Exact Real Computation in Computer Algebra*

Gábor Bodnár, Petru Pau, Josef Schicho
[gbodnar,ppau,jschicho]@risc.uni-linz.ac.at
Research Institute for Symbolic Computation
Johannes Kepler University Linz
A-4040 Linz, Austria

Abstract

This paper describes our approach to represent computable real numbers in Maple. We aimed at a model that can be used in symbolic computation.

The main topics we discuss are: representation and arithmetic of exact real numbers, vectors and matrices; polynomial arithmetic and root computation. Some operations raise ill-posed problems; we used regularization methods to solve them.

We also give non-trivial examples of the theory that supports our algorithms.

1 Introduction

Traditional ways of representing real numbers symbolically reach only a restricted subset of reals, such as the rationals or the real algebraic numbers. Many computations lead outside these subsets. Some operations which stay within these subsets turn out to be expensive in this representation (see [12]).

The most common way to represent reals in a computer, the so-called *floating point*, has its own disadvantages: the represented set is a subset of the rationals; the density of the numbers is not uniform; during some computations, the propagation of the errors cannot be controlled.

There have been attempts to work with real numbers exactly. Turing [15] shows that only a proper subset of the real numbers can be reached by any representation system – the *computable* reals. Specific ways to represent the reals have been invented; we mention here infinite (lazy) strings of digits ([6], with golden ratio notation, [9], [5]) and a functional approach, where the reals are represented as functions that produce approximations on demand ([3]).

From a numerical point of view, exact real computation can be viewed as a *safe numerical method with adaptive precision*. However, in traditional numeri-

*Granted by the Austrian *Fonds zur Förderung der wissenschaftlichen Forschung*.

cal computation it is often difficult to ensure safety, i.e. to provide guaranteed error bounds that are sufficiently small (see [13]).

In contrast to floating point representations, exact real computation is intended to provide *a mathematically consistent representation* of the reals: the models behave exactly like the mathematical objects which they represent. This makes it possible to devise and verify algorithms on a high level of abstraction.

The above arguments make it plausible that exact real computation is sometimes much faster than traditional symbolic computation, and sometimes much easier to analyze/interpret than numerical computation. Indeed, there are examples were it outperforms symbolic computation, and other examples where its superiority with respect to accuracy and consistency to numerical computation becomes apparent.

Our belief is that *exact real computation can be realized within existing computer algebra systems*. Existing computer algebra software already contains most of the necessary ingredients for such an implementation: high level algorithms using arbitrary precision arithmetic, conversion between numerical and symbolic data. For many problems, it suffices to supply small wrappers which encode the necessary information about the error propagation. Very often this information is already available in the literature of numerical analysis (see e.g. [7], [1]).

We have solved easy but nontrivial problems such as *linear equations, root finding, gcd computation*, in the sense of exact real computation. Our Maple implementation is based on an objectual approach, which makes it easy to change details in the representation independently of the higher level algorithms.

In more difficult problems, like *multivariate polynomial factorization, curve and surface parameterization*, such a straightforward solution is not possible. The main source of difficulty often lies in the fact that these problems are *numerically ill-posed*. The solution to those problems is a topic of current research. We give some ideas from both disciplines, numerics and symbolic computation, which seem to be promising for obtaining a solution.

The paper is organized as follows: In Section 2, we describe the representation of real objects (numbers, vectors, matrices). Section 3 contains the description of the linear algebra part of our work, together with some theoretical facts. In Section 4, we introduce polynomials with exact real coefficients and the most important operations.

Throughout the paper, we insert some portions of the code. We have to specify that these are only excerpts from the original code; the original procedures contain other lines (usually handling floating-point variables and parameters) that have not been included here. An archive with the Maple package described here will be made available in the web site of our institute (RISC, Johannes Kepler University, Austria, `http://www.risc.uni-linz.ac.at`).

2 Representation of Real Numbers

The main goal of our attempt is to provide reliable representations of real numbers that will be used in symbolic (algebraic) computation. This leads to a two-fold constraint: First of all, we have to embed everything in a software which has powerful symbolic machinery. On the other hand, the objects that represent real numbers should be easily handled in algebraic contexts.

Maple is a natural choice regarding the software environment. Apart from the symbolic expression manipulation, Maple has a good deal of numerical methods. This allows us to define some operations on real numbers simply as *wrappers* for Maple procedures.

The first task would be to provide a sound arithmetic with real numbers. We should give means to answer queries like *determine the result of a computation with accuracy* ε. This is not a trivial task, because Maple does not give guaranteed answers to such queries, as the following example proves.

Example. A good test for arithmetic with real numbers in floating-point representation is the following sequence of rational numbers (discovered by Jean-Michel Muller [10], also used by Ménissier-Morain in [9]):

$$a_0 = \frac{11}{2},\ a_1 = \frac{61}{11},\ a_{n+1}\frac{1130 - 3000/a_{n-1}}{a_n}.$$

Maple floating-point arithmetic fails: here are the values of the first 20 elements of this sequence, computed with 20 digits of precision (10 less significant digits have been cropped from the results):

a_0=5.5	a_1=5.545454545	a_2=5.590163934
a_3=5.633431085	a_4=5.674648620	a_5=5.713329052
a_6=5.749120919	a_7=5.781810920	a_8=5.811314238
a_9=5.837656549	a_{10}=5.860951537	a_{11}=5.881377465
a_{12}=5.899158157	a_{13}=5.914597014	a_{14}=5.928959639
a_{15}=5.959595154	a_{16}=6.293384805	a_{17}=11.43353126
a_{18}=53.86026177	a_{19}=94.89139147	a_{20}=99.67863258

The next values are closer and closer to 100. However, it can be easily proved that the limit of the sequence is 6.

The first 21 values of this sequence, computed within the arithmetic that we developed (the values are guaranteed with accuracy 10^{-10}) follow:

a_0 = 5.5	a_1 = 5.5454545454	a_2 = 5.5901639344
a_3 = 5.6334310850	a_4 = 5.6746486205	a_5 = 5.7133290523
a_6 = 5.7491209197	a_7 = 5.7818109204	a_8 = 5.8113142382
a_9 = 5.8376565489	a_{10} = 5.8609515225	a_{11} = 5.8813772158
a_{12} = 5.8991539057	a_{13} = 5.9145249506	a_{14} = 5.9277414077
a_{15} = 5.9390504854	a_{16} = 5.9486874924	a_{17} = 5.9568707319
a_{18} = 5.9637987208	a_{19} = 5.9696491440	a_{20} = 5.9745790286

We try to impose a high level of abstractization to real number data type. Unfortunately, Maple does not provide means for object oriented programming. We simulate this by *partitioning* the primitive/operations with real numbers *into layers* so that, in a certain measure, the level of abstractization is kept high.

2.1 A Data Type for Real Numbers

Theoretically, the mathematical objects that we want to represent are *computable real numbers*, in the sense of Turing ([15], see also Ménissier-Morain [9], Weihrauch [16]).

From the software engineering point of view, an exact real number r is an object that fulfills the following specifications:

- for every rational positive ε, it should be possible to compute a rational value r_0 which is an *approximation of accuracy ε for r* (we also call this ε-approximation), i.e., $|r - r_0| \leq \varepsilon$;
- if an approximation of accuracy ε is known, an approximation of accuracy $\varepsilon/2$ should be easily obtainable (or, in general, ε/c, where c is greater than 1);
- the most precise approximation already computed should be stored, together with its accuracy.

At the same time, the user should be able to handle real numbers symbolically, without necessarily computing more or less accurate approximations of them.

It is not necessary to require the fulfillment of both the first two specifications: for some real numbers, one can give easily a method to increase the accuracy, rather than to produce an approximation of desired accuracy. Anyway, each of the first two specifications can be obtained, if the other is provided.

In one sense, the implementation that we envisage is closer to *lazy* computation: The real numbers will be handled by means of some identifiers associated to them. Only when required, an approximation is computed.

The operations on real numbers can be partitioned into three layers:

1. basic constructors;
2. queries;
3. arithmetic operations (these can also be seen as constructors, but they use only functions from the two previous layers).

The algebraic operations will constitute a fourth class; we did not include it in the previous enumeration because the operands are now composite objects (polynomials with exact real numbers coefficients).

Before going into the details of the operations with real numbers, we define the notion of an *epsilon*-interval:

Definition 1. (Epsilon-interval) *An* epsilon-*interval is a tuple* $(q, \varepsilon) \in \mathbf{Q} \times \mathbf{Q}^+$*, representing the interval* $[q - \varepsilon, q + \varepsilon]$.

It is clear now, from the specifications of an exact real number, that such an object must have

1. a data-member – an *epsilon*-interval: `knownApprox`;
2. at least one of the following:
   - a function of parameter $\varepsilon > 0$ for computing an approximation with accuracy ε: `approx`;
   - a function of parameter k, with k an approximation of the real number given as an *epsilon*-interval, for computing a more accurate approximation: `iterate`.

Basic constructors

The procedure `buildRealObject` creates an exact real number. It has two or three arguments:

1. an *epsilon*-interval, with an initial approximation of the number;
2. at least one of the following:
   - a function for computing approximations of accuracy ε, with parameter ε;
   - a function for computing a better approximation, using as basis a given one, transmitted as parameter, as an *epsilon*-interval.

Figure 1 contains Maple code for `buildRealObject`. This constructor assigns the new real number to a new identifier whose name has been chosen at random from the set of unassigned names.

```
buildRealObject := proc(b::epsilonInterval,
                        f::procedure,
                        g::procedure)
  table([knownApprox=b,
         approx     =f,
         iterate    =g]);
end:
```

Figure 1: Maple code for `buildReal`.

The simplest way to create exact real numbers is by using Maple constants; the procedure `symb2real`, with one argument, does this.

Queries

We have foreseen three essential queries:

- `dump` – for reporting the *epsilon*-interval stored as the most accurate approximation computed so far;

- `epsint` – for computing an approximation of accuracy ε. This procedure has as side-effect the modification of the stored data, if the newly computed approximation is more accurate;
- `refine` – for refining the *epsilon*-interval: if (q_0, ε) is the stored value, `refine` computes and reports the *epsilon*-interval corresponding to a fraction of ε. The new *epsilon*-interval replaces the old value of `knownApprox`.

2.2 Real Multidimensional Objects

Not only computable real numbers can be represented in this way, but also real vectors and matrices. The whole theory in this case is based on the *metric* properties of these spaces.

Let $||\cdot||$ denote a norm on $\mathbf{R}^n$. A n-dimensional *epsilon*-interval is a tuple $(v, \varepsilon) \in \mathbf{Q}^n \times \mathbf{Q}^+$ that represents the disc centered at v, of radius ε, in the given norm. For a real vector v, a vector v_0 with rational coordinates is said *approximation of accuracy* ε if $||v - v_0|| \leq \varepsilon$.

Similar definitions can be introduced for matrices. However, sometimes problems may occur because of the norm. Usually, in numerical analysis, people use the Euclidean norm for vector spaces and its corresponding operator norm for matrices. We use here Manhattan norm for vectors:

$$||(v_1, v_2, \ldots, v_n)||_1 := \sum_{i=1}^{n} |v_i|,$$

and also Manhattan norm for matrices, which are seen as vectors with n^2 entries. The reason is that the unit disk in Manhattan norm is included in the unit disk of any other (usual) norm: If $(\mathbf{v}_0, \varepsilon)$ is an approximation of a real vector $\mathbf{v}$, this means that $||\mathbf{v} - \mathbf{v}_0||_1 \leq \varepsilon$, in other words, $\mathbf{v}$ lies in the disk of center $\mathbf{v}_0$ and radius ε, with respect to the Manhattan norm. Since this disk is included in the disk of radius ε with respect to any other norm $||\cdot||$, we have $||\mathbf{v} - \mathbf{v}_0|| \leq \varepsilon$.

This is useful because in applications the error analysis is done in different norms.

It is easy to see that, for matrices, Manhattan norm is *consistent*, i.e., $||AB|| \leq ||A|| \cdot ||B||$; however, it is not an operator norm.

The constructor `buildRealObject` has been extended to cover real vectors and matrices. The arguments to the constructor should be given accordingly. Note that a real vector or matrix is *not* a vector or matrix with exact real number entries.

The queries which apply to real numbers can be used for real vectors and matrices as well.

2.3 Arithmetic Operations

The result of an arithmetic operation should be an exact real number (vector, matrix), therefore the functions that implement them must behave like *constructors*.

Let a, b be two exact real numbers (vectors, matrices), and let $\mathtt{op}$ be an arithmetic operation ($\mathtt{op} \in \{+, -, *, /, \cdot^{-1}, \dots\}$). In order to construct the object $c := a \,\mathtt{op}\, b$, we should:

- provide an initial approximation for c (presumably using the known approximations of a and b);
- design a function that computes ε-approximations of c.

We have to solve two problems (we used here the terminology from [7], pag. 7):

Problem 1. (Forward error computation) *Given* (a_0, δ_a) *and* (b_0, δ_b) *two* epsilon-*intervals for* a *and* b *respectively, find an* epsilon-*interval* (c_0, ε) *for* c *such that* $c_0 = a_0 \,\mathtt{op}\, b_0$.

Problem 2. (Backward error computation) *Given* $\varepsilon > 0$*, find* $\delta_a, \delta_b > 0$ *such that if* (a_0, δ_a) *and* (b_0, δ_b) *are* epsilon-*intervals for* a *and* b *respectively, then* $(a_0 \,\mathtt{op}\, b_0, \varepsilon)$ *is an* epsilon-*interval for* c.

Together with this error computation, we should always take into account that arithmetic operations on rational numbers are not always computed exactly, when these numbers are represented as floats. This happens even if the mantissa may have non-limited length.

The arithmetic operations are continuous; this is why the backward error computation is sometimes called the computation of the *modulus of continuity*.

The computation of the errors is usually straightforward. We insert here the complete analysis of the product of real numbers.

Error computation for the product

Let a, b be the operands, and a_0, b_0 their best known approximations, with accuracies δ_a and δ_b.

Then $a_0\, b_0$ will be an approximation of $a\, b$ with precision computed as follows:

We know:

$$|a_0 - a| \le \delta_a, \ \ |b_0 - b| \le \delta_b;$$

from here we deduce:

$$|a_0\, b_0 - a\, b| \le |b_0|\, |a_0 - a| + |a|\, |b_0 - b|,$$

$$|a| \le |a_0| + \delta_a,$$

thus:

$$|a_0\, b_0 - a\, b| \le |b_0|\, |a_0 - a| + (|a_0| + \delta_a)\, |b_0 - b| = |b_0|\, \delta_a + |a_0|\, \delta_b + \delta_a\, \delta_b.$$

What we have computed so far is an estimate for the forward error.

For the backward error, we have a positive rational number ε and we seek δ'_a, δ'_b such that, if $|a_1 - a| \le \delta'_a$, $|b_1 - b| \le \delta'_b$, then $|a\, b - a_1\, b_1| \le \varepsilon$.

Using the previous computation, it would be enough to choose δ'_a, δ'_b such that:

$$\delta'_a\,|b_1| \le \frac{\varepsilon}{2},$$

$$\delta'_b\,|a_1| \le \frac{\varepsilon}{2}.$$

We use the most accurate approximation already known for a:

$$|a| \le |a_0| + \delta_a,$$

and set a value for δ'_b from the second inequality:

$$\delta'_b := \frac{\varepsilon}{2\,(|a_0| + \delta_a)}.$$

Now, using $|b_1| <= |b_0| + \delta_b + \delta'_b$, we can set a value for δ'_a:

$$\delta'_a := \frac{\varepsilon}{2\,(|b_0| + \delta'_b + \delta_b)}.$$

```
realProd := proc(a::number,
                 b::number)
  local bna0a,bna0b,  # known approximations of a and b
        p,e;          # (p,e) is the epsilon-interval for product

  if a=0 or b=0 then RETURN(0) fi;
  if a=1        then RETURN(b) fi;
  if b=1        then RETURN(a) fi;
  bna0a := dump(a);
  bna0b := dump(b);
  # compute the initial approximation of the product
  p := bna0a[1]*bna0b[1];
  d := (abs(bna0a[1])+bna0a[2])*bna0b[2] + abs(bna0b[1])*bna0a[2];

  buildReal([p,d],
    proc(eps::nonneg)
      local d1,d2;    # the accuracies of a and b
      d2 := e1/(abs(bna0a[1])+bna0a[2]);
      d1 := e1/(abs(bna0b[1])+2*max(d2,bna0b[2]));
      evalf(epsint(a,d1)*epsint(b,d2))
    end)
end:
```

Figure 2: Maple code for the product of two real numbers.

Figure 2 contains a first draft of the code of the product of two real numbers. Things are a little bit more complicated, because of the floating-point arithmetic in Maple: Without explicitly requesting the precision of an arithmetic operation, Maple does everything with ten digits. This would clearly reduce the accuracy of the results, and would make the error analysis impossible. We provide small wrappers for the usual arithmetic operations with floats, so that the computation takes place with as many digits of precision as given by the sizes of the operands.

It is easy (and desired) to extend the domain of `realProd` to other types of exact real objects: real vectors and matrices. Other arithmetic operations are also defined for the real numbers and overloaded for vectors and matrices.

3 Real Linear Algebra

In this section we give some details about the representation of vectors and matrices of real numbers, and we describe some nontrivial standard operations: matrix inversion and computation of the Moore-Penrose generalized matrix inverse, also known as the pseudo-inverse.

The straightforward way of describing multidimensional entities of real numbers would be to build them up from exact real objects presented in section 2. However, in this case all the standard methods of linear algebra should be re-implemented to work with exact real objects, and the resulting exact real entries of vectors and matrices would be deeply cascaded. All this would result in severe disadvantage in computational efficiency.

In our approach, any multidimensional real entity is treated as an exact real object in its corresponding approximation space, with a carefully chosen distance function. For instance, an exact real matrix $A \in \mathbf{R}^{m\times n}$ is represented by: (1) a rational approximation, a matrix $A_0 \in \mathbf{Q}^{m\times n}$, together with $\varepsilon \in \mathbf{Q}_+$, such that $\|A - A_0\| \le \varepsilon$, and (2), the `approx` function that produces approximates of desired accuracy.

The usual arithmetic operations on real vectors and matrices can easily be described by means of forward and backward error analysis, like in the case of real numbers. In the following subsections we discuss some nontrivial operations.

3.1 Inverse of a nonsingular matrix

Let $A \in \mathbf{R}^{n\times n}$ be a nonsingular matrix with best known approximate $(A_0, \delta_0) \in \mathbf{Q}^{n\times n} \times \mathbf{Q}_+$. We determine first the forward error; we have

$$\begin{aligned}\|A^{-1} - A_0^{-1}\| &= \|A^{-1}A(A^{-1} - A_0^{-1})A_0A_0^{-1}\| \le \\ &\le \|A^{-1}\| \cdot \|A_0 - A\| \cdot \|A_0^{-1}\| \le \delta_0\|A^{-1}\| \cdot \|A_0^{-1}\|.\end{aligned} \tag{1}$$

We have to find an upper bound for $\|A^{-1}\|$, so by using inequality 1 we get

$$\|A^{-1}\| \le \|A_0^{-1}\| + \|A^{-1} - A_0^{-1}\| \le \|A_0^{-1}\| + \delta_0\|A^{-1}\| \cdot \|A_0^{-1}\|$$

yielding

$$\|A^{-1}\| \le \frac{\|A_0^{-1}\|}{1 - \delta_0\|A_0^{-1}\|},$$

whenever $\delta_0\|A_0^{-1}\| < 1$. Because A is nonsingular, $\|A_0^{-1}\|$ converges to $\|A^{-1}\| \in \mathbf{R}_+$ as δ_0 goes to zero, thus the previous condition can be fulfilled within finitely many iterative refinements of the approximation A_0 of A.

Assuming that (A_0, δ_0) is a good enough approximation to have $\delta_0\|A_0^{-1}\| < 1$, for the forward error we get

$$\|A^{-1} - A_0^{-1}\| \leq \frac{\delta_0\|A_0^{-1}\|^2}{1 - \delta_0\|A_0^{-1}\|}.$$

In the backward error computation, for a given $\varepsilon \in \mathbf{Q}_+$, we determine $\delta_\varepsilon \in \mathbf{Q}_+$ such that $\|A - A_\varepsilon\| \leq \delta_\varepsilon$ implies $\|A^{-1} - A_\varepsilon^{-1}\| \leq \varepsilon$. We apply inequality (1) for A and A_ε, getting

$$\|A^{-1} - A_\varepsilon^{-1}\| \leq \delta_\varepsilon\|A^{-1}\| \cdot \|A_\varepsilon^{-1}\|,$$

and because $\|A_\varepsilon^{-1}\| \leq \|A^{-1}\| + \varepsilon$, according to the upper bound for $\|A^{-1}\|$, we can write

$$\|A^{-1} - A_\varepsilon^{-1}\| \leq \delta_\varepsilon \frac{\|A_0^{-1}\|}{1 - \delta_0\|A_0^{-1}\|}\Big(\frac{\|A_0^{-1}\|}{1 - \delta_0\|A_0^{-1}\|} + \varepsilon\Big).$$

If we choose δ_ε such that the above product is less than ε, we can compute an ε-approximate for A^{-1} from A_ε. For instance, we may set:

$$\delta_\varepsilon := \frac{\varepsilon(1 - \delta_0\|A_0^{-1}\|)^2}{\|A_0^{-1}\|^2 + \varepsilon\|A_0^{-1}\| - \varepsilon\delta_0\|A_0^{-1}\|^2}.$$

These error bounds are valid only if the inverse computation on a nonsingular matrix from $\mathbf{Q}^{n\times n}$ is exact.

Figure 3 contains the most important part of the code for computing the inverse of a real matrix. If the initial matrix is nonsingular, the inverse will be stored as a real object.

3.2 Pseudo-inverse

In this section we use $\|\cdot\|_2$ as the default norm. Given $A \in \mathbf{R}^{m\times n}$, the pseudo-inverse of A is the unique matrix $A^\dagger \in \mathbf{R}^{n\times m}$, which, for any $b \in \mathbf{R}^m$, provides $x = A^\dagger b$ in $\mathbf{R}^n$ that minimizes $\|Ax - b\|$, with the smallest $\|x\|$.

Clearly, if A is nonsingular, then the pseudo-inverse coincides with the inverse. In the general case, pseudo-inverse computation is an ill-posed problem; in order to solve it we use a special technique, known as *Tikhonov regularization*. We solve the following optimization problem:

$$\text{for } \alpha > 0 \text{ find } x \text{ such that } \|Ax - b\|^2 + \|\alpha x\|^2 \text{ is minimal.} \tag{2}$$

For all $\alpha > 0$ problem 2 is continuous in A, b, and as $\alpha \to 0$, its solution A^α converges to $A^\dagger$.

```
realMatrixInverse := proc(a::rMatrix)
  local da,   # the current epsilon interval for a at any stage
        na1,  # the norm of the current approximate of a
        a1,   # the current approximate for the inverse of a
        e,    # the forward error
        omda0;# a precomputed subexpression of the formula of the
              # forward and the backward error
  # if the real matrix is 0, this loop can be infinite
  da := dump(a);
  while norm(da[1]) <= da[2] do
    da := refine(a)[1]
  od;
  # if the real matrix is singular, this loop can be infinite
  a1 := traperror(inverse(da[1]));
  while a1 = `singular matrix` do
    a1 := traperror(inverse(refine(a)[1]))
  od;
  na1:= norm(a1); da := dump(a);
  while da[2]*na1 >= 1 do
    da := refine(a);
    a1 := traperror(inverse(da[1]));
    if a1 = lasterror then next; fi;
    na1:= norm(a1)
  od;
  omda0 := 1.0-da[2]*na1;
  e := da[2]*na1^2/omda0;
  buildRealObject([evalm(a1),e],
    proc(eps::nonneg)
    local d,a1;
    d := omda0^2*eps/(na1*(na1+eps*omda0));
    # if the real matrix is singular, this loop can be infinite
    a1 := traperror(inverse(epsint(a,d)));
    while a1 = lasterror do
      a1 := traperror(inverse(refine(a)[1]))
    od;
    a1
    end,0)
end:
```

Figure 3: Computing the inverse of a real matrix.

We find the optimum of the quadratic function in x, at the zero of the derivative,

$$2\langle A\dot{x}, Ax - b\rangle + 2\alpha\langle \dot{x}, x\rangle = 2\langle \dot{x}, A^T(Ax - b) + \alpha x\rangle$$

which vanishes if $A^T(Ax - b) + \alpha x = 0$, or in other terms at

$$x = (A^T A - \alpha I^{n\times n})^{-1} A^T b.$$

Because the solution of problem 2 approximates $A^\dagger b$ as α goes to zero, we have $\lim_{\alpha\to 0} A^\alpha = A^\dagger$, where $A^\alpha := (A^T A - \alpha I^{n\times n})^{-1} A^T$. We also remark that because $A^T A - \alpha I^{n\times n} \in \mathbf{R}^{n\times n}$ is an invertible matrix, except for finitely many values of α, its inverse is computable.

Now let A_0 be an approximate of A, and let σ be the minimum of the positive singular values of A; assume that we have knowledge of this value. Now:

$$\|A^\dagger - A_0^\alpha\| \leq \|A^\dagger - A^\alpha\| + \|A^\alpha - A_0^\alpha\|, \tag{3}$$

where the second summand can be bounded by standard error bounds computed for the basic operations on matrices. In the following, we determine an upper bound for the first summand.

Let $A = U\Delta V$ be a singular value decomposition, where $U \in \mathbf{R}^{m\times m}, V \in \mathbf{R}^{n\times n}$ are orthogonal matrices, and $\Delta \in \mathbf{R}^{m\times n}$ is a diagonal matrix, with elements $(\sigma_1, \ldots, \sigma_i, \ldots, 0)$. Using this decomposition of A we have

$$\begin{aligned}A^\alpha &= (V^T \Delta^2 V + \alpha I^{n\times n})^{-1} V^T \Delta U^T = (V^T(\Delta^2 + \alpha I^{n\times n})V)^{-1} V^T \Delta U^T = \\ &= V^T(\Delta^2 + \alpha I^{n\times n})^{-1}\Delta U^T.\end{aligned}$$

Thus, by the convergence of A^α to $A^\dagger$, we have $A^\dagger = V^T \Delta^\dagger U^T$, and since we use 2-norm we can write

$$\begin{aligned}\|A^\dagger - A^\alpha\| &= \|V^T(\Delta^\dagger - (\Delta^2 - \alpha I^{n\times n})^{-1})U^T\| = \\ &= \|\Delta^\dagger - (\Delta^2 - \alpha I^{n\times n})^{-1}\|.\end{aligned}$$

Let us expand the following expressions,

$$\begin{aligned}\Delta^2 + \alpha I^{n\times n} &= \operatorname{diag}(\sigma_i^2 + \alpha), \\ (\Delta^2 + \alpha I^{n\times n})^{-1} &= \operatorname{diag}\Big(\frac{1}{\sigma_i^2 + \alpha}\Big), \\ (\Delta^2 + \alpha I^{n\times n})^{-1}\Delta &= \operatorname{diag}\Big(\frac{\sigma_i}{\sigma_i^2 + \alpha}\Big).\end{aligned}$$

On the other hand, we have $\Delta^\dagger = \operatorname{diag}(\sigma_i^\dagger)$, where

$$\sigma_i^\dagger = \begin{cases} 1/\sigma_i & \text{if } \sigma_i \neq 0 \\ 0 & \text{otherwise.} \end{cases}$$

Therefore we get

$$\Delta^\dagger - (\Delta^2 - \alpha I^{n\times n})^{-1} = \mathrm{diag}\Big(\sigma_i^\dagger - \frac{\sigma_i}{\sigma_i^2+\alpha}\Big) = \mathrm{diag}(\delta_i),$$

where

$$\delta_i = \begin{cases} \frac{\alpha}{\sigma_i(\sigma_i^2+\alpha)} & \text{if } \sigma_i \neq 0 \\ 0 & \text{otherwise.} \end{cases}$$

Finally, we get

$$\|A^\dagger - A^\alpha\| = \|\mathrm{diag}(\delta_i)\| = \max_{\sigma_i>0} \frac{\alpha}{\sigma_i(\sigma_i^2+\alpha)} = \frac{\alpha}{\sigma(\sigma^2+\alpha)}.$$

The forward error of the pseudo-inverse computation of A, for a given δ_0-approximate, A_0, with a fixed α', denoting the forward error of the computation $(A_0^T A_0 - \alpha' I^{n\times n})^{-1} A_0^T$ by ε_{δ_0} (which can be determined easily from the forward errors of the basic matrix operations) is just

$$\frac{\alpha'}{\sigma(\sigma^2+\alpha')} + \varepsilon_{\delta_0}.$$

In the backward error computation for $A^\dagger$, with given $\varepsilon \in \mathbf{Q}_+$, we distribute the error between $\|A^\dagger - A^\alpha\|$ and $\|A^\alpha - A_0^\alpha\|$ equally.

The first step is to find a lower bound $\sigma' \in \mathbf{Q}_+$ for σ. First of all, we allow the user to provide regularization information on the problem, namely to specify σ' for A. If no such σ' is available we try to determine one by an iterative approach, computing singular values for better and better approximates of A. Of course, if zero singular values are present, the process would never end, thus we are forced to apply arbitrary termination criteria. On the other hand, because singular values depend continuously on the entries of the matrix, nonzero singular values can be isolated within finitely many steps with the above approach. This will also allows us, for example, to override the σ' of the user if it happens to be too large.

The second step is to determine a suitably small α', so that we have

$$\frac{\alpha'}{\sigma'({\sigma'}^2+\alpha')} \le \frac{\varepsilon}{2}.$$

For instance, if $\varepsilon\sigma' < 2$, we may choose

$$\alpha' = \frac{\varepsilon {\sigma'}^3}{2-\varepsilon\sigma'}.$$

The third step is to set up an exact real matrix object $A^{\alpha'}$ for $(A^T A - \alpha' I^{n\times n})^{-1} A^T$ by means of the basic matrix operations, and ask for an $\frac{\varepsilon}{2}$-approximate: $A_\varepsilon^{\alpha'}$. According to inequality 3, by the choice of α', $A_\varepsilon^{\alpha'}$ is an ε-approximate for $A^\dagger$.

Figure 4 contains the essential part of the code for computing the Moore-Penrose pseudo-inverse of a real singular matrix, using Tikhonov regularization.

```
realMatrixPseudoinverse := proc(a::rMatrix, s::nonneg)
  local alpha, # the parameter of the approximation of A^{\dag}
        e,     # the forward error
        aa,    # the alpha approximate exact real matrix
        ea,    # -alpha*Identity-matrix(n)
        er,    # exact real matrix for ea
        i,     # index variable
        n;     # column dimension of the input matrix
  e := 1.0/s;
  alpha := e*s^3/(2.0-e*s);
  n := coldim(dump(a)[1]);
  ea := matrix(n,n,[seq(0,i=1..n*n)]);
  for i from 1 to n do ea[i,i] := -alpha od;
  er := matrix2realMatrix(ea);
  aa := realMatrixProd(realMatrixInverse(realMatrixAdd(
        realMatrixProd(Transpose(a),a),er)),Transpose(a));
  buildRealObject([epsint(aa, e/2.0), e],
    proc(eps::nonneg)
    local i,e,alpha,aa,ea,er;
    e := min(1.0/s, eps);
    alpha := e*s^3/(2.0-e*s);
    ea := matrix(n,n,[seq(0,i=1..n*n)]);
    for i from 1 to n do ea[i,i] := -alpha od;
    er := matrix2realMatrix(ea);
    aa := realMatrixProd(realMatrixInverse(realMatrixAdd(
          realMatrixProd(Transpose(a),a),er)),Transpose(a));
    epsint(aa, e/2.0)
    end,0)
end:
```

Figure 4: Computing the Moore-Penrose generalized inverse of a real matrix.

3.3 Solutions of Linear Equations

Solving linear equations can easily be achieved by the previously defined operations. Assume we are given an exact real matrix object `A`, representing a nonsingular matrix $A \in \mathbf{R}^{n \times n}$, and an exact real vector object `b`, representing $b \in \mathbf{R}^n$. The exact real vector `x` representing the solution of the linear equation $Ax = b$ can be built as

$$\texttt{realMatrixProd}(\texttt{realMatrixInverse}(\texttt{A}), \texttt{b}).$$

In a similar way we can compute the optimal solution x of $Ax = b$ for $A \in \mathbf{R}^{m \times n}, b \in \mathbf{R}^m$ in the sense given at the beginning of the previous subsection. Let `A` be an exact real matrix and `b` an exact real vector; the exact real vector `x` representing the solution of the system $Ax = b$ is defined as:

$$\texttt{realRealMatrixProd}(\texttt{realMatrixPseudoinverse}(\texttt{A}), \texttt{b}).$$

If in the above setup we want to know whether there is no solution of $Ax = b$, we can build an exact real number `n` as

$$\texttt{norm}(\texttt{realMatrixMinus}(\texttt{realMatrixProd}(\texttt{A}, \texttt{x}), \texttt{b})),$$

and apply pseudo-zero-test on it. If there is no solution we must have for a sufficiently small δ that a δ-approximate $n_\delta > \delta$. If exact solutions exist, the iterative refinement of the approximation of `n` never ends, thus arbitrary termination criteria have to be applied.

4 Polynomials with Exact Real Number Coefficients

We have already mentioned that every real number is assigned to a (unique) identifier. This identifier will be used as a symbol in algebraic computations.

A polynomial over exact real numbers will have, as coefficients, such identifiers. It will be handled as a polynomial with symbolic coefficients, until the values of these coefficients are needed. At that moment, approximations of the real numbers are computed and the computation follows with a polynomial with rational coefficients (represented as floats).

In order to be able to apply complex operations to polynomials, we have to write a polynomial arithmetic. The most complex operation is the division, which, in our case, will return a couple of polynomials – the quotient and the remainder.

For the moment, we have implemented some operations on multivariate polynomials: *addition, multiplication, evaluation,* and other operations on univariate polynomials: *root computation for square-free polynomials, division, GCD.* We describe in the following subsections two of them: root computation and GCD.

4.1 Root computation

Maple provides a good interface for computing roots of univariate polynomials, through `fsolve`. We tried to write a wrapper to this function, so that a root of a polynomial is extracted as an exact real number. However, there is no way to define a consistent matching between the roots computed with `fsolve` and the exact roots of the polynomial, in the case of non-squarefreeness.

In other words, since `fsolve` works with polynomials with rational coefficients, it can compute some approximate roots of an approximate polynomial. These roots are, hopefully, near to the exact roots of the exact polynomial. It may happen that a multiple value reported by `fsolve` corresponds to some simple exact roots, which are close to each other but do not coincide. The reverse can also happen. It is impossible to make a perfect correspondence between the values reported by `fsolve` and the exact roots, correspondence that is preserved when the precision increases. In fact, Aberth ([1]) mentions as unsolvable the problem of finding the roots and their multiplicities for a non-square-free polynomials with real coefficients.

It is known that for polynomials with rational coefficients this problem is solved by applying first the *square-free factorization* (see [2]), a method to find the square-free factors of p, i.e., to write p as:

$$p = N_1 \cdot N_2^2 \cdot \dots \cdot N_k^k,$$

where $N_1, \dots, N_k$ are square-free and pairwise relatively prime. A possible solution to root computation for non-square-free polynomials with real coefficients can be through square-free factorization of some approximate polynomials. This matter is currently a subject of research. In fact, there are some results in the literature (see, e.g., [11] or [8]) but they use mainly analytic tools; we try to get some formulas involving only the coefficients of the polynomial.

For square-free polynomials, we have implemented algorithms to extract their roots as exact real numbers.

4.2 Extracting roots of square-free univariate polynomials as exact real number objects

In the case of a square-free polynomial, we have powerful theoretical tools to handle its roots. There is a well-known method for computing iteratively good approximations of the roots, known to converge quadratically: *Newton method*. Moreover, for a given rational number, there is a formula that gives an estimate of the distance to the closest root.

Lemma 1. *Let z be a complex number, p a univariate polynomial with real coefficients, $z_1, \ldots, z_n$ all its roots, and assume that*

$$|z - z_1| \leq |z - z_2| \leq \cdots \leq |z - z_n|.$$

Then

$$|z - z_1| \leq \left| \frac{n! p(z)}{(n-k)! p^{(k)}(z)} \right|^{1/k},$$

for all k such that $p^{(k)}(z) \neq 0$.

For a proof, see [14].

The computation of a root as an exact real number takes place in two phases: First, some initial approximations of the roots are computed, together with estimates of the distances to the nearest exact roots (errors). It is required that the disks defined in this way (centers are the approximative values of the roots and radii are the errors) do not overlap. This requirement can be achieved if the computation takes place with sufficient accuracy: the theoretical argument exploits the fact that the polynomial is square-free, thus the errors of some sufficiently accurate approximations of the roots should be eventually small enough, so that the neighborhoods of the approximates contain one exact root.

Secondly, an exact real number is built for a root, by giving a pair (approximate,error) computed before as initial *epsilon*-interval, and a procedure for computing a more accurate approximate. This procedure implements Newton method.

4.3 The algorithm for computing roots of square-free polynomials

Isolation phase

1. get the dump of p;
2. set the digits according to accuracies/magnitudes of the coefficients of p;
3. call Maple function `fsolve` to compute some initial approximations of the roots of p;
4. for each root $z0$, apply formula given in Lemma 1 to get the error of z_0:
   (a) create a real number $u := p(z0)/p'(z0)$, by evaluating both p and p', as polynomials with exact real number coefficients, at z_0;
   (b) compute the distance ε_0 to the next exact root, using the dump of u;
5. check for overlaps;
6. if there are none, report the list
7. otherwise, refine p and goto step 1.

Figure 5 contains the most important part of the procedure for root isolation. The output consists in a list with elements of the form (r, e), where r is an

```
sqFreePolRootsIsol := proc(p, var::symbol)
  local q,      # rational approximation of p, with dump-ed coefficients
        pDiff,  # first derivative of p
        d,      # degree of p
        L,      # list of approximative roots
        minRootDist, # minimum distance between approximate roots
        z0,     # an element of L
        un, ud, # p and pDiff evaluated at z0
        u,      # stores un/ud
        u0,     # dump of u
        eps,    # evaluation of the error of z0
        eS,     # sum of two errors
        rDA,    # distance between two approximations
        i,j;

  d     := degree  (p,var);
  pDiff := rPolDiff(p,var);
  q     := rPolDump(p,var);

  minRootDist := 1;
  while minRootDist<infinity do
    L      := [fsolve(q,var,complex,fulldigits)];
    for i from 1 to d do
      z0  := L[i];
      un  := cPolEval(var=z0,p);  ud  := cPolEval(var=z0,pDiff);
      u   := realDiv(un,ud);      u0  := dump(u);
      eps := abs(d*(u0[1]+u0[2]));
      L   := subsop(i=[z0,eps],L);
    od;
    minRootDist := infinity;
    for i from 1 to d-1 while minRootDist=infinity do
      for j from i+1 to d while minRootDist=infinity do
        eS  := L[i][2]+L[j][2];
        rDA := abs(L[i]-L[j]);
        if eS>rDA and rDA<minRootDist then minRootDist := rDA fi;
      od
    od;
    if minRootDist<infinity then
    # refine p and increase the precision

    fi
  od;
  L
end:
```

Figure 5: Maple code of the procedure for isolating roots of a square-free polynomial.

approximation of a root and e is the radius of a disk centered at r that contains an exact root of the input polynomial p.

The procedure makes use of some predefined functions on polynomials, like `rPolDiff` (for differentiating a polynomial with exact real number coefficients), `rPolDump` (for obtaining a rational approximation by `dump`-ing the coefficients), `cPolEval` (for evaluating a polynomial with exact complex coefficients at an exact complex value).

```
sqFreePolRoot := proc(p,var,e)
  local r,pDiff,d;

  d     := degree(p,var);  pDiff := rPolDiff(p,var);
  if type(e[1],float) then
    r := buildRealObject(e,
      0,
      proc(k::ErnaInterval)
        local z,newEps,p0,pd0,u,un,ud,u0;

        while true do
          p0  := rPolRefine(p);     pd0 := rPolRefine(pDiff);
          un0 := evalf(subs(var=k[1],p0));
          ud0 := evalf(subs(var=k[1],pd0));
          z   := k[1]-un0/ud0;
          un  := rPolEval(var=z,p); ud := rPolEval(var=z,pDiff);
          u   := realDiv(un,ud);    u0 := epsint(u,Float(1,-f));
          newEps := abs(d*(u0+dump(u)[2]));
          if newEps>k[2]/10 then
          % refine p, get new values for p0 and pd0
             ...
          else
            break
          fi
        od;
        [z,newEps]
      end
    )
  else
  % similarly, use functions for exact complex instead
    ...
  fi;
  r
end:
```

Figure 6: Constructor for a root of a square-free polynomial.

Iterator of a particular exact root, based on Newton's method, having an approximation z_0, and an estimate of its error, ε_0

1. refine the coefficients of p and p', getting two polynomials with rational coefficients, p_0 and p'_0;
2. set the digits according to accuracies/magnitudes of the coefficients of p_0 and p'_0;
3. compute $z := z_0 - \frac{p_0}{p'_0}$;
4. create a real number $q := \frac{p(z)}{p'(z)}$;
5. compute the distance ε to the next exact root, using the dump of q;
6. if ε is not smaller than the accuracy of the old root, refine p and goto step 1;
7. report (z, ε)

Figure 6 contains the important parts of the code for the constructor of a root as an exact real (or complex) number. Notice that only the `iterate` member function is defined.

4.4 GCD computation (Corless & al algorithm)

We have implemented one of the algorithms given in Corless & al ([4]) for computing the GCD of two polynomials with real coefficients. The procedure accepts as input two polynomials with real coefficients, and outputs a polynomial with real coefficients whose degree is sensitive to the precision of the computation.

Due to theoretical reasons that will be explained later, the output of the GCD algorithm is an exact real vector, storing the coefficients of the GCD. Naturally, this brings in the necessity to define polynomial arithmetic that can handle both representations. However, this can be done, with some extra effort, during the coding: there are no theoretical incompatibilities between Maple representation of univariate polynomials and the vector of coefficients representation.

Let p, q denote the initial polynomials, and d their GCD. We will use p_0, q_0, d_0 for some approximations of these polynomials.

The algorithm makes use of some properties of the Sylvester matrix (see [17] for a definition of the Sylvester matrix) of two univariate polynomials.

Let

$$\begin{aligned} p(x) &= x^n + a_{n-1}x^{n-1} + \cdots + a_0, \\ q(x) &= x^m + b_{m-1}x^{m-1} + \cdots + b_0, \end{aligned}$$

be two polynomials with exact real entries, and let

$$r(x) = x^k + c_{n-1}x^{k-1} + \cdots + c_0$$

be their greatest common divisor, whose degree is known in advance. Then there exist two polynomials u and v, of degrees at most $m-k-1$ and $n-k-1$ respectively, such that:

$$pu + qv = r.$$

These two polynomials can be found by solving a system of linear equations.

From the equality $pu+qv=r$ we extract $m+n-2k$ equations, by identifying the $m+n-k$ coefficients of the left-hand side to 0. More formally, let us denote:

$$\begin{aligned} u(x) &= u_{m-k-1}x^{m-k-1} + u_{m-k-2}x^{m-k-2} + \cdots + u_0, \\ v(x) &= v_{n-k-1}x^{n-k-1} + v_{n-k-2}x^{n-k-2} + \cdots + v_0. \end{aligned}$$

Let S^T be the transposed Sylvester matrix of p and q, and let M be a sub-matrix of S^T, containing the first $m-k$ columns of coefficients of p, the first $n-k$ columns of coefficients of q, and the first $m+n-2k-1$ lines:

$$M = \left.\underbrace{\begin{pmatrix} 1 & 0 & \dots & 1 & 0 & \dots \\ a_{n-1} & 1 & \ddots & b_{m-1} & 1 & \ddots \\ a_{n-2} & a_{n-1} & & b_{m-2} & b_{m-1} & \\ \vdots & \vdots & & \vdots & \vdots & \\ a_1 & a_2 & & & & \\ a_0 & a_1 & & b_1 & b_2 & \\ 0 & a_0 & & b_0 & b_1 & \\ 0 & 0 & & 0 & b_0 & \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \end{pmatrix}}_{m+n-2k \text{ columns}}\right\} m+n-2k \text{ lines}$$

A simple computation shows that the high order coefficients of $pu+qv$ can be obtained by multiplying M with the vector:

$$V = \begin{pmatrix} u_{m-k-1} \\ u_{m-k-2} \\ \vdots \\ u_0 \\ v_{n-k-1} \\ v_{n-k-2} \\ \vdots \\ v_0 \end{pmatrix}.$$

Identifying the entries of MV with zero, we obtain $m+n-2k$ equations in the equally many coefficients of u and v.

After solving this system and constructing u and v, the GCD is obtained by a simple arithmetic operation with polynomials.

Practically, first of all the Sylvester matrix is generated, as an exact real matrix. Using this matrix, for any given ε, one finds an approximation of accuracy ε of the GCD of the initial polynomials, with the coefficients in a vector:

1. an approximation S of the Sylvester matrix is generated;
2. the singular values of S are computed;
3. ε is considered as a threshold that sets the approximate rank of the Sylvester matrix (see [4]).

   The difference between the dimension of the matrix and this rank is the degree l of the approximate CGD;
4. assuming that $l \notin \{0, \mathrm{degree}(p), \mathrm{degree}(q)\}$, the algorithm proceeds by solving the equation
$$p_0 \cdot u + q_0 \cdot v = d_0;$$
   more exactly, it uses a square sub-matrix of S as the matrix, and a combination of the coefficients of p and q as the right-hand side, of a linear system of equations, and gets the coefficients of u and v by solving this system;
5. computes the coefficients of d_0, by substituting the values computed in the previous step in the formula
$$p_0 \cdot u + q_0 \cdot v = d_0.$$

The GCD is generated as an (exact real) vector of its coefficients.

Geometrically speaking, every approximation of accuracy ε of the GCD represents a point, in the space of polynomials of degree l, situated as close as possible to the real GCD. The dimension of the vector, which gives the degree of the GCD, can vary as the accuracy increases. This is normal, because the accuracy constitutes a threshold for the singular values of the Sylvester matrix, and the degree of the GCD is set by counting the singular values that lie beneath this threshold.

We do not insert here the code, because it is quite involved and requires many sub-algorithms (for generating real Sylvester matrix, for computing the right-hand side of the system mentioned in step 4, etc.).

5 Further Research

Roots of non-squarefree polynomials

The problem of extracting roots of univariate polynomials is, for the moment, incompletely solved: it remains to treat the case of non-square free polynomials. We plan to use the method suggested by Hribernig and Stetter in [8], which basically consists in computing an approximate squarefree factorization and then the roots of the squarefree factors. We are currently working on the problem of finding the distance from a point in the complex plane to the k'th nearest root of a polynomial. Also a topic of research is to prove the convergence of the squarefree factorizations computed only approximately to the exact squarefree factorization.

Parameterization of curves and surfaces

The symbolic computation of parametric representations of curves and surfaces given implicitly often raise the problem of handling roots of polynomials with high degrees. Even for medium size problems, the size of the equations (and, therefore, of the symbolic representation of the roots) tend to explode.

The alternative computation with floating-point produces meaningless results, due to the rapidly accumulating error bounds.

It is to be expected that, using exact real numbers, we may get a parameterization in reasonably short time, and the curve or surface obtained in this way can be made arbitrarily close to the one given implicitly.

References

[1] Aberth, O. *Precise Numerical Methods using C++.* Academic Press, 525 B Street, Suite 1900, San Diego, Ca, 1998.

[2] Akritas, A. *Elements of Computer Algebra With Applications.* John Wiley & Sons, 1989.

[3] Boehm, H., and Cartwright, R. Exact real arithmetic, formulating real numbers as functions. In *Research Topics in Functional Programming*, D. Turner, Ed. Addison-Wesley, 1990, pp. 43–64.

[4] Corless, R., Gianni, P., Trager, B., and Watt, S. The singular value decomposition for polynomial systems. In *Proceedings of ISSAC'95* (1995), ACM.

[5] Escardó, M. Exact numerical computation. In *ISSAC'00* (2000), ACM Press. tutorial.

[6] Gianantonio, P. Real number computability and domain theory. In *Proc. Math. Found. Comp. Sci. '93* (1993), Springer, pp. 413–422.

[7] Higham, N. *Accuracy and stability of numerical algorithms.* SIAM, 1996.

[8] Hribernig, V., and Stetter, H. Detection and validation of clusters of polynomials zeros. *jsc 24* (1997), 667–681.

[9] Menissier-Morain, V. Arbitrary precision real arithmetic: design and algorithms. *Journal of Symbolic Computation* (2000). to appear.

[10] Muller, J.-M. *Arithmétique des Ordinateurs.* Etudes et recherches en informatique. Masson, 1989.

[11] Neumayer, A. An existence test for root clusters and multiple roots. *ZAMM 68* (1988), 257–257.

[12] Roy, M.-F., and Szpirglas, A. Complexity of computation of real algebraic numbers. *Journal of Symbolic Computation 10* (1990), 39–51.

[13] Shewchuk, J. R. Adaptive precision floating point arithmetic and fast robust geometric predicates. *Discr. Comp. Geom. 18* (1997), 305–363.

[14] Strzeboński, A. Computing in the field of complex algebraic numbers. *Journal of Symbolic Computation 24* (1997), 647–656.

[15] TURING, A. M. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings, London Mathematical Society,* (1936), pp. 230–265. Published as Proceedings, London Mathematical Society,, volume 2, number 42.

[16] WEIHRAUCH, K. A foundation of computable analysis. *Bulletin of the European Association for Theoretical Computer Science 57* (1995), 167–182.

[17] WINKLER, F. *Polynomial Algorithms in Computer Algebra.* Springer Verlag, 1996.