An Overview of Computation at the Symbolic−Numeric Boundary

Daniel Lichtblau

Wolfram Research, Inc.
100 Trade Centre Dr.
Champaign IL 61820
danl@wolfram.com

In the general world of scientific computation one frequently encounters "numeric computation" and "symbolic computation" as separate if loosely related entities. Sometimes one even hears about both used within one program. What is less often noted is that the two types of computation have begun to cross−pollinate in the past 15 years or so, with each invading the other’s historic turf. In this talk I will present a few examples of such interaction. My goal is to demonstrate that what is now called "hybrid symbolic/numeric computation" is a fruitful and active area of research and development in the current field of scientific computation.

A brief and mostly incorrect history of symbolic computation (including an irrelevant forecast of dubious validity pertaining to future trends)

~35 years ago:

Scientific computation mostly sophisticated number crunching (generally performed between mastodon hunts).

Algorithms entirely numeric.

Symbolic computation began to make its entry into the scientific world. Algorithmic development also heavily mathematical but very different from numerical stuff.

~20 years ago:

Emerging emphasis on having numeric capabilities within the symbolic programs.

Local history: *Mathematica* appears in 1988. Regarded as a powerful blend of symbolic, numeric, and graphical capabilities. Reason: these parts get along as an integrated system.

Today:

One requires much more in a general purpose technical program.

(i) editable user interface

(ii) ability to communicate with other programs

(iii) ability to import and export various flavors of data

(iv) ability to interact with the web

We will ignore all this.

What we discuss began as a small trend in both the symbolic and numeric computation communities (which by and large did not overlap) to "reduce" problems in one domain to computations in the other. Gradually it became clear that both fields grew stronger when each made use of the other's functionality. In the following I will give some examples of specific ways in which this is done.

What is meant by "symbolic" computation

Numeric computation is what happens when you have a bunch of machine numbers and you tell your friend's computer "Do something useful with them, and hurry up because my homework is due an hour ago".

Symbolic computation: Start with rational arithmetic (a major component is "exact" computation). What makes rational arithmetic different from floating point?

Must parse into a numerator and denominator.

Then must find and remove the greatest common divisor.

Symbolic computation is far more general than simple manipulation of exact numbers such as $\frac{7}{3}$. Must work with polyno−mials and all manner of other functions. Requires an ability to

(i) do arithmetic with such things

(ii) put them into a "canonical" form

(iii) use them in a calculus framework

(iv) do linear algebra with such expressions

etc.

For purposes of this talk "symbolic methods" may indicate any programming that is mathematical in nature but not strictly a matter of number crunching computation.

An aside into what we do at Wolfram Research, Incorporated

There are many departments at Wolfram Research. As with any software house we have Sales, Marketing, Customer Service, Design, Shipping/Receiving, Administrative, Human Resources, Finance, and Library departments. Then there are the groups that handle various technical aspects of *Mathematica*. Here is a quick run downon them.

(1) Software Quality Assurance. Design and perform testing.

(2) Documentation and Publications. Technical writing editing, and translating. Also do The *Mathematica* Journal and The *Mathematica* Book.

(3) Release Engineering. Building *Mathematica* and installers on various machines and operating systems.

(4) Technical Support. Handle questions from users.

(5) Applications. Commercial add−ons such as "Control Systems Professional".

(6) Project Management. Keep tabs on timetables for various projects. Requires a blend of the skills from each of the above groups. Also: sense of humor; Black Belt in some martial arts discipline is strongly advised.

(7) *Mathematica* Programming. Technical programming for standard (noncommercial) add−on packages and related work.

(8) Information Technology. Umbrella for three subgroups.

(i) Systems Administration

(ii) Web Programming

(iii) Management Information Systems

(9) Front−end. Design, code, and port *Mathematica* user interface. Includes the type−setting code responsible for this particular document.

(10) Kernel Development. My group. We debug and extend *Mathematica* core functionality. Includes the programming language, pattern matching, mathematical functionality, and more.

What are the credentials needed to work in these various groups? No hard rules but here are some basic facts.

Several groups have people with doctoral degrees (Applications group exclusively so). Kernel Development mostly at that level, in particular math and applied math.

Front−end, Documentation and Publications, Release Engineering, and Software Quality Assurance groups have a mix of educational backgrounds and degrees.

The *Mathematica* Programming group mostly master’s degrees in various fields.

No idea about folks in Project Management have, frankly too frightened to ask.

Information Technology folks are even scarier because they know how to mess with our machines.

More about my department. We require a lot of mathematicians. If you don’t want to go to graduate school for several years, pick another department.

Brings to mind a story told by a retiring executive at a Massachusetts software house where I worked many years ago. A prospective applicant (who did not get a job offer) had asked "Would you really hire someone with a math Ph.D?" and he replied "Yes, they are almost as smart as CS Bachelors and we don’t have to pay them as much."

Domains of computation

Machine floating point arithmetic

Computations using double precision arithmetic and system libraries of compiled code

In[22]:=
```
expr = 2.0 + Sin[33.]
```

Out[22]=
```
2.99991
```

High precision approximate arithmetic

Again approximate arithmetic but implemented in software. Frequently utilized at higher than hardware precision.

In[23]:=
```
exprbig = N[2, 50] + Sin[N[33, 100]]
```

Out[23]=
```
2.9999118601072671457280843767078283143509446164918
```

In[24]:=
```
Precision[exprbig]
```

Out[24]=
```
50
```

Note adjusted precision of the result based on the lowest precision of the operands. Unlike machine arithmetic there is an active precision tracking mechanism. Not so with all implementations of extended−precision but we regard it as an asset. Originally developed at WRI by Jerry Keiper, who died tragically in 1995.

Exact arithmetic

Represent quantities exactly.

In[25]:=
```
exprexact = 2 + Sin[33]
```

Out[25]=
```
2 + Sin[33]
```

Well, what did you expect? But we can "evaluate numerically" and to whatever precision we desire.

In[26]:=
```
N[exprexact, 200]
```

Out[26]=
```
2.99991186010726714572808437670782831435094461649184588392103189409924
 88727735085729616861132641349562723778721377655482194907884340911876
 96693395395844845409042092646491595240364937429485014534021
```

Symbolic results

We now deal with expressions that contain nonnumeric entities.

In[27]:=
```
exprsymbolic = 2 + Sin[33 * x]
```

Out[27]=
```
2 + Sin[33 x]
```

Again we can obtain numeric values if we give numeric values to the symbols.

In[28]:=
```
Table[exprsymbolic, {x, 0., 3.}]
```

Out[28]=
```
{2., 2.99991, 1.97345, 1.00079}
```

But it’s not really a hierarchy

Often thought that "exact" results are somehow more desirable than approximate and symbolic are better than exact because they are more general. Historically analogous to food chain wherein mastodon eats bushes and the caveman eats mastodon. In fact this computational hierarchy developed when cave dwellers were eating mastodons.

We have come a long way in the intervening decades.

Consumption of mastodons has plummetted (due in no small part to scarcity).

Food web supplants the chain.

Modes of computation no longer live in separate worlds and they now borrow from one another.

The hierarchy of generality is not always appropriate: remember that on a bad day the caveman gets trampled.

Hybrid methods: numeric computation invades the symbolic domain

The ubiquitous multiplication of integers

Our first example involves every day multiplication of integers. How do you multiply integers? Basically you multiply every digit by every other digit, keeping track of place value when you sum the results. Suppose you are a computer? You can do it the same way, but use base 2. So you form aAround n^2 products if your inputs each have around n bits. Does *Mathematica* use an $O(n^2)$ method to multiply her integers? To test we'll need some honking big integers. We will start with a pair of random integers with around a million bits.

```
m20 = RandomInteger[2^2^20];
n20 = RandomInteger[2^2^20];
```

In[12]:=
```
Timing[prod = m20 * n20;]
```

Out[12]=
```
{0.24 Second, Null}
```

Now we double the size to two million bits and see if it takes four times as long to get the product.

```
m21 = RandomInteger[2^2^21];
n21 = RandomInteger[2^2^21];
```

In[15]:=
```
Timing[prod = m21 * n21;]
```

Out[15]=
```
{0.55 Second, Null}
```

It took a bit more than twice as long. The secret is that *Mathematica* is using a fast Fourier transform (FFT) behind the scenes. This is done in floating point arithmetic, our first example of using a numerical method to do something that is exact.

Actually what we just saw is a more subtle example of a hybrid exact−numeric algorithm. I used a development kernel of *Mathematica* and in that version we have changed the way we do integer arithmetic. Instead of using the FFT there is something called the number theory transform (NTT). This uses exact integer arithmetic modulo a few primes. So we have an example where exact computation has appropriated a numerical algorithm into an exact setting.

Let’s see how long it would take to print out these numbers by breaking digits into a list.

In[22]:=
```
Timing[digits = IntegerDigits[m21];]
```

Out[22]=
```
{5.88 Second, Null}
```

It took significantly longer than to multiply a pair. The reason is that one must do a lot of division by large powers of 10 in order to get the (decimal) digits. So we

(i) form those powers (large multiplications)

(ii) do divisions that are roughly two or three times as costly

Since numbers are stored in a binary format, if the base of interest is a power of 2 we can do a lot better.

In[23]:=
```
Timing[digits = IntegerDigits[m21, 16];]
```

Out[23]=
```
{0.01 Second, Null}
```

I will point out that division is converted to multiplication problems using Newton iterations. This tactic appears in arithmetic as a numeric method but a variation is used in polynomial division and that takes an exact approach. Now our development code will translate polynomial to integer arithmetic which uses the number theoretic transform method discussed above. So it is difficult to sort out exactly whether our upcoming polynomial algebra code is exact or numeric in spirit, and I think it now also qualifies as a hybrid.

Integer lattice manipulation

One sometimes needs to work with integer lattices. One important problem is to find "small" vectors in a given lattice. In a sense this is tantamount to finding a nearly orthogonal generating set. Example: {1, 0, 104} and {0, 1, 222} are nearly parallel to the z axis. We find a smaller pair with a much larger angle between them that span the same lattice of integer coordinates in $\mathbb{R}^3$.

In[1]:=
```
LatticeReduce[{{1, 0, 104}, {0, 1, 222}}]
```

Out[1]=
```
{{-2, 1, 14}, {15, -7, 6}}
```

Internal technology uses Gram−Schmidt orthogonalization and more. The original algorithm worked primarily with rational arithmetic but it can be done (if carefully!) using approximate arithmetic. Hence we have numeric computation working for us behind the scenes in an exact computation.

This exact computation can be further blended with numeric approximation for the exact algebra application of factoring a polynomial.

In[3]:=
```
poly = 42 - 13 x - 20 x^2 + 9 x^3 + 13 x^4 + 2 x^5 - x^6 + 2 x^7 + x^8;
```

Start with a numeric root−finder to approximate a particular root.

In[4]:=
```
rt = x /. FindRoot[poly == 0, {x, 2}, MaxIterations → 100,
    WorkingPrecision → 40]
```

Out[4]=
```
-2.086745339882666383582362204395378742708
```

(i) take powers of this root

(ii) multiply them all by some large number

(iii) round off to get integers

(iv) try to find a small integer combination

This last step is known as finding an integer relation between real numbers. May be done using lattice reduction.

To record the integer relations we attach a column vector of these powers to an identity matrix. Idea is similar to "place−value" arithmetic except that successive powers are in the approximate root; they correspond to powers in a polynomial variable. When done each column in the matrix prior to the last will record the coefficient of the corresponding power of that variable that is needed to make the last column approximately zero. Also similar to standard method for matrix inversion where we append an identity matrix that records row operations.

In[13]:=
```
powers = Round[10^40 * Table[rt^j, {j, 0, 7}]];
lattice = Transpose[Append[IdentityMatrix[8], powers]]
LatticeReduce[lattice]
```

Out[14]=
```
{{1, 0, 0, 0, 0, 0, 0, 0, 10 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000},
 {0, 1, 0, 0, 0, 0, 0, 0, -20 867 453 398 826 663 835 823 622 043 953 787 427 081},
 {0, 0, 1, 0, 0, 0, 0, 0, 43 545 061 135 220 248 454 440 608 499 700 095 794 098},
 {0, 0, 0, 1, 0, 0, 0, 0, -90 867 453 398 826 663 835 823 622 043 953 787 427 081},
 {0, 0, 0, 0, 1, 0, 0, 0, 189 617 234 927 006 895 305 205 962 807 376 607 783 662},
 {0, 0, 0, 0, 0, 1, 0, 0, -395 682 881 345 368 403 016 907 881 541 854 457 985 763},
 {0, 0, 0, 0, 0, 0, 1, 0, 825 689 408 718 793 542 155 971 317 115 053 119 773 226},
 {0, 0, 0, 0, 0, 0, 0, 1,
  -1 723 003 525 834 416 670 153 349 621 193 490 712 471 397}}
```

Out[15]=
```
{{7, -1, 0, 1, 0, 0, 0, 0, 0}, {0, 7, -1, 0, 1, 0, 0, 0, -3},
 {0, 0, 7, -1, 0, 1, 0, 0, 4}, {0, 0, 0, 7, -1, 0, 1, 0, -3},
 {0, 0, 0, 0, 7, -1, 0, 1, 0}, {3 775 678 231, -580 501 866 972,
  1 066 579 054 143, -606 931 614 580, -119 441 513 761, -1 073 606 811 312,
  -1 120 453 848 415, -237 516 214 993, -1 749 844 545 569},
 {75 133 458 513, -210 035 938 933, 71 000 322 194, -735 970 148 531,
  -78 595 481 320, 926 824 097 508, 3 453 348 182 341, 1 476 992 466 742,
  -539 949 125 356}, {-94 796 544 126, -1 250 665 041 382,
  295 437 637 752, -587 089 232 479, 1 430 521 362 673, 7 504 276 056 257,
  -2 079 425 574 756, -2 509 373 482 425, -2 539 857 188 259}}
```

First vector is $\{7, -1, 0, 1, 0, 0, 0, 0, 0\}$, corresponds to polynomial $x^3 - x + 7$. Is it really a factor?

In[16]:=
```
Factor[poly]
```

Out[16]=
$$\left(7 - x + x^3\right)\left(6 - x - 3\,x^2 + 2\,x^4 + x^5\right)$$

We used a blend of exact and approximate technologies to attack the symbolic algebra problem of factoring a polynomial. Method appeared in the early 1980's.

Symbolic computation stakes claims on numeric turf

Solving polynomial systems of equations

Important for computational mathematics: find simultaneous solutions to a system of polynomial equations. Method used by *Mathematica* and many other programs involves something known as Gröbner bases. What are those? I'm not telling, as that would be a talk all by itself. Suffice it to say that they generalize the Gaussian elimination method for solving systems of linear equations.

This algorithmic technique of polynomial manipulation was first developed in the 1960's by Bruno Buchberger (he named the resulting polynomial set for W. Gröbner, his thesis advisor).Buchberger was attacking, algorithmically, something called the "ideal membership" problem. Later (~1978) was it seen how they might be used to solve polynomial systems.

Drawback: it seems to require exact arithmetic. This is trouble as often intermediate coefficients grow to enormous size (general phenomenon called "intermediate expression swell"). Around late 1980's a new method appears. Uses Gröbner bases but ones that are easier to compute. Reduces system solving to an eigenspace problem which is tackled with numeric software. So we had then a hybrid algorithm with a symbolic computation in the first step. Still not a good situation: even with easier Gröbner basis step one often sees problems for which it was not feasible.

More hybridization follows. By the mid−1990's experimentation was under way to compute Gröbner bases using approxi−mate arithmetic (I had working code to do that by late 1993, so in this one instance at least I was ahead of the field). Worked because it made good use of approximate arithmetic at moderate precision, typically one or two hundred digits. At this point we had another sort of hybrid in that an algorithm that had required exact arithmetic was now working with approximate arithmetic. Still not perfect insofar as even approximate Gröbner bases may be expensive to compute but this performs quite well in practice.

Today there are several approaches to solving systems of polynomial equations. Some fare even better than the one described above on particularly large or otherwise hard problems. But all share the attribute that they use a hybrid of exact and numeric methods.

A brief look at other examples

Algebra

■ Matrix factorization and linear equations

A frequent need in numeric computation is to solve a aytem of linear equations. Primarily into two classes of methods.

(i) direct based of Gaussian elimination

(ii) iterative methods based on subspace projections.

Both developed primarily to do inexact numeric computation.

Direct methods also used heavily in symbolic algebra. One application under active research is in solving linear Diophantine systems. Given linear equations with integer coefficients, find solution vectors with integer components (row reduction will give solutions with rational values). Recent approach: first solve over rationals. Suffers from growth which makes the arithmetic quite slow (computes gcds!). Alternative:

(i) find solutions in prime fields

(ii) somehow "lift" to rational solutions

(iii) use these to generate integer solutions

The steps of solving in a prime field and of lifting can be done using something called the LU factorization of a matrix. It is an example where exact computation in linear algebra has borrowed from a numeric approach.

Iterative methods have been adapted to perform Gaussian elimination of a large but sparse matrix of elements that lie in the field of integers modulo 2. Application is integer factorization. This is another example where a linear algebra method developed for numeric computation has been deployed in an exact setting.

■ Polynomial interpolation

Polynomial interpolation often done by working with Vandermonde matrices. Originally for numeric purposes. But applications are not entirely numerical. ~25 years ago: significant body of symbolic algebra developed around interpolation of "sparse" multivariate polynomials. Principal technology used in *Mathematica* and elsewhere to do polynomial gcd extraction. This is among the most important of polynomial operations.

Related example: solving symbolic linear systems when the matrix is comprised of polynomials in one variable. A useful approach is to interpolate the result using exact solutions for various specializations of that variable.

Geometry and combinatorial math

■ Exact linear programming

To solve a linear programming problem exactly one can use basic algorithms

Too slow when using rational arithmetic. Faster tactic: first use machine arithmetic to find an approximate solution. Use this to find the exact rational solution.

■ Discrete optimization

Genetic methods used in discrete optimization for more than two decades. Drawback to application in the continuous realm: difficulty of sensibly encoding continuous values as "genes". Method of Differential Evolution due to Price and Storn ~1995 addressed this so now a method from the discrete realm comes to continuous problems.

But the story continues. Around 1999 we started playing with Differential Evolution applied to discrete optimization problems.This method hybridized from the discrete world (where arithmetic is exact) to handle continuous problems (where arithmetic is approximate) has now returned to discrete problems.

■ Reconstruction of geometric objects

Problem from computational geometry: reconstruct an object containing a given set of points. Example: find cylinders that contain a given set of five or more points in space. Uses hybrid methods of solving numeric systems, numeric root−finding, and exact methods for forming a parametric or implicit representation of the object in question. Numeric methods alone are simply not adequate for these tasks.

■ Other computational geometry

Frequent need in computational geometry: determine sign of an inequality. Arises in finding whether a point lies inside a given circle, in front of or behind a given plane, etc. Degenerate case: something evaluates to zero, that is, it seems to lie right on the circle or plane. In some applications (Delaunay triangulation, convex hull) such degeneracy, when not recog−nized, is very bad. High precision, exact or even symbolic methods may be employed to recognize and/or work around it.

Calculus

■ Numeric optimization

Common task: find an extreme value of a multivariate function near a given starting vector. Common methods such as Newton's or steepest descent use gradients and maybe Hessians. These can be approximated numerically. But often will do better using analytical forms which need symbolic computation. More extreme example is to work with a function defined by a program and produce new procedural code to evaluate gradients or Hessians. Called "automatic differentia−tion".

■ Singularity analysis in numeric integration

Trauma for numeric integration (quadrature) methods: handling of integrable singularities. Adaptive methods often go crazy near such points. Symbolic analysis of a singularity can save a lot of time and trouble.

■ Miscellaneous other uses of symbolic methods in numeric calculus

Numeric ODE solvers may use symbolic algebraic solvers to obtain a canonical form prior to use of purely numeric methods.

Numeric PDE solvers sometimes use hybrid symbolic/numeric technique for spatial discretization.

Numeric DAE solvers may use a symbolic computation method to do something known as index reduction.

Revisiting the computational "hierarchy"

We will now look at an example of exact computation that is not so useful as one might wish. We begin with a pair of equations and graph some points of intersection.

In[1]:=
```
eqns = {4 x^4 - x y^5 + 3 x^3 y - 9 x^2 y + 4 y^2 + 2 y - 5,
    -y^3 + 3 x^3 y^2 - 2 x^2 y - x^2 - 2 x y + 4 x + 12 y^2 - 4 y + 5};
```

COMPATIBILITY ISSUE

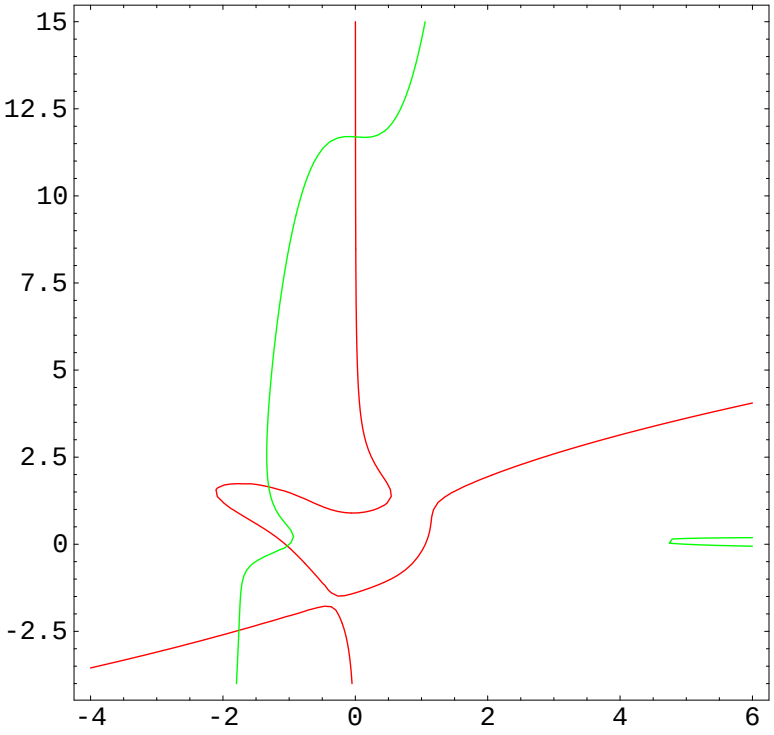
The value of PlotPoints given in this evaluation may cause delay in displaying the graphic. Reducing the value of the PlotPoints option may help.
Show is terminated by a semicolon. Graphics output will be suppressed if it is not removed.

SUGGESTED VERSION

```
plot1 = ContourPlot[eqns[[1]], {x, -4, 6},
   {y, -4, 15}, PlotPoints → 100, AspectRatio → 1,
   Contours → {0}, ContourShading → False,
   ContourStyle → {{Thickness[0.002'], Hue[0]}},
   DisplayFunction → Identity];
plot2 = ContourPlot[eqns[[2]], {x, -4, 6}, {y, -4, 15},
   PlotPoints → 100, AspectRatio → 1,
   Contours → {0}, ContourShading → False,
   ContourStyle → {{Thickness[0.002'], RGBColor[0, 1, 0]}},
   DisplayFunction → Identity];
Show[{plot1, plot2}, DisplayFunction → $DisplayFunction]
```

Replace with Suggested Version | Discard Suggested Version

```
plot1 = ContourPlot[eqns[[1]], {x, -4, 6}, {y, -4, 15}, PlotPoints → 100,
    AspectRatio → 1, Contours → {0}, ContourShading → False,
    ContourStyle → {{Thickness[0.002], Hue[0]}},
    DisplayFunction → Identity];
plot2 = ContourPlot[eqns[[2]], {x, -4, 6}, {y, -4, 15}, PlotPoints → 100,
    AspectRatio → 1, Contours → {0}, ContourShading → False,
    ContourStyle → {{Thickness[0.002], RGBColor[0, 1, 0]}},
    DisplayFunction → Identity];
Show[{plot1, plot2}, DisplayFunction → $DisplayFunction];
```



Say we are interested in the root near {0, 11}. We can use a numeric root−finder with no trouble.

In[40]:=
```
Timing[rt = FindRoot[Evaluate[Thread[eqns == 0]], {x, 0}, {y, 11}]]
eqns /. rt
```

Out[40]=
```
{0. Second, {x → 0.00258531, y → 11.6941}}
```

Out[41]=
$$\{-1.47963 \times 10^{-9},\ -6.36646 \times 10^{-12}\}$$

Can even find ALL solutions though takes longer. Also check residuals to assess quality.

In[44]:=
```
Timing[nsolns = NSolve[eqns == 0, {x, y}];]
Max[Abs[eqns /. nsolns]]
```

Out[44]= {0.62 Second, Null}

Out[45]= 2.55125×10^{-9}

Exact solutions? We can get those too.

■ Wait, don’t touch that solver!

In[46]:=
```
Timing[exactsolns = Solve[eqns == {0, 0}, {x, y}];]
LeafCount[exactsolns]
```

Out[46]= {6.82 Second, Null}

Out[47]= 87 673

Too late. A lot slower, with result quite large. Now suppose we want numeric values.

In[50]:=
```
nexactsolns = N[exactsolns];
Max[Abs[eqns /. nexactsolns]]
```

Out[51]= 3.63321×10^{23}

Quite inaccurate. Exact solutions involve algebraic combinations of complicated objects and contain large coefficients. Numerical evaluation using machine arithmetic is problematic due to cancellation error. No warning from evaluation at machine precision that we were getting gibberish. Next try higher precsion.

In[52]:=
```
Timing[nexactsolns25 = N[exactsolns , 25];]
Max[Abs[eqns /. nexactsolns25]]
```

Out[52]= {0.33 Second, Null}

Out[53]= $0. \times 10^{-21}$

Niw time to evaluate is not small.

We were better off avoiding the exact computation.

Another interesting hybridization involving solving of systems of equations. Gröbner basis/eigensystem method was a hybrid where exact computation methods are used in a numeric task. Can instead do all in infinite precision.

In[56]:=
```
Timing[
 exactsolnsagain = NSolve[eqns == {0, 0}, {x, y},
    WorkingPrecision → Infinity];]
LeafCount[exactsolnsagain]
```

Out[56]=
```
{0.91 Second, Null}
```

Out[57]=
```
88 729
```

Big, yes. But at least it did not take so long to compute.

Related misuses of exact computation

Similar problems in integration, solving of differential equations, and elsewhere. Methods appropriate to exact domain can give results where numerical evaluation is not stable.

Not safe to assume that an "exact" result will give good numerical values

Result computed numerically from the outset may be obtained faster and more reliably

Not to say that exact results are useless or unsafe but often one is better served by numeric or hybrid methods.

Two ways in which machine and higher precision approximate arithmetic interact.

(i) Similar to use of numerical approximation in exact linear programming. For many high precision tasks, e.g. root−finding and optimization, one wants a good starting point. Find using fast machine arithmetic. Further iterations to proceed at higher precision. May be slow but few needed if we began from a good initial point.

(ii) Machine arithmetic can profit from having a higher precision neighbor to call upon. One may encounter a machine overflow or at intermediate steps, even when result is a perfectly reasonable machine value. A program such as *Mathemat−ica* will then slip into extended precision arithmetic where far larger exponents may be represented. Similar to hardware processors that handle machine numbers with larger exponents than regular double precision. Has unfortunate ability to create trouble if one cannot maintain tight control over when a value is in a register. An attempt to store an intermediate result too large for ordinary memory will give an overflow that might have been avoided by delaying the store operation. Are such overflows carefully avoided in hardware arithmetic? Is the IEEE standard up to speed in this area? As best I can tell you are nowadays at the mercy of the compiler writer. Come to think of it, I used to be a compiler writer. Be very afraid.

Summary

Techniques developed for numeric computation have been adapted for exact computations. Reverse as happened as well. Important lessons are

(i) Do not assume that exact computations are needed to find exact results. Approximate methods (when they exist) tend to be much faster in practice.

(ii) Not all numeric problems must be tackled by exclusively numeric programs. Sometimes symbolic methods will reduce the difficulty.

(iii) If you think you need an exact result, determine whether this is really so. Often you are better off with a numeric computation done according by high quality numeric algorithms rather than using the result of a symbolic computation that does not have good algorithms for numeric evaluation.

My hope is that the types of example we have discussed, howsoever briefly, will give you some insight into why the world of hybrid symbolic−numeric computation is recently coming into prominence. Should this current popularity prove to be more than just a brief trend, you may some day find yourself working this realm.