Exploiting Fast Hardware Floating Point in High Precision Computation

Keith O. Geddes
Wei Wei Zheng

Technical Report CS-2002-41

School of Computer Science
University of Waterloo

December 2002

Abstract

We present an iterative refinement method based on a linear Newton iteration for solving a particular group of high precision computation problems. Our method generates an initial solution at hardware floating point precision using a traditional method and then repeatedly refines this solution to higher precision, exploiting hardware floating point computation in each iteration. This is in contrast to direct solution of the high precision problem completely in software floating point. Theoretical cost analysis, as well as experimental evidence, shows a significant reduction in computational cost is achieved by the iterative refinement method on this group of problems.

Contents

List of Tables

1 Introduction

In symbolic and numeric computation, a high precision solution is often desired. The traditional way to compute high precision solutions is to directly carry out the computation in a multiprecision software floating point environment which can be quite time consuming. In contrast, hardware floating point computation is much faster than the software equivalent. Based on this realization, we construct a new method that exploits the hardware floating point environment for most of the computations.

The iterative method presented here is based on Newton's iteration but in a linearly converging variant rather than the quadratic Newton's iteration commonly used in traditional numerical computation. As is well-known, the quadratic Newton's iteration becomes linear if the "derivative information" is held constant. This is the basic form of our iterative method and it corresponds to the Hensel iteration [6] known in computer algebra. As is the case in the algebraic Hensel setting, one finds that a linearly converging variant of Newton's iteration can be advantageous when the computations in the "base ring" are much more efficient than the computations which would result if the update for each iteration were computed in a larger ring.

For the iterative method of this paper, an initial solution at hardware floating point precision is first generated using a traditional algorithm. Subsequent solutions at higher precision are computed by repeatedly finding the correction term. The computation is separated into components, the majority of which are performed on the faster floating point hardware. This method has been applied to problems with easily computed residuals, and by performing most of the time-consuming computations in hardware, achieves significant speedup compared to traditional methods for these types of problems.

In this paper we apply the method to some linear algebra problems. We order the discussion according to the categories of linear systems. First we consider the solution of nonsingular linear systems, followed by least squares solutions of overdetermined linear systems, and finally we consider the singular value decomposition for singular linear systems. In the last section, we apply our method to a class of nonlinear problems, namely, systems of polynomial equations. We find that the new method reduces the computational cost of computing a high precision solution by a significant factor for these problems.

We implemented and tested our methods in `Maple 8` on a 1 GHz Pentium 3 with 512 Mb of memory. Our test cases use a "base precision" of 15 digits corresponding to hardware floating point (double precision). All the timing results have units in seconds.

2 Cost of a Floating Point Operation

Let us compare the cost of one floating point operation in the software and hardware floating point environments. In order to have timings that are sufficiently large to be measured with some degree of reliability, we use matrix multiplication. By a "floating point operation" we refer to a single scalar multiplication (or addition) operation.

When an $n \times n$ matrix is multiplied by itself using the standard method, the unit cost of a single operation may be estimated by dividing the matrix multiplication time by the total number of operations, namely $2n^3$. In the tables below we record the estimates of the unit cost for both hardware and software floating point environments, based on matrix multiplication for increasing matrix size n. See Appendix A for the Maple code.

2.1 Software versus hardware floating point

We measure the time to multiply a random $n \times n$ matrix with itself in both hardware (T_{hard}) and software (T_{soft}) floating point environments and generate two sets of data in Table 1. We calculate $T_h = T_{hard}/2n^3$, the per operation cost using hardware floating point representation, and $T_s = T_{soft}/2n^3$, the per operation cost using software floating point representation.

Note that the values chosen for the matrix size n are much larger in the hardware environment in order to achieve measurable timings, whereas in the software environment choosing such large values of n would lead to unnecessarily large matrix multiplication timings.

In this experiment the floating-point precision is specified in Maple to be `Digits:=15` for both cases, corresponding to the approximate precision, expressed in decimal digits, of the binary-based "double precision" hardware floating point representation.

We see that as the matrix size increases, T_h remains approximately constant at around $0.14 \times 10^{-8}s$. This is our estimate for the hardware floating point operation cost on the particular computer used. The absolute timing is

Hardware Cost			*Software Cost*		
n	T_{hard}	T_h	n	T_{soft}	T_s
500	.460	.1840e-8	25	.161	.5152e-5
750	1.250	.1481e-8	50	1.111	.4444e-5
1000	2.859	.1429e-8	75	4.079	.4834e-5
1500	9.691	.1436e-8	100	10.610	.5305e-5
2000	22.71	.1419e-8	125	22.700	.5811e-5
3000	76.08	.1409e-8	150	43.610	.6460e-5
4000	177.719	.1388e-8	175	78.740	.7346e-5
5000	347.210	.1388e-8	200	130.160	.8135e-5

Table 1: Hardware and software floating point operation cost.

only relevant as a means to determine the *relative* timing of software versus hardware floating point operations.

For the case of software floats, we see that as the matrix size grows T_s increases. This is due to computational overhead including the cost of garbage collection. The smallest value of T_s is approximately 0.5×10^{-5}. Clearly, T_s is at least 1000 times larger than T_h.

To see more clearly the effect of garbage collection in the software floating point environment, we turn off garbage collection and report the results in Table 2. More specifically, we set the frequency of garbage collection in Maple to `gcfreq=10^8` in contrast to the default setting of `gcfreq=10^6`. The value reported for Maple's `gctimes` is 1 from the initial setting of the `gcfreq` flag and then it is incremented with each additional invocation of garbage collection.

From Table 2 we can see that by removing the effect of garbage collection, the value of T_s remains approximately constant as the matrix size increases, with a value of approximately $0.3 \times 10^{-5}s$. The last three rows of the table illustrate, once again, the effect of garbage collection overhead. In this paper, we will use the estimate

$$T_s \approx 2 \times 10^3 \, T_h \; . \tag{1}$$

2.2 Growth of cost with increasing precision

The software floating point operation cost $T_s(d)$ is a function of the precision $d = m \times p$, where p denotes the base precision which is 15 in our test cases,

n	T_{soft}	T_s	$gctimes$
25	0.079	0.2528e-5	1
50	0.670	0.2680e-5	1
75	2.170	0.2572e-5	1
100	5.091	0.2546e-5	1
125	10.180	0.2606e-5	1
150	17.719	0.2625e-5	1
175	28.390	0.2649e-5	1
200	42.339	0.2646e-5	1
250	136.85	0.4380e-5	2
350	640.74	0.7472e-5	2
500	4504.12	1.8016e-5	2

Table 2: Software floating point with minimal garbage collection.

$precision\ m \times p$	$T_s(m \times p)$
1×15	0.4444e-5
5×15	0.6640e-5
10×15	1.598e-5
20×15	3.980e-5
30×15	6.288e-5
40×15	10.91e-5
50×15	14.07e-5
60×15	17.32e-5
70×15	30.93e-5
80×15	30.79e-5
90×15	35.18e-5
100×15	48.48e-5

Table 3: Software floating point operation cost versus precision.

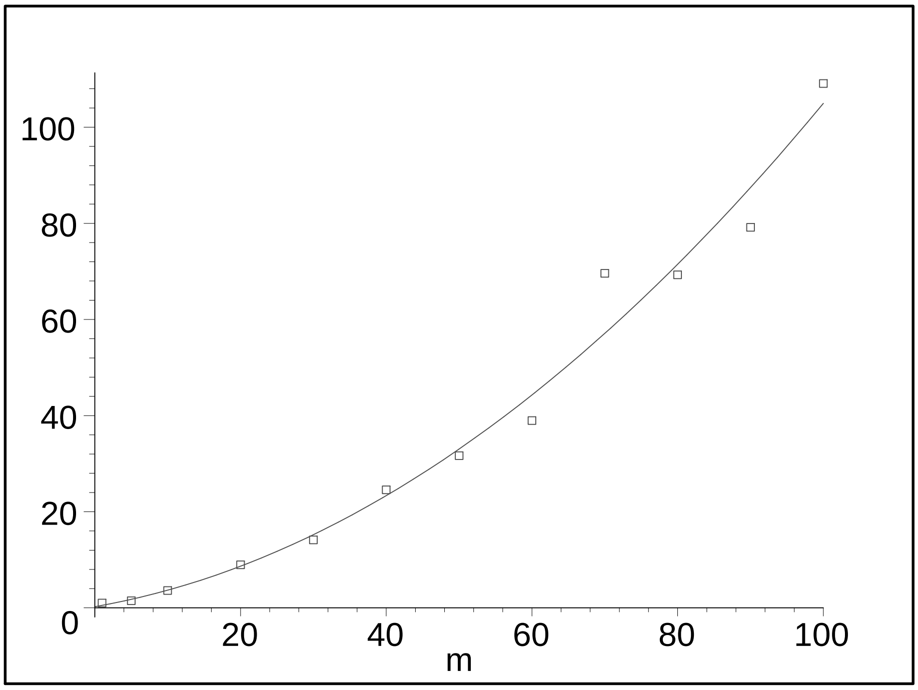

Figure 1: Time Ratio TR(m): Polynomial fit to data.

and m is the precision multiplier. To obtain experimental evidence for the rate of growth of $T_s(d)$, we fix the matrix size at 50×50 and let the precision multipler m grow. The timing results are presented in Table 3.

We wish to model the time ratio $TR(m)$ defined by

$$TR(m) = \frac{T_s(m \times p)}{T_s(1 \times p)}$$

which is the ratio of the cost of one software floating point operation in precision $m \times p$ to the cost of one software floating point operation in base precision p. The denominator here is the constant value T_s discussed above. We know that the cost of multiplying two d-digit floating point numbers is quadratic in d. It follows that the ratio $TR(m)$ is quadratic in m.

Applying a least squares fit by a polynomial of degree 2 to the data for $TR(m)$ derived from Table 3, we obtain the following equation. See Figure 1 which shows that the least squares fit to the data by a quadratic polynomial is quite good.

$$
\begin{aligned}
TR(m) = \frac{T_s(m \times p)}{T_s(1 \times p)} &\approx 0.008m^2 + 0.264m + 0.252 \\
&\approx \frac{1}{125}m^2 + \frac{1}{4}m + \frac{1}{4} \ .
\end{aligned}
$$

Incorporating the relationship $T_s \approx 2 \times 10^3\, T_h$ from equation (1), we get

$$
\begin{aligned}
T_s(m \times p) &= TR(m) \cdot T_s \\
&\approx (\frac{1}{125}m^2 + \frac{1}{4}m + \frac{1}{4})\ T_s \\
&\approx (16\, m^2 + 500\, m + 500)\ T_h \ . \qquad (2)
\end{aligned}
$$

3 Nonsingular Linear Systems

Solving nonsingular linear systems in high precision is a natural starting point to demonstrate the strength of the iterative method. The concept of our iterative method comes directly from the well-known concept of *iterative improvement* for the solution of a linear system, as discussed in standard numerical textbooks [5], [2]. In the traditional setting, one is computing in a fixed-precision floating point environment and the purpose of the iterative improvement step(s) is to refine the solution computed by a direct method into a solution accurate to full precision.

In the following discussion, we assume the base precision is p digits and the desired precision is $m \times p$ digits.

3.1 The Iterative Algorithm

Consider a linear system $Ax = b$, where $A \in R^{n \times n}, b \in R^{n \times 1}$, and we want to solve for $x \in R^{n \times 1}$. Both the direct method and the iterative method start by applying LU decomposition to A and then use forward and back substitution to solve the system. In the direct method, the computation is carried out entirely in the software floating point system at high precision, working with $m \times p$ digits throughout.

In contrast, the iterative method works in the hardware floating point system first, to find an initial solution in the base precision p, and then enters an iterative refinement loop. In each iteration, the method first uses software floating point operations to compute the residual from the previous solution in high precision, and then goes back to the hardware floating point environment to compute the correction term in base precision, and finally adds the correction term to the previous solution in software floats. The resulting sequence of solutions has monotonically increasing precision. The LU decomposition computed in the initial step at base precision is re-used in each iterative step, thus making the iterative steps very cost efficient.

In summary, the iterative method can be expressed as Algorithm A.

Algorithm A.

1. [Traditional linear solve] Solve $A\,x = b$ by a direct method in precision p yielding initial solution $x^{(1)}$; save the decomposition result $A = P\,L\,U$.
2. [Compute $x^{(i+1)} = x^{(i)} + \Delta x^{(i)}$ so that $x^{(i+1)}$ is correct to $\approx (i+1) \times p$ digits]

   For $i = 1 \,..\, M-1$:

   (a) Compute $r^{(i)} = A \cdot x^{(i)} - b$ in $(i+1) \times p$ digits.

   (b) Solve $(P\,L\,U) \cdot \Delta x^{(i)} = r^{(i)}$ for $\Delta x^{(i)}$ in p digits.

   (c) Compute $x^{(i+1)} = x^{(i)} + \Delta x^{(i)}$ in $(i+1) \times p$ digits.

In practice, one can let the loop iterate until the size of the correction term, relative to the computed solution, is small; for example, loop until $\|\Delta x^{(i)}\| \leq \epsilon\, \|x^{(1)}\|$ where ϵ is the unit roundoff error for the desired precision of the final result. The loop has been specified above in terms of a number M for purposes of the cost analysis to be carried out. We need an estimate for the number of iterations required.

The number of iterative refinement steps required is determined by how well-conditioned (or ill-conditioned) is the matrix A. Let $\kappa(A)$ denote the *condition number* of the matrix A. If $\kappa(A) \approx 1$, we expect the initial estimate $x^{(1)}$ to be correct to approximately p digits, and we expect to add approximately p correct digits with each iteration. In this very well-conditioned case we would have $M = m$ to achieve a result accurate to $m\,p$ digits.

More generally, we need the following error estimate. Suppose that the linear system $A\,y = b$ is solved in a p-digit floating-point environment by a direct method (Gaussian elimination with pivoting) yielding the computed solution y_{approx}. An error analysis [4] yields the following estimate for the relative error, where y denotes the true solution:

$$\frac{\|y - y_{approx}\|}{\|y_{approx}\|} \leq \kappa(A)\ \epsilon \tag{3}$$

where $\epsilon = 10^{1-p}$.

Now suppose that $\kappa(A) \approx 10^q$. Equation (3) implies that the initial estimate $x^{(1)}$ computed in step A1 will lose about q digits of accuracy; i.e., it will be correct only to approximately $p-q$ digits. Similarly, in each iterative step the correction term $\Delta x^{(i)}$ will only be correct to about $p-q$ digits, so we will expect to add approximately $p-q$ correct digits with each iteration. The conclusion is that to achieve the desired precision of $m\,p$ digits, the number M of iterations required (counting step A1) can be estimated by

$$M \approx \frac{m\,p}{p-q} \tag{4}$$

where $q = \log_{10}(\kappa(A))$.

It is clear from equation (4) that Algorithm A can only be expected to succeed if $q < p$. Indeed, when the condition number is large enough so that one cannot get at least one digit of accuracy in step A1 then it would be necessary to abandon hardware floating point and use a high-precision software floating point environment for the entire computation.

3.2 Cost Analysis

The cost to compute the LU decomposition in step A1 is approximately $\frac{2}{3}\,n^3$ flops (floating point operations), and the cost of forward and back substitution is $2\,n^2$ flops. The total cost C_{iter} for Algorithm A can be estimated as follows, where we use equation (2) to express $T_s(i \times p)$ in terms of T_h.

$$\begin{aligned}
C_{iter} &= T_h \times \mathrm{Cost}(LUdecomp + for_back_sub) \\
&\quad + \sum_{i=2}^{M} \left[T_s(i \times p) \times \mathrm{Cost}(A2.a + A2.c) + T_h \times \mathrm{Cost}(A2.b)\right] \\
&\approx \left(\frac{2}{3}\,n^3 + 2\,n^2\right)\ T_h + \sum_{i=2}^{M} \left[2\,(n^2+n)\,T_s(i \times p) + 2\,n^2\,T_h\right] \\
&\approx \left[\frac{2}{3}\,n^3 + 2\,n^2 + 2\,(M-1)\,n^2\right]\ T_h
\end{aligned}$$

$$+\,2\,(n^2+n)\sum_{i=2}^{M}\Big(16\,i^2+500\,i+500\Big)\;T_h$$
$$\approx\;\left[\frac{2}{3}\,n^3+\frac{2}{3}\,n^2(16\,M^3+774\,M^2+2261\,M-3048)\right]T_h+O(M^3\,n).$$

The cost for the direct method can be estimated as follows.

$$\begin{aligned} C_{direct} &= T_s(m\times p)\times \mathrm{Cost}(LUdecomp+for\_back\_sub)\\ &\approx (\frac{2}{3}\,n^3+2\,n^2)\,T_s(m\times p)\\ &\approx (\frac{8}{3}\,n^3+8\,n^2)\,(4\,m^2+125\,m+125)\,T_h\ . \end{aligned}$$

We can now estimate the speedup ratio

$$\frac{C_{direct}}{C_{iter}}\approx\frac{(\frac{8}{3}\,n^3+8\,n^2)\,(4\,m^2+125\,m+125)}{\frac{2}{3}\,n^3+\frac{2}{3}\,n^2\,(16\,M^3+774\,M^2+2261\,M-3048)}\ .$$

Using equation (4) to express M in terms of the precision multiplier m, and rearranging to exhibit the asymptotic behaviour as n grows large, we define the following Theoretical Speedup (TS) formula.

$$TS\left(\frac{C_{direct}}{C_{iter}}\right)= \tag{5}$$
$$\frac{(16\,m^2+500\,m+500)+\frac{1}{n}\,(48\,m^2+1500\,m+1500)}{1+\frac{1}{n}\left[16\left(\frac{p}{p-q}\right)^3m^3+774\left(\frac{p}{p-q}\right)^2m^2+2261\left(\frac{p}{p-q}\right)m-3048\right]}\ .$$

Note that

$$TS\left(\frac{C_{direct}}{C_{iter}}\right)\to 16\,m^2+500\,m+500\quad \text{as } n\to\infty\ .$$

3.3 Experimental Data

For our first set of experiments, we generate random nonsingular $n\times n$ matrices A and random n-vectors b, for various values of n. With base precision $p=15$, we compute the solution x of the linear system $A\,x=b$ to precision $m\times p=120$; i.e., the precision multiplier is $m=8$.

For the random nonsingular matrices generated, we find that $\kappa(A)\approx 10^3$ in each case so we use the value $q=3$ for calculating the Theoretical Speedup

Matrix size	*Time*		*Speedup*	
n	T_{iter}	T_{direct}	$\frac{T_{direct}}{T_{iter}}$	$TS(\frac{C_{direct}}{C_{iter}})$
50	0.49	2.80	5.71	2.59
75	1.78	14.45	8.12	3.81
100	5.83	48.93	8.39	5.03
125	13.00	619.01	47.62	6.25
150	36.11	321.12	8.89	7.47
175	84.14	799.35	9.50	8.69
200	154.41	8219.59	53.23	9.91
225	286.76	16853.00	58.77	11.1

Table 4: Nonsingular linear systems: size, cost and speedup.

TS. Note that the expected number of iterations required for solving this set of linear systems is $M \approx p/(p-q) \times m = 1.25\,m$; i.e., $M \approx 10$.

Equation (5) becomes

$$TS\left(\frac{C_{direct}}{C_{iter}}\right) = \frac{5524 + \frac{16572}{n}}{1 + \frac{112962}{n}} \quad .$$

Table 4 presents timing results for the iterative method and the direct method for solving this set of random linear systems. The actual ratio of the timings is presented as well as the Theoretical Speedup predicted by our cost analysis. See Appendix B for the program.

We see that the Theoretical Speedup formula predicts the speedup factor reasonably well except for various "spikes" in the timings for the direct method. In fact, the iterative method proves to be even more advantageous than predicted since it essentially avoids any serious memory issues.

The "spikes" appearing in Table 4 in the timings for the direct method are due to Maple's garbage collection algorithm as well as possibly some other memory organization issues. To test the effect of garbage collection, we present in Table 5 the results of "turning off" garbage collection (more precisely, we set `gcfreq=10^8`). Note that this is not a practical idea because it leads to a large increase in memory usage, but it serves to verify that garbage collection is the primary cause of the timing "spikes" noted in Table 4. See Appendix C for the program.

We see in Table 5 that when garbage collection is "turned off", the computing times for the direct method increase monotonically as the matrix size

Matrix size	default: $gcfreq = 10^6$			gc "off": $gcfreq = 10^8$		
n	gc	T_{direct}	space(Mb)	gc	T_{direct}	space(Mb)
50	4	2.39	7.9	1	1.35	40.4
75	8	13.60	11.2	1	4.68	127.2
100	13	51.83	12.6	1	12.94	287.0
125	20	1340.07	12.7	2	43.67	473.7
150	27	511.67	16.9	2	113.72	505.4
175	36	1267.52	20.5	2	243.31	546.2
200	47	12097.02	23.1	2	478.04	591.3
225	58	4826.99	31.5	2	1013.19	641.2

Table 5: Effect of garbage collection on Direct Method.

increases, avoiding the timing "spikes" seen in Table 4. As a tradeoff, the required memory space allocation becomes more than 20 times larger than for the default gc setting. For later experiments, we revert to the default garbage collection setting.

Table 4 showed that the iterative refinement method is significantly faster than the direct method for a set of random matrices which were reasonably well-conditioned. In Table 6 we present the results for the case of matrices which have a higher condition number. The program in Appendix D creates "ill-conditioned" matrices by creating a random matrix A and then forming $A^T A$ as the new coefficient matrix.

In Table 6 we report the condition number $\kappa(A)$ of the matrix, the relative error in the solution by each method, and the number of iterations actually required by the iterative method to reach the desired high precision solution.

As before, we use base precision $p = 15$ and we compute the solution of the linear system to precision $m \times p = 120$; i.e., $m = 8$. As can be seen in Table 6, $\kappa(A) \approx 10^6$ for this set of matrices. Therefore we have used the value $q = 6$ for calculating the Theoretical Speedup TS. Note that the expected number of iterations required for solving this "ill-conditioned" set of linear systems is $M \approx p/(p-q) \times m \approx 1.67\,m$; i.e., $M \approx 13.3$. We see that this is a reasonable estimate for the number of iterations actually required as reported in Table 6.

In this case, equation (5) becomes

$$TS\left(\frac{C_{direct}}{C_{iter}}\right) = \frac{5524 + \frac{16572}{n}}{1 + \frac{202624.6}{n}} \quad .$$

n	$\kappa(A)$ $\times 10^6$	# $it.$	*Relative Error*		*Time*		*Speedup*	
			RE_{iter}	RE_{direct}	T_{iter}	T_{direct}	$\frac{T_{direct}}{T_{iter}}$	TS
50	0.30	11	0.16e-119	0.22e-115	1.14	4.51	3.97	1.45
75	0.15	11	0.12e-119	0.44e-116	2.93	33.00	11.28	2.13
100	1.30	12	0.35e-119	0.17e-114	8.77	175.25	19.98	2.81
125	0.09	11	0.11e-119	0.34e-115	16.53	1254.08	75.85	3.49
150	0.63	12	0.19e-119	0.96e-114	36.59	2320.82	63.43	4.17
175	0.13	14	0.56e-120	0.55e-112	105.94	3415.27	32.24	4.85
200	0.43	13	0.10e-119	0.17e-113	178.84	13282.48	74.27	5.53
225	1.90	12	0.41e-119	0.10e-113	210.72	13807.22	65.52	6.21

Table 6: Ill-conditioned linear systems.

From Table 6 we observe that the iterative method not only is faster than the direct method, but also it returns fully accurate solutions. As expected, the solutions computed by the direct method lose approximately 6 digits of accuracy due to the condition number of the matrices. See Appendix D for the program.

4 Overdetermined Systems: Least Squares

Consider the case of an overdetermined linear system for which we wish to compute the least squares solution. Just as in the case of solving a nonsingular linear system, we can exploit an iterative method to compute a high precision least squares solution more efficiently than using a direct method. In the following discussion, we assume the base precision is p digits, and the desired precision is $m \times p$ digits.

4.1 The Iterative Method

Let $k > n$, $A \in R^{k\times n}$, $b \in R^{k\times 1}$ and we wish to solve $Ax \approx b$ in the least squares sense. This is an overdetermined system of linear equations. The desired solution is a vector $x \in R^{n\times 1}$ which minimizes $\|b - Ax\|_2$. This solution can be computed by finding x and r such that (see [2])

$$r = b - Ax, \ A^T r = 0 \ .$$

Note that since our iterative method will compute successive approximations for both x and r, we will compute in each iteration a residual not only with respect to x but also with respect to r.

In this section, we assume that the columns of A are linearly independent and therefore the system has a unique least squares solution set x, r. The linearly dependent case will be handled by the SVD method in section 5.

The original least squares problem can be rewritten as the following nonsingular linear system (see [2]):

$$\underbrace{\begin{pmatrix} A & I_k \\ 0 & A^t \end{pmatrix}}_{C} \underbrace{\begin{pmatrix} x \\ r \end{pmatrix}}_{y} = \underbrace{\begin{pmatrix} b \\ 0 \end{pmatrix}}_{d} . \tag{6}$$

Algorithm A from section 3 may be applied to the $(k+n)\times(k+n)$ nonsingular linear system $C\,y = d$ defined by (6) yielding the following algorithm.

1. Solve for $y^{(1)}$ in base precision by a direct method.
2. Compute $y^{(i+1)} = y^{(i)} + \Delta y^{(i)}$, $i = 1..M-1$, where $\Delta y^{(i)}$ is defined by $C\,\Delta y^{(i)} = s^{(i)}$ and $s^{(i)} = d - C\,y^{(i)}$.

In the above algorithm, the precision for each step of the computation would be as defined in Algorithm A. Also as before, $M = \left(\frac{p}{p-q}\right) m$ where

$$q = \log_{10}(\kappa(C)) \ .$$

The large size of the matrix C in the above formulation of the least squares problem makes this approach inefficient as stated. However, one can separate the large square system (6) into into smaller blocks to take advantage of the fact that there are identity and zero submatrices in the matrix C. QR decomposition is then applied to solve the problem.

We have

$$C = \begin{pmatrix} A & I_k \\ 0 & A^t \end{pmatrix}, \quad \Delta y^{(i)} = \begin{pmatrix} \Delta x^{(i)} \\ \Delta r^{(i)} \end{pmatrix},$$

$$s^{(i)} = d - Cy^{(i)} = \begin{pmatrix} s_1^{(i)} \\ s_2^{(i)} \end{pmatrix} = \begin{pmatrix} b - Ax^{(i)} - r^{(i)} \\ -A^T r^{(i)} \end{pmatrix}.$$

Solving $C\,\Delta y^{(i)} = s^{(i)}$ for $\Delta y^{(i)}$ is thus equivalent to solving the following system for $\Delta x^{(i)}$ and $\Delta r^{(i)}$:

$$\begin{cases} A\,\Delta x^{(i)} + \Delta r^{(i)} = b - A\,x^{(i)} - r^{(i)} \\ A^T\,\Delta r^{(i)} = -A^T\,r^{(i)} \ . \end{cases}$$

The iterative method for the least squares problem can be expressed as Algorithm B.

Algorithm B.

1. [Traditional QR method] In precision p, decompose $A = Q\,R$, solve $R\,x^{(1)} = Q^T\,b$ for $x^{(1)}$ and compute $r^{(1)} = b - A\,x^{(1)}$.

2. [Compute $x^{(i+1)} = x^{(i)} + \Delta x^{(i)}$, $r^{(i+1)} = r^{(i)} + \Delta r^{(i)}$ so that $x^{(i+1)}$, $r^{(i+1)}$ are correct to $\approx (i+1) \times p$ digits]

   For $i = 1 \,..\, M-1$:

   (a) In $(i+1) \times p$ digits, compute the right hand side vectors:

   $$s_1^{(i)} = b - A\,x^{(i)} - r^{(i)}$$

   $$s_2^{(i)} = -A^T\,r^{(i)}\ .$$

   (b) In p digits, solve $A^T\,\Delta r^{(i)} = s_2^{(i)}$ for $\Delta r^{(i)}$; i.e.,

   $$R^T\,Q^T\,\Delta r^{(i)} = s_2^{(i)}\,;$$

   i.e., solve $R^T\,z = s_2^{(i)}$ for $z = Q^T\,\Delta r^{(i)}$, then $\Delta r^{(i)} = Q\,z$.

   (c) In p digits, solve $A\,\Delta x^{(i)} = s_1^{(i)} - \Delta r^{(i)}$ for $\Delta x^{(i)}$; i.e.,

   $$\begin{aligned} & Q\,R\,\Delta x^{(i)} = s_1^{(i)} - \Delta r^{(i)} \\ \Rightarrow\quad & R\,\Delta x^{(i)} = Q^T\,(s_1^{(i)} - \Delta r^{(i)}) \\ \Rightarrow\quad & R\,\Delta x^{(i)} = Q^T\,s_1^{(i)} - z\ . \end{aligned}$$

   (d) In $(i+1) \times p$ digits, $x^{(i+1)} = x^{(i)} + \Delta x^{(i)}$, $r^{(i+1)} = r^{(i)} + \Delta r^{(i)}$.

The direct method uses QR decomposition and back substitution to solve the least squares problem, with all computations performed in high precision in the software floating point environment.

4.2 Cost Analysis

The cost to compute the QR decomposition in step B1 is approximately $2\,k\,n^2$ flops. The cost to compute $Q^T\,b$ and then solve for $x^{(1)}$ is $2\,k\,n + n^2$ flops and the cost to compute $A\,x^{(1)}$ and then $r^{(1)}$ is $2\,k\,n + k$ flops. The total cost C_{iter} for Algorithm B can be estimated as follows, where we use equation (2) to express $T_s(i \times p)$ in terms of T_h.

$$
\begin{aligned}
C_{iter} &= T_h \times \mathrm{Cost}(B1) + \\
&\quad \sum_{i=2}^{M} [T_s(i \times p) \times \mathrm{Cost}(B2.a + B2.d) + T_h \times \mathrm{Cost}(B2.b + B2.c)] \\
&\approx (2\,k\,n^2 + 4\,k\,n + n^2)\,T_h \\
&\quad + \sum_{i=2}^{M} \Big[(4\,k\,n + 3\,k + n)\,T_s(i \times p) + (4\,k^2 + 2\,n^2 + k)\,T_h\Big] \\
&\approx \Big[(2\,k\,n^2 + 4\,k\,n + n^2) + (4\,k^2 + 2\,n^2)\,(M-1)\Big]\;T_h \\
&\quad + 4\,k\,n \sum_{i=2}^{M} (16\,i^2 + 500\,i + 500)\,T_h \quad + \quad O(M^3\,k) \\
&\approx \Big[2\,k\,n^2 + 4\,(M-1)\,k^2 + \frac{4}{3}\,k\,n\,(16\,M^3 + 774\,M^2 + 2258\,M - 3045) \\
&\quad + (2\,M-1)\,n^2\Big]\;T_h \quad + \quad O(M^3\,k)\;.
\end{aligned}
$$

The cost for the direct method can be estimated as follows.

$$
\begin{aligned}
C_{direct} &= T_s(m \times p) \times \mathrm{Cost}(QR\ decomp + solve\ for\ x\ and\ r) \\
&\approx (2\,k\,n^2 + 4\,k\,n + n^2 + O(k))\,T_s(m \times p) \\
&\approx (8\,k\,n^2 + 16\,k\,n + 4\,n^2)\,(4\,m^2 + 125\,m + 125)\,T_h + O(m^2\,k)\;.
\end{aligned}
$$

For our experiments we choose to set $k = 2\,n$, so using this relationship the estimate for the speedup ratio is

$$
\frac{C_{direct}}{C_{iter}} \approx \frac{(16\,n^3 + 36\,n^2)\,(4\,m^2 + 125\,m + 125)}{4\,n^3 + \frac{1}{3}\,n^2\,(128\,M^3 + 6192\,M^2 + 18118\,M - 24411)}\;.
$$

Using equation (4) to express M in terms of the precision multiplier m, and rearranging to exhibit the asymptotic behaviour as n grows large, we define the following Theoretical Speedup (TS) formula.

$$TS\left(\frac{C_{direct}}{C_{iter}}\right) = \tag{7}$$
$$\frac{(16\,m^2+500\,m+500)+\frac{9}{n}(4\,m^2+125\,m+125)}{1+\frac{1}{12\,n}\left[128\left(\frac{p}{p-q}\right)^3 m^3+6192\left(\frac{p}{p-q}\right)^2 m^2+18118\left(\frac{p}{p-q}\right) m-24411\right]}\ .$$

Note that

$$TS\left(\frac{C_{direct}}{C_{iter}}\right) \to 16\,m^2+500\,m+500 \quad \text{as } n \to \infty\ .$$

4.3 Experimental Data

We generate random $k \times n$ matrices A and random k-vectors b, for various values of n and with $k = 2\,n$. Using base precision $p = 15$, we compute the least squares solution to precision $m \times p = 120$; i.e., the precision multiplier is $m = 8$. The timing results are presented in Table 7. See Appendix E for the program.

As was the case in Table 4 (nonsingular linear systems), we find that $q = 3$ is an appropriate estimate for the random matrices generated, for the purpose of calculating the Theoretical Speedup TS. Note that the expected number of iterations required for solving this set of problems is

$$M \approx p/(p-q) \times m = 1.25\,m$$

i.e., $M \approx 10$. Experimental observations confirm that this is a reasonable estimate.

Equation (7) becomes

$$TS\left(\frac{C_{direct}}{C_{iter}}\right) = \frac{5524+\frac{12429}{n}}{1+\frac{75330.75}{n}}\ .$$

From the data in Table 7, we see that the iterative method has an efficiency advantage over the direct method by a factor that is significantly larger than predicted by our cost analysis. As the investigation in section 3 illustrated (see Table 5), memory management overhead including garbage collection can add very significantly to the cost of solving large problems using a high-precision software floating point environment.

Matrix size $k \times n$	*Time*		*Speedup*	
	T_{iter}	T_{direct}	$\frac{T_{direct}}{T_{iter}}$	$TS(\frac{C_{direct}}{C_{iter}})$
50×25	1.08	4.22	3.91	2.00
80×40	3.24	33.76	10.42	3.10
100×50	5.28	100.72	19.08	3.83
120×60	9.34	278.20	29.78	4.56
150×75	24.36	652.57	26.79	5.66
160×80	31.95	3144.05	98.41	6.03
200×100	84.92	9376.09	110.41	7.49
250×125	262.92	33128.67	126.00	9.32

Table 7: Least squares problems: size, cost and speedup.

5 Singular Linear Systems: SVD

In this section, we apply an iterative method based on the singular value decomposition (SVD) for computing high precision solutions for singular linear systems. In the following discussion, we assume the base precision is p digits and the desired precision is $m \times p$ digits.

5.1 The Iterative Method

Let $k \geq n$, $A \in R^{k \times n}$, $b \in R^{k \times 1}$ and furthermore assume that the matrix is rank deficient: $\kappa(A) < n$. We investigate the iterative approach proposed by Corless and Schicho [1] for computing a solution of $A\,x \approx b$ based on applying the Moore-Penrose pseudo-inverse of the singular matrix A.

First we outline the direct method for this problem. The following steps would be applied in the desired precision $m \times p$.

1. Compute the singular value decomposition $A = U\,\Sigma\,V^T$.
2. Decide the numerical rank r of A by examining the singular values $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r \geq \sigma_{r+1} \approx 0 \geq \ldots \geq \sigma_n$.
3. Compute the Moore-Penrose pseudo-inverse $A^{\dagger} = V\,\Sigma^{\dagger}\,U^T$ where
$$\Sigma^{\dagger} = \mathrm{diag}(\frac{1}{\sigma_1}, \frac{1}{\sigma_2}, \ldots, \frac{1}{\sigma_r}, 0, \ldots, 0)\ .$$
4. Compute the solution $x = A^{\dagger}\,b$.

The determination of the numerical rank in step 2 above (and in step C1 of Algorithm C) requires that a tolerance τ has been chosen such that all singular values satisfying $\sigma_i \leq \tau$ are considered to be equivalent to zero.

In the iterative method for this problem, we first apply in base precision p the direct method outlined above. Then we apply the iteration presented in Algorithm C. Note that the approximate pseudo-inverse $A^\dagger$ computed in step C1 is used repeatedly in each iteration [1].

Algorithm C.

1. [Moore-Penrose method] In base precision p, decompose $A = U\,\Sigma\,V^T$, then compute the approximate pseudo-inverse $A^\dagger = V\,\Sigma^\dagger\,U^T$ and the initial solution $x^{(1)} = A^\dagger\, b$.
2. [Compute $x^{(i+1)} = x^{(i)} + \Delta x^{(i)}$ so that $x^{(i+1)}$ is correct to $\approx (i+1) \times p$ digits]

   For $i = 1 \,..\, M-1$:

   (a) In $(i+1) \times p$ digits, compute $r^{(i)} = A\,x^{(i)} - b$.

   (b) In p digits, compute $\Delta x^{(i)} = A^\dagger\, r^{(i)}$.

   (c) In $(i+1) \times p$ digits, compute $x^{(i+1)} = x^{(i)} + \Delta x^{(i)}$.

5.2 Cost Analysis

Consider the case $k = 2n$. The cost of the singular value decomposition is $12n^3 + O(n^2)$ flops. Computing the pseudo inverse costs $4n^3 + O(n^2)$ flops and computing the initial solution $x^{(1)}$ costs $2n^2$ flops [3].

The cost of the direct method can be estimated as follows where, as usual, we use equation (2) to express $T_s(m \times p)$ in terms of T_h .

$$
\begin{aligned}
C_{direct} &= T_s(m \times p) \times \mathrm{Cost}(SVD + pseudo\ inverse + compute\ x) \\
&\approx [12\,n^3 + 4\,n^3 + O(n^2)]\,T_s(m \times p) \\
&\approx 16\,n^3\,(16\,m^2 + 500\,m + 500)\ T_h + O(m^2\,n^2)\ .
\end{aligned}
$$

The iterative method of Algorithm C has total cost estimated as follows.

$$
\begin{aligned}
C_{iter} &= T_h \times \mathrm{Cost}(C1) + \\
&\quad \sum_{i=2}^{M} [T_s(i \times p) \times \mathrm{Cost}(C2.a + C2.c) + T_h \times \mathrm{Cost}(C2.b)] \\
&\approx 16\, n^3\, T_h + \sum_{i=2}^{M} [2\, n^2\, T_s(i \times p) + 2\, n^2\, T_h] \quad + \quad O(n^2) \\
&\approx \left[16\, n^3 + \frac{2}{3}\, n^2\, (16\, M^3 + 774\, M^2)\right] T_h \quad + \quad O(M\, n^2)\ .
\end{aligned}
$$

We have not done an analysis to determine how to estimate M, the number of iterations required. For the experimental results presented here, we simply note that for our set of tests the number of iterations never exceeded $M = \frac{3}{2}\, m$, so we use this estimate below. In any case, we find that the actual speedup achieved by the iterative method is much larger than predicted by our theoretical analysis.

The estimate for the speedup ratio is

$$
\frac{C_{direct}}{C_{iter}} \approx \frac{16\, n^3\, (16\, m^2 + 500\, m + 500)}{16\, n^3 + \frac{2}{3}\, n^2\, (16\, M^3 + 774\, M^2)} \approx \frac{16\, n^3\, (16\, m^2 + 500\, m + 500)}{16\, n^3 + 9\, n^2\, (4\, m^3 + 129\, m^2)}\ .
$$

Therefore we define the Theoretical Speedup (TS) formula:

$$
TS\left(\frac{C_{direct}}{C_{iter}}\right) = \frac{16\, m^2 + 500\, m + 500)}{1 + \frac{9}{16\, n}\, (4\, m^3 + 129\, m^2)}\ . \tag{8}
$$

Note that

$$
TS\left(\frac{C_{direct}}{C_{iter}}\right) \to 16\, m^2 + 500\, m + 500 \quad \text{as } n \to \infty\ .
$$

5.3 Experimental Data

We set $k = 2n$ and generate matrices A of size $k \times n$ having deficient rank: $\kappa(A) < n$. The k-vectors b are chosen to ensure that the singular system has a solution. Using base precision $p = 15$, we solve the problem in the sense described above to precision $m \times p = 120$; i.e., the precision multiplier is $m = 8$. The timing results are presented in Table 8. See Appendix F for the program.

Matrix size $k \times n$	*Time*		*Speedup*	
	T_{iter}	T_{direct}	$\frac{T_{direct}}{T_{iter}}$	$TS(\frac{C_{direct}}{C_{iter}})$
30×15	0.29	7.74	26.69	14.3
50×25	0.61	52.19	85.56	23.7
80×40	1.11	213.74	192.56	37.9
100×50	1.47	786.75	535.20	47.3
120×60	2.00	3700.11	1850.01	56.6
150×75	2.94	17907.16	6090.87	70.6

Table 8: SVD problems: size, cost and speedup.

Equation (8) becomes

$$TS\left(\frac{C_{direct}}{C_{iter}}\right) = \frac{5524}{1+\frac{5796}{n}} \quad .$$

From the data in Table 8, we see that the iterative method has an efficiency advantage over the direct method by a factor that is significantly larger than predicted by our cost analysis. As the investigation in section 3 illustrated (see Table 5), memory management overhead including garbage collection can add very significantly to the cost of solving large problems using a high-precision software floating point environment.

6 Nonlinear Equations: Polynomial Systems

The problem of computing numerical solutions for a system of nonlinear equations is typically solved by a successive approximation method. Indeed, Newton's iteration is one common choice of method. Starting from a sufficiently accurate initial approximation $x^{(0)}$ one computes a sequence of iterates $x^{(1)}, x^{(2)}, x^{(3)}, \ldots$ which converge to a solution.

The point of considering systems of nonlinear equations in this paper is to note that, just as in the preceding sections, it can be advantageous when computing high-precision solutions to employ a linearly converging Newton iteration rather than the commonly-used quadratic iteration. The idea is to build up the high precision solution in blocks of "base precision" digits and thus exploit the speed of the hardware floating point environment.

In this section, we consider systems of multivariate polynomial equations and demonstrate the strength of our particular "iterative method" for computing high precision solutions. As before, the base precision is p digits and we wish to compute solutions accurate to precision $m \times p$ digits.

6.1 The Iterative Method

We are given a system of nonlinear equations

$$f_i(x_1, x_2, ..., x_n) = 0, \quad i = 1, 2, ..., n \tag{9}$$

to be solved for $x_i,\ i = 1, 2, ..., n$. In this paper, we will assume that we are given a sufficiently accurate initial approximation $x^{(0)}$.

Newton's iteration in matrix-vector formulation takes the following form where $J_f(x)$ denotes the $n \times n$ Jacobian matrix for system (9) evaluated at a point (vector) x:

$$x^{(k+1)} = x^{(k)} - [J_f(x^{(k)})]^{-1} \cdot f(x^{(k)}) \ .$$

Denote $J^{(k)} = J_f(x^{(k)})$, $\Delta x^{(k)} = x^{(k+1)} - x^{(k)}$, $f^{(k)} = f(x^{(k)})$.

The so-called "direct method" (i.e., the traditional quadratic Newton iteration method) updates the Jacobian matrix $J^{(k)}$ and the residual $f^{(k)}$ in each iteration, and solves the linear system $J^{(k)}\,\Delta x^{(k)} = f^{(k)}$ by a direct linear solver. Most of the calculations are performed in the high precision software floating point environment, noting that the precision is allowed to grow appropriately with each iteration. (See the program in Appendix G.)

Our "iterative method", in contrast, performs as much of the computation as possible in the hardware floating point environment. We compute the first iterate $x^{(1)}$ by applying the "direct" Newton's method at base precision. (In all cases, we are assuming that a sufficiently accurate initial approximate $x^{(0)}$ has been given to us.) We then evaluate the Jacobian matrix $J^{(1)}$ at base precision and compute the LU decomposition of $J^{(1)}$. The result $J^{(1)} = P\,L\,U$ of the LU decomposition is kept and repeatedly used in later iterations. The residual is computed at higher precision in the software floating point environment, and the correction term is calculated at base precision in the hardware floating point environment.

We present our algorithm as Algorithm D.

Algorithm D.

1. Compute $x^{(1)}$ in precision p via a standard ("direct") Newton's method starting with the given initial guess $x^{(0)}$.

2. In precision p, compute the Jacobian matrix $J^{(1)} = J_f(x^{(1)})$ and apply LU decomposition yielding $J^{(1)} = P\,L\,U$.

3. [Compute $x^{(i+1)} = x^{(i)} + \Delta x^{(i)}$ so that $x^{(i+1)}$ is correct to $\approx (i+1) \times p$ digits]

   Set $\epsilon = 0.5 \times 10^{1-(m \times p)}$.

   For $i = 1, 2, ...$

   (a) Compute $r^{(i)} = f(x^{(i)})$ in $(i+1) \times p$ digits.

   (b) Solve $(P\,L\,U) \cdot \Delta x^{(i)} = r^{(i)}$ for $\Delta x^{(i)}$ in p digits.

   (c) Compute $x^{(i+1)} = x^{(i)} - \Delta x^{(i)}$ in $(i+1) \times p$ digits.

   until $\mathrm{Norm}(\Delta \mathrm{x}^{(\mathrm{i})}) \leq \epsilon$.

6.2 Cost Analysis

Suppose the cost of evaluating function f_i at a point $(x_1, x_2, ..., x_n)$ is C_{f_i} flops. This cost will depend on the particular function f_i. For a system of sparse multivariate polynomials, we may assume that $C_{f_i} \sim O(n)$. We will use the estimate $C_{f_i} \approx n$ in the following cost analysis.

In the "direct method", each iteration performs n function evaluations to compute the residual $f^{(k)}$, n^2 function evaluations to evaluate the Jacobian matrix $J^{(k)}$, and solves one $n \times n$ linear system. Since the solution converges quadratically, only a few iterations are required and we may estimate the total cost by the cost of the last iteration which is performed at the highest precision. Therefore the cost estimate for the "direct method" is:

$$
\begin{aligned}
C_{direct} &\geq \Big[(n^2+n)\,C_{f_i} + \mathrm{Cost}(\mathit{linear\ solve})\Big]\,T_s(m \times p) \\
&\approx \Big[(n^2+n)\,n + \frac{2}{3}n^3 + 2n^2\Big]\,(16\,m^2 + 500\,m + 500)\,T_h \\
&\approx \Big[\Big(\frac{5}{3}\,n^3 + 3\,n^2\Big)\,(16\,m^2 + 500\,m + 500)\Big]\,T_h\ .
\end{aligned}
$$

At base precision p, our " iterative method" computes the initial solution $x^{(1)}$ using the "direct method", evaluates the Jacobian matrix $J^{(1)}$, and decomposes $J^{(1)} = P\,L\,U$. At higher precision, it computes the residual. As in the previous sections, this method saves the LU decomposition of $J^{(1)}$ for later use. The total cost C_{iter} for Algorithm D can be estimated as follows.

$$\begin{aligned}
C_{iter} &= T_h \times \text{Cost}\left(compute\ x^{(1)} + compute\ J^{(1)} + LUdecomp\right) \\
&\quad + \sum_{i=2}^{M} \left[T_s(i \times p) \times \text{Cost}(D3.a + D3.c) + T_h \times \text{Cost}(D3.b)\right] \\
&\approx \left[\frac{5}{3}n^3 + 3n^2 + n^2 C_{f_i} + \frac{2}{3}n^3\right] T_h \\
&\quad + \sum_{i=2}^{M} \left[(n\, C_{f_i} + n)\, T_s(i \times p) + 2n^2 T_h\right] \\
&\approx \left[\frac{10}{3}n^3 + 3n^2 + 2(M-1)n^2\right] T_h \\
&\quad + (n^2 + n)\sum_{i=2}^{M}(16\, i^2 + 500\, i + 500)\, T_h \\
&\approx \left[\frac{10}{3}n^3 + \frac{1}{3}n^2\left(16M^3 + 774M^2 + 2264M - 3045\right)\right] T_h + O(M^3 n)\ .
\end{aligned}$$

The number M of iterations required will depend on $\kappa(J^{(1)})$, the condition number of the Jacobian matrix, because the correction term for each iteration is computed in step D3.b by solving a linear system with coefficient matrix $J^{(1)}$. For our cost estimates, we will assume $q = \log(\kappa(J^{(1)})) \approx 6$ and the expected number of iterations is $M \approx \left(\frac{p}{p-q}\right) m \approx \frac{5}{3}m$.

Using this relationship for M in terms of m and rearranging to exhibit the asymptotic behaviour as n grows large, we define the following Theoretical Speedup (TS) formula.

$$TS\left(\frac{C_{direct}}{C_{iter}}\right) = \frac{(8m^2 + 250m + 250) + \frac{18}{5n}(4m^2 + 125m + 125)}{1 + \frac{1}{54n}(400m^3 + 11610m^2 + 20376m - 16443)}\ . \tag{10}$$

Note that

$$TS\left(\frac{C_{direct}}{C_{iter}}\right) \to 8m^2 + 250m + 250,\ \ as\ n \to \infty\ .$$

Size	*Time*		*Speedup*		*Norm of Errors*	
n	T_{iter}	T_{direct}	$\frac{T_{direct}}{T_{iter}}$	TS	NE_{iter}	NE_{direct}
25	0.91	1.88	2.07	3.65	0.459e-119	0.465e-119
50	1.83	9.12	4.98	7.04	0.483e-119	0.483e-119
75	2.92	33.59	11.50	10.4	0.604e-119	0.483e-119
100	5.41	82.00	15.16	13.8	0.457e-119	0.483e-119
125	7.91	568.98	72.00	17.2	0.483e-119	0.483e-119
150	12.03	1425.43	118.49	20.5	0.483e-119	0.483e-119
225	37.86	9423.69	248.91	30.6	0.333e-118	0.333e-118
250	54.241	18162.47	334.85	33.9	0.338e-118	0.333e-118

Table 9: Polynomial systems: size, cost, speedup and errors.

6.3 Experimental Data

In our experiments, the base precision is $p = 15$ and we compute results to precision $m \times p = 120$; *i.e.*, $m = 8$. Equation (10) becomes

$$TS\left(\frac{C_{direct}}{C_{iter}}\right) = \frac{2762 + \frac{24858}{5n}}{1 + \frac{1094405}{54n}} .$$

Some timing results for randomly generated polynomial systems are presented in Table 9. The size of the error in the computed solution for each method is also presented in the table. See Appendix G for the program.

We see that each of the methods computes solutions that are accurate up to the last two digits. As in previous sections, the efficiency advantage of our proposed iterative method is even greater than anticipated, mainly because the traditional method operating in a high-precision software floating point environment incurs expensive garbage collection costs.

6.4 Aside: Comparison with fsolve

In this experiment, we compare the "direct method" with Maple's built-in function `fsolve`. We find that the "direct method" has better performance than `fsolve` on systems of polynomial equations. As Table 9 shows, our new iterative method offers even greater performance.

Maple's `fsolve` needs a hint (*i.e.* an initial guess) for solving a large system of equations, as do our methods. We start with a hint accurate

Size	*Time*		*Digits of Hint*		*Norm of Errors*	
n	T_{fsolve}	T_{direct}	$fsolve$	$direct$	NE_{fsolve}	NE_{direct}
25	4.91	1.88	3	3	0.220e-115	0.465e-119
50	42.02	9.12	3	3	0.110e-112	0.483e-119
75	132.35	33.59	3	3	0.310e-114	0.483e-119
100	433.56	82.00	3	3	0.110e-113	0.483e-119
125	$fail$	568.98	-	3	-	0.483e-119
150	$fail$	1425.43	-	3	-	0.483e-119
225	$fail$	9423.69	-	3	-	0.333e-118
250	$fail$	18162.47	-	3	-	0.333e-118

Table 10: Direct method *vs* fsolve.

to the first 3 digits. The "direct method" works properly for the entire input set, while `fsolve` fails when the system size is greater than 100. Also, when `fsolve` succeeds it gives a less accurate solution, as shown in Table 10. Where "*Digits of Hint*" for `fsolve` in Table 10 is indicated by a dash, we tried successively more accurate hints (up to 20 digits) and `fsolve` still failed.

7 Conclusion

The iterative refinement method based on a linear Newton iteration, exploiting the speed of hardware floating point while constructing a high precision solution, is found to be significantly faster than traditional direct methods. The methods were compared on some linear algebra problems and also on systems of nonlinear equations.

In the iterative refinement method, the main computation which must be performed in the high-precision software floating point environment is the computation of the residual in each iteration. Problems for which the residual is easy to compute are suitable for the proposed method.

Computing eigenvalues to high precision could be a possible extension to our current work if a fast method of computing determinants (for the residual calculations) can be found.

References

[1] Corless, R. and Schicho, J. Iterated improvement using the SVD for singular linear systems. *Technical Report TR-00-09*, Ontario Research Centre for Computer Algebra, London, ON, Canada, 2000.

[2] Dahlquist, G. and Bjoerck, A. *Numerical Methods.* Prentice-Hall, Englewood Cliffs, NJ, 1974.

[3] Demmel, J. *Applied Numerical Linear Algebra.* SIAM, 1997.

[4] Forsythe, G., Malcolm, M. and Moler, C. *Computer Methods for Mathematical Computations.* Prentice-Hall, Englewood Cliffs, NJ, 1974.

[5] Forsythe, G. and Moler, C. *Computer Solution of Linear Algebraic Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1967.

[6] Geddes, K., Czapor, S. and Labahn, G. *Algorithms for Computer Algebra.* Kluwer Academic Publishers, Norwell, MA, 1992.

[7] Golub, G. and Van Loan, C. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, MD, 1989.

Appendices

A Floating Point Cost Code

```
Table 1: Hardware and software floating point operation cost.
         Compute in base precision, default gcfreq.
Table 2: Software floating point with minimal garbage collection.
         Set gcfreq=10^8, compute only in software floats.
Table 3: Software floating point operation cost versus precision.
         Increase precision multiplier, compute in software floats.
```

```
> with(LinearAlgebra):
> baseprec := trunc(evalhf(Digits));
```

$$baseprec := 15$$

A.1 Hardware floating point environment

```
> n := 500;
```

$$n := 500$$

```
> Digits := baseprec;  UseHardwareFloats := true:
```

$$Digits := 15$$

```
> A := RandomMatrix(n, n, generator=-1.0..1.0,
                    outputoptions=[datatype='float[8]']):
> hfTime := time( MatrixMatrixMultiply(A, A) );
```

$$hfTime := 0.860$$

```
> T_h := evalf[4](hfTime/(2*n^3));
```

$$T\_h := 0.3440\,10^{-8}$$

A.2 Software floating point environment

```
> #kernelopts(gcfreq=10^8):
> n := 50;
```

$$n := 50$$

```
> multiplier := 1:
> prec := multiplier*baseprec:
> Digits := prec;  UseHardwareFloats := false:
```

$$Digits := 15$$

```
> t := SFloat(1, -prec):
> A := RandomMatrix(n, n, generator=rand(-10^prec..10^prec),
                  outputoptions=[datatype='sfloat']):
> A := t.A:
> sfTime := time( MatrixMatrixMultiply(A, A) );
```

$$sfTime := 2.209$$

```
> T_s := evalf[4](sfTime/(2*n^3));
```

$$T\_s := 0.8835\,10^{-5}$$

```
> kernelopts(gctimes);
```

$$3$$

B Nonsingular Linear Systems Code

Table 4: Nonsingular linear systems: size, cost and speedup.
Default gcfreq.

B.1 Procedure precLinearSolve

```
> precLinearSolve := proc (A::Matrix, b::Vector, prec::integer)
    local baseprec, ipiv, x, delta_x, Normx, eps, k, r;

    baseprec := trunc(evalhf(Digits));

    Digits := baseprec;  UseHardwareFloats := true;
    ipiv := LinearAlgebra:-LUDecomposition(A, output=['NAG']);
    x := LinearAlgebra:-LinearSolve([ipiv], b);

    delta_x := x;  Normx := LinearAlgebra:-Norm(x);
    eps := SFloat(5, -prec)*Normx;
    for k from 2 while LinearAlgebra:-Norm(delta_x) > eps do
      Digits := k*baseprec;  UseHardwareFloats := false;
      r := A.x - b;

      Digits := baseprec;  UseHardwareFloats := true;
      delta_x := LinearAlgebra:-LinearSolve([ipiv], r);

      Digits := k*baseprec;  UseHardwareFloats := false;
      x := x - delta_x;
    end do;

    return x;
  end proc:
```

B.2 Problem setup

```
> with(LinearAlgebra):
> baseprec := trunc(evalhf(Digits));
```

$$baseprec := 15$$

```
> multiplier := 8:
> prec := multiplier*baseprec:
> Digits := prec;  UseHardwareFloats := false:
```

$$Digits := 120$$

```
> n := 100;
```

$$n := 100$$

```
> t := SFloat(1, -prec):
> A := RandomMatrix(n, n, generator=rand(-10^prec..10^prec),
>                   outputoptions=[datatype='sfloat']):
> A := t.A:
> b := RandomVector(n, generator=rand(-10^prec..10^prec),
>                   outputoptions=[datatype='sfloat']):
> b := t.b:
```

B.3 Iterative Method

```
> st := time():
> iterative_x := precLinearSolve(A, b, prec):
> iterativeTime := time() - st:
```

B.4 Direct Method

```
> st := time():
> direct_x := LinearSolve(A, b):
> directTime := time() - st():
```

B.5 Results

```
> directTime, iterativeTime;
```

$$579.060,\ 51.650$$

```
> SpeedUp := evalf[3](directTime/iterativeTime);
```

$$SpeedUp := 11.2$$

```
> diffNorm := Norm(iterative_x - direct_x):
> evalf[3](diffNorm);
```

$$0.412\,10^{-115}$$

```
> # Note: iterative_x is fully accurate.
```

C Effect of Garbage Collection

Table 5: Effect of garbage collection on Direct Method.

C.1 Problem setup

```
> kernelopts(gcfreq=10^8):
> with(LinearAlgebra):
> baseprec := trunc(evalhf(Digits));
```

$$baseprec := 15$$

```
> multiplier := 8:
> prec := multiplier*baseprec:
> Digits := prec;  UseHardwareFloats := false:
```

$$Digits := 120$$

```
> n := 100;
```

$$n := 100$$

```
> t := SFloat(1, -prec):
> A := RandomMatrix(n, n, generator=rand(-10^prec..10^prec),
>                  outputoptions=[datatype='sfloat']):
> A := t.A:
> b := RandomVector(n, generator=rand(-10^prec..10^prec),
>                  outputoptions=[datatype='sfloat']):
> b := t.b:
```

C.2 Direct Method

```
> st := time():
> direct_x := LinearSolve(A, b):
> directTime := time() - st():
```

C.3 Results

```
> kernelopts(gctimes);
```

$$2$$

```
> evalf[4](kernelopts(bytesalloc));
```

$$0.2882\,10^9$$

```
> directTime;
```

$$39.190$$

D Ill-conditioned Linear Systems Code

```
Table 6: Ill-conditioned linear systems.
         Default gcfreq.
```

D.1 Procedure precLinearSolve

```
> precLinearSolve := proc (A::Matrix, b::Vector, prec::integer)
    local baseprec, ipiv, x, delta_x, Normx, eps, k, r;

    baseprec := trunc(evalhf(Digits));

    Digits := baseprec;  UseHardwareFloats := true;
    ipiv := LinearAlgebra:-LUDecomposition(A, output=['NAG']);
    x := LinearAlgebra:-LinearSolve([ipiv], b);

    delta_x := x;  Normx := LinearAlgebra:-Norm(x);
    eps := SFloat(5, -prec)*Normx;
    for k from 2 while LinearAlgebra:-Norm(delta_x) > eps do
      Digits := k*baseprec;  UseHardwareFloats := false;
      r := A.x - b;

      Digits := baseprec;  UseHardwareFloats := true;
      delta_x := LinearAlgebra:-LinearSolve([ipiv], r);
      print(`Norm(delta_x)` = LinearAlgebra:-Norm(delta_x));

      Digits := k*baseprec;  UseHardwareFloats := false;
      x := x - delta_x;
    end do;

    print(`#Iterations` = k-1);
    return x;
  end proc:
```

D.2 Problem setup

```
> with(LinearAlgebra):
> baseprec := trunc(evalhf(Digits));
```

$$baseprec := 15$$

```
> multiplier := 8:
> prec := multiplier*baseprec:
> Digits := prec;  UseHardwareFloats := false:
```

$$Digits := 120$$

```
> n := 100;
```

$$n := 100$$

```
> t := SFloat(1, -prec):
> A := RandomMatrix(n, n, generator=rand(-10^prec..10^prec),
>                   outputoptions=[datatype='sfloat']):
> A := t.A:
> b := RandomVector(n, generator=rand(-10^prec..10^prec),
>                   outputoptions=[datatype='sfloat']):
> b := t.b:

> # Create matrix A with larger condition number.
> A := Transpose(A).A:
> condA := evalf[3](ConditionNumber(A));
```

$$condA := 14800.$$

D.3 Iterative Method

```
> st := time():
> iterative_x := precLinearSolve(A, b, prec):
```

$$Norm(delta\_x) = 0.916610133592613816\,10^{-8}$$

$$Norm(delta\_x) = 0.649768918493847026\,10^{-19}$$

$$Norm(delta\_x) = 0.461555877050158644\,10^{-30}$$
$$Norm(delta\_x) = 0.327887888693404913\,10^{-41}$$
$$Norm(delta\_x) = 0.232927678675440092\,10^{-52}$$
$$Norm(delta\_x) = 0.165466123224243248\,10^{-63}$$
$$Norm(delta\_x) = 0.117558431188850830\,10^{-74}$$
$$Norm(delta\_x) = 0.835141325409320218\,10^{-86}$$
$$Norm(delta\_x) = 0.593240981105982526\,10^{-97}$$
$$Norm(delta\_x) = 0.421396189769436202\,10^{-108}$$
$$Norm(delta\_x) = 0.299349272043340206\,10^{-119}$$
$$\#Iterations = 12$$

```
> iterativeTime := time() - st:
```

D.4 Direct Method

```
> st := time():
> direct_x := LinearSolve(A, b):
> directTime := time() - st():
```

D.5 Results

```
> directTime, iterativeTime;
```

$$439.620,\ 64.440$$

```
> SpeedUp := evalf[3](directTime/iterativeTime);
```

$$SpeedUp := 6.83$$

```
> diffNorm := Norm(iterative_x - direct_x):
> evalf[3](diffNorm);
```

$$0.851\,10^{-111}$$

```
> # Note: iterative_x is fully accurate.
```

E Least Squares Code

Table 7: Least squares problems: size, cost and speedup.

E.1 Procedure precLeastSquares

```
> precLeastSquares := proc(A::Matrix, rows::integer, cols::integer,
                           b::Vector, prec::integer)
    local baseprec, At, Q0, R0, Q0t, b2, R0upper, R0upper_t, x, r,
          delta_x, Normx, eps, k, s1, s2, k1, u, i, delta_r;

    baseprec := trunc(evalhf(Digits));
    At := LinearAlgebra:-Transpose(A);

    Digits := baseprec;  UseHardwareFloats := true;
    (Q0, R0) := LinearAlgebra:-QRDecomposition( Matrix(rows,[A]) );
    Q0t := LinearAlgebra:-Transpose(Q0);
    b2 := Q0t.b;
    R0upper := R0[1..cols, 1..cols];
    R0upper_t := LinearAlgebra:-Transpose(R0upper);
    x := LinearAlgebra:-BackwardSubstitute(R0upper, b2[1..cols]);
    r := Q0[1..rows, cols+1..rows] . b2[cols+1..rows];

    delta_x := x;  Normx := LinearAlgebra:-Norm(x);
    eps := SFloat(5, -prec)*Normx;
    for k from 2 while LinearAlgebra:-Norm(delta_x) > eps do
      Digits := k*baseprec;  UseHardwareFloats := false;
      s1 := b - r - A.x;  s2 := -At.r;

      Digits := baseprec;  UseHardwareFloats := true;
      k1 := LinearAlgebra:-ForwardSubstitute(R0upper_t, s2);
      u := Q0t.s1;
      delta_x := LinearAlgebra:-BackwardSubstitute(R0upper,
                                                   u[1..cols]-k1);
      for i from 1 to cols do u[i] := k1[i] end do;
      delta_r := Q0.u;

      Digits := k*baseprec;  UseHardwareFloats := false;
```

```
    x := x + delta_x;  r := r + delta_r;
  end do;

  return (x, r);
end proc:
```

E.2 Problem setup

```
> with(LinearAlgebra):
> baseprec := trunc(evalhf(Digits));
```

$$baseprec := 15$$

```
> multiplier := 8:
> prec := multiplier*baseprec:
> Digits := prec;  UseHardwareFloats := false:
```

$$Digits := 120$$

```
> cols := 25;  rows := 2*cols;
```

$$cols := 25$$

$$rows := 50$$

```
> t := SFloat(1, -prec):
> A := RandomMatrix(rows, cols,
>                   generator=rand(-10^prec..10^prec),
>                   outputoptions=[datatype='sfloat']):
> A := t.A:
> b := RandomVector(rows, generator=rand(-10^prec..10^prec),
>                   outputoptions=[datatype='sfloat']):
> b := t.b:
```

E.3 Iterative Method

```
> st := time():
> (x, r) := precLeastSquares(A, rows, cols, b, prec):
> iterativeTime := time() - st:
```

E.4 Direct Method

```
> st := time():
> (Q, R) := QRDecomposition(A):
> b2 := Transpose(Q).b:
> direct_x := BackwardSubstitute(R, b2):
> direct_r := b - A.direct_x:
> directTime := time() - st:
```

E.5 Results

```
> directTime, iterativeTime;
```

$$50.900,\ 5.900$$

```
> SpeedUp := evalf[3](directTime/iterativeTime);
```

$$SpeedUp := 8.63$$

```
> diffNorm_x := Norm(x - direct_x):
> evalf[3](diffNorm_x);
```

$$0.685\,10^{-119}$$

```
> diffNorm_r := Norm(r - direct_r):
> evalf[3](diffNorm_r);
```

$$0.239\,10^{-118}$$

F Singular Linear Systems: SVD Code

```
Apply Corless and Schicho's
  "Iterated Improvement using the SVD."
```

F.1 Procedure IteratedSVD

```
> IteratedSVD := proc (A::Matrix, Ap::Matrix, b::Vector,
                                              prec::integer)
    local baseprec, eps, x, r, k, delta_x;

    baseprec := trunc(evalhf(Digits));

    # Desired size of residual is eps.
    eps := SFloat(1, -prec);

    # Compute initial solution.
    #   Ap = approximate Moore-Penrose pseudo-inverse of A
    #   accurate to (at most) baseprec.
    Digits := baseprec;  UseHardwareFloats := true;
    x := Ap.b;
    Digits := 2*baseprec;  UseHardwareFloats := false;
    r := A.x - b;

    # Iterate until Norm(r) <= eps.
    for k from 2 while LinearAlgebra:-Norm(r) > eps do
      Digits := baseprec;  UseHardwareFloats := true;
      delta_x := Ap.r;
      Digits := k*baseprec;  UseHardwareFloats := false;
      x := x - delta_x;
      Digits := Digits + baseprec;
      r := A.x - b;
    end do;

    return x;
  end proc:
```

F.2 Problem setup

```
# For k >= n define k-by-n matrix A with rank(A) < n and
# k-vector b, with x_true a solution of A.x = b
> with(LinearAlgebra):
> baseprec := trunc(evalhf(Digits));
```

$$baseprec := 15$$

```
> multiplier := 8:
> prec := multiplier*baseprec:
> Digits := prec;  UseHardwareFloats := false:
```

$$Digits := 120$$

```
> cols := 25;  rows := 2*cols;
```

$$cols := 25$$

$$rows := 50$$

```
> rank := cols-1;
```

$$rank := 24$$

```
> A := RandomMatrix(rows,rank).RandomMatrix(rank, cols):
> x_true := RandomVector(cols):
> b := A.x_true:

> # ranktol: threshold for deciding rank based on singvals.
> ranktol := SFloat(1, -baseprec);
```

$$ranktol := 0.1\,10^{-14}$$

F.3 Iterative Method

```
> st := time():
> Digits := baseprec;  UseHardwareFloats := true:
```

$$Digits := 15$$

```
> # SVD computation in base precision.
> (U, S, Vt) := SingularValues(A, output=['U','S', 'Vt']):
```

```
> # Compute Ap = Moore-Penrose pseudo-inverse of A .
> tau := ranktol*Norm(S):
> Sp := Vector(rows, 0):
> for i from 1 to rows do
>   if S[i] > tau then Sp[i] := 1.0/S[i] end if;
> end do:
> Spt := DiagonalMatrix( Sp[1..cols], rows, cols ):
> Ap := Transpose(Vt).Transpose(Spt).Transpose(U):
```

```
> x_iter := IteratedSVD(A, Ap, b, prec):
> iterativeTime := time()-st:
```

F.4 Direct Method

```
> st := time():
> Digits := prec;  UseHardwareFloats := false:
```

$$Digits := 120$$

```
> # SVD computation in high precision.
> (U, S, Vt) := SingularValues(A, output=['U','S', 'Vt']):
```

```
> # Compute the Moore-Penrose pseudo-inverse of A.
> tau := ranktol*Norm(S):
> Sd := Vector(rows, 0):
> for i from 1 to rows do
>   if S[i] > tau then Sd[i] := 1.0/S[i] end if;
> end do:
> Sdt := DiagonalMatrix( Sd[1..cols], rows, cols ):
> Ap := Transpose(Vt).Transpose(Sdt).Transpose(U):
```

```
> x_dir := Ap.b:
> directTime := time()-st:
```

F.5 Results

```
> iterativeTime, directTime;
```

$$1.060,\ 252.110$$

```
> SpeedUp := evalf[3](directTime/iterativeTime);
```

$$SpeedUp := 238.$$

```
> # Check computed results.
> check_dir := Norm( Ap.(A.x_dir - b) ):
> check_dir := evalf[3](check_dir);
```

$$check\_dir := 0.108\,10^{-115}$$

```
> check_iter := Norm( Ap.(A.x_iter - b) ):
> check_iter := evalf[3](check_iter);
```

$$check\_iter := 0.179\,10^{-116}$$

```
> r_dir := Norm(A.x_dir - b);
```

$$r\_dir := 0.13\,10^{-110}$$

```
> r_iter := Norm(A.x_iter - b);
```

$$r\_iter := 0.1\,10^{-111}$$

G Polynomial Systems Code

G.1 Procedure DirectNonLinearSolve

```
########
Procedure AssignDigits: a utility routine.
########
```

Purpose: Returns `digits` with `digits = newDigits` except that if `oldDigits` is within hardware precision then `digits` will not exceed hardware precision.

```
> AssignDigits := proc (oldDigits::integer, newDigits::integer)
    local digits;
    digits := newDigits;
    if oldDigits <= evalhf(Digits) and
                                 digits > evalhf(Digits) then
      digits := trunc(evalhf(Digits));
    end if;
    return digits;
  end proc:
```

```
########
Procedure DirectNonLinearSolve.
########
```

Purpose: Direct solution of a nonlinear system of polynomial equations to precision specified by `Digits`.

```
# Parameters:
#   input: p -- vector of polynomials
#          n -- number of polynomials
#          vars -- list of variables in polynomial system
#          hint -- initial guess for the solution, as a set
#                  {x[1] = <float>, x[2] = <float>, ... }
#   output: (x, iterations) with
#          x -- a vector, the high precision solution
#          iterations -- the number of iterations used
```

```
> DirectNonLinearSolve := proc (p::Vector, n::integer,
                    vars::list(name), hint::set(equation))

    local i, x, xseq, f, J0, J, final_Digits, deltax, sqrt_eps,
          Norm_deltax, guard, k, working_prec, r;

    x := Vector(n, (i) -> eval(vars[i], hint));
    xseq := seq(x[i], i=1..n);
    f := unapply(p, vars);
    J0 := linalg[jacobian]([seq(p[i], i=1..n)], vars);
    J := unapply(convert(J0,'Matrix'), vars);

    final_Digits := Digits;
    if final_Digits <= evalhf(Digits) then
      UseHardwareFloats := true
    else
      UseHardwareFloats := false
    end if;
    deltax := x;  sqrt_eps := sqrt(SFloat(5, -final_Digits));
    Norm_deltax := LinearAlgebra:-Norm(deltax);
    guard := 4;  # Number of guard digits

    for k from 1 while Norm_deltax > sqrt_eps do
      working_prec := max(0, -2*ilog10(Norm_deltax^2));
      Digits := AssignDigits(final_Digits, working_prec + guard);
      r := Vector(n, evalf(f(xseq)));
      deltax := LinearAlgebra:-LinearSolve(J(xseq), r);
      Norm_deltax := LinearAlgebra:-Norm(deltax);
      print(`Norm(deltax)` = Norm_deltax);
      x := x - deltax;  xseq := seq(x[i], i=1..n);
    end do;

    return (x, k-1);
  end proc:
```

G.2 Procedure precNonLinearSolve

```
########
Procedure precNonLinearSolve.
########
```

Purpose: Solve a nonlinear system of polynomial equations to high precision via an iteration exploiting hardware floats in each iteration.

```
# Parameters:
#   input: p -- vector of polynomials
#          n -- number of polynomials
#          vars -- list of variables in polynomial system
#          hint -- initial guess for the solution, as a set
#                  {x[1] = <float>, x[2] = <float>, ... }
#          prec -- the required precision
#   output: (x, iter) with
#          x -- a vector, the high precision solution
#          iter -- the number of iterations required to
#                   achieve the desired accuracy.

> precNonLinearSolve := proc (p::Vector, n::integer,
        vars::list(name), hint::set(equation), prec::integer)

    local baseprec, x, iterations, i, xseq, f, J0, J, ipiv,
          r, deltax, eps, Norm_deltax, k;

    # Compute the solution to baseprec.
    baseprec := trunc(evalhf(Digits));
    Digits := baseprec;  UseHardwareFloats := true;
    (x, iterations) := DirectNonLinearSolve(p, n, vars, hint);
    print("solution computed to base precision");

    xseq := seq(x[i], i=1..n);
    f := unapply(p, vars);
    J0 := linalg[jacobian]([seq(p[i], i=1..n)], vars);
    J := unapply(convert(J0,'Matrix'), vars);
    ipiv := LinearAlgebra:-LUDecomposition(J(xseq),
                                           output=['NAG']);
```

```
    deltax := x;  eps := SFloat(5, -prec);
    Norm_deltax := LinearAlgebra:-Norm(deltax);

    for k from 2 while Norm_deltax > eps do
      Digits := k*baseprec;  UseHardwareFloats := false;
      r := Vector(n, evalf(f(xseq)));

      Digits := baseprec;  UseHardwareFloats := true;
      deltax := LinearAlgebra:-LinearSolve([ipiv], r);
      Norm_deltax := LinearAlgebra:-Norm(deltax);
      print(`Norm(deltax)` = Norm_deltax);

      Digits := k*baseprec;  UseHardwareFloats := false;
      x := x - deltax;  xseq := seq(x[i], i=1..n);
    end do;

    return (x, k-1);
  end proc:
```

G.3 Problem setup

```
########
Procedure GeneratePolys: a utility routine.
########
```

Purpose: Generate a random system of n polynomials in n variables, for given n.

```
> GeneratePolys := proc (n::integer)
    local p, i, xlist, k;
    global x;
    p := Vector(n);
    xlist := [seq(x[i], i=1..n)];
    for k to n do
      p[k] := randpoly(xlist);
    end do;
    return p;
  end proc:
```

```
########
Procedure RandomRat: a utility routine.
########
```

Purpose: Generate random rational numbers.

```
> IntLength := 2:
> RandomNum := rand(-10^IntLength .. 10^IntLength):
> RandomDen := rand(1..10^IntLength):
> RandomRat := proc()
    RandomNum()/RandomDen()
  end proc:
```

```
########
Begin problem setup.
########
```

```
> with(LinearAlgebra):
> n := 50;  prec:= 120;
```

$$n := 50$$

$$prec := 120$$

```
> # Generate a random system of n polynomials in n variables.
> # Add constant terms to the polynomials generated so that
> # there is a known solution.
> soln := {seq(x[k] = RandomRat(), k=1..n)}:
> p := GeneratePolys(n):
> for k to n do
>   p[k] := p[k] - eval(p[k], soln)
> end do:
> vars := [seq(x[k], k=1..n)]:
>
> for i from 1 to n do
>   x_exact[i] := eval(vars[i], soln)
> end do:
> x_exact := Vector(n, (i)->x_exact[i]):
>
> # Choose how accurate a hint to give to fsolve.
> hint := evalf[3](soln):
```

G.4 Iterative Method

```
> st := time():
> (x_iter, iter) :=
>             precNonLinearSolve(p, n, vars, hint, prec):
```

$$Norm(deltax) = 0.00559406778213197754$$

$$Norm(deltax) = 0.00829141999910298738$$

$$Norm(deltax) = 0.0000423683774278293988$$

$$Norm(deltax) = 0.384625601384041979\,10^{-8}$$

"solution computed to base precision"

$$Norm(deltax) = 0.888199999999981386\,10^{-13}$$

$$Norm(deltax) = 0.986333333333364779\,10^{-26}$$

$$Norm(deltax) = 0.582009999999938352\,10^{-38}$$

$$Norm(deltax) = 0.616480000000167566\,10^{-51}$$

$$Norm(deltax) = 0.167565999999997060\,10^{-63}$$

$$Norm(deltax) = 0.550363333333444397\,10^{-77}$$

$$Norm(deltax) = 0.613233000000017910\,10^{-89}$$

$$Norm(deltax) = 0.299329999999975222\,10^{-102}$$

$$Norm(deltax) = 0.148838000000010426\,10^{-114}$$

$$Norm(deltax) = 0.104259999999954016\,10^{-127}$$

```
> iterativeTime := time()-st:
> `#Iterations` = iter;
```

$$\#Iterations = 11$$

G.5 Direct Method

```
> Digits := prec;
```

$$Digits := 120$$

```
> st := time():
> (x_dir, iterations) :=
>                DirectNonLinearSolve(p, n, vars, hint):
```

$$Norm(deltax) = 0.005526$$

$$Norm(deltax) = 0.0083234778819211$$

$$Norm(deltax) = 0.000046691834781847$$

$$Norm(deltax) = 0.1674699976488610996235\,10^{-7}$$

$$Norm(deltax) = 0.23510999999999841639164036833411477\,10^{-15}$$

$$Norm(deltax) = 0.158399999999999999999999999995384145390162307504 0\backslash \\ 4547707935197834 1\,10^{-30}$$

$$Norm(deltax) = 0.46158999999999999999999999999999999999999999999\backslash \\ 99999999995320206057383232912478125267161437164787648\backslash \\ 4444882301025846417528 74\,10^{-61}$$

```
> directTime := time()-st:
> `#Iterations` = iterations;
```

$$\#Iterations = 7$$

G.6 Results

```
> directTime, iterativeTime;
```

$$43.550,\ 4.640$$

```
> SpeedUp := evalf[3](directTime/iterativeTime);
```

$$SpeedUp := 9.40$$