The Ax Computer Algebra System

Roman Pearce

axcas.net

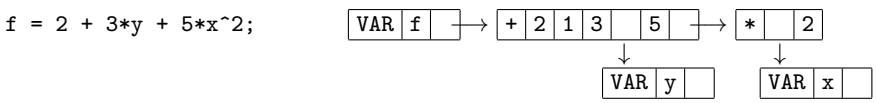
Abstract. We describe the design of the Ax computer algebra system located online at `axcas.net`. The system has arbitrary precision integers, rationals, and floating point numbers, with immediate representations for small instances of all three. It automatically simplifies vectors to a sparse format, which is inherited by matrices and higher dimensional objects. Polynomials are supported with a sum-of-products data structure which is simplified to a sparse representation with packed exponents. Ax has modular algorithms for polynomials and matrices based on 63-bit primes, including a high performance Gröbner basis engine which is released into the public domain. The system has a mark-and-sweep garbage collector and allocates data from slabs for dags of particular sizes. The 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 long term goal of this website is to make interactive computational mathematics available to a wide audience. To that end, there is no program to compile or install, all computations are performed on the server. 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 [2], 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.

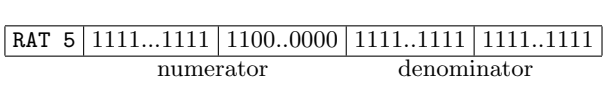


1.1 Number Representations

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. 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 $(2^{64+3})/(2^{128-1})$. This representation has drawbacks: in the worst case we double the storage for each rational number. Polynomials with a common denominator will do this T times. And large common denominators will be expensive to recognize.

Fig. 2. A Multiprecision Rational Number in Ax.



So why choose this representation? In the case of unique denominators, such as a power series, we cut the number of objects in memory by a factor of three. Large common denominators usually come from rational reconstruction and the first optimization is to scale them out, so we did not prioritize 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 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)

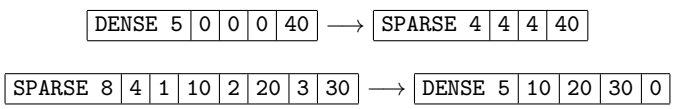
Notice that the Mersenne prime $2^{61}-1$ just fits in the integer representation. In our experiments we found that floating point matrix multiplication was often faster than copying floats out of dags, so immediate floats solve a real problem. The 32-bit version of Ax does not have them, so running the test suite alerts us to numerical defects. So far we have not experienced any serious discrepancies.

For the naive numerical explorer, Ax supports multiprecision decimal floats. It uses power series to compute most functions, and the system limits them to 10000 digits. The development of high precision algorithms, such as finding all the complex eigenvalues of a large matrix, has been challenging.

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.



Matrices are recognized by simplification of a vector of row vectors, producing an Iliffe scheme [3], which allows us to refer to A_{ij} as `A[i][j]` in C provided the matrix is dense. In general there may be sparse rows in the matrix so functions are provided in the kernel to access $A_{i,j}$ or convert the matrix to dense storage. The implementation of sparse matrix and graph algorithms is ongoing work.

One major drawback of simplifying vectors is that assignment becomes $O(n)$ in the top level language. Similarly assigning to an m by n matrix is $O(m+n)$. We don't worry about this because writing fast elementwise code outside of the kernel is not realistic. There is too much loop and dag overhead. Our approach is to supply fast high level operations that are coded in C.

Integer matrices use modular algorithms, chinese remaindering, and rational reconstruction to achieve good performance. Our portable code supports 63-bit primes on 64-bit platforms, and the code is in the public domain. Floating point matrices are recognized and treated with machine code at the default setting of 15 digits. These algorithms are dense and optimized for throughput.

With a nod to Matlab, we support the backslash operator `A \ b` for solving linear systems and the transpose operator `A'`. These operators are essential for writing high level linear algebra code.

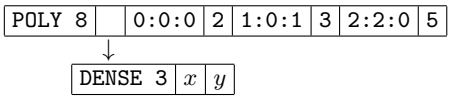
1.3 Polynomials

Polynomials in Ax are automatically simplified to the POLY representation [4] developed for Maple by Monagan and Pearce. 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.

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 [5].

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



1.4 Parallelism

Our approach to parallelism in Ax is to spawn threads in low level routines with preallocated storage, like matrix multiplication and row reduction mod $p < 2^{63}$. The result is free parallel speedup in modular algorithms with no extra memory. We present timings on a Mac Mini M4 with 10 cores.

Table 2. Free parallel speedup for linear algebra over $\mathbb{Z}$ (dense).

characteristic polynomial over $\mathbb{Z}$			
	sequential	parallel	speedup
50×50	0.128s	0.125s	1.0x
100×100	3.105s	0.998s	3.111x
200×200	114.605s	27.370s	4.187x

minimal polynomial over $\mathbb{Z}$			
	sequential	parallel	speedup
50×50	0.201s	0.209s	1.0x
100×100	5.788s	3.640s	1.590x
200×200	218.593s	131.004s	1.668x

By working at a low level, we can divide the work among threads optimally. Other cases, such as computing Gröbner bases, call for lock free algorithms that modify data structures in parallel. That is the subject of another paper, but we present some timings on the Mac Mini which are new.

Table 3. Parallel speedup of grevlex Gröbner bases mod p.

$p = 2^{31} - 1$	cyclic-8	cyclic-9	katsura-12	eco-12	gametwo	noon-9
sequential	0.660s	32.672s	13.750s	1.069s	4.670s	7.594s
parallel	0.275s	8.609s	3.080s	0.464s	1.269s	4.883s
speedup	2.400x	3.795x	4.464x	2.300x	3.680x	1.555x

1.5 Memory Management

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.

As 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 when Ax is waiting for input. Ax prints any output before gc is invoked to make the system feel snappy. Garbage collection runs after the table of simplified dags has grown by over 20%.

1.6 Programming Language

The language in Ax is strictly imperative, and looks like C or Javascript. Type testing is explicit with the type command. The language is not really intended to support a large codebase, but rather to enable quick experiments.

```
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, again on the Mac, but with memoization toggled. Ax has a hash table of function calls and results which is cleared on gc. The interpreter is not very fast by itself as loop variables are dags, but automatic memoization makes it tolerable in many cases.

```
T = time();
for (s=0, i=0; i < 1000000; i+=1) s += collatz(i);
time()-T;
```

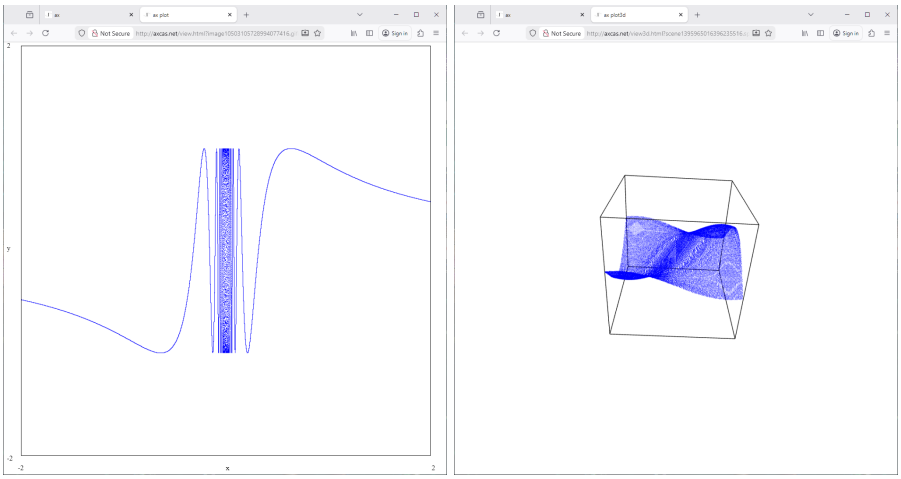
Table 4. Interpreter benchmark with automatic memoization.

language	C	Ax	Ax	python3	ruby
memoization	no	yes	no	no	no
time	0.140s	1.030s	145.441s	9.201s	3.974s

1.7 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, a set of points is written to a file which is loaded by a Javascript and viewed with WebGL. In both cases the file can be downloaded and saved.

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



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 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.

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. For now we only hope you've found this interesting.

References

1. Forman Acton. Real Computing Made Real: Preventing Errors in Scientific and Engineering Calculations. Dover, 2005.
2. Bruce W. Char, Keith O. Geddes, W. Morven Gentleman, Gaston H. Gonnet (1983). The Design of Maple: A Compact, Portable, and Powerful Computer Algebra System. Proceedings of EUROCAL '83.

3. John K. Iliffe (1961). The Use Of The Genie System in Numerical Calculations. Annual Review in Automatic Programming 2:25.
4. Michael B. Monagan, Roman M. Pearce (2014). The design of Maple's sum-of-products and POLY data structures for representing mathematical objects. Communications in Computer Algebra, 48(4).
5. Michael B. Monagan, Roman Pearce (2007). Polynomial Division Using Dynamic Arrays, Heaps, and Packed Exponent Vectors. Proc. CASC 2007, 295–315.