# CAMAL 40 Years on – Is Small Still Beautiful?

John Fitch

Department of Computer Science, University of Bath, UK

**Abstract.** Over forty years ago an algebra system was written in Cambridge, UK, designed to assist in a number of calculations in celestial mechanics and later in relativity. I present the hardware environment and the main design decisions that led this system, later dubbed CAMAL, to be used in many applications for twenty years. Its performance is investigated, both in its own era, and more recently. It is argued that a compact data representation as in CAMAL has real benefits even in today's larger memory world.

## 1 Introduction

This paper considers a period in my early academic career, when not only was computer algebra a novel idea but the idea of having a computer at all was close to fantasy. In 1964 I started as an undergraduate at the University of Cambridge, and my first encounter in the first week with the teaching staff was with David Barton, a research student, who was to supervise me in applied mathematics for that term. I still remember that hour with a mixture of amazement and embarrassment at my stupidity. Four years later, having survived a degree and a postgraduate qualification in Computing, I started working for and later with David Barton on his algebra system, which was at that time unnamed.

I wish to impart some of the flavour of that system, why it was written and how it worked. The system survived for over twenty years in some form or other, but I wish to concentrate on the original three systems, and to give some benchmark figures for the performance of the system.

## 2 The Original Problem and System

David Barton was a research student at the Institute of Theoretical Astronomy, University of Cambridge. His area of interest was celestial mechanics, and the algebra system came into existence in order to solve one problem, to determine the orbit of the moon[2].

Charles Delaunay[7] published the methodology and results of a major hand-calculation. The two-volume book produced an algebraic expression for the orbit of the Moon round the Earth, as perturbed by the Sun in its orbit. The basic methodology was to consider the orbit as an instantaneous ellipse as explained by Newton's laws of motion, but the ellipse will evolve under the influence of the distant Sun (see figure 1).

J. Carette et al. (Eds.): Calculemus/MKM 2009, LNAI 5625, pp. 32–44, 2009.

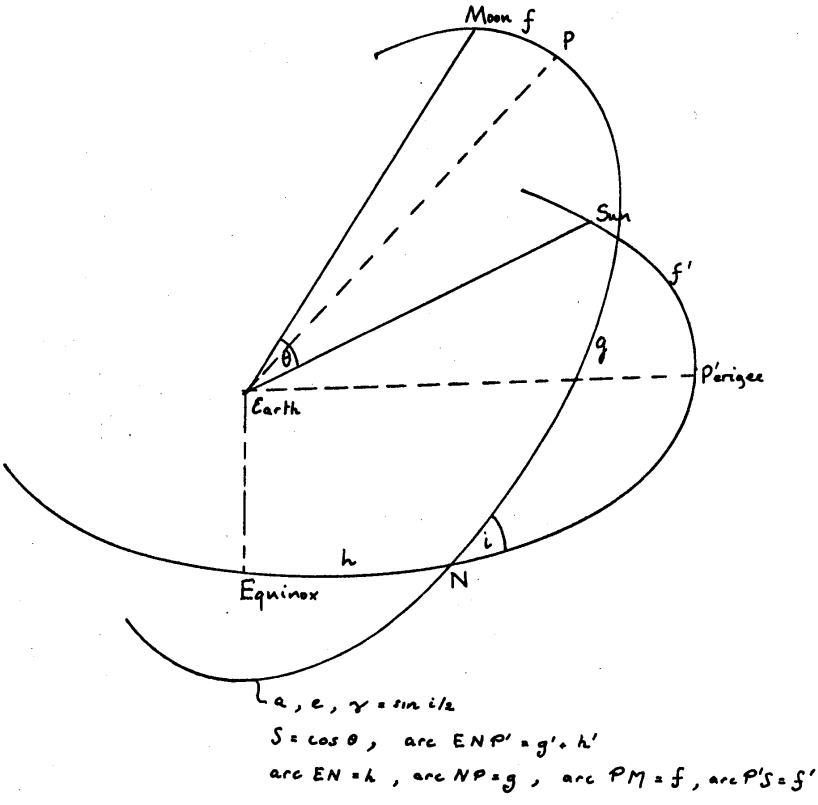


**Fig. 1.** The Lunar Theory of Delaunay

The problem is posed in terms of 6 variables and 6 angles, and all the variables are "small"; that is the solution is developed as an approximation[1] expanded to a certain level in variables like the eccentricity of the orbits, the ratio of the distances Earth-Moon to the Earth-Sun, and so on. This problem leads to a number of design decisions that at the time were obvious.

All expressions were Fourier series whose coefficients are polynomials in a small fixed number of variables over the rational numbers. The arguments to the sine and cosine functions are linear sums of the six angles. The general form of all expressions is thus

$$\sum P(a,b,c,d,e,f,g,h) \begin{matrix}\sin\\ \cos\end{matrix}\left(\lambda u + \mu v + .. + \gamma z\right)$$

### 2.1 Hardware Base

The University of Cambridge at that time had an Atlas2 computer, also known as Titan[15]. This computer had a 48 bit word which could hold a floating point

[1] After all the 3-body problem is not solvable in gravitation.

number, an instruction, or two 24 bit half-words. The memory was addressed in words by the top 21 bits of a halfword, or in half-words by the top 22 bits. Conceptually a word could also hold 8 six-bit characters, but there was little hardware support for this. The design would probably nowadays be described as three-address, as most instructions had two registers and an address, and the machine was RISC-like, having 128 registers (mainly 24bit); there was also a floating point accumulator.

Titan initially had 64K words of memory (that is 1.5 megabits), although this was soon extended to 128K words. The memory had a cycle time of about 4 $\mu$secs[2], and the second bank was a little faster. There was in addition a 32 word "slave store" which was an instruction cache, working at 300 nanosecond cycle time, although this hardware was often turned off as it was unreliable.

Memory was divided into blocks of 512 words, and this was the basic unit of allocation. Programs had to be in contiguous memory, and there were other restrictions due to the use of an OR rather than a ADD in the base register system, but they do not concern us regarding the algebra system.

To the programmers at the time the real delight of Titan was its B-registers; there where 88 general purpose registers, together with zero in register 0, and register 90 was the subroutine return address. Registers 119-122 were special, and the user's program counter was in register 127. What this means is that programming in assembler was fun and offered many opportunities for optimisation. This was good, but the programming languages available were less so. There was Titan Autocode, a version of Fortran, and a promise of CPL.

The original algebra system was written in the commonest assembler, IIT (Intermediate Input for Titan).

### 2.2 A Polynomial

The polynomials of the system were held as packed structures. There are only 8 variables, and as the application was approximation, there is no need for exponents to be larger than 31 – there was no hope of that accurate an approximation for the lunar orbit, 10 or 12 being the aim. This remark shows that all variables would be held in one 48 bit word, allowing 5 bits for the exponent and a guard bit to check for exponent overflow. The allocation is fixed, and every term in the polynomial will have a representation for all 8 variables. The advantage in algorithmic terms was that the multiplication of two terms could be achieved using two halfword additions[3], with a mask to check overflow.

The coefficients were held as two integers, in a word each. This was extended later as described below (section 3). Perhaps the only innovation in this section was that the rational was not reduced to lowest terms unless it was being printed or it was about to overflow.

The last part of polynomial design is the list of terms, ordered in increasing total order, that is, the sum of the exponents. By maintaining polynomials in

[2] The manual says 2.25 but that was not achieved.

[3] Integers were always in halfwords, and words were only for instructions and floating point.

increasing maximum order it is possible to truncate the multiplication of two polynomials when the terms get too small for the degree of approximation. Assume that we are multiplying the two polynomials

$$A = a_0 + a_1 + a_2 + \ldots$$
$$B = b_0 + b_1 + b_2 + \ldots$$

If we are generating the partial answer

$$a_i(b_0 + b_1 + b_2 + \ldots)$$

then if for some $j$ the product $a_i b_j$ vanishes, then so will all products $a_i b_k$ for $k > j$. This means that the later terms need not be generated. In the product of $1 + x + x^2 + x^3 + \ldots + x^{10}$ and $1 + y + y^2 + y^3 + \ldots + y^{10}$ to a total order of 10 instead of generating 100 term products only 55 are needed. The ordering can also make the merging of the new terms into the answer easier.

### 2.3 A Fourier Series

The Fourier series part was similarly held as a packed structure with all 6 angles having multipliers packed into the structure. With 8-bit fields there is space for a signed number and a guard bit. The rest of the structure is made from two half-word pointers, one to the polynomial coefficient and one for the primary sum chain. The difference between a sine and cosine term was coded in a single bit on the polynomial pointer; remember that on Titan the bottom 2 bits are ignored on memory access. This is shown in figure 2.

The algorithms for performing the algebra are straightforward for addition, multiplication, differentiation and restricted integration. The product of the Fourier terms are subjected to the linearisation rules.

$$\cos\theta \cos\phi \Rightarrow (\cos(\theta + \phi) + \cos(\theta - \phi))/2,$$

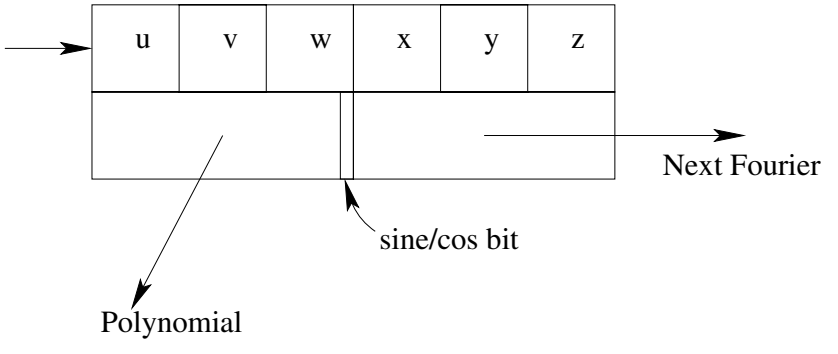


**Fig. 2.** The Fourier Representation

$$\begin{aligned}
\cos\theta\sin\phi &\Rightarrow (\sin(\theta+\phi)-\sin(\theta-\phi))/2,\\
\sin\theta\sin\phi &\Rightarrow (\cos(\theta-\phi)-\cos(\theta+\phi))/2,\\
\cos^2\theta &\Rightarrow (1+\cos(2\theta))/2,\\
\sin^2\theta &\Rightarrow (1-\cos(2\theta))/2.
\end{aligned}$$

Substitution of one Fourier series into another is a more complex operation, but it relies on the approximation mechanism:

$$\begin{aligned}
\sin(\theta+A) \Rightarrow \sin(\theta)\{1-A^2/2!+A^4/4!\ldots\}+\\
\cos(\theta)\{A-A^3/3!+A^5/5!\ldots\}\\
\cos(\theta+A) \Rightarrow \cos(\theta)\{1-A^2/2!+A^4/4!\ldots\}-\\
\sin(\theta)\{A-A^3/3!+A^5/5!\ldots\}
\end{aligned}$$

The actual coding of the operation was not as expressed above, but by the use of Taylor's theorem. It should be noted that the differentiation of a harmonic series is particularly easy.

## 3 The Bourne-Again CAMAL

Steve Bourne's PhD work was on the Hill formulation of the Lunar theory[3]. This is similar to the Delaunay theory except that it uses a Cartesian coordinate system, and needs more variables and also complex coefficients. In addition some of the coefficients got larger. In order to accommodate these changes Bourne rewrote the system, still in assembler, to add an extended rational coefficient with numerator and denominator up to $2^{76}-1$, a possibility for overflow to floating point, and four more variables, $i$ through $l$. The variable $i$ was the complex number and replacement of $i^2$ by -1 was managed internally always. The final polynomial structure is shown in figure 3.

The other innovation for this second incarnation was the production of a programming language. Prior to this programs were sketched in an informal language and then coded in assembler. The new language[1] was very similar to Titan Autocode, except that the lowercase letters were algebraic variables and angles, and programming variables were single uppercase letters, with A-H and U-Z being algebraic type and I-T being integer. Multiplication was by juxtaposition as in Autocode and there was a looping construct and jumps. There were no subroutines as commonly understood now, but the form → 60 → could be used to jump to label 60, and store a return link on a stack. The compiler was implemented in a syntax-directed compiler system called Psyco[16,18]

### 3.1 Memory

The other innovation that was incorporated in the compiler was related to memory allocation. As is appropriate for list processing on a small memory

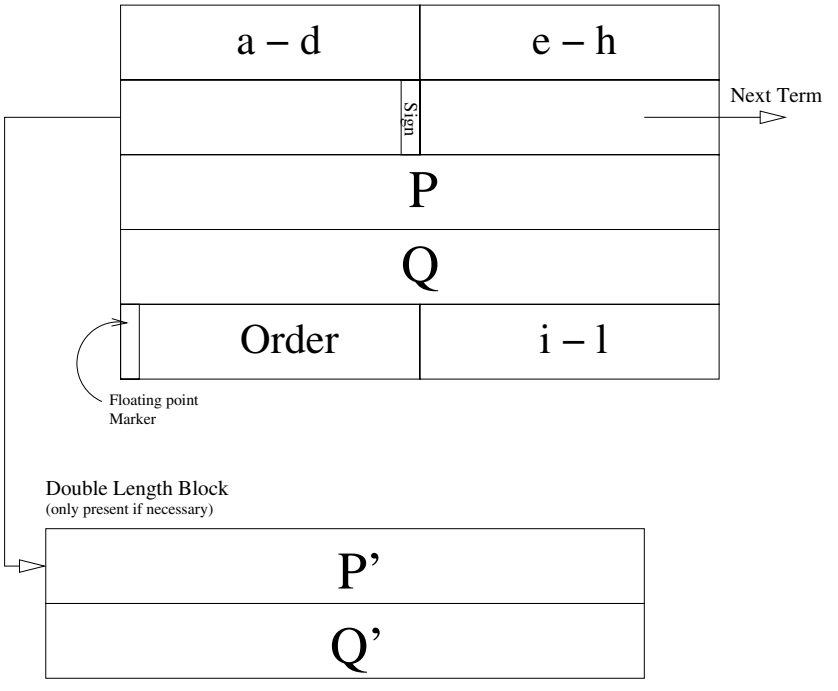


**Fig. 3.** The Polynomial Term

machine[10] the system worked with an explicit return system. With two sizes for allocation the mechanism was simply two free-chains. However the important feature was the division of operations into those that "naturally" destroyed their arguments and those that did not.

Consider the addition of two polynomials; this is a merge of two chains, and so is a simple example of the first kind of operation. Others include polynomial differentiation and integration. The operation of polynomial multiplication on the other hand does not consume its arguments. The compiler ensured that the necessary copying of structures was made before destructive transformations, *if necessary*. If the operation was then the first argument must be copied but

```
A = B + 5abs
```

not the second. More importantly the user language allowed a colon to follow a variable to indicate that it was not required after use, and the compiler could either not copy or insert a lose call as appropriate. This mechanism, used carefully, could save much memory and allow calculations to complete despite the memory limitations of Titan.

Another feature of the language scheme that was developed over time was to compile into a half-word encoding which was interpreted at run time. The

argument at the time was that the basic operations took much longer that the time in the user program. Later a mixed system was used, especially in the third version.

## 4 Getting the Hump

The work of my PhD was largely in relativity, gravity waves and Einstein space[12]. For this neither the polynomials nor Fourier series were sufficient and so the third algebra system in Cambridge was constructed. The system handled sums and products of elementary functions, with possibly nested arguments, but used the polynomial component of the earlier system as a subsystem. In this way it inherited many of the compact store attributes while providing a faster implementation base.

This code, later called CAMAL(H) or Hump, was not written in assembler, but in a language like the user language, which was compiled into assembler for integer and pointer operations, and into interpreter half-words for polynomial algebra. The compiler was again written in Psyco. This system also introduced use-counts for memory structures and a hashing system to save memory on repeated polynomials.

## 5 Overall Structure

At this stage in the development of the algebra system we had a collection of systems, polynomial, Fourier, elementary functions and a tensor package (written in the same way as Hump, see [14]), and these could be used together in various ways. A small (1024 word) control program handled space, stacks, and the interpreter, and each component supplied a *jump table* to the functions, in a pre-defined order. In some ways this pre-echoes the internal structure of Scratchpad/2 much later. It was this system that was first named The Cambridge Algebra System, abbreviated to CAMAL[4,5]. The sizes of the code for various configurations is given in table 1.

**Table 1.** Sizes of Various CAMAL Systems in Blocks(512 words)

| | | |
|---|---|---|
| Polynomial System | 2 + 5 | = 7 |
| Fourier System | 2 + 7 | = 9 |
| Elementary Functions System | 2 + 4 + 12 | = 18 |
| Fourier Tensors System | 2 + 7 + 7 | = 16 |
| Elementary Function Tensor System | 2 + 4 + 12 + 7 | = 25 |

The last system described there used three levels of code, linked via interpretation. There are few extant examples of this particular scheme but it was operational.

## 6 Subsequent History

The CAMAL system, as it was now called, continued for about twenty years as an application tool. Titan was turned off in 1973, and replaced by an IBM370/165. CAMAL was rewritten, first in a dialect of ALGOL68 and then in BCPL[19]. The assembler sections were hand recoded, and the components written in the CAMAL language were compiled to BCPL, again with Psyco. The algorithms and data structures were largely the same, except that the opportunity was taken to allow arbitrary numbers of polynomial variables, fixed for any run, and to allow similar licence for the maximum exponents. The masks and shifts were computed at initialisation time, but the fundamental algorithms were the same. Also arbitrary (unfixed) precision coefficients were introduced, and some hand adjustment of the automatic BCPL code made.

This incarnation was later ported to other architectures, including SUN workstations and VAXen. It fell out of maintenence in the late 1980s, although there was a working version reconstructed in 1999 using an automated BCPL to C translator, which when compared to REDUCE on the same computer showed remarkable speed.

## 7 Performance

In this paper so far the concentration has been on the design and data structures. This is only of interest if we also can see how this translates into performance.

In the 1970s there was a flurry of small benchmarks that were used in comparing the variety of algebra systems then in existence. The first of these was to calculate the f and g series.

### 7.1 f and g

The f and g series arise in the solution of orbits, and were first proposed as an algebra benchmark by Sconzo *et al.*[20] in FORMAC. It was widely used for a few years. The series are defined by simple recurrence relations:

$$\begin{aligned}
\dot{\mu} &= -3\mu\sigma \\
\dot{\sigma} &= \epsilon - 2\sigma \\
\dot{\epsilon} &= -\sigma(\mu + 2\epsilon) \\
f_n &= \dot{f}_{n-1} - \mu g_{n-1} \\
g_n &= f_{n-1} + \dot{g}_{n-1} \\
f_1 &= 1 \\
g_1 &= 0
\end{aligned}$$

A CAMAL program to calculate this is shown in figure 2.

A table of reported timings is shown in table 3 from about 1973. It is hard to get complete comparisons, but note that the storage requirement of CAMAL is small and the time is short. CAMAL was noted at the time for its speed, but that was never the design. We held to the mantra that memory was finite

**Table 2.** CAMAL Program for f & g series

```
F[19]; G[19]

F[0] = 1; G[0] = 0; U = -3ab; V = c-2bb; W = -b(a+2c)

FOR N=1:1:19
        F[N] = UdF[N-1]/da + VdF[N-1]/db + WdF[N-1]/dc - aG[N-1]
        G[N] = UdG[N-1]/da + VdG[N-1]/db + WdG[N-1]/dc + F[N-1]
REPEAT

PRINT[F[19]]; PRINT[G[19]]
PRINT[TIME]
STOP
END
```

**Table 3.** 1973 Comparisons for f & g series

| System | Computer | Word | Cycle | × time | Order | Time | Memory |
|---|---|---|---|---|---|---|---|
| ALTRAN | GE 625/635 | 36 | 1.5 | 6 | 19 | 158 | 51K |
| CAMAL | Titan | 48 | 4 | 7 | 19 | 6.4 | 3.8K |
| CLAM | CDC 6600 | 60 | 0.8 | 1 | 15 | 10.6 | 30K |
| FORMAC | IBM 7094 | 32 | 2 | 10 | 12 | 58.2 | ?? |
| Korsvald | IBM 7094 | 32 | 2 | 10 | 12 | 178.2 | ?? |
| MATHLAB | PDP 10 | 36 | 1 | 10 | 12 | 20 | ?? |
| PM | IBM 7094 | 32 | 2 | 10 | 27 | 105 | ?? |
| REDUCE | PDP 10 | 36 | 1 | 10 | 10 | 68 | 38K |
| SAC-1 | CDC 1604 | 48 | 4.8 | 36 | 12 | 75.9 | 21 |
| SAC-1/Asm | CDC 1604 | 48 | 4.8 | 36 | 12 | 38.5 | 21 |

but time infinite[4] and we were pleased by the Memory column of this table. To assist in interpretation the graph of figure 4 shows the growth graphs in time in seconds and memory on 100s of words.

At the 1971 SYMSAC/2 conference there were attempts to run this program on all systems being demonstrated; unfortunately some crashed or apparently looped[5].

For comparison, a recent investigation of the f and g series using contemporary algebra systems and hardware can be found in [6].

### 7.2 Legendre Polynomials

Another simple benchmark is the calculation of the Legendre Polynomials, which can be defined by the relations

[4] although on Titan the time between failures was quite short!
[5] The same was true of a much hyped system in the 1980s.

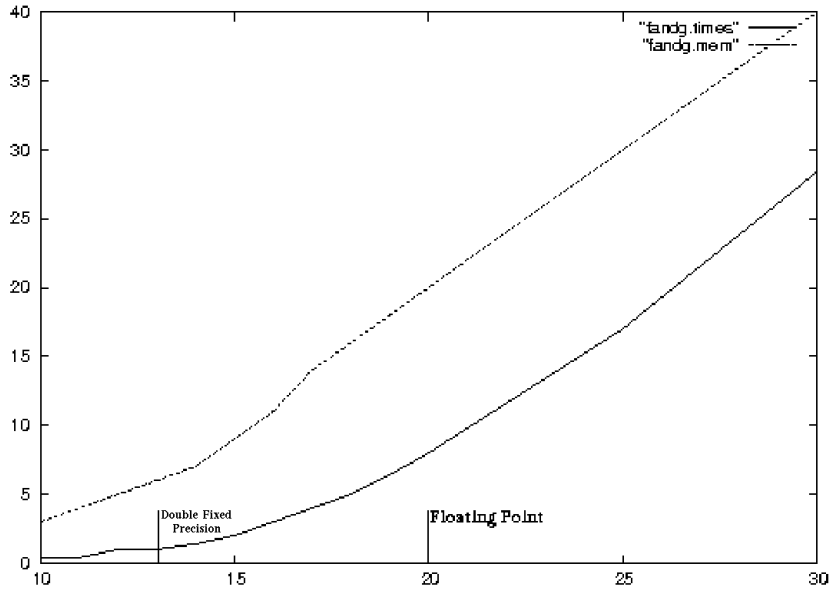


**Fig. 4.** Time and Memory for f & g series in CAMAL

$$\begin{aligned} nP_n &= (2n-1)xP_{n-1} - (n-1)P_{n-2} \\ P_0 &= 1 \\ P_1 &= x \end{aligned}$$

The same polynomials can be calculated by Rodrigues' formula

$$P_n(x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}(x-1)^n$$

and the simple benchmark is to check that these give the same answers. This is one of the tests made on the 1998 Camal-in-C translation[6]. To order 30 this verification and printing took 0.380s, while the same program in REDUCE on the same computer took 1.071s. Unfortunately my records do not give the memory used.

### 7.3 Other Benchmarks

Amongst the other benchmarks there is a series published in the SIGSAM Bulletin together with solutions and timings. The f and g series was re-designated as Problem #1, and there were a total of 11 problems proposed. Those interested can find papers reporting timings from March 1972 until the end of the decade. I want here to consider just problem #3, the reversion of a double series[17]. Solutions were proposed by Fitch[8] and by Hall[13] in ALTRAN. Hall's method

[6] Unpublished work of Arthur Norman and myself.

**Table 4.** Comparisons of two algorithms for Problem #3 on PDP10

| | Fitch | | Hall | |
|---|---|---|---|---|
| n | Time | Store | time | Store |
| 1 | 0.224 | 203 | 0.250 | 143 |
| 2 | 0.993 | 561 | 0.712 | 491 |
| 3 | 3.526 | 1781 | 2.147 | 1723 |
| 4 | 12.376 | 5431 | 6.759 | 5401 |

was to reduce the problem to one of calculating a recurrence problem, while Fitch used repeated approximation, to which CAMAL is well suited. The timing superiority of the CAMAL program led to an inaccurate statement that the algorithm was superior. Later implementations of Hall's method on CAMAL[11] showed that Hall's program used less memory and ran faster (see table 4, time in seconds, memory in words). A compact representation can be deceiving.

But do note the memory sizes. As this is an explicit return system these memory sizes are invariant; in addition there was 14K of program and 3K of fixed data.

### 7.4 Lunar Disturbing Function

The last comparison I wish to make is in the calculation of the Delaunay Lunar Disturbing Function, which is the first stage of the Delaunay Lunar theory. The details can be found elsewhere(for example [9]) but this is a Fourier series calculation which was bread and butter to the original CAMAL. In 1993 I wrote a module for REDUCE that used as close as I could the CAMAL data structure for Fourier series, while using the standard REDUCE polynomials for coefficients. This system runs significantly faster than the same problem in normal REDUCE(see table 5)

This showed that even within a general purpose algebra system there is scope for a compact subcomponent. For the whole calculation, using both lists and balanced trees, the figures were interesting:

| Order of DDF | Reduce | Camal Linear | Camal Tree |
|---|---|---|---|
| 2 | 23.68 | 11.22 | 12.9 |
| 4 | 429.44 | 213.56 | 260.64 |
| 6 | >7500 | 3084.62 | 3445.54 |

What came as a surprise at the time was the direct comparison with CAMAL. The CAMAL Disturbing function program could calculate the tenth order with a maximum of 32K words (about 192Kbytes) whereas this system failed to calculate the eighth order in 4Mbytes (taking 2000s before failing). I also have in my archives the output from the standard CAMAL test suite, which includes a sixth order DDF on an IBM 370/165 run on 2 June 1978, taking 22.50s and

**Table 5.** Solving Kepler's Equation in Reduce and CAMAL-module

**Solving Kepler's Equation**

| Order | REDUCE | Fourier Module |
|---|---|---|
| 5 | 9.16 | 2.48 |
| 6 | 17.40 | 4.56 |
| 7 | 33.48 | 8.06 |
| 8 | 62.76 | 13.54 |
| 9 | 116.06 | 21.84 |
| 10 | 212.12 | 34.54 |
| 11 | 381.78 | 53.94 |
| 12 | 692.56 | 82.96 |
| 13 | 1247.54 | 125.86 |
| 14 | 2298.08 | 187.20 |
| 15 | 4176.04 | 275.60 |
| 16 | 7504.80 | 398.62 |
| 17 | 13459.80 | 569.26 |
| 18 | *** | 800.00 |
| 19 | *** | 1116.92 |
| 20 | *** | 1536.40 |

using a maximum of 15459 words of memory for heap — or about 62Kbytes. One is tempted to ask if we have made any progress...

## 8 Conclusions

This paper has presented the design, performance and experience with an early computer algebra system that was designed to run on a small memory computer. Throughout its various incarnations the need to keep memory usage as low as possible was always present. Data structures were packed, memory allocation was simple and explicit, and we always ensured that all memory use was accounted. The polynomial subsystem was especially tuned to approximation techniques, with algorithms that took care of components that were too small to form part of an answer. There were actually many other features that are not mentioned here to keep the focus on the compact data structures and associated algorithms. Clearly the system was designed at a time when computers were much smaller memoried and much slower at execution than that to which we are used today; but there are still applications where memory is at a premium, or battery power considerations indicate slow processing. While not suggesting that CAMAL is necessarily the answer to modern problems, I assert that it still has many things to teach us.

I would like to express my thanks to David Barton (who taught me not to comment code), Steve Bourne from whom I learnt much about assembler, and many others in the CAMAL and algebra communities. I would also give my special thanks to Arthur Norman for the many suggestions he has made over many years that have served to improve CAMAL.

## References


1. Barton, D., Bourne, S.R., Burgess, C.J.: A Simple Algebra System. Computer Journal 11, 293–298 (1968)
2. Barton, D.: A New Approach to the Lunar Theory. PhD thesis, University of Cambridge, 196
3. Bourne, S.R.: Automatic Algebraic Manipulation and its Applications to the Lunar Theory. PhD thesis, University of Cambridge (1970)
4. Bourne, S.R., Horton, J.R.: The CAMAL System Manual. The Computer Laboratory, University of Cambridge (1971)
5. Bourne, S.R., Horton, J.R.: The Design of the Cambridge Algebra System. In: Proceedings of SYMSAM/2, pp. 134–143. SIGSAM/ACM (1971)
6. Carette, J., Davenport, J.H., Fitch, J.: Barton and Fitch revisited (2009), `http://opus.bath.ac.uk/14083`
7. Delaunay, C.: Théorie du Mouvement de la Lune (Extraits des Mém. Acad. Sci.). Mallet-Bachelier, Paris (1860)
8. Fitch, J.P.: A solution to Problem #3. SIGSAM Bulletin 26, 24–27 (1973)
9. Fitch, J.P.: REDUCE meets CAMAL. In: Fitch, J. (ed.) DISCO 1992. LNCS, vol. 721, pp. 104–115. Springer, Heidelberg (1993)
10. Fitch, J.P., Garnett, D.J.: Measurements on the Cambridge Algebra System. In: Proceedings of the International Computing Symposium, Venice, pp. 139–147 (1972)
11. Fitch, J.: A solution of problem #3 using camal. SIGSAM Bulletin 32, 14 (1975)
12. Fitch, J.P.: An Algebraic Manipulator. PhD thesis, University of Cambridge (1971)
13. Hall, A.D.: Solving a problem in eigenvalue approximation with a symbolic algebra system. SIGSAM Bulletin 26, 145–223 (1975)
14. Horton, J.R.: A System for the Manipulation of Tensors on an Automatic Computer. University of Cambridge, Dissertation for the Diploma in Computer Science (1969)
15. I.C.T. Ltd and Cambridge University. Atlas 2 System Programmers Manual, E.P. 59 (April 1964)
16. Irons, E.T.: A Syntax Directed Compiler for ALGOL 60. Comm. ACM 4(1), 51–55 (1961)
17. Lew, J.S.: Problem #3 – reversion of a double series. SIGSAM Bulletin 23, 6–7 (1972)
18. Matthewman, J.R.: Syntax Directed Compilers. PhD thesis, University of Cambridge (1966)
19. Richards, M.: BCPL — a tool for compiler writing and systems programming. In: Proceedings of the Spring Joint Computer Conference, vol. 34, pp. 557–566 (1969)
20. Sconzo, P., LeSchack, A.R., Tobey, R.G.: Symbolic Computation of f and g Series by Computer. Astronomical Journal 70(4), 269–271 (1965)