On using Common Lisp for Scientific Computing

N. Neuss

December 17, 2002

Abstract

Lisp is a very flexible and powerful language, but up to now it has not been used intensively for applications in scientific computing. The main reason is the prejudice that Lisp is slow. While this prejudice may have been true in early stages of Lisp's history, it is not really true today. Furthermore, the virtues of Lisp are becoming more and more important. In this contribution, we support this point of view: first, by comparing the efficiency of BLAS routines written in C and Common Lisp and second, by discussing the recently developed toolbox FEMLISP for solving partial differential equations with finite element methods and multigrid.

Key Words: Common Lisp, floating-point performance, BLAS, run-time compilation, partial differential equations, finite element methods, multigrid.

1 Introduction

Lisp is the second-oldest high-level computer language after Fortran. It is a very flexible and powerful language, and many problems arising in the computational sciences were first solved using Lisp as a vehicle. Besides being well-known as *the* language for solving problems in artificial intelligence, Lisp was often used for prototyping, i.e. as the language in which an application was first implemented. For example, the world's first computer algebra system, MACSYMA, was implemented in Lisp. Lisp is also the language of the CAD system AUTOCAD and the extensible editor EMACS. In spite of these successes, Lisp has not yet become a mainstream language. The reasons are mostly historical: in the first decades of computer history, resources were

scarce, and a language for which the first implementations were interpreters, which has automatic memory management as an essential component, and which is used best inside large development environments could not compete with lightweight languages that were more or less sufficient for solving the problems of small complexity which could be treated at that time.

Nowadays, the situation is drastically different. Computing power has increased tremendously, rendering it possible to have powerful Lisp environments on personal computers while still using only a fraction of the available memory and CPU power. Due to Java, automatic memory management has become a mainstream feature. In the meantime, Lisp itself has also grown up into *Common Lisp*, which is a powerful, object-oriented language, suited to real-world applications, with many implementations that support compilation to native code. Thus, using Lisp also for problems outside its original realm of artificial intelligence is a natural choice today. Indeed, this has been known for some time and was discussed at several points in literature, e.g. in [6], but, unfortunately, only with small impact outside the Lisp community. Yet today, there seems to be a growing interest in Lisp especially in the mathematical community, as the recent applications KENZO [13] or FEMLISP [12], as well as the flourishing of the free computer algebra system MAXIMA [9] show.

The structure of this article is as follows. In Section 2, we give an overview of Common Lisp. In Section 3, we study the floating-point efficiency for compiled Lisp code and, in Section 4, we briefly sketch the FEMLISP toolbox. We end with conclusions and acknowledgements.

2 Common Lisp

In this section, we want to give a brief overview of Common Lisp, which is one dialect of Lisp. Lisp itself was designed at the end of the 1950s and is thus the second-oldest high-level computer language after Fortran, which was formulated some years earlier in 1954. Thus, it had a long time for growing-up and changed considerably from its beginnings. During its first decades, it diversified into a large variety of dialects which were then unified during the 1980s. The result of this unification effort is Common Lisp, which became an ANSI Standard in 1994. Nowadays, it is the dominant dialect in the Lisp family of languages, though other dialects coexist, notably Scheme (which has a small language core and is adapted well to educational purposes), Emacs Lisp (the configuration language of the Emacs editor), and AUTOLISP (the configuration language of AUTOCAD).

In brief, Common Lisp has the following features:

1. It is an interactive language which is usually used in an integrated development environment. That is, compiler, debugger, and application all are parts of the same Lisp environment. The editor is tightly integrated as well (often being a part of the same environment). This allows expressions to be evaluated with one key stroke, as well as information and documentation of functions to be available when editing.

2. It has powerful commands for list handling. (This does not mean that lists are the only data structure. Instead, Common Lisp possesses a wealth of other built-in data structures, e.g. multi-dimensional arrays, hash-tables, or bit-vectors.)

3. It allows for *symbolic manipulation*: every token which is read is interned as an entity called "symbol" in name spaces called "packages". Thus, comparison of symbols is fast (equality of pointers), and composite data structures can be constructed efficiently.

4. Probably its most obvious characteristic is a consistent prefix syntax of the form `(operator argument1 ...)`, e.g. `(+ 3 5)`, `(sin x)` or `(if (< x 0) (- x) x)`.

5. Because of the three preceding points, Lisp programs have a tree representation which can be efficiently built and manipulated. Consequently, a very strong mechanism for syntactic transformations called "macros" becomes feasible. Most of Common Lisp basic operators are indeed macros. For example, the loop expressions from Table 2 are transformed into more elementary expressions involving conditionals and even jumps.

6. Lisp is dynamically typed: variables can contain data of arbitrary type, and the data themselves contain type information which is analysed at runtime.

7. Common Lisp possesses a complete numerical tower: there are short and long integers, rational numbers, real and complex floating-point numbers, together with generic arithmetic operating on them.

8. Lisp has automatic memory management, i.e. memory does not have to be freed manually, but is reclaimed automatically by garbage collection (GC) in case it is not referenced any more. GC has turned out to be a crucial ingredient for many advanced software techniques and is built-in in most computer languages designed in recent years.

9. Lisp supports "functional programming", i.e. every expression has a return value, and functions are first-class objects: they can be generated (and compiled) in a local environment and can appear as a value of a variable or return-value of a function.

10. Common Lisp is also a very strong object-oriented programming language. The *Common Lisp Object System* (CLOS) has multiple inheritance, multiple dispatch, class redefinition, introspection, and a meta-object protocol allowing for extension and modification.

11. Lisp supports low-level programming: C or Fortran code can usually be translated into the equivalent Lisp code.

3 Numerical efficiency of Common Lisp

In this section, we test Common Lisp for its numerical efficiency. We do this by considering two important operations from the *Basic Linear Algebra Subroutines* (BLAS), see [8], [5].

Most computations in numerical analysis and scientific computing have to perform at least one of the following operations:
Scalar product: Compute the *scalar product* or *dot product* between two vectors $x = (x_1, \ldots, x_n)^t$ and $y = (y_1, \ldots, y_n)^t$. The formula for this `dot`-operation is

$$\texttt{dot(x, y)} = x \cdot y = \sum_{i=1}^{n} x_i y_i \,. \tag{1}$$

Matrix-vector multiplication: Compute $b = Ax$ for

$$A = \begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & & \ldots \\ a_{m1} & \ldots & a_{mn} \end{pmatrix} \,,\; x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \,,\; b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \,. \tag{2}$$

The formula for this operation is

$$b_i = \sum_{i=1}^{n} A_{ij} x_j \,,\; i = 1, \ldots, m \,. \tag{3}$$

From this formula, we can see that the computation of b can be done in two different ways. The first possibility is to compute each component b_i as the dot product between x and the i-th row of A. Alternatively, one can sum

```
double ddot (double *x, double *y,
             int n)
{
  int j;
  double sum = 0.0;
  for (j=0; j<n; j++)
    sum += x[j]*y[j];
  return sum;
}
```

```
void daxpy (double a, double *x,
            double *y, int n)
{
  int j;
  for (j=0; j<n; j++)
     y[j] += a*x[j];
}
```

Table 1: C code for `ddot` and `daxpy`.

```
(defun ddot (x y n)
  (declare (type fixnum n)
           (type (simple-array double-float (*)) x y))
  (loop for j of-type fixnum below n
        summing (* (aref x j) (aref y j)) of-type double-float))

(defun daxpy (a x y n)
  (declare (type fixnum n) (type double-float a)
           (type (simple-array double-float (*)) x y))
  (loop for i of-type fixnum below n do
        (incf (aref y i) (* a (aref x i)))))
```

Table 2: Common Lisp code for `ddot` and `daxpy`.

up scalar multiples of columns of A. This latter operation is composed of elementary `axpy`-operations

$$y_i := y_i + ax_i\,,\ i = 1, \ldots, n\,. \tag{4}$$

Solving linear equations: Solve $Ax = b$ for x where

$$A = \begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & & \ldots \\ a_{n1} & \ldots & a_{nn} \end{pmatrix}\,,\ x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}\,,\ b = \begin{pmatrix} b_1 \\ \vdots \\ b_n\,. \end{pmatrix} \tag{5}$$

Often, this is done by transforming A into a product of a lower and an upper triangular matrix. Here, the elementary operation is again a call to `axpy`.

Both `dot` and `axpy` have an operation count of n additions and n multiplications or n *arithmetic operations* where an arithmetic operation consists of one addition and one multiplication.

An implementation of these routines in C for vectors of double-precision floating-point numbers is given in Table 1 while an equivalent implementation in Common Lisp is given in Table 2.[1] We see that both versions present the same information to their compiler, such that both compilers

[1]The C and Lisp code used in these tests can be downloaded from the author's web page [10]. The C code has been published in [11].
Note also that a translation of C or Fortran code to Common Lisp can be done automatically for a large part of the existing code base, see e.g. [6].

	Pentium 2 (400 MHz)				Pentium 4 (2.4 GHz)			
	$n = 256$		$n = 10^6$		$n = 256$		$n = 10^6$	
	`ddot`	`daxpy`	`ddot`	`daxpy`	`ddot`	`daxpy`	`ddot`	`daxpy`
C	172	117	45	24	910	1140	300	140
CMUCL	94	87	35	18	690	480	300	140

Table 3: Some BLAS performance results (MFLOPS).

should be able to produce optimal machine code. The Lisp version is more verbose, however. In fact, Lisp's syntax was not designed for putting type declarations everywhere, and Lisp programmers mostly write code without type declarations. Also in this case, this would make the above code shorter and more generally applicable, e.g. to complex-valued vectors. On the other hand, it would result in lower performance, because the compiler is not able to deduce this type information without help and has to produce code handling general numbers. Thus, the type declarations become necessary if such routines present a performance bottleneck for the respective application.

Now, we measure the speed with which those routines are performed either on a pair of short vectors ($n = 256$) or on a pair of long vectors ($n = 10^6$). The size of n makes a crucial difference because of the higher data locality in the first case. To see why, remember that the CPU can usually process data at a higher speed than it can be delivered from main memory by random access. As a remedy, the so-called *cache* consisting of fast memory chips is installed as a buffer between CPU and main memory. Now, in the case $n = 256$, both vectors fit in the so-called primary cache whereas, in the second case, they do not even fit in the so-called secondary cache (at least for the hardware which we used for our tests).

Table 3 shows the performance which we measured on two machines: a two-year-old laptop (Pentium 2, 400 MHz) and a new PC (Pentium 4, 2.4 GHz). We compiled the C code with the GNU C compiler (`gcc`) and `-O3` optimization. The Lisp code was compiled with the optimization settings

```
(declaim (optimize (debug 0) (safety 0) (space 0)
                   (compilation-speed 0) (speed 3))
```

using CMU Common Lisp (CMUCL) from [4].

As a result of these tests, we see that the machine code produced by `CMUCL` is upto 60% slower than the machine code produced by `gcc` if all data is inside the cache (but equally fast if memory bandwidth is the limiting factor). The reason is that the compiler back end of `gcc` is more elaborate, presumably because many more people have improved on it. Further, writing

```
/* dynamic version: n, stencil_size, stencil_values unknown
   at compile time */
for (pos1=n; pos1<(n-1)*n; pos1+=n)
  for (pos2=pos1+1; pos2<pos1+n-1; pos2++)
  {
      register double s = 0.0;
        for (k=0; k<stencil_size; k++)
          s += stencil_values[k]*u[pos2+stencil_shift[k]];
        u[pos2] = s;
  }

/* static version: n=1000, nine-point stencil for Laplace equation */
for (pos1=1000; pos1<(1000-1)*1000; pos1+=1000)
  for (pos2=pos1+1; pos2<pos1+999; pos2++)
        u[pos2] += 8.0/3.0 * u[pos2] - 1.0/3.0 *
          (u[pos2-1001]+u[pos2-1]+u[pos2+999]+u[pos2-1000]+
             u[pos2+1000]+u[pos2-999]+u[pos2+1]+u[pos2+1001]);
```

Table 4: Applying a stencil to a function on a structured grid.

a good compiler for Common Lisp is a very demanding task, not because of the compiler back end, but because of obtaining good efficiency for advanced language features, for example CLOS. Thus, many commercial or free implementations put the emphasis there and accept a slightly reduced low-level performance.

Nevertheless, it is interesting that judicious use of the advanced language features of Common Lisp can also result in a *higher* performance than a pure C implementation can obtain. The key feature here is that Common Lisp source code can be generated and compiled to machine code at runtime, while conventional languages do not allow this in a portable fashion. This allows for the compilation of code adapted to some special runtime situation. This is the equivalent of doing a controlled just-in-time compilation of arbitrary complexity.

Examples where this can be used abound. For example, in interactive programs for solving partial differential equations (see Section 4), coefficient functions can be read and compiled to native code. This presents a large problem for other approaches, because interpreting those expressions is complicated and may be too slow at runtime. Another example is the application of stencils to functions defined on structured grids, which arises for filters applied in image-processing, wavelet transformations, and finite-difference discretizations. We elaborate this in the following.

Two versions of C code for applying a stencil to a function defined on a structured two-dimensional grid of dimension $n \times n$ are shown in Table 4. In the second version, the dimensions of the grid as well as the stencil are assumed to be known (the stencil arises from discretizing the Laplace equation with bilinear finite elements). It is obvious that the C compiler can produce

	C (dynamic)	C (static)	CMUCL
P2 (400 MHz)	4.3 sec	1.1 sec	1.6 sec
P4 (2.4 GHz)	0.59 sec	0.30 sec	0.48 sec

Table 5: Timings of C and CL for 10 applications of a difference stencil.

much more efficient code for the second version, but this is only possible if both n and the stencil are known at compilation time. In contrast, with Common Lisp, it is easy to generate and compile the code for the second version at runtime.[2] Comparing the efficiency of the dynamic and static C code compiled by `gcc` with the efficiency of the code generated by CMUCL we obtain the values shown in Table 5, and we see that the Lisp code performs somewhat worse than the static C version because of the reasons mentioned above, but better than the (dynamic) C code with the same functionality.

4 Femlisp

The theoretical and practical treatment of partial differential equations (PDEs) is a highly complex domain of mathematics with a multitude of possible phenomena arising. Consequently, the efficient numerical solution is also intricate and the implementation of such methods inherits this intricacy. Thus, originally, software in this domain could handle only special situations well and was not applicable to different problems. But this changed in the last decade, when more and more software packages appeared, claiming to be multi-purpose tools for solving PDEs. Before C++ became popular, this work was done in Fortran (e.g. [1]) or C (e.g. [2]), but nowadays, most developments in this application area are done in C++.

Unfortunately, C++ is a double-edged sword. C++ improves on its predecessor C, while still maintaining a very machine-oriented programming style. It does not have most of the advanced features described in Section 2. Additionally, it suffers from the compromises it has to make to ensure efficient code, see [7]. Thus, it appears to be a rather inadequate choice for implementing a complicated framework for solving partial differential equations.

This observation was the reason for the development of the toolbox Femlisp which can solve partial differential equations using the finite element

[2] Of course, compilation introduces an overhead (for our example about 0.02 seconds on the P4), but these are fixed costs, independent of the size of the grid and independent of the number of stencil applications.

method (FEM), see e.g. [3]. It benefits from the choice of Common Lisp as the implementation language, especially because of the powerful interactive environment and due to the expressiveness of the language which results in a considerable reduction of source code.[3]

At the moment, FEMLISP has the following functionality. It handles unstructured meshes consisting of arbitrary tensor-product cells in arbitrary space dimensions. This functionality has not yet been implemented in any other software. At the moment, the discretization is done with Lagrange finite elements of arbitrary order, but the extension to other types of finite elements should pose no problems. Isoparametric and non-parametric cell mappings for obtaining a good resolution of boundaries and interfaces are available. For the solution of the arising linear systems, geometric and algebraic multigrid methods are available. The graphic display of meshes, coefficients and finite element functions is done by linking to Gnuplot and IBM's Data Explorer.

FEMLISP was first presented at the CISC 2002 in Berlin and the EMG 2002 in Hohenwart. It is described in more detail in [12]. Its first public release is scheduled for January, 2003.

5 Conclusions

Lisp is certainly one of the most expressive computer languages available today, if not the most expressive. Because of its long history, it is a very mature language suited very well to large-scale applications. More and more, such a language will become a necessity also for scientific computing, where the complexity of real-world applications requires the *combination* of many intricate algorithms to obtain a solution.

In this article, we have shown that the efficiency of Lisp code is no longer a problem, whereas the strength of Common Lisp opens up new possibilities for interesting applications in scientific computing. One of these applications is the solution of partial differential equations, where this power is certainly required for handling the multitude of phenomena and algorithms from a software engineering standpoint.

[3] The amount of source code reduction is difficult to quantify exactly, but is said to be in the order of 1.5–5. One hint for a large reduction factor is that the FEMLISP core comprises only about 12.000 lines of source code, which appears to be very little when compared with similar programs written in C or C++.

Acknowledgements

This article was written while I was a research assistant in Heidelberg with the Technical Simulation group of Prof. G. Wittum, whom I want to thank for giving me the opportunity to work on this non-standard topic. Further, I am indebted to the interesting and entertaining *comp.lang.lisp* newsgroup as well as the CMUCL mailing list for much information from the year 2000 onwards.

References

[1] R. E. Bank. *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations – Users' Guide 7.0*, volume 15 of *Frontiers in Applied Mathematics*. SIAM Books, Philadelphia, 1994.

[2] P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuss, H. Rentz-Reichert, and C. Wieners. UG – a flexible software toolbox for solving partial differential equations. *Comput. Visual. Sci.*, 1:27–40, 1997.

[3] S. C. Brenner and L. R. Scott. *The Mathematical Theory of Finite Element Methods*. Texts in Applied Mathematics. Springer–Verlag, New York, 1994.

[4] CMUCL. Homepage. http://www.cons.org/cmucl.

[5] J. J. Dongarra. Performance of various computers using standard linear equations software. Technical report, Computer Science Department, University of Tennessee, 1998.

[6] R. Fateman, K. A. Broughan, D. K. Willcock, and D. Rettig. Fast floating-point processing with Common Lisp. *ACM Trans. on Math. Software*, 21:26–62, 1995.

[7] I. Joyner. A critique of C++ and Programming and Language Trends of the 1990s. http://www.literateprogramming.com/c++critique.pdf, 1996.

[8] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algbra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5:308–323, 1979.

[9] Maxima. Homepage. http://sourceforge.net/~maxima.

[10] N. Neuss. http://www.iwr.uni-heidelberg.de/~Nicolas.Neuss.

[11] N. Neuss. A new sparse matrix storage method for adaptive solving of large systems of reaction-diffusion-transport equations. *Computing*, 68,1:19–36, 2002.

[12] N. Neuss. Femlisp — a multi-purpose tool for solving partial differential equations. *Comput. Vis. Sci.*, (submitted).

[13] F. Sergeraert. Common Lisp, Typing, and Mathematics. Satellite talk at the EACA Congress in Ezcaray, 2001.