The Ax Computer Algebra System

Roman Pearce

axcas.net

Abstract. We describe the design of the Ax computer algebra system located online at `axcas.net`. The system has arbitrary precision integers, rationals, and floating point numbers, with immediate representations for small instances of all three. It automatically switches between sparse and dense vector formats so that sparsity is inherited by matrices and higher dimensional objects. Polynomials are supported as a sum-of-products or a sparse representation with packed exponents. Modular algorithms for polynomials and matrices use word-size primes and run in parallel with preallocated storage. The system includes a Gröbner basis engine which is released into the public domain. Ax has its own memory allocator and garbage collector which we briefly describe. The programming language is similar to C and Javascript, but with automatic memoization. At this point the system should be considered experimental.

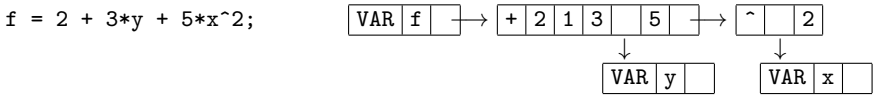
1 Introduction

On the internet at `axcas.net` is the Ax computer algebra system. The purpose of this website is to make interactive computational mathematics available to a wide audience. As such, one does not need to install anything, although binaries are provided. Computations can be performed on the server, and output can be copied or downloaded while input and scripts reside on the user's computer.

The website currently runs on Linux with Apache and ttyd. The system itself is written in C and depends only on the C standard library. It is compiled and tested on a variety of machines to ensure portability and catch bugs. About 15% of its code is released into the public domain.

Ax follows in the footsteps of many other programs by representing objects as *directed acyclic graphs*. In the diagram below, a formula is parsed to create a data structure for a sum of terms. The objects are recursively simplified, i.e. by combining like terms. Then, in an approach pioneered by Maple [4], objects are made unique via hashing to allow fast recognition of identical objects and speed up further simplifications.

Fig. 1. A directed acyclic graph in memory.



1.1 Number Representations

The performance of a system is greatly affected by the memory references it makes while processing large formulae. We have sought to minimize this through the use of *immediate representations*. In the diagram above, the numbers are all immediate while the variables are not.

Ax supports arbitrary precision integers and rational numbers. The integers are stored in base 2^{64} little endian format with the sign stored in the dag header. We elected not to use GMP [8] for the reasons given in Section 1.6.

Rational numbers store the numerator and denominator together in one block of memory that is twice the length of the longer of the two, again with the sign in the dag header. The representation is shown below for $(2^{61} - 1)/(2^{127} - 1)$. This representation has drawbacks: in

Fig. 2. A Multiprecision Rational Number in 64-bit Ax.

`RATPOS 5`	1111...1000	0000..0000	1111..1111	1111..1110
	numerator		denominator	

the worst case we double the storage for each rational number. Polynomials with a common denominator will cut the number of objects in memory by a factor of three. Large denominators usually come from rational reconstruction and the first optimization is to scale them out, so we do not optimize for that case.

Integers and rational numbers have immediate representations. On a 64-bit machine the bottom three bits of a pointer are zero. For a machine word `x`, we summarize what can be found in the remaining bits.

Table 1. Immediate numbers in 64-bit Ax.

(x & 7)	some kind of immediate value
(x & 3) = 2	a signed integer in the top 62 bits (18 decimal digits)
(x & 7) = 4	a rational a/b, a is signed, 30 bits magnitude (9 decimal digits)
(x & 1)	a floating point number with 52 bit mantissa (15 decimal digits)

Notice that the Mersenne prime $2^{61} - 1$ just fits in the integer representation. In our experiments we found that floating point matrix multiplication was often faster than copying floats out of dags, so immediate floats solve a real problem. The 32-bit version of Ax does not have them, so running the test suite should alert us to issues. So far we have not found any serious discrepancies, and we would side with Acton [1] and say that if your numerical algorithm depends on that last bit then you would be better off reformulating the problem.

For the numerically adventurous, Ax supports multiprecision decimal floats. It uses power series to compute most functions, which is reasonable up to 10000 digits. The development of high precision algorithms, such as finding all complex eigenvalues of a large matrix, has been a challenge.

1.2 Vectors and Matrices

Ax has sparse and dense vector representations, shown below, and simplification automatically converts vectors to whichever is shorter. Some algorithms, such as dot product, take advantage of sparsity with no overhead. Others come down to a binary search. Floating point zeroes are elided from the sparse format and the first entry is the total number of columns.

Fig. 3. Sparse and dense vector simplification in Ax.

`VECTOR 5` | 0 | 0 | 0 | 40 $\longrightarrow$ `SPARSE 4` | 4 | 4 | 40

`SPARSE 8` | 4 | 1 | 10 | 2 | 20 | 3 | 30 $\longrightarrow$ `VECTOR 5` | 10 | 20 | 30 | 0

Matrices are recognized by simplification of a vector of row vectors. Ax uses an Iliffe scheme [9] which allows us to refer to A_{ij} as `A[i][j]` in C as long as A is dense. Functions are provided to convert matrices to dense storage or access A_{ij} in general. Our motivation for wanting a sparse matrix format comes from graph theory, where dense storage for the adjacency matrix unduly limits the sizes of graphs. The implementation of sparse algorithms for matrices and graphs is an area of ongoing work.

A major drawback of making vectors unique is that assignment becomes $O(n)$ in the top level language. Similarly, assigning to an m by n matrix is $O(m+n)$. However, we think writing fast

elementwise code in the top level language is not feasible due to loop and dag overhead, so our approach has been to supply high level operations that are coded in C. With a nod to Matlab, Ax supports the backslash operator `A \ b` for solving $Ax = b$ and the transpose operator `A'`.

Integer matrices use modular algorithms, chinese remaindering, and rational reconstruction to achieve good performance. Floating point matrices are handled by double precision code at the default setting of 15 digits. For both $\mathbb{Z}_p$ and $\mathbb{R}$ we parallelized matrix multiplication and Gaussian elimination, producing free parallel speedup in higher level routines. We present timings on a Mac Mini M4 with 10 cores.

Table 2. Free parallel speedup for random dense integer matrices $|A_{ij}| < 10$.

determinant over $\mathbb{Z}$			
	sequential	parallel	speedup
100×100	0.035s	0.033s	1.060x
500×500	3.787s	2.364s	1.602x
1000×1000	66.743s	24.650s	2.707x

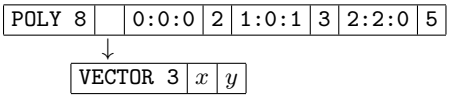
characteristic polynomial over $\mathbb{Z}$			
	sequential	parallel	speedup
50×50	0.101s	0.067s	1.507x
100×100	1.789s	0.494s	3.621x
200×200	64.204s	12.433s	5.164x

The determinant is computed from row reducing images of the matrix in $\mathbb{Z}_p$. For the characteristic polynomial we use LeVerrier's method [2] over $\mathbb{Z}_p$, which is $O(n^4)$. Note that a faster method is given in [5].

1.3 Polynomials

Expanded polynomials are automatically simplified to the POLY representation developed for Maple by Monagan and Pearce [12]. It is a sparse distributed format with the variables stored in the first word and exponent vectors with the total degree packed into machine words. The purpose of this structure is to compress large polynomials, eliminate memory references, and provide fast algorithms for common operations such as computing the degree or extracting coefficients of a variable. The structure is shown in Figure 4.

Fig. 4. The POLY representation for $2 + 3y + 5x^2$.



There are several differences from Maple. In Ax, POLY can hold any numeric coefficient whereas Maple restricts them to integers. This is slower as arithmetic has to check types. Terms are sorted in ascending order in Ax, and extra bits in the degree field are not used, so the maximum total degree equals the maximum exponent for a given number of variables. We implemented the heap algorithms for sparse polynomial multiplication and division from [11].

Two optimizations are critical for performance. First, all products should be extracted from the heap before their successors are inserted. Second, the heap should create *chains* of terms with equal monomials. We show the effect of these optimizations in Maple and Fateman's benchmarks [6] on the Mac Mini M4.

One remaining problem is Maple automatically switches to Kronecker subsitution backed by GMP. While we have an FFT-based integer multiplication routine, integer division is still classical and we have not implemented Kronecker substitution. Adding fast routines for large integers and polynomials is a short term goal.

1.4 Programming Language

The language in Ax is strictly imperative and looks like C or Javascript. Testing types is done explicitly with the type command. The language is not intended to support a large codebase,

Table 3. Sparse polynomial multiplication and division benchmark.

$f = (1+x+y+z)^{20}$	chained heap	naive heap	no POLY
`f = expand(f);`	0.009	0.011s	0.024s
`g = expand(f*(f+1));`	0.068	0.142s	0.533s
`divide(g,f);`	0.041	0.152s	0.521s
$f = (1+x+y+z+t)^{20}$	chained heap	naive heap	no POLY
`f = expand(f);`	0.032	0.036s	0.082s
`g = expand(f*(f+1));`	1.515	4.995s	26.910s
`divide(g,f);`	1.959	5.344s	32.547s

but rather to enable quick experiments. The binaries we distribute can read code from a file, but the website currently can not.

```
function euclid(a,b)
{
    var r;
    while (b != 0) {
        r = a % b;
        a = b, b = r;
    }
    return a;
}
```

```
function collatz(n)
{
    if (n < 2) return 0;
    if (n % 2) return collatz(3*n+1)+1;
    return collatz(n/2)+1;
}
```

We benchmark the interpreter with the Collatz function above. Ax enables function memoization by default, which produces a significant gain for this case. Without memoization we can see that the interpreter is not particularly fast. We think that automatic memoization is a valuable feature for a computer algebra system since most objects are already immutable.

Table 4. Interpreter benchmark of the sum of the first 10^7 Collatz numbers.

language	C	Ax	Ax	Maple	Maple	python3	ruby
memoization	no	yes	disabled	yes	no	no	no
time	0.140s	1.030s	145.441s	20.369s	366.594s	9.201s	3.974s

Like most computer algebra systems Ax provides its own memory allocator and garbage collector. Small dags up to 2772 words are allocated from freelists, with one list for each size. If an allocation can not be met by its list, a block of size $27720 = lcm(2, 3, 4, 5, \ldots, 16)$ is allocated and added to the list. Larger dags use malloc and free. Ax does not try to release memory from small dags back to the operating system, however this may change in the future.

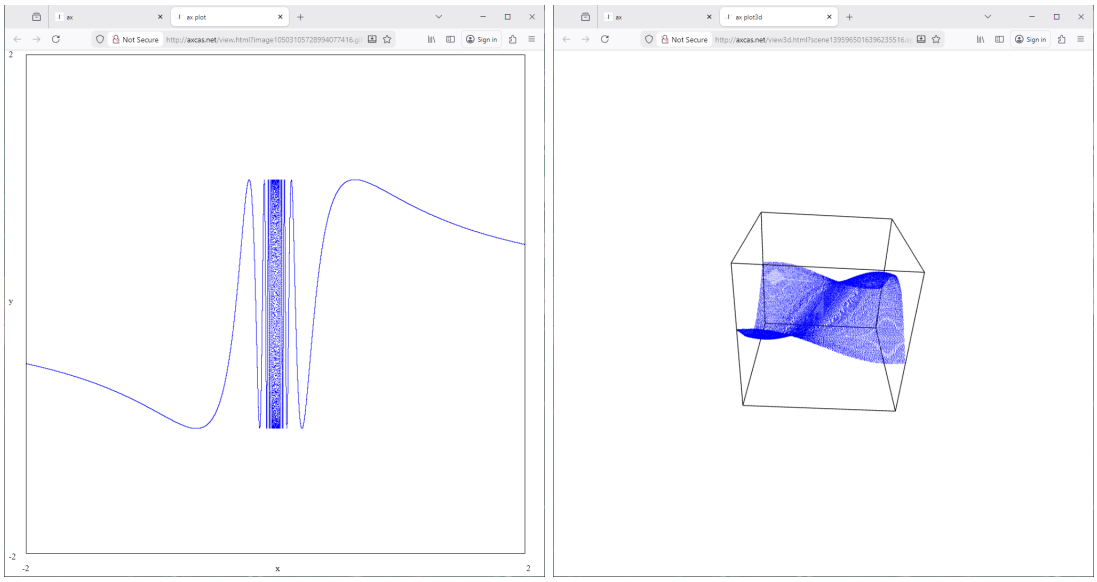
Ax uses a mark-sweep garbage collector that first frees all unsimplified dags. This means that garbage collection can only occur at the end of each statement. That is why Ax is faster than Maple on the Collatz benchmark. Collection runs after the table of simplified dags has grown by at least 20%. Ax prints any output before collection is invoked to make the system feel snappy.

1.5 Plotting

Plotting in Ax creates a file on the server which is viewed through a link. In the case of a 2D plot, Ax creates a .gif or animated .gif which is viewed in HTML. For 3D plots, the set of points is written to a file which is loaded by Javascript and viewed in WebGL. In the 2D case the image can be saved by the browser.

The algorithms for plotting in Ax use oversampling to create the illusion of a solid line or surface from sample points. There is no attempt to draw a line or a surface. We think

Fig. 5. 2D and 3D plots in Ax.



this embarrassingly parallel approach looks better when the function being plotted breaks the method, as in the plot of $\sin(1/x)$ as $x \to 0$. For 3D plots, the viewer puts a bit of space between each sample point to create some transparency which helps define the shape. One bit of ongoing work is to animate solutions to ordinary and partial differential equations.

1.6 Portability

The binaries distributed on the website support seven 64-bit platforms:

```
ax.exe          Windows x64 (Intel/AMD)
axarm.exe       Windows Arm64 (Qualcomm/Nvidia)
ax.mac          Mac Arm64 (Apple)
ax.macx64       Mac x64 (Intel)
ax              Linux x64 (Intel/AMD)
axarm           Linux Arm64 (Qualcomm/Nvidia)
axrv            Linux Risc-V (SiFive/Sophgo/Ky)
```

Ax can be optimized by implementing up to four functions with assembly or intrinsics, the most important of which is umul2. For modular algorithms, reduction mod p uses Möller and Granlund's 2/1 reciprocal division [10].

```
lo = umul2(a,b,&hi);       for a*b = (lo:hi)
q  = udiv2(lo,hi,v,&r);    for (lo:hi)/v = (q,r)
c  = uadd2(&lo,&hi,a,b);   for (lo:hi) += (a:b) returning a carry
c  = usub2(&lo,&hi,a,b);   for (lo:hi) -= (a:b) returning a borrow
```

Here is an example of how these functions are used in the Ax kernel:

```
/* array c = a * b classical */
INT int_mul_small(UINT *a, UINT *b, UINT *c, INT na, INT nb)
{
        INT i, j, k;
        UINT lo, hi, t0, t1, t2;
        t0 = t1 = 0;
        for (k=0; k <= na+nb; k++) {
```

```
                i = k-nb; if (i < 0) i=0;
                j = na;   if (j > k) j=k;
                for (t2=0; i <= j; i++) {
                        lo = umul2(a[i],b[k-i],&hi);
                        t2 += uadd2(&t0,&t1,lo,hi);
                }
                c[k] = t0; t0 = t1; t1 = t2;
        }
        c[k] = t0;
        while (k && c[k]==0) k--;
        return k;
}
```

So why not use GMP? GMP supports essentially one toolchain (gcc+m4) and optimizes for many specific CPUs. We support multiple toolchains, such as MSVC on Windows, and optimize for CPU architectures. We recently released binaries for Windows on Arm64 and it took only a few minutes to get everything working and optimized. The result is not as fast as GMP, but for small integers there is very low overhead because of function inlining.

There is also the problem of shipping a binary. GMP is LGPL and can only be integrated into a GPL2 binary. We would have to ship the library and make an installer for all platforms, and on some platforms we would have to sign the code for each release and pay an annual fee for the certificates.

1.7 Gröbner Bases

Ax began as a wrapper for a Gröbner basis engine with the goal of developing a polynomial system solver. The Gröbner engine has now been released into the public domain while work on the solver continues. Below we give an example Gröbner basis computation in Ax. For simplicity, the `groebnerbasis` command uses degree reverse lexicographical order, while `fglm` or `groebnerwalk` convert the result to lexicographical order to eliminate variables.

```
sys = [x^2-y, x*y-1];
groebnerbasis(sys,[x,y]);
        [-x+y^2, -1+x*y, -y+x^2]        @0
hilbertdimension(@0,[x,y]);
        0       @1
fglm(@0,[x,y]);
        [-1+y^3, x-y^2]         @2
```

The example shows another feature of Ax, which is that results not assigned to variables are automatically given equation labels (@0, @1, etc). There is also an `info` command which provides runtime information from various algorithms. Below we share some timings for the F_4 algorithm [7] on the Mac Mini M4. The performance is roughly on par with Magma [3] when both run in parallel.

Table 5. Timings and parallel speedup for grevlex Gröbner bases mod $p = 2^{61} - 1$.

$p = 2^{61} - 1$	cyclic-8	cyclic-9	katsura-12	eco-12	gametwo	noon-9	reimer-9
parallel	0.323s	13.266s	7.426s	0.627s	2.202s	4.569s	3.548s
sequential	1.026s	58.936s	34.247s	1.705s	9.297s	6.795s	13.866s
speedup	3.176x	4.442x	4.611x	2.719x	4.222x	1.487x	3.908x

1.8 Conclusion

There is still a lot of work needed to make Ax a broadly useful system, and we continue to implement and experiment with many things. We hope to report back in the future with new developments.

References

1. F. S. Acton. *Real computing made real: Preventing Errors in Scientific and Engineering calculations*. Courier Corporation, 2013.
2. S. Barnett. Leverrier's algorithm: a new proof and extensions. *SIAM Journal on Matrix Analysis and Applications*, 10(4):551–556, 1989.
3. W. Bosma, J. Cannon, and C. Playoust. The magma algebra system i: The user language. *Journal of Symbolic Computation*, 24(3-4):235–265, 1997.
4. B. W. Char, K. O. Geddes, W. M. Gentleman, and G. H. Gonnet. The design of maple: A compact, portable, and powerful computer algebra system. In *European Conference on Computer Algebra*, pages 101–115. Springer, 1983.
5. H. Cohen. *A course in computational algebraic number theory*. Springer Science & Business Media, 2013.
6. R. Fateman. Comparing the speed of programs for sparse polynomial multiplication. *ACM SIGSAM Bulletin*, 37(1):4–15, 2003.
7. J.-C. Faugere. A new efficient algorithm for computing gröbner bases (f4). *Journal of pure and applied algebra*, 139(1-3):61–88, 1999.
8. T. Granlund. Gnu mp. *The GNU Multiple Precision Arithmetic Library*, 2(2), 1996.
9. J. K. Iliffe. The use of the genie system in numerical calculations. *Annual Review in Automatic Programming*, 2, 1961.
10. N. Möller and T. Granlund. Improved division by invariant integers. *IEEE Transactions on Computers*, 60(2):165–175, 2010.
11. M. Monagan and R. Pearce. Polynomial division using dynamic arrays, heaps, and packed exponent vectors. In *International Workshop on Computer Algebra in Scientific Computing*, pages 295–315. Springer, 2007.
12. M. Monagan and R. Pearce. The design of maple's sum-of-products and poly data structures for representing mathematical objects. *ACM Communications in Computer Algebra*, 48(3/4):166–186, 2015.