrad([1,0;0.5,3],1e-4)
intval B =
    1.000_    0.000_
    0.500_    3.0000
```

Finally an interval may be input by its infimum and supremum.

```
>> C = infsup([-1,0;2,4],[-0.9,0;2.4,4.01])
intval C =
   -0.9___    0.0000
    2.____    4.00__
```

Complex intervals can be entered by midpoint/radius notation with the midpoint as a complex number.

```
>> c = midrad(3-2i,0.01)
 c =
   3.00__ -  2.00__i
```

If a complex number has midpoint on the real line the above method will result in a real interval. Instead the following should be used.

```
>> z = cintval(2,0.01)
intval z =
   2.00__ +  0.0___i
```

This can also be used to enter any complex interval by its midpoint and radius.

The default output is to display uncertainties by "_". This means that an enclosure of the interval is given by the midpoint as the digits displayed and the radius as 1 unit of the last displayed figure.

The midpoint, radius, infimum and supremum of an interval may be obtained.

```
>> x = infsup(0.99,1.01);
>> mid(x)
ans =
     1
>> rad(x)
ans =
    0.0100
>> inf(x)
ans =
    0.9900
>> sup(x)
ans =
    1.0100
```

Also an interval may be output in midpoint/radius of infimum/supremum notation.

```
>> infsup(x)
intval x = [0.9899,    1.0101]
>> midrad(x)
intval x = <1.0000,   0.0101>
```

The default output may be changed to display all intervals in one of these two notations. To change to midpoint/radius:

```
>> intvalinit('displaymidrad')
===> Default display of intervals by midpoint/radius
>> x
intval x = <1.0000,   0.0101>
```

To change back to the original default:

```
>> intvalinit('display_')
===> Default display of intervals with uncertainty
>> x
intval x =
    1.00__
```

To change to infimum/supremum:

```
>> intvalinit('displayinfsup')
===> Default display of intervals by infimum/supremum
>> x
intval x = [0.9899,    1.0101]
```

The MATLAB output format may also be changed to give varying precision in the output of intervals.

```
>> format long; x
intval x =
[   0.98999999999998,   1.01000000000001]
>> format short; x
intval x =
[   0.9899,   1.0101]
```

4.2 Arithmetic

All the standard arithmetic operations can be used on intervals. An arithmetic operation uses interval arithmetic if one of the operands is of interval type.

```
>> y = 3*midrad(3.14,0.01)^2
intval
y = [   29.3906,   29.7676]
```

For multiplication of two matrices containing thick intervals, the default is to use a fast matrix multiplication which gives a maximum overestimation by a factor of 1.5 in radius.

```
>> A = midrad(rand(100),1e-4);
>> tic; A*A; toc;
elapsed_time =
    0.0500
```

This can be changed to the slower but sharper matrix multiplication.

```
>> intvalinit('sharpivmult')
===> Slow but sharp interval matrix multiplication in use
>> tic; A*A; toc;
elapsed_time =
    1.1000
```

To change back to the default:

```
>> intvalinit('fastivmult')
===> Fast interval matrix multiplication in use
    (maximum overestimation factor 1.5 in radius)
```

Standard functions, such as $\sin, \cos$ and $\log$ can be used with intervals. The default is to use rigorous standard functions which have been verified to give correct and narrow enclosures. The alternative is to use faster approximate functions which give a wider enclosure.

```
>> x = intval(1e5);
>> intvalinit('approximatestdfcts'), rad(sin(x))
===> Faster but not rigorous standard functions in use
ans =
 2.2190e-011
>> intvalinit('rigorousstdfcts'), rad(sin(x))
===> Rigorous standard functions in use
ans =
  1.3878e-017
```

INTLAB provides functions for many of the properties of intervals from Section 2.2. The intersection of two intervals can be calculated:

```
>> x = infsup(-1,2); y = infsup(1.5,3);
>> intersect(x,y)
intval ans =
    [    1.5000,    2.0000]
```

The union of two intervals may be obtained as follows:

```
>> hull(x,y)
intval ans =
    [   -1.0000,    3.0000]
```

The magnitude of an interval is given by:

```
>> abss(x)
ans =
     2
```

The mignitude is given by.

```
>> mig(x)
ans =
     0
```

A logical function can be used to test if an interval is included in another.

```
>> p = infsup(-1,1); in(p,x)
ans =
     1
```

Alternatively it is possible to test whether an interval lies in the interior of another.

```
>> in0(p,x)
ans =
     0
```

In this case the result is false since the infimum of `p` and `x` are the same.

4.3 Verified functions

INTLAB provides a function which either performs a verified calculation of the solution of a linear system of equations, or computes an enclosure of the solution hull of an interval linear system. The following example gives verified bounds on the solution of linear system of equations.

```
>> A = rand(5); b = ones(5,1);
>> verifylss(A,b)
intval ans =
    [    0.6620,    0.6621]
    [    0.7320,    0.7321]
    [    0.5577,    0.5578]
    [   -0.2411,   -0.2410]
    [    0.0829,    0.0830]
```

If a component of the linear system is of interval type then the backslash command can be used to produce an enclosure on the solution hull.

```
> A = midrad(A,1e-4);
>> A\b
intval ans =
    [    1.8192,    1.8365]
    [   -0.2210,   -0.2167]
    [   -1.1849,   -1.1618]
    [   -0.1182,   -0.1129]
    [    1.1484,    1.1555]
```

If `A` is sparse then a sparse solver is automatically used.

Verified solutions of nonlinear equations may be found using `verifynlss`. One dimensional nonlinear functions can be entered directly with the unknown as “x”.

```
>> verifynlss('2*x*exp(-1)-2*exp(-x)+1',1)
intval ans =
    [    0.4224,    0.4225]
```

The second parameter is an approximation to a root. To solve a system of nonlinear equation, a function has to be created. For example, suppose the following function is entered as an M-file.

```
function y = f(x)
y = x;
y(1) = 3*x(1)^2-x(1)+3*x(2)-5;
y(2) = 4*x(1)+2*x(1)^2+x(2)-7;
```

This nonlinear system can be solved for one of the roots, $y = (1, 1)^T$, provided a close enough approximation is given.

```
>> verifynlss(@f,[2,3])
intval ans =
    [    0.9999,    1.0001]
    [    0.9999,    1.0001]
```

Verified bounds may also be obtained for eigenvalue/eigenvector pairs.

```
>> intvalinit('displaymidrad'); A = wilkinson(9);
===> Default display of intervals by midpoint/radius
>> [V,D] = eig(A);
>> [L,X] = verifyeig(A,D(1,1),V(:,1))
intval L =
    <   -1.1254 +  0.0000i,  0.0001>
intval X =
    <   -0.0074,   0.0001>
    <    0.0381,   0.0001>
    <   -0.1497,   0.0001>
    <    0.4297,   0.0001>
    <   -0.7635,   0.0001>
    <    0.4297,   0.0001>
    <   -0.1497,   0.0001>
    <    0.0381,   0.0001>
    <   -0.0074,   0.0001>
```

This generates the 9 × 9 Wilkinson eigenvalue test matrix, approximates the eigenvalues and eigenvectors of A, and then produces verified bounds for the eigenvalue/eigenvector pair approximated by D(1,1) and V(:,1).

Wilkinson's eigenvalue test matrices have pairs of eigenvalues that are nearly but not exactly the same. The function verifyeig can be used to bound clusters of eigenvalues.

```
>> verifyeig(A,D(8,8),V(:,8:9))
intval ans =
    <   4.7462 +  0.0000i,  0.0010>
```

This produces bounds on eigenvalues approximated by D(8,8) and D(9,9). The required input is an approximation of an eigenvalue, in this case D(8,8), and an approximation of the corresponding invariant subspace, which is given by V(:,8:9).

4.4 Rounding mode

The most important function in INTLAB is the function setround which allows the rounding mode of the processor to be changed between nearest, round down and round up. This routine allows interval arithmetic and the numerous other functions to be implemented.

The current rounding mode may be accessed by:

```
>> y = getround
y =
     0
```

The result y = 0 means that the rounding mode is currently set to nearest. This may be changed to rounding downwards by

```
>> setround(-1)
```

or rounding upwards by

```
>> setround(1)
```

It is good practice to set the rounding mode to the original setting at the end of a routine.

```
>> setround(y)
```

As an example of the effect changing the rounding mode makes, try entering the following:

```
>> format long
>> rndold = getround; setround(-1); x = 0;
>> for i = 1:100000, x = x+0.1; end
>> setround(rndold); x
x =
    9.999999999947979e+003
```

Now try changing the rounding mode to nearest and upward rounding, and observe the difference in result.

4.5 Gradients

INTLAB contains a gradient package which uses automatic differentiation. An example of initialising a variable to be used with the gradient package is given by:

```
>> u = gradientinit(2)
gradient value u.x =
    2
gradient derivative(s) u.dx =
    1
```

Bounds on the derivative of a function may be obtained by initialising an interval to be used with the gradient package.

```
>> intvalinit('displayinfsup');
===> Default display of intervals by infimum/supremum
>> x = gradientinit(infsup(0.999,1.001))
intval gradient value x.x =
[    0.9989,    1.0010]
intval
gradient derivative(s) x.dx =
[    1.0000,    1.0000]
```

Automatic differentiation is performed on an expression if a variable is of gradient type.

```
>> y = sin(x)*(4*cos(x)-2)^2
intval gradient value y.x =
[    0.0209,    0.0229]
intval gradient derivative(s) y.dx =
[   -0.9201,   -0.8783]
```

The values of the expression and the derivative are stored as `y.x` and `y.dx` respectively.

```
>> y.x, y.dx
intval ans =
[    0.0209,    0.0229]
intval ans =
[   -0.9201,   -0.8783]
```

For functions of several unknowns the gradient package can be used to obtain the Jacobian matrix. For example, bounds on the Jacobian of the function stored in the M-file `f.m` from Section 4.3 can be evaluated.

```
>> x = gradientinit(midrad([1,2],1e-6));
 >> y = f(x)
 intval gradient value y.x =
 [    2.9999,    3.0001] [    0.9999,    1.0001]
 intval gradient derivative(s) y.dx =
 intval (:,:,1) =
 [    4.9999,    5.0001] [    7.9999,    8.0001]
 intval (:,:,2) =
 [    3.0000,    3.0000] [    1.0000,    1.0000]
```

Bounds are produced on the Jacobian of the function `f.m`, and are stored as `y.dx`.

5 Linear Systems of Equations

Linear systems of equations are a fundamental part of scientific calculations. In this section we consider bounding the solution set of interval linear systems.

5.1 Systems of Interval Equations

The solution to linear systems of equations is prone to errors due to the finite precision of machine arithmetic and the propagation of error in the initial data. If the initial data is known to lie in specified ranges then interval arithmetic enables computation of intervals containing the elements of the exact solution. Error bounds are provided in the calculation of the interval solution rather than estimating the error from an analysis of the error propagation after an approximate solution is obtained.

An interval linear system is of the form $\boldsymbol{Ax} = \boldsymbol{b}$, where $\boldsymbol{A} \in \mathbb{IR}^{n\times n}$ and $\boldsymbol{b} \in \mathbb{IR}^n$. The solution set

$$\Sigma(\boldsymbol{A}, \boldsymbol{b}) = \{\tilde{x} : \tilde{A}\tilde{x} = \tilde{b} \text{ for some } \tilde{A} \in \boldsymbol{A}, \tilde{b} \in \boldsymbol{b}\},$$

is typically star-shaped and expensive to compute. For example, consider the system where

$$\boldsymbol{A} = \begin{pmatrix} [1,3] & [-1,2] \\ [-1,0] & [2,4] \end{pmatrix},$$

and

$$\boldsymbol{b} = \begin{pmatrix} [-2,2] \\ [-2,2] \end{pmatrix}.$$

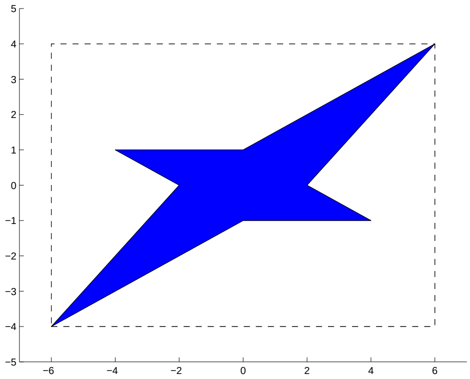


Figure 2: Solution set and hull of a linear system.

The solution set is shown by the shaded area of Figure 2. The *hull* of the solution set is the interval vector with smallest radius containing $\Sigma(\boldsymbol{A}, \boldsymbol{b})$ and is denoted by $\square\Sigma(\boldsymbol{A}, \boldsymbol{b})$. For the previous example this is shown by the dashed line in Figure 2. An outer estimate of $\square\Sigma(\boldsymbol{A}, \boldsymbol{b})$ is sought.

5.2 Interval Gaussian Elimination

An obvious approach is to use a generalisation of Gaussian elimination adapted to deal with interval coefficients. A triangular system can be formed in the usual way but with interval arithmetic. By the inclusion property, Theorem 2.1, the solution of this triangular system will give an inclusion of the solution set.

The usual care has to be taken with division by zero. Column mignitude pivoting can be used to choose a pivot as the contender with the largest mignitude, where we recall that the mignitude of $\boldsymbol{x}$ is defined as

$$\operatorname{mig}(\boldsymbol{x}) = \begin{cases} \min(|\underline{x}|, |\overline{x}|) & \text{if } 0 \notin \boldsymbol{x} \\ 0 & \text{otherwise.} \end{cases}$$

An implementation written in INTLAB of interval Gaussian elimination with mignitude pivoting is given by the function `intgauss.m` in Appendix A.1.

When interval Gaussian elimination is applied to a general $\boldsymbol{A} \in \mathbb{IR}^{n \times n}$ and $\boldsymbol{b} \in \mathbb{IR}^{n}$ problems are soon encountered as n is increased. As interval calculations are carried out in the Gaussian elimination process the widths of the interval components grow larger due to the nature of interval arithmetic. If a solution is obtained then it is likely that the width of the components are very large. Alternatively, at some stage in the Gaussian elimination process all contenders for pivot, or the bottom right element in the upper triangular system, contain zero, which causes the algorithm to break down due to division by zero. For example, if the coefficient matrix is

$$\boldsymbol{A} = \begin{pmatrix} [0.95, 1.05] & [1.95, 2.05] & [2.95, 3.05] \\ [1.95, 2.05] & [3.95, 4.05] & [6.95, 7.05] \\ [1.95, 2.05] & [-0.05, 0.05] & [0.95, 1.05] \end{pmatrix},$$

then the upper triangular system is given by

$$\boldsymbol{U} = \begin{pmatrix} [1.95, 1.05] & [3.95, 4.05] & [6.95, 7.05] \\ [0, 0] & [-4.31, -3.71] & [-6.46, -5.56] \\ [0, 0] & [0, 0] & [-1.23, 0.23] \end{pmatrix}.$$

This causes division by zero when using back-substitution. All the elements began with a radius of 0.05, but the radius of $\boldsymbol{U}_{3,3}$ is 0.7301.

The feasibility of using `intgauss.m` depends on the matrix $\boldsymbol{A} \in \mathbb{IR}^{n\times n}$. For a general $\boldsymbol{A}$, problems may occur for dimensions as low as $n = 3$ if the radii of the elements are too large. As the width of the elements decreases the algorithm becomes feasible for larger n. However, even when `intgauss.m` is used with thin matrices, it is likely for the algorithm to break down for n larger than 60.

Despite interval Gaussian elimination not being effective in general, it is suitable for certain classes of matrices. In particular, realistic bounds for the solution set are obtained for M-matrices, H-matrices, diagonally dominant matrices, tridiagonal matrices and 2×2 matrices. In the case where $\boldsymbol{A}$ is an M-matrix the exact hull $\square\Sigma(\boldsymbol{A}, \boldsymbol{b})$ is obtained for many $\boldsymbol{b}$; Neumaier [13] shows that if $\boldsymbol{b} \geq 0$, $\boldsymbol{b} \leq 0$ or $0 \in \boldsymbol{b}$ then the interval hull of the solution set is obtained.

5.3 Krawczyk's Method

The linear interval system $\boldsymbol{Ax} = \boldsymbol{b}$ can be preconditioned by multiplying by a matrix $C \in \mathbb{R}^{n\times n}$. Here, we choose C to be the inverse of the midpoint matrix of $\boldsymbol{A}$, which often leads to the matrix $C\boldsymbol{A}$ being an H-matrix. If this is the case then Gaussian elimination can be used, but it is quicker to compute an enclosure of the solution by Krawczyk's method.

Assuming an interval vector $\boldsymbol{x}^{(i)}$ is known such that $\square\Sigma(\boldsymbol{A}, \boldsymbol{b}) \subseteq \boldsymbol{x}^{(i)}$ then

$$\tilde{A}^{-1}\tilde{b} = C\tilde{b} + (I - C\tilde{A})\tilde{A}^{-1}\tilde{b} \in C\boldsymbol{b} + (I - C\boldsymbol{A})\boldsymbol{x}^{(i)}$$

holds for all $\tilde{A} \in \boldsymbol{A}$ and $\tilde{b} \in \boldsymbol{b}$, so that

$$\square\Sigma(\boldsymbol{A}, \boldsymbol{b}) \subseteq \boldsymbol{x}^{(i)} \implies \square\Sigma(\boldsymbol{A}, \boldsymbol{b}) \subseteq (C\boldsymbol{b} + (I - C\boldsymbol{A})\boldsymbol{x}^{(i)}) \cap \boldsymbol{x}^{(i)}. \tag{9}$$

This gives the *Krawczyk iteration*

$$\boldsymbol{x}^{(i+1)} = (C\boldsymbol{b} + (I - C\boldsymbol{A})\boldsymbol{x}^{(i)}) \cap \boldsymbol{x}^{(i)}. \tag{10}$$

To start the iteration we require an initial vector $\boldsymbol{x}^{(0)}$ such that the solution $\tilde{x} \in \boldsymbol{x}^{(0)}$ and $\boldsymbol{x}^{(0)} \supseteq \square\Sigma(\boldsymbol{A}, \boldsymbol{b})$. A possible $\boldsymbol{x}^{(0)}$ can be found with the aid of the following theorem.

Theorem 5.1. *If C satisfies $||I - C\tilde{A}|| = \beta < 1$, $\tilde{x} = \tilde{A}^{-1}\tilde{b}$ and $\|\cdot\|$ is any subordinate norm, then*

$$\|\tilde{x}\| \leq \frac{\|C\tilde{b}\|}{1-\beta}.$$

Proof. From $\tilde{A}\tilde{x} = \tilde{b}$ we have $\tilde{x} = C\tilde{b} + (I - C\tilde{A})\tilde{x}$, and hence

$$\begin{aligned}\|\tilde{x}\| &\leq \|C\tilde{b}\| + \|I - C\tilde{A}\|\|\tilde{x}\|,\\ &\leq \|C\tilde{b}\| + \beta\|\tilde{x}\|,\end{aligned}$$

which gives the result. □

Since $\|C\tilde{b}\|_\infty \leq \|C\boldsymbol{b}\|_\infty$ and $\beta < 1$ is very likely for C being the inverse of the midpoint matrix of $\boldsymbol{A}$, we define the initial interval vector to be

$$\begin{aligned} \boldsymbol{x}^{(0)} &= ([-\alpha, \alpha], \ldots, [-\alpha, \alpha])^T \quad \text{with} \\ \alpha &= \frac{\|C\boldsymbol{b}\|_\infty}{1-\beta}. \end{aligned}$$

The iterations can be terminated if the radii of the components of $\boldsymbol{x}^{(i)}$ are no longer rapidly decreasing. The sum of these radii can be computed after each iteration and compared with the previous sum. An INTLAB implementation of the algorithm is given by the function `kraw.m` in Appendix A.1.

5.4 The Hansen-Bliek-Rohn-Ning-Kearfott-Neumaier Method

A bound for the interval hull of the solution set of linear interval equations is given by Hansen [6] for the case where the midpoint matrix of $\boldsymbol{A}$ is the identity. This result was also found by Bliek [1], but it was Rohn [16] who first gave a rigorous proof. Ning and Kearfott [15] generalised the result for the case when $\boldsymbol{A}$ is an H-matrix. This is of particular interest since the coefficient matrix can be preconditioned in an attempt to produce an H-matrix.

The method is based on the following theorem which uses the comparison matrix, $\langle \boldsymbol{A} \rangle$, defined in Section 2.3.

Theorem 5.2. *Let $\boldsymbol{A} \in \mathbb{IR}^{n\times n}$ be an H-matrix, $\boldsymbol{b} \in \mathbb{IR}^n$ a right hand side,*

$$u = \langle \boldsymbol{A} \rangle^{-1} |\boldsymbol{b}|, \quad d_i = (\langle \boldsymbol{A} \rangle^{-1})_{ii},$$

and

$$\alpha_i = \langle \boldsymbol{A}_{ii} \rangle - 1/d_i, \quad \beta_i = u_i/d_i - |\boldsymbol{b}_i|.$$

Then $\square\Sigma(A, b)$ is contained in the vector $\boldsymbol{x}$ with components

$$\boldsymbol{x}_i = \frac{\boldsymbol{b}_i + [-\beta_i, \beta_i]}{\boldsymbol{A}_{ii} + [-\alpha_i, \alpha_i]}.$$

A simplified proof is given by Neumaier [14].

In order to give a rigorous enclosure of the interval hull using floating point arithmetic, rigorous upper bounds are required for α_i and β_i. These are obtained if a rigorous bound B for $\langle \boldsymbol{A} \rangle^{-1}$ is used. The following explanation of how this is achieved is based on that given in [14].

A property of the H-matrix $\boldsymbol{A}$ is that $\langle \boldsymbol{A} \rangle^{-1}$ is nonnegative. This suggests that an upper bound B for $\langle \boldsymbol{A} \rangle^{-1}$ can be expressed in terms of $\tilde{B}$, an approximation to $\langle \boldsymbol{A} \rangle^{-1}$, and vectors $v \in \mathbb{R}^n, w \in \mathbb{R}^n$ satisfying $I - \langle \boldsymbol{A} \rangle \tilde{B} \leq \langle \boldsymbol{A} \rangle v w^T$ by

$$B = \tilde{B} + vw^T. \tag{11}$$

By the definition of an H-matrix, there exists a vector $v > 0$ such that $u = \langle \boldsymbol{A} \rangle v > 0$. This vector v can be used to satisfy (11) by taking the vector w with components

$$w_k = \max_i \frac{-R_{ik}}{u_i},$$

where

$$R = \langle \boldsymbol{A} \rangle \tilde{B} - I.$$

It is now left to find the vector v. Assuming there is a positive vector $\tilde{u}$ such that $v = \tilde{B}\tilde{u} \approx \langle \boldsymbol{A} \rangle^{-1}\tilde{u} > 0$ then A is an H-matrix, and if $u = \langle \boldsymbol{A} \rangle v \approx \tilde{u}$ is positive then the approximation $\tilde{B}$ is good enough. Since $\langle \boldsymbol{A} \rangle^{-1}$ is nonnegative, the vector $\tilde{u} = (1, \dots, 1)$ is sufficient to produce $\langle \boldsymbol{A} \rangle^{-1}\tilde{u} > 0$.

The values u and R must be calculated with downward rounding, w and B calculated with upward rounding, while $\tilde{B}$ and v can be calculated with nearest rounding. The function `hsolve.m` in Appendix A.1 implements the method described above.

5.5 INTLAB function

INTLAB provides the function `verifylss` for solving interval linear systems or for giving verified bounds on a point system. If any of the elements of $\boldsymbol{A}$ or $\boldsymbol{b}$ are of interval type then the "\" command is equivalent to calling `verifylss`. The function is also used for solving sparse systems, overdetermined systems and underdetermined systems, but here we discuss the case where $\boldsymbol{A} \in \mathbb{IR}^{n\times n}$ is dense.

The dense linear system solver first attempts to solve the system by an iterative method based on the Krawczyk operator, but the method is different to the method described in Section 5.3. Using as the preconditioner C the inverse of the midpoint matrix of $\boldsymbol{A}$, an approximate solution x_s is calculated by multiplying C by the midpoint vector of $\boldsymbol{b}$. Applying (9) to an enclosure $\boldsymbol{d^{(i)}}$ of $\square\Sigma(\boldsymbol{A}, \boldsymbol{b} - \boldsymbol{A}x_s)$ gives the *residual Krawczyk iteration*

$$\boldsymbol{d^{(i+1)}} = (C(\boldsymbol{b} - \boldsymbol{A}x_s) + (I - C\boldsymbol{A})\boldsymbol{d^{(i)}}) \cap \boldsymbol{d^{(i)}}.$$

The implication

$$C\boldsymbol{b} + (I - C\boldsymbol{A})\boldsymbol{x} \subseteq \text{ interior of } \boldsymbol{x} \implies \square\Sigma(\boldsymbol{A}, \boldsymbol{b}) \subseteq \boldsymbol{x},$$

a consequence of Brouwer's fixed point theorem [2], is used to determine whether or not to terminate the algorithm. After an inclusion of the difference has been found, the approximate solution x_s is added to give an enclosure of $\square\Sigma(\boldsymbol{A}, \boldsymbol{b})$.

We can now show how the enclosure given by this method and the method described in Section 5.3 compare. Since

$$\begin{aligned} x_s + C(\boldsymbol{b} - \boldsymbol{A}x_s) &= x_s + C\boldsymbol{b} - C(\boldsymbol{A}x_s), \\ &= C\boldsymbol{b} - (C\boldsymbol{A})x_s + x_s, \\ &= C\boldsymbol{b} + (I - C\boldsymbol{A})x_s, \end{aligned}$$

by the sub-distributive law (4), we have

$$\begin{aligned} x_s + C(\boldsymbol{b} - \boldsymbol{A}x_s) + (I - C\boldsymbol{A})\boldsymbol{d^{(i)}} &= C\boldsymbol{b} + (I - C\boldsymbol{A})x_s + (I - C\boldsymbol{A})\boldsymbol{d^{(i)}}, \\ &\supseteq C\boldsymbol{b} + (I - C\boldsymbol{A})(x_s + \boldsymbol{d^{(i)}}), \\ &= C\boldsymbol{b} + (I - C\boldsymbol{A})\boldsymbol{y^{(i)}}, \end{aligned}$$

where $\boldsymbol{y^{(i)}} = x_s + \boldsymbol{d^{(i)}}$. This implies that

$$\boldsymbol{y^{(i+1)}} = x_s + \boldsymbol{d^{(i+1)}} \supseteq (C\boldsymbol{b} + (I - C\boldsymbol{A})\boldsymbol{y^{(i)}}) \cap \boldsymbol{y^{(i)}},$$

and by a comparison with (10) suggests that if the initial enclosures are the same then the Krawczyk iteration gives better results than the residual version. In practice, if $\boldsymbol{A}$ and $\boldsymbol{b}$ are thin, the residual $\boldsymbol{b} - \boldsymbol{A}x_s$ can be enclosed with fewer rounding errors than $\boldsymbol{b}$, which leads to a tighter enclosure of the solution. This is shown is Section 5.6 where the functions are tested.

For most matrices $\boldsymbol{A}$ an enclosure will be found; however, if after 7 iterations none is found, the function reverts to the Hansen-Bliek-Rohn-Ning-Kearfott-Neumaier method described in the previous section.

5.6 Comparing Three Functions – Timings and Accuracy

The functions `verifylss`, `kraw` and `hsolve` are tested on a variety of problems, to compare timings and accuracy. All timings were performed using a 1400MHz AMD Athlon machine running Linux.

We first consider the problem of finding verified bounds on a dense system with a point matrix and point right hand side vector. The matrix A and vector b have random components in the range $[-1, 1]$ for various dimensions. The three verification functions and the "\" command, which does not produce a verified result, are timed with the results displayed in the Table 1.

n	\	`verifylss.m`	`kraw.m`	`hsolve.m`
500	0.1658	1.0452	1.2212	1.3256
1000	1.0826	7.4799	8.2485	9.8314
1500	3.0739	23.7469	25.4587	34.9030

Table 1: Time taken in seconds to give verified bounds of solution to point system.

The aim of the verification algorithms is to produce a tight bound on the true solution. Therefore, to measure the accuracy, the sum of the radii of the solution components is calculated and displayed in Table 2.

n	`verifylss.m`	`kraw.m`	`hsolve.m`
500	2.8833e-9	2.9634e-9	3.5746e-8
1000	4.8767e-7	5.0295e-7	1.0325e-5
1500	7.9302e-7	7.9751e-7	2.4509e-5

Table 2: Sum of radii of solution to point system.

Tables 1 and 2 show that the INTLAB function `verifylss` produces the tightest bounds and does so in the least time. On all occasions, the first stage of `verifylss` produces a suitable enclosure. The function requires $2n^3$ flops for calculating the inverse of a matrix, $4n^3$ flops for interval matrix multiplication and $O(n^2)$ flops for all the other calculations. This gives a dominant term of $6n^3$ flops, which in theory should produce a factor of 9 between the time taken for the "\" and `verifylss` functions. The factor is smaller in practice since matrix inversion and matrix multiplication have a larger proportion of "level 3" flops.

We now consider thick systems by using $\boldsymbol{A}$ and $\boldsymbol{b}$ with random midpoints in the range $[-1, 1]$ with radius $1e$-10. The timings for various dimensions are displayed in Table 3, which show that again `verifylss` is the quickest function.

n	`verifylss.m`	`kraw.m`	`hsolve.m`
500	1.6063	1.7432	2.1593
1000	10.5656	11.1824	14.9916
1500	32.6368	34.0961	47.4057

Table 3: Time taken in seconds to solve thick system.

The accuracy of the solutions are measured as before and are displayed in Table 4. The results shows that `kraw` produces the tightest bounds while being only marginally slower than `verifylss`.

n	`verifylss.m`	`kraw.m`	`hsolve.m`
500	0.20377221255906	0.20377128653829	0.20377959207465
1000	0.01096443183751	0.01096442299211	0.01096552977129
1500	0.01086009356507	0.01086008825560	0.01086241425924

Table 4: Sum of radii of solution for system with radius 1e-10.

Table 5 shows the accuracy results when the radii are increased to $1e$-8. The function `hsolve` now produces the tightest bounds with `kraw` outperforming the INTLAB function. The times are similar to those in Table 3, so although `hsolve` gives the most accurate solution it is significantly slower than the other two functions.

n	`verifylss.m`	`kraw.m`	`hsolve.m`
500	0.60583956501556	0.60581192580785	0.60581088581245
1000	0.24248418623455	0.24247349810479	0.24247325507546
1500	0.90346986564659	0.90338534443322	0.90338334791627

Table 5: Sum of radii of solution for system with radius 1e-8.

For all linear systems tested the INTLAB function was quickest, and when used to produce a verified bound of a point system it gave the sharpest bounds. As the radius of the components were increased, the function `kraw` gave tighter bound than `verifylss`, and when increased further `hsolve` gave better bounds still. Since the sum of radii are very similar for the three functions then the quickest function, `verifylss`, should be used. If very high accuracy is required then the other two functions should be considered.

6 Univariate Nonlinear Equations

To obtain rigorous bounds on a root of a function of one variable, f, an interval version of Newton's method for finding roots of equations may be used provided f is twice differentiable in the initial interval $\boldsymbol{x^{(0)}} \in \mathbb{IR}$. Enclosures of a real simple root may be obtained, where a simple root x^* satisfies $f(x^*) = 0$ and $f'(x^*) \neq 0$. Furthermore, by using a recursive procedure, bounds on all real simple roots of a function can be found.

The mean value theorem states that

$$0 = f(x) = f(\tilde{x}) + f'(\xi)(x - \tilde{x}),$$

for some ξ between x and $\tilde{x}$. If x and $\tilde{x}$ lie within the interval $\boldsymbol{x^{(0)}} \in \mathbb{IR}$ then so does ξ and hence

$$x = \tilde{x} - \frac{f(\tilde{x})}{f'(\xi)} \quad \Rightarrow \quad x \in N(\boldsymbol{x^{(0)}}) = \tilde{x} - \frac{f(\tilde{x})}{f'(\boldsymbol{x^{(0)}})}.$$

Therefore the intersection $\boldsymbol{x^{(0)}} \cap N(\boldsymbol{x^{(0)}})$ must contain the root, if $\boldsymbol{x^{(0)}}$ contained a root of f. Since $\tilde{x} \in \boldsymbol{x^{(0)}}$, a suitable choice for $\tilde{x}$ is to take the midpoint $\check{\boldsymbol{x}}^{\boldsymbol{(0)}}$. To take account of rounding errors, $\check{\boldsymbol{x}}^{\boldsymbol{(0)}}$ must be a thin interval containing an approximation to the midpoint of $\boldsymbol{x^{(0)}}$, and therefore we write it in boldface to show this.

Given an initial interval $\boldsymbol{x^{(0)}}$ that contains a root, this leads to the interval version of Newton's method

$$\boldsymbol{x^{(i+1)}} = \boldsymbol{x^{(i)}} \cap \left(\check{\boldsymbol{x}}^{\boldsymbol{(i)}} - \frac{f(\check{\boldsymbol{x}}^{\boldsymbol{(i)}})}{f'(\boldsymbol{x^{(i)}})} \right), \quad i = 1, 2, 3, \ldots \tag{12}$$

The stopping criterion is taken to be $\boldsymbol{x^{(i+1)}} = \boldsymbol{x^{(i)}}$.

If f' has a zero in the interval $\boldsymbol{x}$ then clearly $N(\boldsymbol{x})$ is not defined. Therefore if f has a multiple root, or two distinct roots in $\boldsymbol{x}$ then the iteration (12) can not be used. A necessary condition for the iteration to work is that $\boldsymbol{x^{(0)}}$ must contain at most one root, and the root must be a simple one.

Moore [12] shows the following properties in the simple root case. If $N(\boldsymbol{x})$ is defined then either $N(\boldsymbol{x}) \cap \boldsymbol{x}$ is empty, in which case $\boldsymbol{x}$ does not contain a root, or $N(\boldsymbol{x}) \cap \boldsymbol{x}$ is an interval which contains a root if $\boldsymbol{x}$ does. Furthermore if $\boldsymbol{x}$ does not contain a root, then after a finite number of steps the iteration (12) will stop with an empty $\boldsymbol{x^{(i)}}$. The convergence is also shown to be quadratic since

$$\text{rad}(\boldsymbol{x^{(k+1)}}) \leq \alpha \text{rad}(\boldsymbol{x^{(k)}})^2,$$

where α is a constant.

The iteration (12) can be used to find a simple root of a function if at most one simple root is contained in the initial interval $\boldsymbol{x^{(0)}}$. By modifying this iteration and using a recursive procedure with extended interval arithmetic, bounds can be obtained for all the simple roots within $\boldsymbol{x^{(0)}}$. If $f'(\boldsymbol{x}) = [a, b]$ contains zero then using extended interval arithmetic from (3), this interval can be split so that $f'(\boldsymbol{x}) = [a, 0] \cup [0, b]$ and hence

$$N(\boldsymbol{x}) = \check{\boldsymbol{x}} - \frac{f(\check{\boldsymbol{x}})}{[a, 0] \cup [0, b]}.$$

This produces two intervals which can then be considered separately.

As an example consider $f(x) = x^2 - 2$, with $\boldsymbol{x^{(0)}} = [-2, 3]$. This gives $\check{\boldsymbol{x}}^{\boldsymbol{(0)}} = 0.5$, $f(\check{\boldsymbol{x}}^{\boldsymbol{(0)}}) = -1.75$ and $f'(\boldsymbol{x^{(0)}}) = [-4, 0] \cup [0, 6]$. $N(\boldsymbol{x^{(0)}})$ is calculated as

$$\begin{aligned} N(\boldsymbol{x^{(0)}}) &= 0.5 - \frac{(-1.75)}{[-4, 0] \cup [0, 6]}, \\ &= [-\infty, 0.0625] \cup [0.7917, \infty], \end{aligned}$$

which gives

$$\boldsymbol{x}^{(1)} = [-2, 0.0625] \cup [0.7917, 3].$$

One of the intervals, say $[0.7917, 3]$, is then considered to give the sequence of intervals

$$\begin{aligned}
\boldsymbol{x}^{(2)} &= [0.8889803, 1.6301360],\\
\boldsymbol{x}^{(3)} &= [1.3863921, 1.4921355],\\
\boldsymbol{x}^{(4)} &= [1.4134846, 1.4153115],\\
\boldsymbol{x}^{(5)} &= [1.4142135, 1.4142137],\\
&\vdots
\end{aligned}$$

After a tight bound containing $\sqrt{2}$ has been obtained, a similar procedure is performed on the interval $[-2, 0.0625]$. The procedure is described for a general function in the following algorithm.

Algorithm 6.1.

```
function  intnewton($f, \boldsymbol{x}, tol$)
if $0 \in f'(\boldsymbol{x})$
    $[a, b] = f'(\boldsymbol{x})$
    $N = \check{\boldsymbol{x}} - f(\check{\boldsymbol{x}})/f'([a, 0])$
    $\boldsymbol{y} = \boldsymbol{x} \cap N$
    if $\boldsymbol{y} \neq \emptyset$
        intnewton($f, \boldsymbol{y}, tol$)
    end
    $[a, b] = f'(\boldsymbol{x})$
    $N = \check{\boldsymbol{x}} - f(\check{\boldsymbol{x}})/f'([0, b])$
    $\boldsymbol{y} = \boldsymbol{x} \cap N$
    if $\boldsymbol{y} \neq \emptyset$
        intnewton($f, \boldsymbol{y}, tol$)
    end
else
    $N = \check{\boldsymbol{x}} - f(\check{\boldsymbol{x}})/f'(\boldsymbol{x})$
    $\boldsymbol{y} = \boldsymbol{x} \cap N$
    if $\boldsymbol{y} = \boldsymbol{x}$ or $\mathrm{rad}(\boldsymbol{y}) < tol$
        display $\boldsymbol{y}$
    elseif $\boldsymbol{y} \neq \emptyset$
        intnewton($f, \boldsymbol{y}, tol$)
    end
end
```

Algorithm 6.1 is implemented as the function `intnewton.m` in Appendix A.2. The derivative of the function is obtained by automatic differentiation described in Section 3.4. As an example of using the function, it can be applied to the polynomial $f(x) = x^5 - 15x^4 + 85x^3 - 225x^2 + 274x - 120$ which has roots at 1, 2, 3, 4 and 5. Using an initial interval containing all the roots, $\boldsymbol{x}^{(0)} = [0, 7]$, produces the following output.

```
>> intnewton(infsup(0,7))
Root in the interval
```

```
intval Xnew =
[0.99999999999999,   1.00000000000001]
Root in the interval
intval Xnew =
[1.99999999999994,   2.00000000000005]
Root in the interval
intval Xnew =
[2.99999999999960,   3.00000000000044]
Root in the interval
intval Xnew =
[3.99999999999955,   4.00000000000036]
Root in the interval
intval Xnew =
[4.99999999999973,   5.00000000000034]
```

Tight bounds are obtained for all five roots.

INTLAB does not have a univariate solver, however the multivariate solver, described in Section 7.3, can be used for this purpose. If a univariate equation and a close enough approximation of a root are entered then tight bounds on the root can be obtained.

7 Multivariate Nonlinear Equations

In this section we consider the problem of solving nonlinear systems of equations. Three Newton operators are given, which are the most commonly used for such problems. The INTLAB function for nonlinear equations is explained before the basic ideas for finding all solutions, in a given range, are described.

7.1 Multivariate Newton's Method

The problem is to find bounds on the solution of a nonlinear continuous function $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ in a given box $\boldsymbol{x}^{(\mathbf{0})} \in \mathbb{IR}^n$. Due to interval arithmetic's power to bound ranges of functions, the solutions of nonlinear systems of equations and global optimisation, by similar methods, are the most common uses of interval analysis.

There are many applications where we would like a nonlinear equation solving technique that can find, with mathematical certainty, all the roots in a given region. Using a noninterval method it is difficult to find all solutions and often impossible to check whether all solutions have been found. Interval methods can be used to count the number of roots in a given box and provide tight bounds for each individual root. An example of an application requiring all solutions is robust process simulation, for which Schnepper and Stadtherr [19] use interval techniques.

Using the mean value theorem we have for any x^* that

$$f(x^*) \in f(\tilde{x}) + J(\boldsymbol{x})(x^* - \tilde{x}),$$

where $J(\boldsymbol{x})$ is the interval Jacobian matrix with

$$J_{ij} = \frac{\partial^2 f}{\partial \boldsymbol{x}_i \partial \boldsymbol{x}_j} \quad i, j = 1, \ldots, n,$$

which can be formed using automatic differentiation, and $\tilde{x} \in \boldsymbol{x}$. If x^* is a zero of f then $f(x^*) = 0$ and therefore

$$-f(\tilde{x}) \in J(\boldsymbol{x})(x^* - \tilde{x}). \tag{13}$$

The interval linear system (13) can then be solved for x^* to obtain an outer bound on the solution set, say $N(\tilde{x}, \boldsymbol{x})$. The notation includes both $\tilde{x}$ and $\boldsymbol{x}$ to show the dependence on both terms. This gives

$$0 \in f(\tilde{x}) + J(\boldsymbol{x})(N(\tilde{x}, \boldsymbol{x}) - \tilde{x}),$$

which suggests the iteration

$$\text{solve for } N \qquad f(\tilde{x}^{(k)}) + J(\boldsymbol{x^{(k)}})(N(\tilde{x}^{(k)}, \boldsymbol{x^{(k)}}) - \tilde{x}^{(k)}) = 0, \tag{14}$$

$$\boldsymbol{x^{(k+1)}} = \boldsymbol{x^{(k)}} \cap N(\tilde{x}^{(k)}, \boldsymbol{x^{(k)}}), \tag{15}$$

for $k = 0, 1, \ldots$ and $x^{(k)} \in \boldsymbol{x^{(k)}}$. A reasonable choice for $\tilde{x}^{(k)}$ is the centre, $\check{\boldsymbol{x}}^{\boldsymbol{(k)}}$, of $\boldsymbol{x}$, however better choices are available. One example of determining $\tilde{x}^{(k)}$ is to use a noninterval version of Newton's method to find an approximation of a solution in $\boldsymbol{x}$. The various interval Newton methods are determined by how $N(\tilde{x}^{(k)}, \boldsymbol{x^{(k)}})$ is defined. Essentially variations on the three operators described below are commonly used.

7.2 Newton Operators

The linear system (14) can be solved using any of the methods described in Section 5 to give the *Newton operator*

$$N(\tilde{x}, \boldsymbol{x}) = \tilde{x} - J(\boldsymbol{x})^{-1} f(\tilde{x}).$$

In practice this operator is not often used since an interval linear system has to be solved for each iteration. Also $J(\boldsymbol{x})$ is likely to contain singular matrices unless the width of $\boldsymbol{x}$ is small.

An alternative is to follow Krawczyk [11] and use what is now called the *Krawczyk operator.* The advantage of this operator is that no interval linear equations have to be solved at any stage, thus increasing speed and reliability. Substituting the terms of the linear system (14) into the Krawczyk iteration given in Section 5.3 leads to the Krawczyk operator for nonlinear systems $K : \mathbb{IR}^n \times \mathbb{R}^n \to \mathbb{IR}^n$, and is defined by

$$K(\tilde{x}, \boldsymbol{x}) = \tilde{x} - Cf(\tilde{x}) - (I - CJ(\boldsymbol{x}))(\tilde{x} - \boldsymbol{x}),$$

where C is the preconditioning matrix, the midpoint inverse of $J(\boldsymbol{x})$, and $\tilde{x} \in \boldsymbol{x}$. Although an inverse is still required, it is the inverse of a real matrix rather than an interval matrix.

Replacing $N(\tilde{x}, \boldsymbol{x})$ by $K(\tilde{x}, \boldsymbol{x})$ in (15) leads to the Krawczyk iteration

$$\boldsymbol{x^{(k+1)}} = K(\tilde{x}^{(k)}, \boldsymbol{x^{(k)}}) \cap \boldsymbol{x^{(k)}},$$

for $k = 0, 1, \ldots$ and $\tilde{x}^{(k)} \in \boldsymbol{x^{(k)}}$.

A third operator developed by Hansen and Sengupta [8], the *Hansen–Sengupta operator*, uses a Gauss–Seidel procedure. Preconditioning (13) with C the midpoint inverse of $J(\boldsymbol{x})$ gives

$$CJ(\boldsymbol{x})(N(\tilde{x}, \boldsymbol{x}) - \tilde{x}) = -Cf(\tilde{x}).$$

Changing the notation $N(\tilde{x}, \boldsymbol{x})$ to $H(\tilde{x}, \boldsymbol{x})$ and defining

$$M = CJ(\boldsymbol{x}), \quad b = Cf(\tilde{x}),$$

the interval Gauss–Seidel procedure proceeds component by component to give the iteration

$$H(\tilde{x}^{(k)}, \boldsymbol{x^{(k)}})_i = \tilde{x}_i^{(k)} - \frac{b_i + \sum_{j=1}^{i-1} M_{ij}(\boldsymbol{x_j^{(k+1)}} - \tilde{x}_j^{(k+1)}) + \sum_{j=i+1}^{n} M_{ij}(\boldsymbol{x_j^{(k)}} - \tilde{x}_j^{(k)})}{M_{ii}}, \tag{16}$$

$$\boldsymbol{x_i^{(k+1)}} = H(\tilde{x}^{(k)}, \boldsymbol{x^{(k)}})_i \cap \boldsymbol{x_i^{(k)}}, \tag{17}$$

for $k = 0, 1, \ldots$ and $\tilde{x}^{(k)} \in \boldsymbol{x^{(k)}}$.

In this iteration, after the ith component of $H(\tilde{x}^{(k)}, \boldsymbol{x^{(k)}})$ is computed using (16), the intersection (17) is performed. The result is then used to calculate subsequent components of $H(\tilde{x}^{(k)}, \boldsymbol{x^{(k)}})$. Neumaier [13] shows that this operator yields a tighter enclosure than the Krawczyk operator.

A benefit of using interval analysis to solve nonlinear equations is that the Krawczyk and Hansen–Sengupta operators can be used to test the existence and uniqueness of a zero in an interval $\boldsymbol{x}$.

Theorem 7.1. *Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be a nonlinear continuous function, $\tilde{x} \in \boldsymbol{x}$ and $\boldsymbol{x'}$ denote either $K(\tilde{x}, \boldsymbol{x})$ or $H(\tilde{x}, \boldsymbol{x})$.*

(i) If f has a root $x^* \in \boldsymbol{x}$ then $x^* \in \boldsymbol{x'} \cap \boldsymbol{x}$.

(ii) If $\boldsymbol{x'} \cap \boldsymbol{x} = \emptyset$ then f contains no zero in $\boldsymbol{x}$.

(iii) If $\emptyset \neq \boldsymbol{x'} \subseteq$ interior of $\boldsymbol{x}$ then f contains a unique zero in $\boldsymbol{x}$.

A proof is given by Neumaier [13].

Using the above ideas, two functions have been created, given in Appendix A.3, to find bounds on roots of nonlinear equations. The functions `nlinkraw.m` and `nlinhs.m` are based on the Krawczyk and Hansen–Sengupta operators respectively, while a function based on the Newton operator is not given due to the problems discussed earlier. The stopping criterion is taken to be $\boldsymbol{x^{(k+1)}} = \boldsymbol{x^{(k)}}$. The following simple numerical examples highlight some of the problems of using the operators alone.

Consider the nonlinear equations

$$2x_1 - x_2 - 2 = 0, \tag{18}$$

$$\frac{7}{2}x_2 - x_1^2 - 4x_1 + 5 = 0,$$

which have real solutions $x = (1, 0)^T$ and $x = (2, 2)^T$. Using the function `nlinkraw.m` with the initial interval $\boldsymbol{x} = ([0.8, 1.2], [-0.2, 0.2])^T$ produces the following output.

```
Number of iterations
    8
intval ans =
[0.99999999999999,   1.00000000000001]
[-0.00000000000001,   0.00000000000001]
```

In this case, appropriate bounds are given around one of the roots in 8 iterations. Using the same starting interval with nlinhs.m, the same bounds are given in 3 iterations.

```
Number of iterations
     3
intval ans =
[0.99999999999999,   1.00000000000001]
[-0.00000000000001,   0.00000000000001]
```

If the width of the initial interval is increased slightly to $\boldsymbol{x} = ([0.5, 1.5], [-0.5, 0.5])^T$ the function nlinkraw.m fails to find tighter the bounds on a root, and the initial interval is output. The function nlinhs.m produces a tight enclosure on the root $y = (1, 0)^T$, as before.

Increasing the width further so that the initial interval, $\boldsymbol{x} = ([0.5, 3], [-0.5, 3])^T$, contains both real roots, produces the following output for the function nlinkraw.m.

```
Number of iterations
     1
intval ans =
[0.50000000000000,   3.00000000000000]
[-0.50000000000001,   3.00000000000000]
```

Although the output is correct in that both roots are contained within the interval, a tight enclosure of a root is not obtained. Applying the same initial interval with the function nlinhs.m produces an error due to division by zero.

These numerical examples show the importance of providing an interval of small radius containing a root. If such an interval is given then the functions can be used to give tight verified bounds on a root. However, if this is not the case then the width of the interval is likely not to be decreased. The INTLAB function for solving nonlinear equations avoids the problem of providing an initial interval with small radius, by requiring the user to supply a noninterval approximation to a zero.

7.3 INTLAB function

The function verifynlss.m is the INTLAB function for finding verified bounds on the roots of nonlinear equations. The method is based on the Krawczyk operator with modifications for enclosing the error with respect to an approximate solution, similar to the function for solving linear equations described in Section 5.5.

An approximation of the root is provided by the user before a floating point Newton iteration is carried out to give a closer approximation to a root. With the new approximation, an interval iteration is carried out using a residual version of the Krawczyk operator. The implication

$$K(\tilde{x}, \boldsymbol{x}) \subseteq \text{interior of } \boldsymbol{x} \implies x^* \in \boldsymbol{x},$$

where x^* is a root of f, is used as the criterion for terminating the iteration.

The approximation does not have to be very close to a root: for example, bounds on the root $(1, 0)^T$ of the nonlinear equation 18 can be obtained with the approximation $x = (-10, -10)$.

```
>> verifynlss(@f,[-10,-10])
intval ans =
[   0.99999999999999,   1.00000000000001]
[  -0.00000000000001,   0.00000000000001]
```

If the approximation is closer to the root $(2,2)^T$ then bounds on this root are obtained. For example, using the approximation $(1.51,1)^T$ is only marginally closer to $(2,2)^T$ but bounds are given for this root.

```
>> verifynlss(@f,[1.51,1])
intval ans =
[   1.99999999999999,   2.00000000000001]
[   1.99999999999999,   2.00000000000001]
```

7.4 All Solutions of a Nonlinear System

A drawback of the multivariate methods discussed thus far is that only one root can be found at a time, making it difficult to find all roots in a given range. By further consideration, interval methods may be used to overcome this problem, bounding all roots in a given interval. In order to achieve this, an algorithm using generalised bisection in conjunction with Newton methods may be developed. Theorem 7.1 enables all solutions to be computed with mathematical certainty.

The basic idea is to iterate using a Newton operator on a box until at some stage the width of $\boldsymbol{x}^{(\boldsymbol{k}+1)}$ is not sufficiently smaller than $\boldsymbol{x}^{(\boldsymbol{k})}$. This may be because several roots are contained in the box, or the box is too wide to converge to a single root. At this stage the box is bisected to form two new boxes. The iteration is then carried out on one of the boxes while the other is stacked for later consideration.

If a step of the iteration is applied to a box $\boldsymbol{x}^{(\boldsymbol{k})}$ and the box $\boldsymbol{x}^{(\boldsymbol{k}+1)}$ is returned, then a criterion is required to decide whether the size of the box is sufficiently reduced. One possible criterion is to say the box $\boldsymbol{x}^{(\boldsymbol{k}+1)}$ is sufficiently reduced if for some $i = 1, \dots, n$

$$\operatorname{rad}(\boldsymbol{x}_i{}^{(\boldsymbol{k})}) - \operatorname{rad}(\boldsymbol{x}_i{}^{(\boldsymbol{k}+1)}) > \alpha \max_i(\operatorname{rad}(\boldsymbol{x}_i{}^{(\boldsymbol{k})})),$$

where α is a constant such that $0 < \alpha < 1$. If a box is sufficiently reduced then the iteration is continued on the box, and if not, then it is split into subboxes.

In order to split a box, $\boldsymbol{x}$, a dimension must be chosen to be split. The obvious choice of splitting the largest component of $\boldsymbol{x}$ may result in the problem being badly scaled. A good choice is to try and choose the component in which f varies most over $\boldsymbol{x}$. The quantity

$$T_j = \operatorname{rad}(\boldsymbol{x}_j) \sum_{i=1}^{n} |J_{ij}(\boldsymbol{x})|,$$

can be computed, from which $\boldsymbol{x}$ can be split in the rth component if

$$T_r \geq T_j, \quad j = 1, \dots, n.$$

These basic ideas together with the Krawczyk operator are implemented as the function `allroots.m` in Appendix A.3. This function bounds all the roots of a nonlinear system and is intended only for low dimensions, since it is not efficient for a large number of equations. The function solves the problem of finding both solutions of (18). By using the initial box

$$\boldsymbol{x}^{(0)} = \begin{pmatrix} [-10,10] \\ [-10,10] \end{pmatrix},$$

bounds are given on both solutions, with the function being called recursively on 36 occasions.

```
>> allroots(@f,infsup([-10,-10],[10,10]))
intval Xnew =
[0.99999999999999,   1.00000000000001]
[-0.00000000000001,   0.00000000000001]
intval Xnew =
[1.99999999999999,   2.00000000000001]
[1.99999999999999,   2.00000000000001]
```

Applying the function to the nonlinear system

$$\begin{aligned} y_1 &= 3x_1(x_2 - 2x_1) + \frac{x_2^2}{4}, \\ y_2 &= 3x_2(x_3 - 2x_2 + x_1) + (x_3 - x_1)^2, \\ y_3 &= 3x_3(20 - 2x_3 + x_2) + (20 - x_2)^2, \end{aligned}$$

with the initial box

$$\boldsymbol{x}^{(0)} = \begin{pmatrix} [-10,5] \\ [-10,5] \\ [-10,5] \end{pmatrix},$$

produces bounds on the solutions at a much slower rate. Four roots are found in the given box with the function being called 1312 times.

```
>> attempt3(@f,infsup([-10,-10,-10],[5,5,5]))
intval Xnew =
[-3.49832993885909,  -3.49832993885907]
[-6.10796755232559,  -6.10796755232557]
[-7.73708170177157,  -7.73708170177155]
intval Xnew =
[0.38582828617158,   0.38582828617159]
[-5.30358277070613,  -5.30358277070610]
[-7.28997002509309,  -7.28997002509307]
intval Xnew =
[-0.06683877075395,  -0.06683877075394]
[0.91876351654562,   0.91876351654564]
[-4.15284241315239,  -4.15284241315238]
intval Xnew =
[0.73257037812118,   0.73257037812119]
[1.27904348004932,   1.27904348004933]
[-3.99217902192722,  -3.99217902192720]
```

Further ideas for speeding up the process of finding all solutions, including a full algorithm, are given in [7]. The bisection part of the algorithm makes the process suitable to use on a parallel machine, for which an algorithm and application is given in [19]. This makes it feasible for larger problems to be solved.

8 Viswanath's Constant Using Interval Analysis

In this section an application of interval analysis is shown. The rate at which a random Fibonacci sequence increases is calculated, with rounding errors being taken into account by the use of interval arithmetic.

8.1 Random Fibonacci Sequences

The Fibonacci sequence is defined by $f_1 = f_2 = 1$ and $f_n = f_{n-1} + f_{n-2}$ for $n > 2$, and it is well known that f_n increases exponentially as $n \to \infty$ at the rate $(1+\sqrt{5})/2$. A random Fibonacci sequence is defined by $t_1 = t_2 = 1$ and $t_n = \pm t_{n-1} \pm t_{n-2}$ for $n > 2$, where each ± sign is chosen independently with equal probability. Here we consider the sequence $t_n = \pm t_{n-1} + t_{n-2}$, which we call the *random Fibonacci recurrence.* Viswanath [20] showed that the terms of this recurrence satisfy

$$\lim_{n\to\infty} |x_n|^{1/n} = 1.13198824\ldots \quad ,$$

a number often called *Viswanath's Constant.*

To find this constant, Viswanath used floating point arithmetic and then conducted an extensive analysis of rounding errors to show that the result was correct. Here we show how interval analysis can be used to find Viswanath's constant without the need for rounding error analysis. A similar approach was recently carried out by Oliveira and Figueiredo [10] who use `RVInterval`, a library of interval data types and operations for the programming language C. The method of reducing the amount of memory required was based on this work.

The random Fibonacci recurrence can be written in matrix form as

$$\begin{pmatrix} t_{n-1} \\ t_n \end{pmatrix} = M \begin{pmatrix} t_{n-2} \\ t_{n-1} \end{pmatrix},$$

with M chosen as one of the two matrices

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}$$

with equal probability $\frac{1}{2}$ at each step. If the random matrix M_i, chosen at the ith step, is A or B with probability $\frac{1}{2}$ then the recurrence can be written as

$$\begin{pmatrix} t_{n-1} \\ t_n \end{pmatrix} = M_{n-2} \cdots M_1 \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Known results from the theory of random matrix products imply that

$$\begin{array}{l} \log ||M_n \cdots M_1|| \to \gamma_f \quad \text{as } n \to \infty, \\ \sqrt[n]{|t_n|} \to e^{\gamma_f} \quad \text{as } n \to \infty, \end{array}$$

for a constant γ_f with probability 1. The aim is to determine e^{γ_f} as accurately as possible.

8.2 Viswanath's Calculations

To find γ_f, Viswanath uses Furstenberg's formula

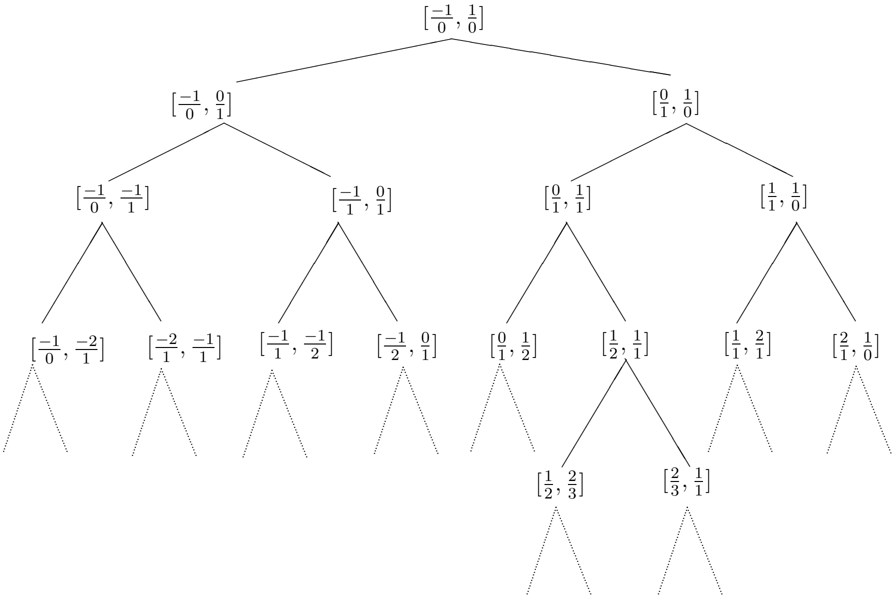
$$\gamma_f = 2\int_0^\infty \operatorname{amp}(m)d\nu(m), \tag{19}$$

where

$$\operatorname{amp}(m) = \frac{1}{4}\log\left(\frac{1+4m^4}{(1+m^2)^2}\right)$$

and ν is an invariant probability measure on an interval $[a,b]$ with $\pm 1 \notin [a,b]$, given by

$$\nu([a,b]) = \frac{1}{2}\nu\left(\left[\frac{1}{-1+b}, \frac{1}{-1+a}\right]\right) + \left(\left[\frac{1}{1+b}, \frac{1}{1+a}\right]\right). \tag{20}$$



$[\frac{-1}{0}, \frac{1}{0}]$
$[\frac{-1}{0}, \frac{0}{1}]$
$[\frac{0}{1}, \frac{1}{0}]$
$[\frac{-1}{0}, \frac{-1}{1}]$
$[\frac{-1}{1}, \frac{0}{1}]$
$[\frac{0}{1}, \frac{1}{1}]$
$[\frac{1}{1}, \frac{1}{0}]$
$[\frac{-1}{0}, \frac{-2}{1}]$
$[\frac{-2}{1}, \frac{-1}{1}]$
$[\frac{-1}{1}, \frac{-1}{2}]$
$[\frac{-1}{2}, \frac{0}{1}]$
$[\frac{0}{1}, \frac{1}{2}]$
$[\frac{1}{2}, \frac{1}{1}]$
$[\frac{1}{1}, \frac{2}{1}]$
$[\frac{2}{1}, \frac{1}{0}]$
$[\frac{1}{2}, \frac{2}{3}]$
$[\frac{2}{3}, \frac{1}{1}]$

Figure 3: The Stern-Brocot Tree.

To find the invariant measure ν, Viswanath uses the Stern-Brocot tree. This is an infinite binary tree that divides the real line $\mathbb{R}$ recursively. In the Stern-Brocot tree, ∞ is represented by $\frac{1}{0}$ and 0 as $\frac{0}{1}$. The root of the tree is the real line $[\frac{-1}{0}, \frac{1}{0}]$ and its left and right children are $[\frac{-1}{0}, \frac{0}{1}]$ and $[\frac{0}{1}, \frac{1}{0}]$ respectively. For a node $[\frac{a}{b}, \frac{c}{d}]$, other than the root, the left child is defined as $[\frac{a}{b}, \frac{a+c}{b+d}]$ and the right child is defined as $[\frac{a+c}{b+d}, \frac{c}{d}]$. Part of the tree is shown in Figure 3.

Viswanath proves that every interval in the Stern-Brocot tree satisfies the invariant measure (20) and can therefore be used to find an approximation of the integral (19). If I_j, $1 \leq j \leq 2^d$, are the set of positive Stern-Brocot intervals at depth $d+1$ then

$$\gamma_f = 2\int_0^\infty \operatorname{amp}(m)d\nu(m) = 2\sum_{j=1}^{2^d}\int_{I_j} \operatorname{amp}(m)d\nu(m),$$

and hence, by forming an upper and lower approximation to this integral, $\gamma_f \in [p_d, q_d]$ where

$$p_d = 2\sum_{j=1}^{2^d} \min_{m\in I_j} \operatorname{amp}(m)\nu(I_j), \quad q_d = 2\sum_{j=1}^{2^d} \max_{m\in I_j} \operatorname{amp}(m)\nu(I_j).$$

These bounds were calculated by Viswanath with $d = 28$ to obtain $p_{28} = 0.1239755981508$ and $q_{28} = 0.1239755994406$.

For a more detailed description including the original program see [20].

8.3 Viswanath's Constant Using INTLAB

The program `viswanath`, written using INTLAB, calculates Viswanath's constant using interval analysis to take account of rounding errors. Here we describe the main changes to the original program.

In the original program the values of $\nu(I_j)/2$ are stored as the array `table`. These are calculated as

$$table_i = \frac{g^{d-i}(1+g)^{-d}}{4} \quad \text{for } 0 \le i \le d,$$

where

$$g = \frac{1+\sqrt{5}}{2}.$$

To take account of rounding errors, an array `itable` is created which bounds the components of `table`. This is done by the following lines of code:

```
g = (intval(1)+sqrt(intval(5)))/intval(2);
for i = 0:d
   itable(i+1) = (g^(d-i)*(1+g)^(-d))/4;
end
```

The index i into `itable`, for finding $\nu(I_j)/2$ is calculated in the same way as Viswanath. It is obtained by taking the binary representation of j, flipping all the odd bits if d is even and all the even bits if d is odd, with the least significant bit taken as an even bit, and then counting the number of 1's. This procedure is performed by the function `flip.m` given in Appendix A.4.

Viswanath calculates an array `num` containing the Stern-Brocot coefficients, but the array is very large. It is necessary to reduce the amount of memory required to store this array. This is done by calculating the first 2^{24} Stern-Brocot coefficients and then using a recursive function `stern` to find the other coefficients.

Where there is an occurrence of the C variable type `double`, an interval variable type is used. This results in the fractions in the Stern-Brocot tree being bounded above and below. This is carried out by the line

```
m = intval(stern([k+2:k+nrt+1]))./ ...
      intval(stern(n-k:-1:n-(k+nrt-1)));
```

To calculate $\operatorname{amp}(m)$ the function `amp.m` is used. Since the variable `m` input into the function is an interval, an enclosure of $\operatorname{amp}(m)$ is output, thus taking account of rounding errors.

8.4 How to Use `viswanath.m`

The M-file `viswanath.m`, used for calculating Viswanath's constant, is given in Appendix A.4, along with additional functions called from within the program. The value of `d`, the depth `d` + 1 of the Stern-Brocot tree used at the beginning of the code, can be changed. Also the appropriate line in `flip.m` has to be uncommented depending on the value of `d` chosen.

To run the M-file type `viswanath` at the command prompt.

8.5 Results

The M-file `viswanath.m` was ran on a machine with 511Mb of memory running Linux for `d` between 10 and 28. Increasing `d` divides the real line into smaller intervals and hence produces tighter bounds on γ_f. This also increases the time taken by the program. Table 6 shows the enclosures computed for γ_f and Table 7 shows the corresponding enclosures for e^{γ_f}.

d	Left bound	Right bound	Diameter
10	0.12373812224857	0.12421208679539	4.74×10^{-4}
12	0.12391824373568	0.12403259184306	1.14×10^{-4}
14	0.12396175692065	0.12398932455491	2.76×10^{-5}
16	0.12397226014473	0.12397890248644	6.64×10^{-6}
18	0.12397479388451	0.12397639358819	1.60×10^{-6}
20	0.12397540481804	0.12397578993100	3.85×10^{-7}
22	0.12397555206790	0.12397564474935	9.27×10^{-8}
24	0.12397558754683	0.12397560984528	2.23×10^{-8}
26	0.12397559609276	0.12397560984528	5.36×10^{-9}
28	0.12397559815068	0.12397559944075	1.29×10^{-9}

Table 6: Upper and lower bounds for γ_f.

d	Left bound	Right bound	Diameter
10	1.13171946004170	1.13225598207888	5.37×10^{-4}
12	1.13192332539355	1.13205276608402	1.29×10^{-4}
14	1.13197258005419	1.13200378629041	3.12×10^{-5}
16	1.13198446947829	1.13199198853091	7.52×10^{-6}
18	1.13198733763601	1.13198914848177	1.81×10^{-6}
20	1.13198802920523	1.13198846514858	4.36×10^{-7}
22	1.13198819589032	1.13198830080464	1.05×10^{-7}
24	1.13198823605206	1.13198826129364	2.52×10^{-8}
26	1.13198824572594	1.13198825179751	6.07×10^{-9}
28	1.13198824805549	1.13198824951584	1.46×10^{-9}

Table 7: Upper and lower bounds for e^{γ_f}.

Viswanath obtained the enclosure $[0.1239755981508, 0.1239755994406]$ for γ_f with a diameter of 1.29×10^{-9}. Here we have computed an enclosure of γ_f with approximately the same diameter, without the need for rounding error analysis.

A MATLAB M-files

A.1 Linear Systems of Equations

intgaus.m
This routine performs interval Gaussian elimination.

```
function x = intgauss(A,b)
%INTGAUSS   Interval Gaussian Elimination with mignitude pivoting.
%
%           x = INTGAUSS(A,b)
%
%           Solves Ax = b for linear systems of equations.
%
%           INPUT:  A   coefficient matrix
%                   b   right hand side vector
%           OUTPUT: x   interval solution

n = length(A);
for i = 1:n-1;
    [maxmig,index] = max(mig(A(i:n,i)));
    if maxmig <= 0
        error('All possible pivots contain zero.')
    end
    k = index+i-1;
    if k ~= i                   % Swap the rows if necessary.
        A([i k],i:n) = A([k i],i:n);
        b([i k]) = b([k i]);
    end

    for j = i+1:n;
        mult = A(j,i)/A(i,i);
        A(j,i+1:n) = A(j,i+1:n)-mult*A(i,i+1:n);
        b(j) = b(j)-mult*b(i);
    end

end

x(n) = b(n)/A(n,n);           % Backsubstitution.
for i = n-1:-1:1;
  x(i) = (b(i)-A(i,i+1:n)*x(i+1:n,1))/A(i,i);
end
```

kraw.m

This routine solves an interval linear system of equations by Krawczyk's method.

```
function x= kraw(A,b)
%KRAW   Solves Ax = b for interval linear systems.
%
%       x = KRAW(A,b)
%
%       Solves linear systems using Krawczyk's method.
%       If A and b have all real components then x is a
%       verified bound on the solution.  If A or b are of interval
%       type then an outer estimate of the interval hull of the
%       system is given.
%
%       INPUT:  A   coefficient matrix
%               b   right hand side vector
%       OUTPUT: x   interval solution

n = length(A);
midA = mid(A);
C = inv(midA);                  % Preconditions system.
b = C*b;
CA = C*intval(A);
A = eye(n)-CA;
beta = norm(A,inf);
if beta >= 1;
    error('Algorithm not suitable for this A')
end;
alpha = norm(b,inf)/(1-beta);
x(1:n,1) = infsup(-alpha,alpha);
s_old = inf;
s = sum(rad(x));
mult = (1+beta)/2;
while s < mult*s_old
    x = intersect(b+A*x,x);     % Krawczyk's iteration.
    s_old = s;
    s = sum(rad(x));
end
```

hsolve.m

This routine solves an interval linear system of equations by a method based on work by Hansen, Bliek, Rohn, Ning, Kearfott and Neumaier.

```
function x = hsolve(A,b)
%HSOLVE     Solves Ax = b for interval linear systems.
%
%           x = HSOLVE(A,b)
```

```
%
%           Solves linear systems using a method motivated by Hansen,
%           Bliek, Rohn, Ning, Kearfott and Neumaier.
%           If A and be have all real components then x is a
%           verified bound on the solution.  If A or b are of interval
%           type then an outer estimate of the interval hull of the
%           system is given.
%
%           INPUT:  A   coefficient matrix
%                   b   right hand side vector
%           OUTPUT: x   interval solution

n = dim(A);
C = inv(mid(A));
A = C*A;
b = C*b;
dA = diag(A);           % Diagonal entries of A.
A = compmat(A);         % Comparison matrix.
B = inv(A);
v = abss(B*ones(n,1));
setround(-1)
u = A*v;
if ~all(min(u)>0)      % Check positivity of u.
    error('A is not an H-matrix')
else
    dAc = diag(A);
    A = A*B-eye(n);   % A contains the residual matrix.
    setround(1)
    w = zeros(1,n);
    for i=1:n,
        w = max(w,(-A(i,:))/u(i));
    end;
    dlow = v.*w'-diag(B);
    dlow = -dlow;
    B = B+v*w;          % Rigorous upper bound for exact B.
    u = B*abss(b);
    d = diag(B);
    alpha = dAc+(-1)./d;
    beta = u./dlow-abss(b);
    x = (b+midrad(0,beta))./(dA+midrad(0,alpha));
end
```

A.2 Univariate Nonlinear Equations

intnewton.m
This routine computes bounds on the roots of a function in a given range.

```
function intnewton(f,X,tol)
```

```
%INTNEWTON  Finds all roots of a function in given range.
%
%           INTNEWTON(f,X,tol)
%
%           Uses an interval version of Newton's method to provide
%           rigorous bounds on the roots of a function f, to be
%           specified seperately.  Bounds are displayed as they are
%           found.  Roots are displayed if radius of enclosure < tol
%           or if enclosure is no longer becoming tighter.  If tol
%           is not given then the later stopping criterion is used.
%
%           INPUT:  f   function of nonlinear equation
%                   X   intitial range
%                   tol tolerance used as stopping criterion


if nargin < 3
    tol = 0;
end

x = intval(mid(X));
fx = feval(f,intval(x));
F = feval(f,gradientinit(X));

if in0(0,F.dx)                                %Checks if f'(x) contains 0.
    a = 1/inf(F.dx); b = 1/sup(F.dx);
    N = x-fx*a; N1 = x-fx*b;                  %Performs Newton' method
    N = infsup(-inf,min(sup(N),sup(N1)));     %with one side of f'(x).
    Xnew = intersect(N,X);
    if X == Xnew
        Xnew = infsup(NaN,NaN);
    end
    if ~isempty(Xnew)
        intnewtall(@f,Xnew,tol);
    end

    a = 1/sup(F.dx); b = 1/inf(F.dx);         %Performs Newton's method
    N = x-fx*a; N1 = x-fx*b;                  %using other side of f'(x).
    N = infsup(max(inf_(N),inf_(N1)),inf);
    Xnew = intersect(N,X);

    if X == Xnew
        Xnew = infsup(NaN,NaN);
    end
    if ~isempty(Xnew)
        intnewtall(@f,Xnew,tol);
    end
```

```
else
    N = x-fx/F.dx;                          %Performs Newton's method.
    Xnew = intersect(N,X);
    if  rad(Xnew) < tol | X == Xnew;        %Stopping Criterion.
        if intersect(0,feval(f,Xnew)) == 0
            disp('Root in the interval')
            Xnew
        end
    elseif ~isempty(Xnew)
        intnewtall(@f,Xnew,tol);
    end
end
```

A.3 Multivariate Nonlinear Equations

nlinkraw.m
This routine finds bounds on the solution of a nonlinear system of equations using the Krawczyk operator.

```
function Y = nlinkraw(f,X)
%NLINKRAW   Bounds roots of nonlinear systems of equations.
%
%           X = NLINKRAW(f,X)
%
%           Uses the Krawczyk operator to produce bounds on
%           a root in a given interval of a nonlinear equation f.
%
%           INPUT:  f   A MATLAB function
%                   X   Initial interval
%           OUTPUT: Y   Interval containing root


X = X(:);
n = length(X);
ready = 0; k = 0;
N = intval(zeros(n,1));

while ~ready
    k = k+1;
    F = feval(f,gradientinit(X));
    C = inv(mid(F.dx));
    x = mid(X);
    fx = feval(f,intval(x));
    N = x-C*fx+(eye(n)-C*(F.dx))*(X-x); %Krawczyk operator.
    Xnew = intersect(N,X);

    if isempty(Xnew)
```

```
        error('No root in box')
    elseif X == Xnew                    %Stopping criterion.
        ready = 1;
    else
        X = Xnew;
    end

end
disp('Number of iterations')
disp(k)
Y = X;
```

nlinhs.m

This routine finds bounds on the solution of a nonlinear system of equations using qthe Hansen–Sengupta operator.

```
function X = nlinhs(f,X)
%NLINHS   Bounds roots of nonlinear systems of equations.
%
%           X = NLINHS(f,X)
%
%           Uses the Hansen-Sengupta operator to produce bounds on
%           a root in a given interval of a nonlinear equation f.
%
%           INPUT:  f   A MATLAB function
%                   X   Initial interval
%           OUTPUT: Y   Interval containing root

X = X(:);
Y = X;
n = length(X);
ready = 0; k = 0;
N = intval(zeros(n,1));
while ~ready
    k = k+1;
    F = feval(f,gradientinit(X));
    A = F.dx;
    C = inv(mid(A));
    x = mid(X);
    fx = feval(f,intval(x));
    M = C*A;
    b = C*fx;

    if in(0,M(1,1))
        error('division by zero')
    end
```

```
                                %Hansen-Sengupta operator.
    N(1) = x(1)-(b(1)+M(1,2:n)*(X(2:n)-x(2:n)))/M(1,1);
    Y(1) = intersect(N(1),X(1));
    for i = 2:n
        if in(0,M(i,i))
            error('division by zero')
        end
        N(i) = x(i)-(b(i)+M(i,1:i-1)*(Y(1:i-1)-x(1:i-1))+ ...
                    M(i,i+1:n)*(X(i+1:n)-x(i+1:n)))/M(i,i);
        Y(i) = intersect(N(i),X(i));
    end

    if any(isempty(Y))
        error('No root in box')
    elseif X == Y
        ready = 1;              %Stopping Criterion.
    else
        X = Y;
    end
end
disp('Number of iterations')
disp(k)
```

allroots.m

This routine bounds all solutions of a nonlinear system of equations in a given box.

```
function  allroots(f,X)
%ALLROOTS   Finds all solutions to nonlinear system of equations.
%
%           ALLROOTS(f,X)
%
%           Uses the Krawczyk operator with generalised bisection
%           to find bounds on all solutions to a nonlinear
%           system, given by the function f, in the box X.
%
%           INPUT:  f   function of nonlinear system
%                   X   initial box


ready = 0;
X = X(:);
n = length(X);
N = intval(zeros(n,1));

while ~ready
    F = feval(f,gradientinit(X));
    C = inv(mid(F.dx));
    x = mid(X);
    fx = feval(f,intval(x));
    N = x-C*fx+(eye(n)-C*(F.dx))*(X-x); %Krawczyk operator.
    Xnew = intersect(N,X);
    if any(isempty(Xnew))
        ready = 1;
    elseif max(rad(X)-rad(Xnew)) <= 0.5 * max(rad(X))
        ready = 1;        %Checks if width sufficiently reduced.
    else
        X = Xnew;
    end
end

if max(rad(Xnew)) < 10e-10
    Xnew                                %Output.
elseif ~isempty(Xnew)  %Splits box and calls function on subboxes
    T = rad(Xnew).*(ones(1,n)*abss(F.dx))';
    [a,b] = max(T);
    Xnew1 = Xnew;
    Xnew2 = Xnew;
    Xnew1(b) = infsup(inf(Xnew(b)),mid(Xnew(b)));
    allroots(@f,Xnew1);
    i = i+1;
    Xnew2(b) = infsup(mid(Xnew(b)),sup(Xnew(b)));
    allroots(@f,Xnew2);
    i = i+1;
end
```

A.4 Viswanath's Constant

viswanath.m

This M-file produces a bound on Viswanath's constant without the need for rounding error analysis.

```
%Attempts to find rigorous bounds for Viswanath's constant
%without rounding error analysis.

d = 12;     %Change value of d to change accuracy (must be even).
flipno = 0;
for i = 1:2:d
    flipno = flipno+2^i;
end

clear larry1 uarray1 larray2 uarray2 measure left right
l = 0;
u = 0;
```

```
startnum = 10;
if d > 25
    maxnum = 24;
else
    maxnum = d;
end
n = 2^d;
nrt = sqrt(n);

itable = intval(zeros(d+1,1));
g = (intval(1)+sqrt(intval(5)))/intval(2);
for i = 0:d
    itable(i+1) = (g^(d-i)*(1+g)^(-d))/4;
end

global num
num = zeros(2^maxnum,1);
num(1) = 1;
num(2) = 1;
for i = 2:2:2^startnum
    num(i+1) = num(i/2+1);
    num(i+2) = num(i/2+1)+num(i/2+2);
end
for i = startnum+1:maxnum   %First 2^maxnum Stern-Brocot coefficients.
    num(2^(i-1)+1:2:2^i+1) = num([2^(i-1):2:2^i]./2+1);
    num(2^(i-1)+2:2:2^i+2) = num([2^(i-1):2:2^i]./2+1)+ ...
            num([2^(i-1):2:2^i]./2+2);
end

for i = 0:nrt-1
    k = i*nrt;
    m1 = intval((stern(k+1)))/intval((stern(n-k+1)));
    left1 = AMP(m1);
    if i < nrt/4
        left(1) = left1;
        m = intval(stern([k+2:k+nrt+1]))./ ...
            intval(stern(n-k:-1:n-(k+nrt-1)));
        b = flip(k+(0:nrt-1),flipno);
        measure(1:nrt,1) = itable([b(1:nrt)+1]);
        right = AMP(m);
        left(2:nrt,1) = right(1:nrt-1);
        larray2 = measure'*inf(right);
        uarray2 = measure'*sup(left);
    elseif i < nrt-1
        left(1) = left1;
        m = intval(stern(k+2:k+nrt+1))./ ...
            intval(stern(n-k:-1:n-(k+nrt-1)));
        b = flip(k+(0:nrt-1),flipno);
        measure(1:nrt,1) = itable([b(1:nrt)+1]);
        right = AMP(m);
        left(2:nrt,1) = right(1:nrt-1);
        larray2 = measure'*inf(left);
        uarray2 = measure'*sup(right);
    else                %i = nrt-1.
        left(1) = left1;
        m = intval(stern(k+2:k+nrt+1))./ ...
            intval(stern(n-k:-1:n-(k+nrt-1)));
        b = flip(k+(0:nrt-1),flipno);
        measure(1:nrt,1) = itable([b(1:nrt)]+1);
        right = AMP(m);
        right(nrt) = log(intval(4));
        left(2:nrt,1) = right(1:nrt-1);
        larray2 = measure'*inf(left);
        uarray2 = measure'*sup(right);
    end
    l = l+larray2;
    u = u+uarray2;
end

lower = inf(l);        %Output.
upper = sup(u);
disp('Enclosure of gamma is')
gammaf = infsup(lower,upper)
disp('Enclosure of exp(gamma) ie. Viswanaths constant')
viswan = exp(gammaf)
```

stern.m
This routine finds the Stern-Brocot tree coefficients.

```
function y = stern(x);
%STERN      Calculates the Stern-Brocot coefficients.
%
%           y = STERN(x)
%
%           If length(x) < 2^maxnum then the coefficients are taken
%           from the variable num, in viswanath.m.  Otherwise
%           a recursive procedure is carried out.


global num
n = length(x);
y = zeros(n,1);
maxnum = 2^24;
```

```
if x(n) <= maxnum
    y=num(x);
else
    for i = 1:n
        if mod(x(i),2) == 1
            y(i) = stern((x(i)-1)/2+1);
        else
            xim2d2 = (x(i)-2)/2;
            y(i) = stern(xim2d2+1)+stern(xim2d2+2);
        end
    end
end
```

flip.m
This routine finds the index i into the variable `itable` for finding $\nu(I_j)/2$.

```
function j = flip(x,flipno)
%FLIP   Index used in viswanath.m.
%
%       j = FLIP(x,flipno)
%
%       Index into array itable.  The binary representation
%       of x is taken and then odd bits are fliped.  The
%       variable j is taken as the number of 1's.

n = length(x);
y = bitxor(flipno,x);
Q = dec2bin([y(1:n)]);
q = Q == '1';
j = sum(abss(q'));
```

amp.m
This routine produces bounds on $amp(m)$.

```
function y = AMP(m)
%AMP        Produces enclosures on amp(m).

m2 = m.^2;
m2p1 = m2+1;
y = log(((m2.^2).*4+1)./(m2p1.^2));
```

References

[1] C. Bliek. *Computer Methods for Design Automation.* PhD thesis, Massachusetts Institute of Technology, August 1992.

[2] L. E. J. Brouwer. Uber abbildung von mannigfaltigkeiten. *Math. Ann.*, 71:97–115, 1912.

[3] I. Gargantini and P. Henrici. Circular arithmetic and the determination of polynomial zeros. *Numer. Math.*, 18:305–320, 1972.

[4] J. Hales. The Kepler conjecture. `http://www.math.pitt.edu/~thales/countdown`.

[5] E. R. Hansen. On solving systems of equations using interval arithmetic. *Mathematics of Computation*, 22(102):374–384, April 1968.

[6] E. R. Hansen. Bounding the solution of interval linear equations. *SIAM Journal on Numerical Analysis*, 29(5):1493–1503, October 1992.

[7] E. R. Hansen. *Global Optimization Using Interval Analysis.* Marcel Dekker, New York, 1992.

[8] E. R. Hansen and S. Sengupta. Bounding solutions of systems of equations using interval analysis. *BIT*, 21(2):203–211, 1981.

[9] IEEE standard for binary floating point arithmetic. Technical Report Std. 754–1985, IEEE/ANSI, New York, 1985.

[10] Luiz Henrique de Figueiredo Joao Batista Oliveira. Interval computation of Viswanath's constant. *Reliable Computing*, 8(2):121–139, 2002.

[11] R. Krawczyk. Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. *Computing*, 4:187–201, 1969.

[12] R. Moore. *Interval Arithmetic.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1966.

[13] A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge University Press, 1990.

[14] A. Neumaier. A simple derivation of the Hansen-Bliek-Rohn-Ning-Kearfott enclosure for linear interval equations. *Reliable Computing*, 5(2):131–136, 1999.

[15] S. Ning and R. B. Kearfott. A comparison of some methods for solving linear interval equations. *SIAM Journal on Numerical Analysis*, 34(4):1289–1305, August 1997.

[16] J. Rohn. Cheap and tight bounds: The recent result by E. Hansen can be made more efficient. *Interval Computations*, (4):13–21, 1993.

[17] S. M. Rump. Fast and parallel interval arithmetic. *BIT*, 39(3):534–554, 1999.

[18] S. M. Rump. INTLAB — INTerval LABoratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–105. Kluwer, Dordrecht, Netherlands, 1999.

[19] C. A. Schnepper and M. A. Stadtherr. Robust process simulation using interval methods. *Comput. Chem. Eng.*, 20:187–199, 1996.

[20] Divakar Viswanath. Random Fibonacci sequences and the number 1.13198824.... *Mathematics of Computation*, 69(231):1131–1155, July 2000.