A History of MATLAB

CLEVE MOLER, MathWorks, USA
JACK LITTLE, MathWorks, USA

Shepherds: Jens Palsberg, University of California, Los Angeles, USA
Guy L. Steele Jr., Oracle Labs, USA

The first MATLAB® (the name is short for "Matrix Laboratory") was not a programming language. Written in Fortran in the late 1970s, it was a simple interactive matrix calculator built on top of about a dozen subroutines from the LINPACK and EISPACK matrix software libraries. There were only 71 reserved words and built-in functions. It could be extended only by modifying the Fortran source code and recompiling it.

The programming language appeared in 1984 when MATLAB became a commercial product. The calculator was reimplemented in C and significantly enhanced with the addition of user functions, toolboxes, and graphics. It was available initially on the IBM PC and clones; versions for Unix workstations and the Apple Macintosh soon followed.

In addition to the matrix functions from the calculator, the 1984 MATLAB included fast Fourier transforms (FFT). The Control System Toolbox™ appeared in 1985 and the Signal Processing Toolbox™ in 1987. Built-in support for the numerical solution of ordinary differential equations also appeared in 1987.

The first significant new data structure, the sparse matrix, was introduced in 1992. The Image Processing Toolbox™ and the Symbolic Math Toolbox™ were both introduced in 1993.

Several new data types and data structures, including single precision floating point, various integer and logical types, cell arrays, structures, and objects were introduced in the late 1990s.

Enhancements to the MATLAB computing environment have dominated development in recent years. Included are extensions to the desktop, major enhancements to the object and graphics systems, support for parallel computing and GPUs, and the "Live Editor", which combines programs, descriptive text, output and graphics into a single interactive, formatted document.

Today there are over 60 Toolboxes, many programmed in the MATLAB language, providing extended capabilities in specialized technical fields.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages.*

Additional Key Words and Phrases: linear algebra, MATLAB, matrix computation

ACM Reference Format:
Cleve Moler and Jack Little. 2020. A History of MATLAB. *Proc. ACM Program. Lang.* 4, HOPL, Article 81 (June 2020), 67 pages. https://doi.org/10.1145/3386331

Authors' addresses: Cleve Moler, MathWorks, Natick, MA, 01760, USA, moler@mathworks.com; Jack Little, MathWorks, Natick, MA, 01760, USA.

2475-1421/2020/6-ART81
https://doi.org/10.1145/3386331

PROLOGUE

In his first job after grad school, in 1966, Cleve Moler was an Assistant Professor of Mathematics. He taught calculus, numerical analysis, and linear algebra. For linear algebra, he followed the traditional syllabus, which emphasized proofs of theorems about linear transforms in abstract vector spaces. There was no time devoted to computers or applications of matrices. He vowed never to follow the traditional syllabus again. He wanted to use matrix factorizations, like the singular value decomposition, and emphasize applications, like principal component analysis. And he wanted his students to use computers and modern software. He wanted a new course, matrix analysis, to replace traditional linear algebra.

Contents

Origins	1965–1970	Series of papers in *Numerische Mathematik* that present matrix algorithms in Algol
	1970	First Argonne proposal to NSF to explore high-quality numerical software
	1971	Collected papers published as *Handbook for Automatic Computation, Volume II*
	1971	First release of EISPACK
	1975	Second Argonne proposal to NSF to explore high-quality numerical software
	1976	Second release of EISPACK
	1976	*Algorithms + Data Structures = Programs* by Niklaus Wirth
	1977	*Computer Methods for Mathematical Computations* by Forsythe, Malcolm, and Moler
	1979	Publication of *LINPACK Users' Guide* and of BLAS specification
Classic	1978	Cleve Moler creates first version of MATLAB in Fortran
	1979	Moler teaches course in Numerical Analysis at Stanford, using Classic MATLAB
	1980	"Design of an Interactive Matrix Calculator" (National Computer Conference)
	1980	Jack Little begins using MATLAB for his consulting work in control systems
	1981	Publication of the first *MATLAB Users' Guide*
MATLAB	1983	Little suggests creating commercial product for IBM PC based on Classic MATLAB
	1984	Little and Steve Bangert rewrite and enhance all of Classic MATLAB in C
	1987	Built-in support for solving ordinary differential equations (ODEs)
	1990	Moler, John Gilbert, and Rob Schreiber introduce sparse matrix data type
	1996	Cell arrays, structures, dot notation, objects, multidimensional arrays
	1999	Function handles
	2004	Signed integer, unsigned integer, and logical data types
	2008	Enhanced object-oriented programming
	2015	New execution engine
	2016	The `string` data type (an alternative to character vectors)
MathWorks	1984	Little, Moler, and Bangert form MathWorks in California
	1984	PC-MATLAB debuts at the IEEE Conference on Decision and Control in Las Vegas
	1986	Debut of Pro-MATLAB for Unix workstations
	1987	Debut of MATLAB for the Apple Macintosh
	1992	MathWorks has over 100 staff members
	1994	MathWorks provides a software solution for the Pentium division bug
	1995	MathWorks has over 200 staff members
	1998	More than 1,000 people worldwide
	2008	More than 2,000 people worldwide
	2019	Over 5,000 people, 30% located outside the United States
Toolboxes	1985	Control System Toolbox™
	1987	Signal Processing Toolbox™
	1988	Lennart Ljung, Linköping University, Sweden, writes System Identification Toolbox™
	1990	Carl de Boor, U. Wisconsin—Madison, writes Spline Toolbox™
	1993	Image Processing Toolbox™
	1993	Symbolic Math Toolbox™
	2004	Parallel Computing Toolbox™
	2010	GPU support added to Parallel Computing Toolbox
Environment	1994	Stephen C. Johnson writes the first compiler for MATLAB (up to 300× speedup)
	2000	EISPACK and LINPACK replaced by LAPACK
	2000	MATLAB Desktop user interface
	2001	MATLAB Central and MATLAB File Exchange at `mathworks.com`
	2004	Publication of *Numerical Computing with MATLAB*, a textbook by Moler
	2010	MATLAB Online and MATLAB Mobile
	2016	MATLAB Live Editor user interface

Fig. 1. Major events in the development of MATLAB®.

1 THE CREATION OF MATLAB®

The original goals of MATLAB are well expressed by the introduction to its original published design description [Moler 1980; see also Moler 1982]:

> MATLAB is an interactive computer program that serves as a convenient "laboratory" for computations involving matrices. It provides easy access to matrix software developed by the LINPACK and EISPACK projects [Dongarra et al. 1979; Garbow et al. 1977; Smith et al. 1974]. The capabilities range from standard tasks such as solving simultaneous linear equations and inverting matrices, through symmetric and nonsymmetric eigenvalue problems, to fairly sophisticated matrix tools such as the singular value decomposition.
>
> It is expected that one of MATLAB's primary uses will be in the classroom. It should be useful in introductory courses in applied linear algebra, as well as more advanced courses in numerical analysis, matrix theory, statistics, and applications of matrices to other disciplines. In nonacademic settings, MATLAB can serve as a "desk calculator" for the quick solution of small problems involving matrices.
>
> The program is written in Fortran and is designed to be readily installed under any operating system which permits interactive execution of Fortran programs. The resources required are fairly modest. There are about 6000 lines of Fortran source code, including the LINPACK and EISPACK subroutines used. With proper use of overlays, it is possible to run the system on a minicomputer with only 32K bytes of memory.
>
> The size of the matrices that can be handled in MATLAB depends on the amount of storage that is set aside when the system is compiled on a particular machine. We have found that an allocation of 4000 words for matrix elements is usually quite satisfactory. This provides room for several 20 by 20 matrices, for example. One implementation on a virtual memory system provides 50,000 elements. Since most of the algorithms used access memory in a sequential fashion, the large amount of allocated storage causes no difficulties.
>
> In some ways, MATLAB resembles SPEAKEASY [NA Cohen 1973] and, to a lesser extent, APL. All are interactive terminal languages that ordinarily accept single-line commands or statements, process them immediately, and print the results. All have arrays or matrices as principal data types. But for MATLAB, the matrix is the only data type (although scalars, vectors, and text are special cases), the underlying system is portable and requires fewer resources, and the supporting subroutines are more powerful and, in some cases, have better numerical properties.
>
> Together, LINPACK and EISPACK represent the state of the art in software for matrix computation. EISPACK is a package of over 70 Fortran subroutines for various matrix eigenvalue computations that are based for the most part on Algol procedures published by Wilkinson, Reinsch, and their colleagues [Wilkinson and Reinsch 1971]. LINPACK is a package of 40 Fortran subroutines (in each of four data types) for solving and analyzing simultaneous linear equations and related matrix problems. Since MATLAB is not primarily concerned with either execution time efficiency or storage savings, it ignores most of the special matrix properties that LINPACK and EISPACK subroutines use to advantage. Consequently, only 8 subroutines from LINPACK and 5 from EISPACK are actually involved.

The most important way in which MATLAB differed from APL and SPEAKEASY was in portability: APL required special I/O equipment for its distinctive character set, and SPEAKEASY ran only on IBM timesharing systems. Moler wanted to create a tool that could easily be compiled and run on a large variety of computers and operating systems, so that it could be widely used by students. Other important distinguishing features of MATLAB from the beginning were the use of high-quality numerical algorithms, the ability to create plots interactively on the user terminal

(some implementations of SPEAKEASY supported CalComp plotters, which produced high-quality plots but only very slowly, so they hardly qualified as "interactive" in the usual sense), and the backslash operator (see Section 2.1).

1.1 Mathematical Origins

For his entire career, Moler has taken great interest in systems of linear equations, which are often represented in matrix form. After completing his doctorate at Stanford University in 1965, Moler continued to work with his PhD thesis supervisor (and founder of the Stanford University Computer Science Department) George Forsythe on improved computational methods for solving systems of linear equations, and they wrote a book together [Forsythe and Moler 1967; see also Moler 1967, Moler 1969, and NA Moler 2013b].

The mathematical and computational basis for the first version of MATLAB begins with a series of papers by J. H. Wilkinson and 18 of his colleagues published between 1965 and 1970 in the journal *Numerische Mathematik*. The papers were collected in a volume edited by Wilkinson and C. Reinsch, *Handbook for Automatic Computation, Volume II: Linear Algebra*, published in 1971 [Wilkinson and Reinsch 1971]. The papers present algorithms, implemented in Algol 60, for solving matrix linear equation and eigenvalue problems. These were research papers presenting results about numerical stability, details of implementation, and, in some cases, new methods. The importance of using orthogonal transformations wherever possible was emphasized by Wilkinson and the other authors. Part I of the Handbook, with 40 Algol procedures, is about the linear equation problem; part II, with 43 procedures, is about the eigenvalue problem. A list of the individual procedures is provided in [NA Moler 2017].

1.2 EISPACK

In 1970, even before the Handbook was published, a group at Argonne National Laboratory proposed to the U.S. National Science Foundation (NSF) to "explore the methodology, costs, and resources required to produce, test, and disseminate high-quality mathematical software and to test, certify, disseminate, and support packages of mathematical software in certain problem areas."

Every summer for 15 years, Wilkinson lectured in a short course at the University of Michigan and then visited Argonne National Laboratory for a week or two (see Figure 2). The project developed EISPACK (Matrix Eigensystem Package) by translating the Algol procedures for eigenvalue problems from part II of the Handbook into Fortran and working extensively on testing and portability.

In 1971, the first release of the collection was sent to about two dozen universities and national laboratories where individuals had shown an interest in the project and agreed to test the software [Smith et al. 1974]. By 1976 a second release was ready for public distribution [Garbow et al. 1977]. In addition to subroutines derived from part II of the Handbook, the second release included the SVD algorithm from part I and the new QZ algorithm for the generalized eigenvalue problem involving two matrices.

For more on EISPACK see [NA Moler 2018a].

1.3 LINPACK

In 1975, as EISPACK was nearing completion, Jack Dongarra, Pete Stewart, Jim Bunch, and Cleve Moler (see Figure 3) proposed to the NSF another research project investigating methods for the development of mathematical software. A byproduct would be the software itself, dubbed LINPACK for Linear Equation Package [Dongarra et al. 1979]. The project was also centered at Argonne. The three participants from universities worked at their home institutions during the academic year and all four got together at Argonne in the summer.

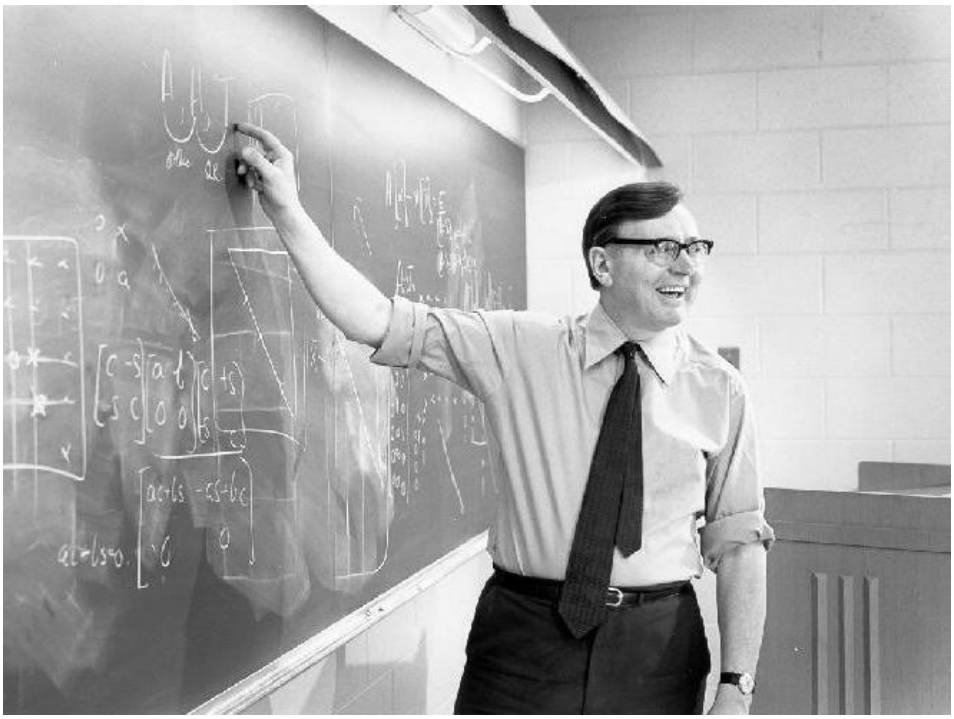


Fig. 2. J. H. Wilkinson describing a matrix algorithm to an audience at Argonne in the early 1970s. (Source: Argonne National Laboratory.)

LINPACK development originated in Fortran; it did not involve translation from Algol. The package contained 44 subroutines in each of four precisions, REAL, DOUBLE, COMPLEX and COMPLEX*16. Fortran at the time was notorious for unstructured, unreadable, "spaghetti" code. The authors adopted a disciplined programming style and expected people as well as machines to read the codes. The scope of loops and if-then-else constructions were carefully shown by indenting. Go-to's and statement numbers were used only to exit blocks and handle possibly empty loops.

All the inner loops are done by calls to the BLAS, the Basic Linear Algebra Subprograms, developed concurrently by Chuck Lawson and colleagues at Caltech's Jet Propulsion Laboratory [Lawson et al. 1979]. On systems that did not have optimizing Fortran compilers, the BLAS could be implemented efficiently in machine language. On vector machines, like the CRAY-1, the loop is a single vector instruction. The two most important subprograms are the inner product of two vectors, DDOT, and vector plus scalar times vector, DAXPY. All the algorithms are column oriented to conform to Fortran storage and thereby provide locality of memory access. [Moler 1972]

In a sense, the LINPACK and EISPACK projects were failures. The investigators had proposed research projects to the NSF to "explore the methodology, costs, and resources required to produce, test, and disseminate high-quality mathematical software." They never wrote a report or paper addressing those objectives. They only produced software [Boyle et al. 1972].

For a summary of the contents of LINPACK, see [NA Moler 2018b].

Today, LINPACK is better known as a benchmark than as a matrix software library. For more on the LINPACK Benchmark, see [NA Moler 2013e].

Fig. 3. The authors of LINPACK: Jack Dongarra, Cleve Moler, Pete Stewart, and Jim Bunch in 1978. (Source: Cleve Moler.)

1.4 Classic MATLAB

In the 1970s and early 1980s, while he was working on the LINPACK and EISPACK projects, Moler was a Professor of Mathematics and then of Computer Science at the University of New Mexico in Albuquerque. He was teaching courses in linear algebra and numerical analysis. He wanted his students to have easy access to LINPACK and EISPACK functions without writing Fortran programs. By "easy access" he meant avoiding the remote batch processing and the repeated edit-compile-link-load-execute process that were ordinarily required on the campus central mainframe computer. This meant the program had to be an interactive interpreter, operating in the time-sharing systems that were becoming available.

So, Moler studied Niklaus Wirth's book *Algorithms + Data Structures = Programs* [Wirth 1976] and learned how to parse programming languages. Wirth calls his example language PL/0. It was a pedagogical subset of Wirth's Pascal, which, in turn, was his response to Algol. Quoting Wirth, "In the realm of data types, however, PL/0 adheres to the demand of simplicity without compromise: integers are its only data type."

Following Wirth's approach, Moler wrote the first MATLAB—the name was an acronym for Matrix Laboratory—in Fortran, as a dialect of PL/0 with matrix as the only data type. The project was just a hobby, a new aspect of programming for him to learn and something for his students to

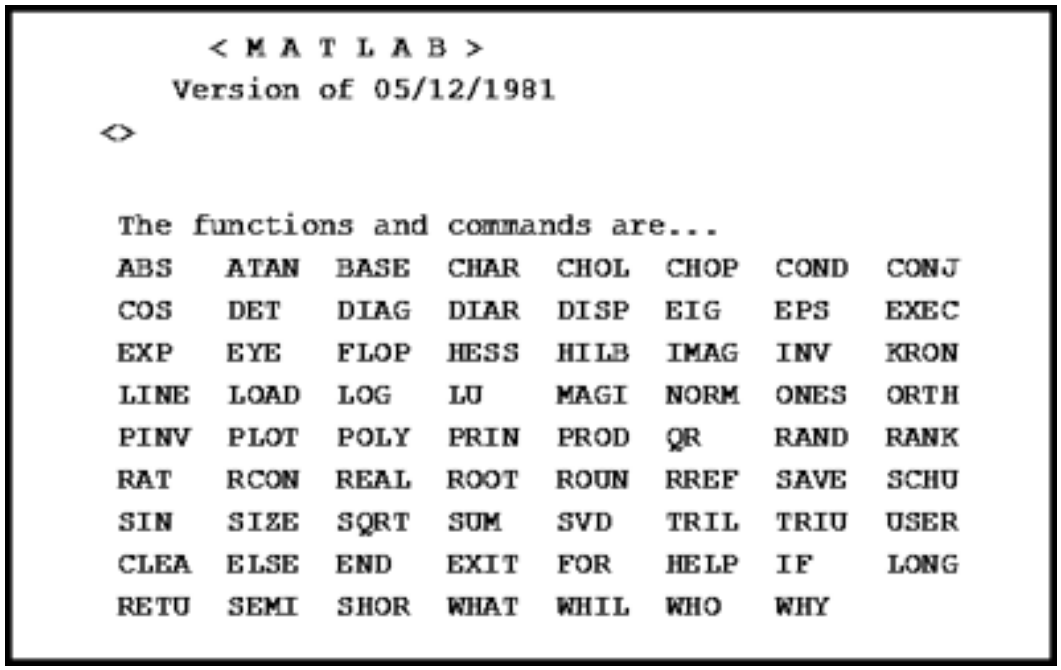


Fig. 4. The start-up screen for the May 12, 1981, version of MATLAB.

use. There was never any formal outside support and certainly no business plan. MathWorks was several years in the future.

Only about a dozen subroutines from LINPACK and EISPACK were needed because *everything* was a complex matrix. The output routine would not print the imaginary parts of the entries if every entry in a matrix had a zero imaginary part. A vector was a matrix with a single column or row. A scalar was a 1-by-1 matrix. A text string was just one way to write a row vector of numeric character codes.

This first MATLAB was not a programming language; it was just a simple interactive matrix calculator. There were no user-defined functions, no toolboxes, no graphics. And no ODEs or FFTs. The snapshot of the start-up screen shown in Figure 4 lists all the functions and reserved words. There are only 71 of them. If you wanted to add another function, you could request the source code from Moler, write a Fortran subroutine, add your new name to the parse table, and recompile MATLAB.

The complete 1981 Users' Guide to this "Classic MATLAB" [NA Moler 1981] is reproduced in [NA Moler 2018g].

*

2 KEY FEATURES OF CLASSIC MATLAB

Classic MATLAB may have been a simple matrix calculator, but it contained several commands and operators that are still in use today. In particular, the backslash operator '\' (for solving a set of linear equations) and the colon operator ':' (for computing a vector of integers or reals in arithmetic sequence) have proved useful in many other fields as MATLAB has expanded beyond numerical linear algebra.

2.1 Backslash

The most important task in matrix computation is solving a system of simultaneous linear equations:

$$Ax = b$$

Here A is a square matrix, b is a column vector and the desired solution x is another column vector. If you disregard considerations of computational efficiency and accuracy, the traditional notation for the solution involves the matrix inverse, A^{-1}:

$$x = A^{-1}b$$

But it is both more efficient and, in the presence of roundoff error introduced by finite precision arithmetic, more accurate to solve the system directly without computing the inverse. (Consider the scalar case. Solve $7x = 21$. The answer $x = 3$ is obtained by dividing both sides of the equation by 7, not by using $7^{-1} = .142857\ldots$ to multiply both sides.) So, the mathematically taboo notion of matrix left division is introduced and the backslash character denotes the solution obtained by the Gaussian elimination process.

```
x = A\b
```

The solution to the similar equation that relates row vectors

$$xA = b$$

is obtained by using matrix right division, expressed (naturally enough) in MATLAB by a forward slash:

```
x = b/A
```

It was fortunate that, thanks to Bob Bemer, all the character sets used at the time had both forward slash and backslash characters [NA Bemer 2000; NA Moler 2013a].

APL and SPEAKEASY both had operators for matrix right division, whose syntax mirrors that of conventional division of floating-point numbers, with the dividend on the left and the divisor on the right. It so happens, however, that historically mathematicians who work with sets of linear equations expressed in matrix form have for many decades preferred to work with column vectors and equations of the form $Ax = b$ rather than row vectors and equations of the form $xA = b$; therefore, in practice it is typically much more convenient to work with matrix left division '\' than with matrix right division '/'. (It is possible to use matrix right division, in any language that provides it, to solve the equation $Ax = b$, but only by transposing both arguments and result: in MATLAB this would be `x = (b'/A')'`. This is needlessly clumsy.)

Today, the matrix assignment statement

```
x = A\b;
```

has come to represent MATLAB. T-shirts with just this statement are very popular—MathWorks has given them away at conferences. (See Figure 5.)

2.2 Colon Operator

The colon character takes on an important role in MATLAB:

`a:d:b` is the row vector $[a, a+d, a+2d, \ldots, a+nd]$ where $n = floor((b-a)/d)$.

If `d` = 1, it may be omitted:

`a:b` is the row vector $[a, a+1, a+2, \ldots, a+n]$ where $n = floor(b-a)$.

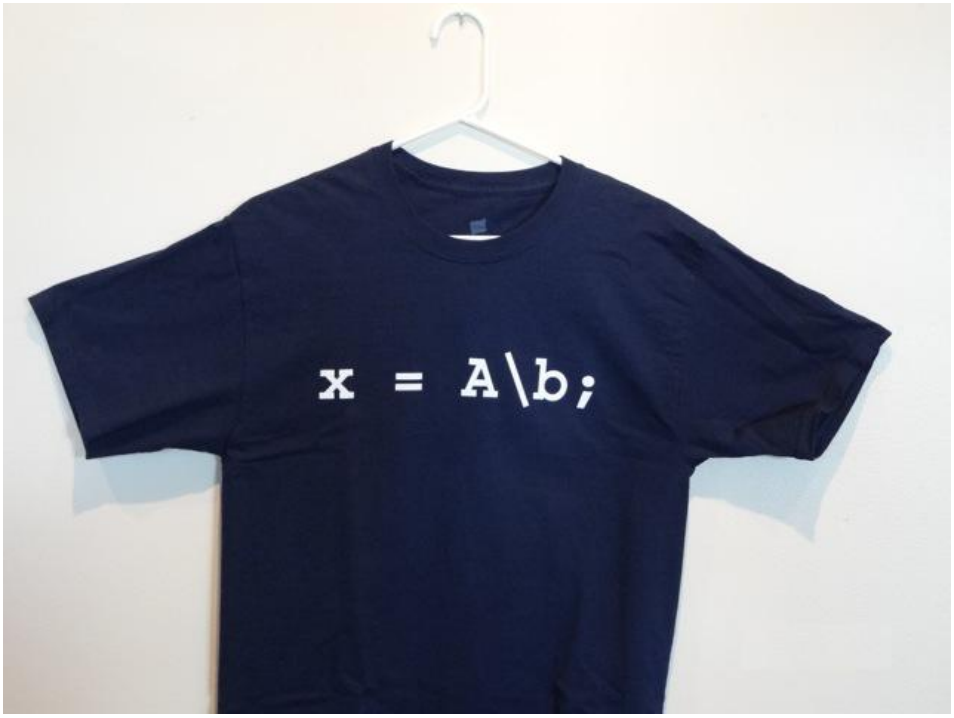


Fig. 5. The iconic MATLAB backslash operation is featured on this popular T-shirt. (Source: Patsy Moler.)

Here are some examples:

- `1:10` is a row vector containing the integers from 1 to 10.
- `0:0.1:10` is the row vector of length 101, $[0, 0.1, 0.2, \ldots, 9.8, 9.9, 10.0]$.
- `'A':'Z'` is the character string made from the 26 uppercase letters.
- `A(i:m,j:n)` is a submatrix of `A`.
- `for k = 1:n`
  <*statements*>
  `end`
  executes the *statements* `n` times.

The colon by itself as an abbreviation for `1:end` was added later.

- `A(:,3)` is the third column of `A`.
- `A(:)` is all of the elements of the matrix `A`, strung out in one tall column.

2.3 The `why` Command

Since Classic MATLAB already had `what`, `while` and `who`, Moler added `why`, which simply returned a random response from a fixed list. It has proved to be very popular. Examples of `why`:

```
WHY NOT?
```

and

```
INSUFFICIENT DATA TO ANSWER.
```

internal code	primary character	alternate character	name
0–9	0–9	0–9	digits
10–35	A–Z	a–z	letters
36			blank
37	(	(	lparen
38	)	)	rparen
39	;	;	semi
40	:	|	colon
41	+	+	plus
42	-	-	minus
43	*	*	star
44	/	/	slash
45	\	$	backslash
46	=	=	equal
47	.	.	dot
48	,	,	comma
49	'	"	quote
50	<	[	less
51	>	]	great

Fig. 6. Table of characters codes used internally by Classic MATLAB.

2.4 Portable Character Set

Classic MATLAB had a fairly conventional notation for text strings, similar to that used in Pascal [Jensen and Wirth 1974] and in Fortran 77. But Classic MATLAB did not have a separate "string" data type.

```
'abcxyz'

 ANS   =

    10     11     12     33     34     35
```

A text string was just one way to notate a row vector (that is, a 1-by-*n* matrix), with each character encoded in a nonnegative integer in a portable "MATLAB character set" of 52 character codes used internally within MATLAB.

```
' < > ( ) = . , ; \ / '

 ANS   =

    COLUMNS  1 THRU 12
   36    50    36    51    36    37    36    38    36    48    36    37

    COLUMNS 13 THRU 21
    36     48     36     39     36     45     36     44     36
```

The `disp` function could be used to display the characters encoded by such a matrix as output on the terminal.

MATLAB maintained this character set as a table with two columns (see Figure 6) that could be modified by the user. Each column has 52 entries; entry k was a machine-specific (or operating-system-specific) character code corresponding to MATLAB internal code k. This table was used to translate between keyboard characters and MATLAB internal codes.

On input, either of two machine characters could be mapped to the same MATLAB internal code; this allowed lowercase letters to be treated as if they were uppercase, and also allowed programmers to use square brackets (on keyboards that had those characters) rather than angle brackets. Moreover, the function `char` took an integer k as an argument and then read a character c (one keypress) directly from the keyboard; it then stored c into entry k of the first column if $k \geq 0$, or into entry $-k$ of the second column if $k < 0$. In this way a programmer running MATLAB on a system with an unusual keyboard could customize the table for the keyboard, an important aspect of the portability of MATLAB.

On output, straightforward indexing of the table was used to convert internal codes to machine characters. For most purposes the first column was used (and this is why, for example, variables names typed in lowercase are printed in uppercase), but the second column was sometimes used (for example to generate lowercase file names on operating systems for which that was the appropriate convention).

The internal character code was chosen so that the digits have consecutive codes and the letters have consecutive codes (at least one common machine character set of the day, EBCDIC, did not have that property). This is why the MATLAB expression `'A':'Z'` always produced a row vector of length 26, containing all the letters in order and no other characters—another important contribution to portability.

2.5 Syntax Diagrams

The formal definition of the language, and a kind of flow chart for the parser-interpreter, was provided by 11 syntax diagrams. They defined these eleven syntactic categories: `line`, `statement`, `clause`, `expression`, `term`, `factor`, `number`, `integer`, `name`, `command`, and `text`.

Briefly, the recursive core of the language was:

- An expression is terms separated by `+` and `-` signs.
- A term is factors separated by `*`, `/` and `\` signs.
- A factor is a number, a name, or an expression in parentheses `( )`.

In the 1970s, Fortran programs could not be recursive. All that meant in practice was that a subroutine could not call itself. So, in the original MATLAB program, EXPR called TERM, TERM called FACTOR, and FACTOR would call EXPR if it encountered a parenthesis. It was necessary to have two arrays, one to manage a stack of subroutine calls, and one to manage the matrices in the workspace.

The complete syntax diagrams, as they appeared in the original line-printer User's Guide, are presented in Appendix A, with commentary.

2.6 User Function

Classic MATLAB allowed for exactly one external user-defined function, named `USER`. The user had to write this function in Fortran using a prescribed declaration:

```
SUBROUTINE USER(A,M,N,S,T)
REAL or DOUBLE PRECISION A(M,N),S,T
```

This interface allowed MATLAB code to call the `USER` function with one matrix of any shape and zero, one, or two scalar arguments, and one matrix would be returned. After the external Fortran

Fig. 7. Jack Little, founder and CEO of MathWorks, in 2000. (Source: MathWorks.)

subroutine had been written, it had to be compiled and linked to the MATLAB object code within the local operating system before running the MATLAB interpreter.

2.7 Precursors

The most important software precursors to Classic MATLAB were Algol 60 and Fortran II. Algol provided the formal structure and, through Wirth's Pascal and PL/0, the parsing techniques [NA Moler 2015]. Fortran, at the time, was well established as the language for scientific computing. LINPACK and EISPACK were written in Fortran. Other numerical software collections, such as IMSL and NAG, were also written in Fortran.

Today, some question why MATLAB indexing is 1-based and not 0-based. It is because both Algol and Fortran are 1-based, as is linear algebra itself.

3 FROM CLASSIC MATLAB TO A COMMERCIAL PRODUCT

Moler spent the 1978–79 academic year at Stanford, as a Visiting Professor of Computer Science. During the Winter quarter (January through March 1979) he taught the graduate course Advanced Topics in Numerical Analysis (CS238b) and introduced the class to his simple matrix calculator [Trefethen 2000, page xiii]; Some of the students were studying subjects like control theory and signal processing, which he knew nothing about. Matrices were central to the mathematics in these subjects, though, and MATLAB was immediately useful to these students.

Jack Little (see Figure 7) had been in the graduate engineering program at Stanford. A friend of his who took the course showed him MATLAB, and he immediately adopted it for his own work in control systems.

3.1 Developing MathWorks MATLAB

In 1983 Little suggested the creation of a commercial product for the IBM PC based on MATLAB. Moler thought that was a good idea but didn't join Little initially. The IBM PC had been introduced only two years earlier and was barely powerful enough to run something like MATLAB, but Little anticipated its evolution. It included a socket for the Intel 8087 chip, which performed floating point calculations in hardware, which was essential for MATLAB. He left his job, bought a Compaq PC clone at Sears, moved into the hills behind Stanford, and, with Moler's encouragement, spent a year and a half creating a new and extended version of MATLAB. The machine had only 256kB of memory and no hard disc; Jack had to swap 5-1/4 inch floppies to compile programs. Little replaced all Moler's Fortran, including the routines from EISPACK and LINPACK, with new code written in C. A friend, Steve Bangert, joined the project and worked on the new MATLAB in his spare time.

Little and Bangert lived several miles apart on the San Francisco Peninsula. They would meet regularly in a parking lot half-way between their homes and exchange floppy discs with their latest versions of the emerging software. Steve wrote the parser and interpreter, while Jack wrote the math libraries, including translations from Fortran into C of about a dozen routines from LINPACK and Classic MATLAB. Jack also wrote the first Control System Toolbox.

Little, Bangert, and Moler incorporated MathWorks in California on Friday, December 7, 1984. This was Jack Little's business plan, dated March 11, 1983 [NA Moler 2006]:

> This brief note describes a technical software product line. The market for this product line is the scientific and technical communities. A combination of events suggests that the timely development and introduction of this product line will be highly successful. The product will be unique and revolutionary. Combining the technology of 1) mice and windows, 2) matrix and APL environments, and 3) direct manipulation, will do for engineers what Lotus 1-2-3 has done for the business world. A product for the UNIX environment, the software is assured the largest target machine base. The product has bright prospects for longevity. The kernel forms the basis for many vertical product lines. There is no approach more likely to capture the engineering market.

PC-MATLAB made its debut the next week at the 23rd IEEE Conference on Decision and Control in Las Vegas. The first sale, early in 1985, was to Nick Trefethen of the Massachusetts Institute of Technology (now of Oxford University) [Trefethen 2000, page xiii], who eventually wrote the book *Spectral Methods in MATLAB* and has recently been involved in the creation and development of the MATLAB-based Chebfun software project [Trefethen 2007, 2015; Moler 2015].

While rewriting the entire code base for Classic MATLAB from Fortran into C, Little and Bangert made many important modifications and improvements to create a new and extended version; we will refer to this version and its immediate successors as *MathWorks MATLAB*. The most significant innovations were in functions, toolboxes, and graphics. The complete contents of version 1.3 of PC-MATLAB is available at [NA Moler 2018e].

PC-MATLAB was soon followed by Pro-MATLAB on Unix workstations in 1986. At one time there were at least half a dozen manufacturers of these systems. The most important was Sun Microsystems. The following year, MATLAB became available on Apple Macintosh.

3.2 Functions

In the mid-1980s, there was a variety of operating systems available on PCs, Macs, and Unix workstations, but they all had some sort of hierarchical file system. The MATLAB function naming mechanism uses this underlying computer file system.

PC-MATLAB was made extensible by scripts and functions. (This facility replaced the single USER function of Classic MATLAB—see Section 2.6.) A script or function is a body of MATLAB

code stored in a file with the extension "`.m`". (At the time such files were referred to as "M-files," but that terminology has fallen out of use today.) If the file content begins with the keyword `function`, then it is a function; otherwise it is a script. The name of the file, minus the `.m`, is the name of the script or function. For example, a file named `hello.m` containing the single line

```
disp('Hello World')
```

is a MATLAB script. Typing the file name without the `.m` extension at the command prompt,

```
hello
```

produces the traditional greeting

```
Hello World
```

Functions usually have input and output arguments. Here is a simple example that uses a vector outer product to generate the multiplication table familiar to all elementary school students.

```
function T = mults(n)
   j = 1:n;
   T = j'*j;
end
```

(Note that each of the two statements in this function ends with a semicolon, which indicates that the computed value should not be printed.)

The following statement produces a 5-by-5 multiplication table:

```
T = mults(5)

T =
  1     2     3     4     5
  2     4     6     8    10
  3     6     9    12    15
  4     8    12    16    20
  5    10    15    20    25
```

Input arguments to MATLAB functions are "passed by value," meaning that functions behave as if their input arguments are copied from the caller. This leads to a simple, straightforward mental model of function behavior. For memory efficiency, though, MATLAB also implements a "copy on write" behavior, in which memory copies are deferred and often never actually made. For example, because this function never changes its "copies" of the input arguments A and B, no actual memory copy is made when the function is invoked.

```
function C = commutator(A,B)
   C = A*B - B*A;
end
```

Functions defined by MATLAB code, like built-in functions, may return more than one value. When calling a function that returns more than one value, the user can use a statement to assign the multiple values to multiple variables.

3.3 Dynamic Linking of Compiled Code

PC-MATLAB introduced an external interface to allow Fortran and C programs to be written, compiled, and then dynamically loaded into MATLAB and executed. Each external such function resided in a separate file called a "mexfile" (nowadays the term is usually written "MEX file"). The interface included a set of library functions and associated header files that allowed Fortran and C programs to determine the dimensions of an array argument, find the real and imaginary

parts, construct result arrays, and so on. A typical mexfile contains "wrapper" code that reads the MATLAB data structures, finds their pieces, and assembles the MATLAB result(s), along with one or more core functions that do the desired computation, often reused with little change from an existing Fortran or C application. [Johnson and Moler 1994, pages 120–121]

Mexfiles worked like MATLAB code files: if a file named `xxx.mex` is on the search path of the interpreter, it is taken to be the definition of the function `xxx`. (Actually, the precise suffix was and is dependent on the architecture and operating system; for example, for a Sun 4 workstation, the suffix `mex4` was used.) If both a MATLAB code file and an appropriate mexfile are both present, the mexfile is used rather than the MATLAB file. (This fact would become especially important when the first MATLAB compiler was constructed—see Section 4.4.) [Johnson and Moler 1994, pages 121]

3.4 Language Changes and Extensions

PC-MATLAB abandoned the use of the MATLAB internal character codes and committed to the use of ASCII as a character set. As a result, square brackets '`[ ]`' were always used rather than angle brackets '`< >`' to delimit lists that produce matrices. Another consequence is that names became case-sensitive (uppercase and lowercase letters were distinguished in names, rather than converting lowercase to uppercase on input); however, a function `casesen` was introduced to allow the user to "turn off" case sensitivity. The "official spellings" of built-in functions and commands became lowercase.

In PC-MATLAB, names were no longer limited to four significant characters. The names of certain functions such as `diary`, `print`, `rcond`, `roots`, `round`, and `schur` were now officially five characters long.

The syntax >*expression*< for performing "macro replacement" (see Appendix A) was eliminated, and its functionality was replaced by the function `eval`.

The operator '`**`' used in Classic MATLAB for exponentiation (matrix power) was replaced in PC-MATLAB by '`^`', and a corresponding operator '`.^`' was introduced to represent elementwise exponentiation.

The Kronecker tensor product and division operators '`.*`' and '`.\`' and '`./`' were removed (but the Kronecker tensor product function `kron` was retained).

In Classic MATLAB, the apostrophe '`'`' was used to indicate (complex conjugate) matrix transposition. PC-MATLAB introduced the use of '`.'`' to indicate matrix transposition without taking the complex conjugate.

PC-MATLAB introduced the the use of '`~`' as a prefix "logical NOT" operator, and replaced '`<>`' with '`~=`' as the "not equal" comparison operator.

PC-MATLAB introduced several new "permanent variables": `inf` and nan (to support IEEE 754 arithmetic), `pi` (a useful constant indeed), and `nargin` and `nargout`, to allow a user-written function to determine how many arguments it had been passed and how many results were expected.

PC-MATLAB added a flag to each matrix to indicate whether it should be regarded as a string for output purposes, and a function `setstr` to enable the user to so flag a matrix. (This was the first "baby step" toward having more than one type in the language.)

PC-MATLAB introduced a `break` statement to exit from a loop (as in C) and an `elseif` clause for use in multi-way `if` constructions.

PC-MATLAB allowed the condition in a `while`, `if`, or `elseif` clause to be any expression, not just a single comparison operation.

3.5 New Functions

Here is a brief summary of most of the new functions introduced into PC-MATLAB as of version 1.3 [NA Moler 2018e] as compared to the Classic MATLAB of 1981 [NA Moler 2018e]:

- Elementary math functions: `fix` (round towards zero), `rem` (remainder), `sign` (signum function), `tan` (tangent), `asin` (arcsine), `acos` (arccosine), `atan2` (four quadrant arctangent), `log10` (logarithm base 10), `bessel` (Bessel functions)
- Attributes and array manipulation: `max` (maximum value in vector), `min` (minimum value in vector), `cumsum` (cumulative sum of elements in vector), `cumprod` (cumulative product of elements in vector), `mean` (mean value in a vector), `median` (median value in a vector), `std` (standard deviation in a vector), `length` (length of a vector), `sort` (sorting), `find` (find array indices of logical values), `hist` (histograms)
- Array building functions: `invhilb` (generate inverse Hilbert matrix), `zeros` (generate an array of all zeros)
- Matrix functions: `expm` (matrix exponential), `logm` (matrix logarithm), `sqrtm` (matrix square root), `funm` (arbitrary matrix functions)
- Polynomials: `polyval` (evaluate polynomial)
- Signal processing: `filter` (digital filter), `fft` (fast Fourier transform), `ifft` (inverse fast Fourier transform), `conv` (convolution)
- Plotting (augmenting the `plot` function already in Classic MATLAB): `shg` (show graphics screen), `cla` (clear current axes), `clg` (clear graphics screen), `loglog` (loglog X-Y plot), `semilogx` (semi-log X-Y plot), `semilogy` (semi-log X-Y plot), `polar` (polar plot), `mesh` (3-dimensional mesh surface), `title` (plot title), `xlabel` (x-axis label), `ylabel` (y-axis label), `grid` (draw grid lines), `axis` (manual axis scaling), `disp` (compact matrix display)
- "System" operations: `eval` (interpret text in a variable), `setstr` (set flag indicating matrix is a string), `exist` (check if a variable or function exists), `demo` (run demonstrations), `pack` (memory garbage collection and compaction)

3.6 Toolboxes

The support in MATLAB for user-defined functions, either as MATLAB code or Fortran and C code in mexfiles, added to emerging operating systems support for hierarchical file systems, made it possible to organize MATLAB into *toolboxes*, collections of functions devoted to particular applications. This provided a form of language extensibility that proved to be crucial to the evolution of MATLAB. The first toolbox, named simply MATLAB Toolbox, implemented all the mathematical and general-purpose functions that were not built into the interpreter.

Little was a controls and signals engineer. So, he wrote the first version of Control System Toolbox, which was released in 1985. And, together with Loren Shure, MathWorks' third employee, he wrote the first version of Signal Processing Toolbox in 1987.

Two external academic authors, experts in their fields, wrote early toolboxes. Professor Lennart Ljung, of the Linköping Institute of Technology in Sweden, wrote System Identification Toolbox in 1988 [Ljung 2014; NA Ljung 2012b; NA Ljung 2012a] to accompany his widely used textbook on the subject [Ljung 1987]. Professor Carl de Boor, of the University of Wisconsin—Madison, wrote Spline Toolbox in 1990 [de Boor 2004]—its successor is today's Curve Fitting Toolbox—to be used with his authoritative book on B-splines [De Boor 1978].

3.7 Graphics

Technical computer graphics have been an essential part of MATLAB since the first MathWorks version. Many users come to MATLAB just for its plotting capabilities. Their data comes from physical experiments or computation with other software. MATLAB produces plots suitable for a report or technical paper.

Operations on arrays proved to be convenient for memory-mapped graphics. The array data type proved to be natural for 2-D (X-Y) and 3-D (X-Y-Z) charting. But even the 1981 version of Classic

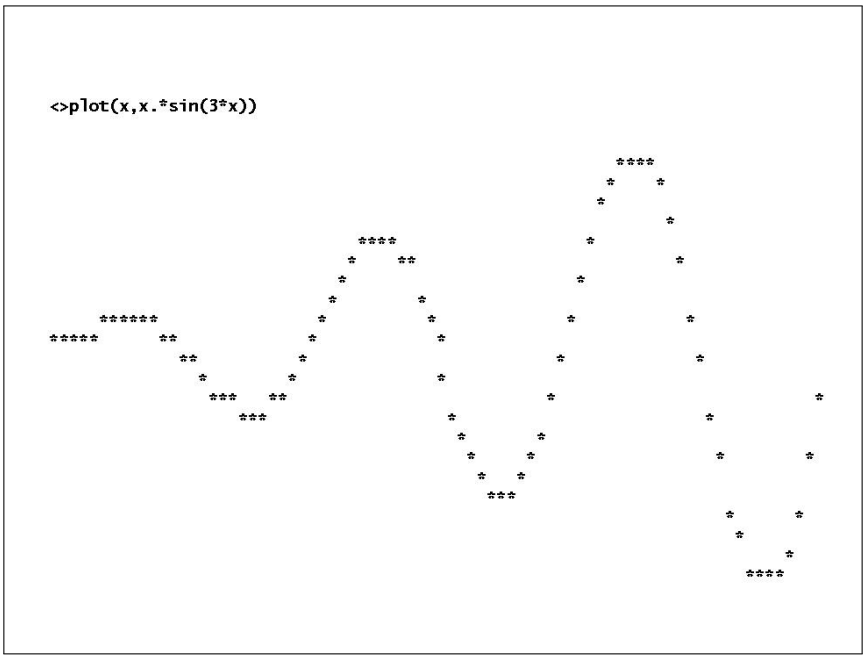


Fig. 8. This portable machine-independent plot from Classic MATLAB was partly a joke, partly anticipation of future graphics capabilities.

MATLAB had a rudimentary `plot` function that produced "ASCII plots" (see Figure 8). Today's MATLAB produces a more sophisticated plot for the same function (see Figure 9).

The textbook *Matrices and MATLAB* by Marvin Marcus, a professor at the University of California, Santa Barbara was developed on a Macintosh-IIci and made use of early computer graphics [Marcus 1993]. Figure 10 shows the graph of the Julia set for the function $f(z) = z^2 + c$ with $c = 0.1 + 0.8i$ as plotted by MATLAB on that Macintosh IIci. Much more detail is visible in a plot produced by MATLAB today (see Figure 11).

Let's look at three more examples. Notice that none of them has anything to do with numerical linear algebra, but they all rely on vector notation and array operations.

The following code produces the plot shown in Figure 12:

```
x = -pi:pi/1024:pi;
y = tan(sin(x))-sin(tan(x));
plot(x,y)
xlabel('x')
title('tan(sin(x)) - sin(tan(x))')
```

The following code produces the plot shown in Figure 13:

```
X = (-3:1/8:3)*ones(49,1);
Y = X';
Z = 3*(1-X).^2.*exp(-(X.^2) - (Y+1).^2) ...
     - 10*(X/5 - X.^3 - Y.^5).*exp(-X.^2-Y.^2) ...
     - 1/3*exp(-(X+1).^2 - Y.^2);
mesh(X,Y,Z)
```

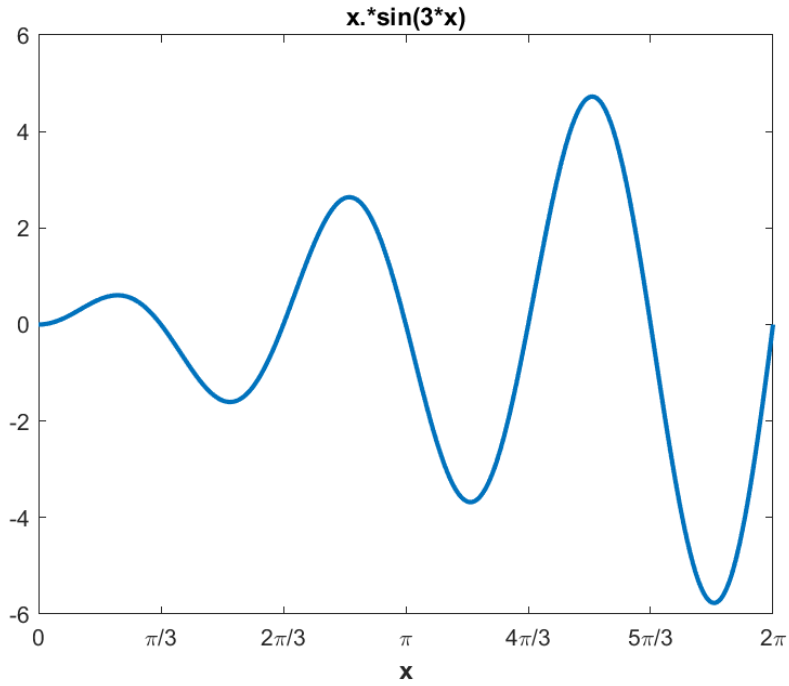


Fig. 9. A plot for the same function, produced by contemporary MATLAB.

From Control System Toolbox, a Bode diagram shows the frequency response of a dynamic system (see Figure 14).

In 1988, Moler characterized MATLAB as not only a "matrix laboratory" but also as a "mathematical visualization laboratory" [Moler 1988].

3.8 Flops Count

The Classic MATLAB interpreter counted every floating point operation ("flop") performed, and reported the count if a statement is terminated with an extra comma. This feature was carried forward into MathWorks MATLAB, but is not available in today's MATLAB, where the inner loops of matrix and other computations must be as efficient as possible. Figure 15 presents a demonstration (in a 1984 version of MATLAB) of the fact that inverting an n-by-n matrix requires slightly more than n^3 floating-point ops.

The flops counts were also made accessible to the user through a built-in variable `flop`, a 1-by-2 matrix whose first entry is the number of flops performed by the immediately previous input, and whose second entry is the (inclusive) cumulative sum of the values taken on by the first element. In later versions of MATLAB, a new function `flops` was introduced to return and reset flop counts.

Support for tracking and reporting flop counts was eliminated when the routines that had been adapted from EISPACK and LINPACK (with extra coding to support both the `FLOP` variable and the `CHOP` command) were replaced by LAPACK (see Section 5.1).

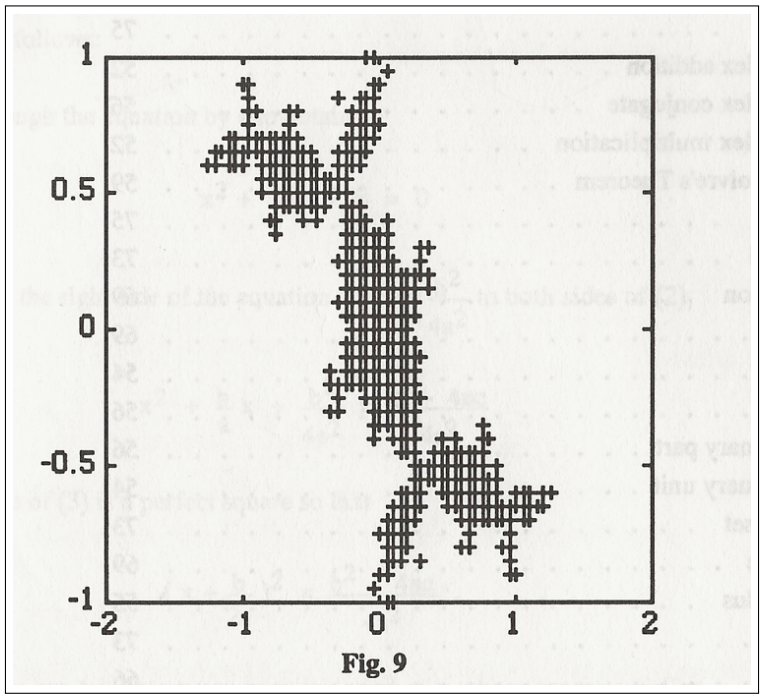


Fig. 10. A plot of a Julia set produced with MATLAB in 1992 by Marvin Marcus [Marcus 1993] on an Apple Macintosh IIci. (Source: Prentice Hall.)

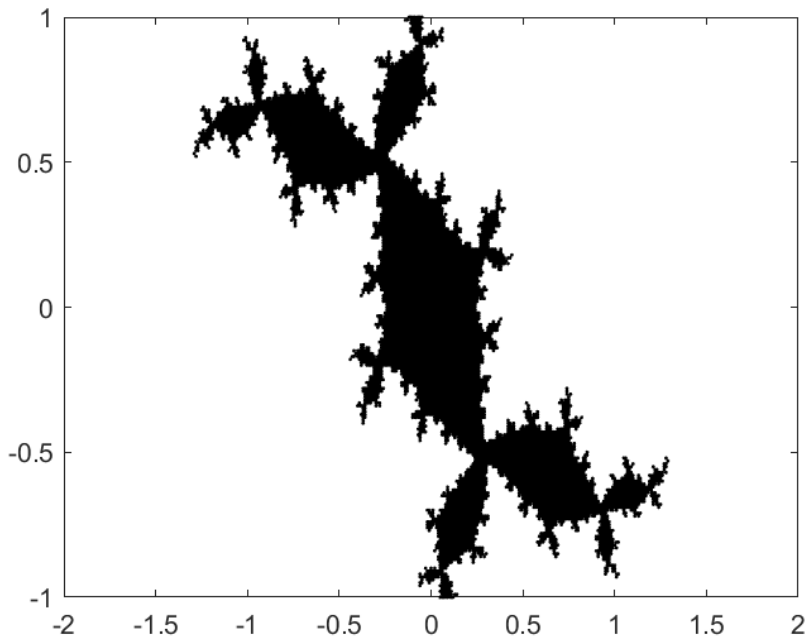


Fig. 11. A plot of the same Julia set produced by contemporary MATLAB.

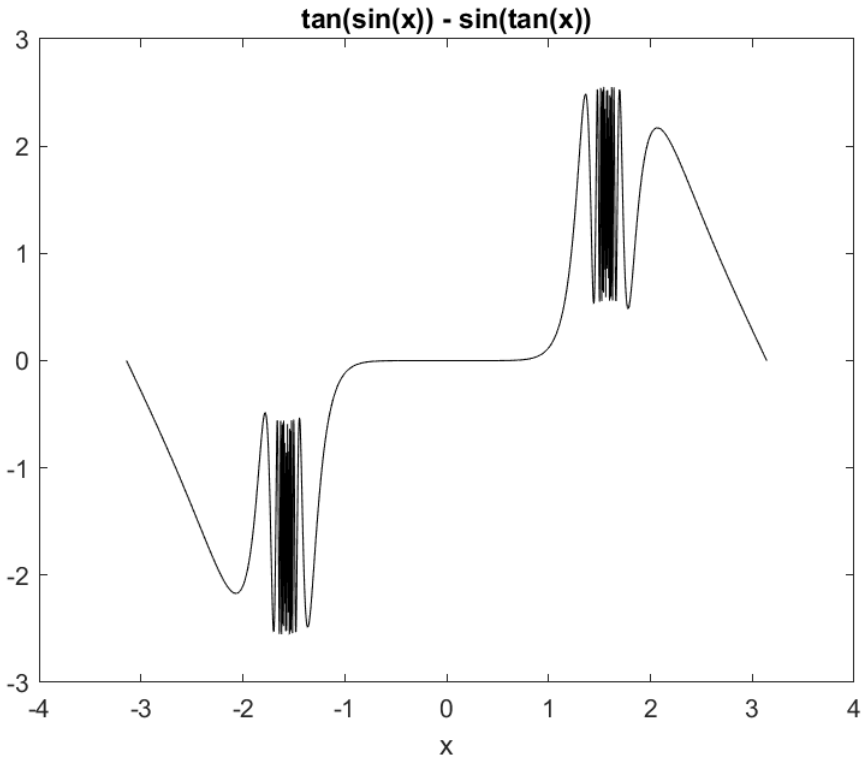


Fig. 12. A two-dimensional plot produced by contemporary MATLAB.

4 EVOLUTION OF MATLAB

While preserving its roots in matrix mathematics, MATLAB has continued to evolve to meet the changing needs of engineers and scientists. Here are some of the key developments [NA Moler 2018c; NA Moler 2018d].

4.1 Data Types

By 1992 MathWorks had over 100 staff members and by 1995, over 200. About half of them had degrees in science and engineering and worked on the development of MATLAB and its various toolboxes. As a group, they represented a large and experienced collection of MATLAB programmers. Suggestions for enhancements to the language often come from these users within the company itself.

MATLAB was being used for the design and modeling of embedded systems, which are controllers with specific real-time functions within larger electrical or mechanical systems. Such controllers often have only single precision floating point or fixed-point arithmetic. The designers of embedded systems desired the ability develop their algorithms using the native arithmetic of the target processors. This motivated the introduction of several new data types into MATLAB, but without the use of actual type declarations.

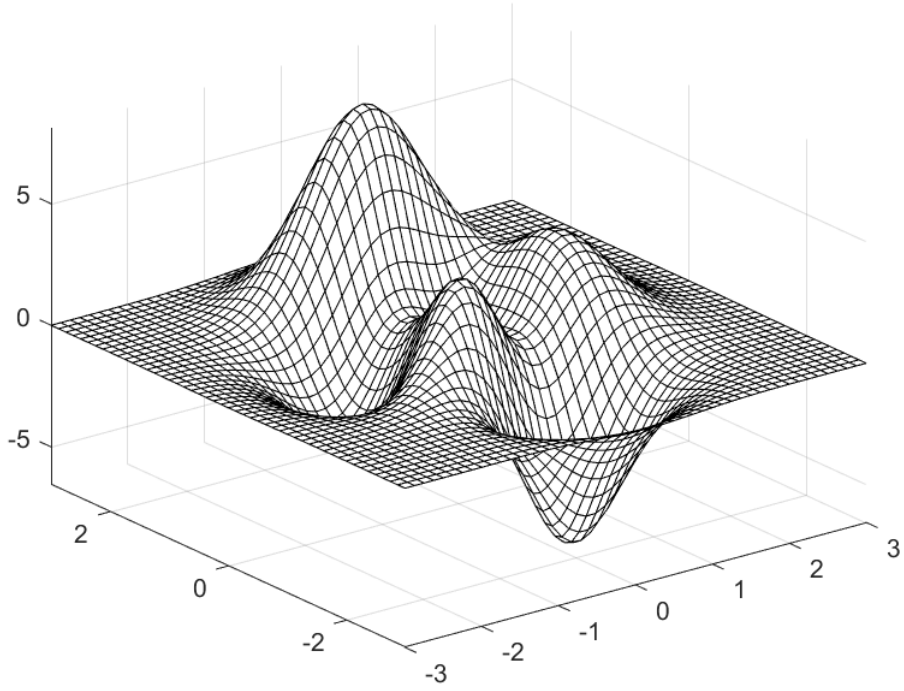


Fig. 13. A three-dimensional plot produced by contemporary MATLAB.

For many years MATLAB had only one numeric data type, IEEE standard 754 double precision floating point, stored in the 64-bit format.

```
format long
phi = (1 + sqrt(5))/2

phi =
   1.618033988749895
```

Support for numeric data types other than double was gradually added to MATLAB between 1996 and 2004. Requiring only 32 bits of storage, single precision floating point cuts memory requirements for large arrays in half. It may or may not be faster.

MATLAB does not have declarations, so single precision variables are obtained from double precision ones by an executable conversion function.

```
p = single(phi)

p =
  single
   1.6180340
```

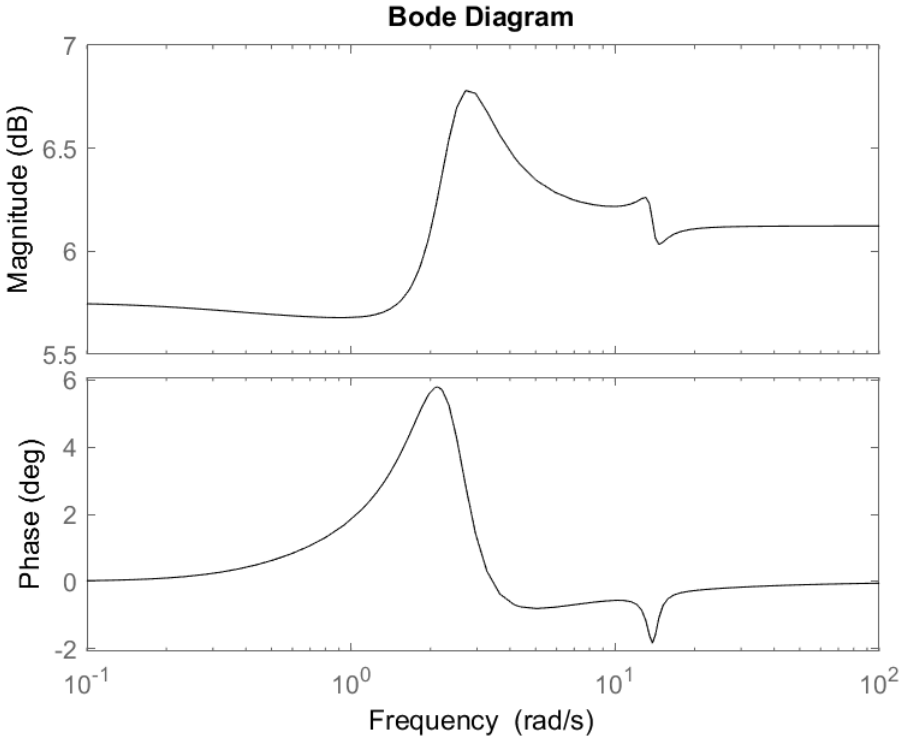


Fig. 14. A Bode plot from Control Systems Toolbox shows the magnitude and phase shift of the frequency response of a system.

Unsigned integer types uint8 and uint16 were initially used just for storage of images; no arithmetic was required.

In 2004 MATLAB 7 introduced full arithmetic support for single precision, four unsigned integer data types (uint8, uint16, uint32, and uint64) and four signed integer data types (int8, int16, int32, and int64). MATLAB 7 also introduced the new logical type.

```
q = uint16(1000*phi)
r = int8(-10*phi)
s = logical(phi)

q =
  uint16
   1618
r =
  int8
   -16
s =
  logical
   1
```

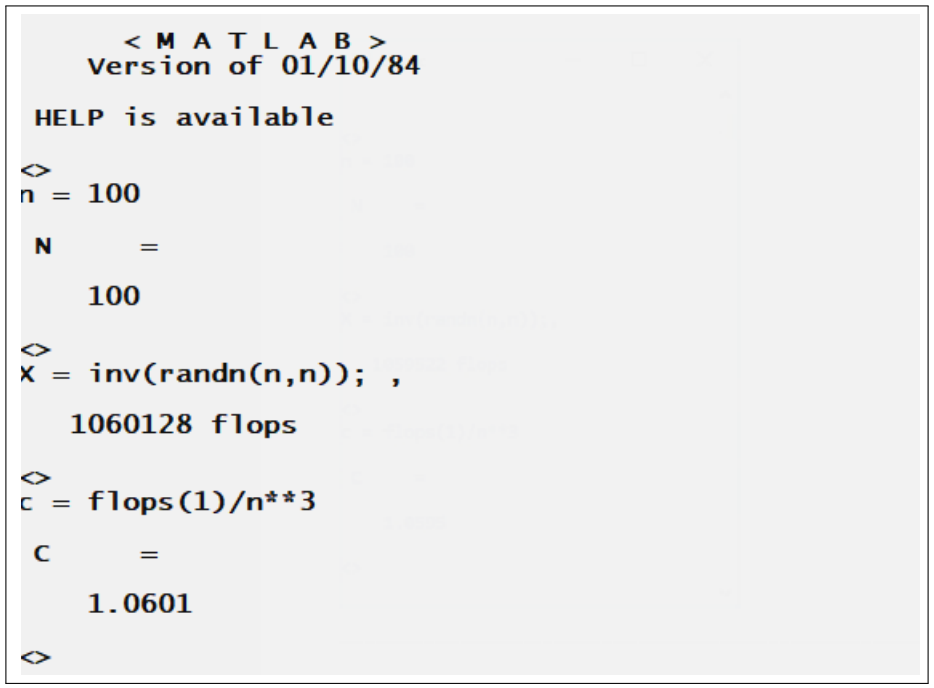


Fig. 15. Demonstration of the use of an extra comma to print the flops count in Classic MATLAB.

Let's see how much storage is required for these variables:

```
Name      Size   Bytes   Class
p         1x1        4   single
phi       1x1        8   double
q         1x1        2   uint16
r         1x1        1   int8
s         1x1        1   logical
```

4.2 Sparse Matrices

In the folklore of matrix computation, it is said that J. H. Wilkinson defined a sparse matrix as any matrix with enough zeros that it pays to take advantage of them. (He may well have said exactly this to his students, but it ought to be compared with this passage he wrote in the *Handbook for Automatic Computation*:

> iv) The matrix may be sparse, either with the non-zero elements concentrated on a narrow band centered on the diagonal or alternatively they may be distributed in a less systematic manner. We shall refer to such a matrix as dense if the percentage of zero elements or its distribution is such as to make it uneconomic to take advantage of their presence.

[Wilkinson and Reinsch 1971, page 191].)

Iain Duff is a British mathematician and computer scientist known for his work on algorithms and software for sparse matrix computation. In 1990, he visited Stanford and gave a talk in the numerical analysis seminar. Moler attended the talk. So did John Gilbert, who was then at Xerox Palo Alto Research Center, and Rob Schreiber, from Hewlett Packard Research. They went to lunch

at Stanford's Tresidder Union and there Gilbert, Schreiber and Moler decided it was about time for MATLAB to support sparse matrices.

MATLAB 4 introduced sparse matrices in 1992 [Gilbert et al. 1992]. A number of design options and design principles were explored. It was important to draw a clear distinction between the *value* of a matrix and any specific *representation* that might be used for that matrix [Gilbert et al. 1992, page 4]:

> We wish to emphasize the distinction between a matrix and what we call its *storage class*. A given matrix can conceivably be stored in many different ways—fixed point or floating point, by rows or by columns, real or complex, full or sparse—but all the different ways represent the same matrix. We now have two matrix storage classes in MATLAB, full and sparse.

Four important design principles emerged.

- The value of the result of an operation should not depend on the storage class of the operands, although the storage class of the result may. [Gilbert et al. 1992, page 10]
- No sparse matrices should be created without some overt direction from the user. Thus, the changes to MATLAB would not affect the user who has no need for sparsity. Operations on full matrices continue to produce full matrices.
- Once initiated, sparsity propagates. Operations on sparse matrices produce sparse matrices. And an operation on a mixture of sparse and full matrices produces a sparse result unless the operator ordinarily destroys sparsity. [Gilbert et al. 1992, page 5]
- The computer time required for a sparse matrix operation should be proportional to the number of arithmetic operations on nonzero quantities. [Gilbert et al. 1992, page 3]

(Some of these principles were rediscovered later in the design effort to add arithmetic support for single precision and the integer types.)

Only the nonzero elements of sparse matrices are stored, along with row indices and pointers to the starts of columns. The only change to the outward appearance of MATLAB was a pair of functions, `sparse` and `full`, to allow the user to provide a matrix and obtain a copy that has the specified storage class. Nearly all other operations apply equally to full or sparse matrices. The sparse storage scheme represents a matrix in space proportional to the number of nonzero entries, and most of the operations compute sparse results in time proportional to the number of arithmetic operations on nonzero entries. In addition, sparse matrices are printed in a different format that presents only the nonzero entries.

As an example, consider the classic finite difference approximation to the Laplacian differential operator. The function `numgrid` numbers the points in a two-dimensional grid, in this case n-by-n points in the interior of a square.

```
n = 100;
S = numgrid('S',n+2);
```

The function `delsq` creates the five-point discrete Laplacian, stored as a sparse N-by-N matrix, where $N = n^2$. With five or fewer nonzeros per row, the total number of nonzeros is a little less than $5N$.

```
A = delsq(S);
nz = nnz(A)

nz =
   49600
```

For the sake of comparison, let's create the full version of the matrix and check the amount of storage required. Storage required for A is proportional to N, while for F it is proportional to N^2.

```
F = full(A);

  Name       Size              Bytes   Class    Attributes
  A       10000x10000          873608   double     sparse
  F       10000x10000       800000000   double
```

Let's time the solution to a boundary value problem. For the sparse matrix the time is $O(N^2)$. For $n = 100$ and $N = 10000$ it's instantaneous.

```
b = ones(n^2,1);
tic
u = A\b;
toc

Elapsed time is 0.029500 seconds.
```

The full matrix time is $O(N^3)$. It requires several seconds to compute the same solution.

```
tic
u = F\b;
toc

Elapsed time is 7.810682 seconds.
```

4.3 Empty Matrices

Classic MATLAB had only one empty matrix, having size 0×0 and written as <> (or as [], starting with MathWorks MATLAB). But a more careful analysis by Carl de Boor [de Boor 1990] (which in turn relied to some extent on similar work done for APL) persuaded us that MATLAB should support empty matrices of all sizes, that is, of sizes $n \times 0$ and $0 \times n$ for any nonnegative n. (And when arrays were introduced, empty arrays of all possible shapes were similarly supported.)

Empty matrices have some surprising, but logically consistent properties.

```
x = ones(4,0)

x =
  4×0 empty double matrix
```

The inner product of an empty column vector and an empty row vector with the same length is empty.

```
z = x'*x

z =
     []
```

The outer product of an empty column vector and an empty row vector with any length is a full matrix of zeros. This is a consequence of the convention that an empty sum is zero.

```
y = ones(0,5)
A = x*y

y =
  0×5 empty double matrix
```

```
A =
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
```

As de Boor observed, one use for empty matrices is to provide appropriate base cases (or termination cases) for iterative or inductive matrix algorithms, so that the processing of such initial or final cases does not require separate code.

4.4 Compiling MATLAB

In the early 1990s, Stephen C. "Steve" Johnson (also known for, among other things, his work on `yacc` [NA Johnson 1975], `lint` [NA Johnson 1978], and the Portable C Compiler [Johnson 1978; NA Johnson 1979]) constructed the first compiler for MATLAB, which was described in a 1994 paper [Johnson and Moler 1994] and released in 1995 [NA Moler 2006].

Much of the infrastructure needed to interface the compiler to the MATLAB interpreter was already in place: user-defined MATLAB functions were already stored in separate source-code files (see Section 3.2) and there was already a facility for dynamically loading files of compiled code called mexfiles (see Section 3.3). Moreover, if a source-code file and a compiled file for the same function were already present, the MATLAB interpreter would use the compiled file. All a MATLAB compiler needed to do was read a file of MATLAB code and produce a corresponding compiled mexfile.

Many MATLAB functions already ran with acceptable speed in the interpreter because most of their work was done by applying EISPACK and LINPACK matrix operations to large matrices. But some codes required the use of scalar operations. For example, solving a tridiagonal system of linear equations could be done using full-matrix operations, but the cost would be quadratic in the size of the system, and the alternative of using loops in a user function written in the MATLAB programming language would have a cost that is linear in the size of the system but multiplied by a large constant factor related to the overhead of the interpreter in processing individual scalar operations. The MATLAB compiler was able to improve the speed of such a tridiagonal solver by a factor of 100 for a system of size 1000 (a diagonal of size 1000 and two diagonals of size 999), and by a factor of almost 150 for a system of size 5000 [Johnson and Moler 1994].

The most important work of this first MATLAB compiler was type analysis: is any given use of a variable, or any given computed quantity, always non-complex? Always integer? Always scalar rather than a matrix? The compiler used def-use chains, knowledge about language constructs and built-in functions, and a least-fixed-point type analysis to answer such questions. For example, the compiler knows that the expression `1:n` is always an array of integers, even if `n` is not an integer; it follows that for a clause `for i=1:n` the variable `i` is always scalar and always takes on integer values. Such information makes it much easier to compile expressions (and especially subscript expressions) involving `i`; for example, computing `i*i` is much, much simpler when the compiler knows that `i` is a scalar integer rather than a complex matrix. If `i` is scalar, then so is `i*i`, and therefore `a(i*i)` is also scalar (though not necessarily an integer—it depends on what the type of `a`). This is an example of bottom-up propagation of constraints. As an example of top-down propagation, consider this code fragment:

```
function a=f(x)
    ...
    y(3) = 4 + x;
    ...
```

The input variable `x` might be *a priori* of any shape, but because the constant `3` is a scalar, `y(3)` is necessarily scalar (bottom-up constraint), therefore `4 + x` must be scalar (top-down), therefore `x` must be scalar at that point of use (top-down).

This original MATLAB compiler iteratively applied both top-down constraints and bottom-up constraints, as well as recursively analyzing multiple functions when appropriate. The compiler also resolved overloaded functions when possible by examining the number and types of the arguments and results of a function call.

The compiler did not have to compile every MATLAB function. If the type analysis of some particular function proved to be too elusive to allow generation of efficient code, the MATLAB source-code version of that function was still available to the calling program. It turns out that many people were more interested in using this compiler to protect intellectual property than expecting it to achieve faster execution.

4.5 The Pentium FDIV Bug

Steve Johnson recalls [Johnson 2020]:

> When I was writing the compiler, I worked from [California] and flew to Boston roughly once a month for several days. (It was an interesting time, since I was working half time for MathWorks, and the other half time I was consulting for HP and writing the modulo scheduling code for what later became the Itanium. Two very interesting and successful projects, but very very different technically.)
> I would share an office with Cleve when I was in Boston. One time, when I visited, the FDIV bug had just broken open and Cleve's phone would not stop ringing. I believe he was interviewed by CNN at that time.

Indeed he was—and by the Associated Press [NA Press 1994], the *New York Times* [Markoff 1994], and many other news outlets.

Intel's Pentium processor had included a brand-new implementation of the floating-point divide instruction `FDIV`, which used a variation of the Sweeney-Robertson-Tocher (SRT) algorithm that was subtle, clever, and fast because it relied on a hardware lookup table with 2048 entries, of which only slightly over half were relevant because only a trapezoid-shaped subset of the 128×16 table could ever be accessed by the algorithm. Unfortunately, because of some mishap, five of the 1066 relevant entries mistakenly held the value 0 rather than 2. Even more unfortunately, these entries were accessed only in very rare cases and were missed by Intel's randomized testing process; furthermore, the bad entries could never be accessed during the first 8 steps of the division algorithm, and so incorrect results differed only slightly from the true results: enough to matter in high-precision calculations, but not enough to stand out unless you were looking for the problem or otherwise being very careful.

But Prof. Thomas Nicely of Lynchburg College did notice, because he was running the same algorithm (a calculation of the sum of the reciprocals of twin primes) on multiple computers, and the results obtained on a Pentium-based system differed from the others. After spending some months checking out other possible reasons for the discrepancy, he notified Intel on October 24, 1994, and send email to some other contacts on October 30. One of the recipients of Nicely's memo posted it on the CompuServe network. Alex Wolfe, a reporter for the EE Times, spotted the posting and forwarded it to Terje Mathisen of Norsk Hydro in Norway. Within hours of receiving Wolfe's query, Mathisen confirmed Nicely's example, wrote a short assembly language test program, and on November 3 initiated a chain of Internet postings in newsgroup `comp.sys.intel` about the `FDIV` bug. A day later, Andreas Kaiser in Germany posted a list of two dozen numbers whose reciprocals are computed to only single precision accuracy on the Pentium. [NA Moler 2013d]

The story quickly made its way around the Internet and hit the press when a story by Wolfe broke the news in *Electronic Engineering Times* on November 7.

Meanwhile, Tim Coe, a floating-point-unit (FPU) designer at Vitesse Semiconductor in southern California, saw in Kaiser's list of erroneous reciprocals clues to how other FPU designers had tackled the same task of designing division circuitry. He correctly surmised that the Pentium's division instruction employed a radix-4 SRT algorithm, producing two bits of quotient per machine cycle and thereby making the Pentium twice as fast at division as previous Intel chips running at the same clock rate.

Coe created a model that explained all the errors Kaiser had reported in reciprocals. He also realized that division operations involving numerators other than one potentially had even larger relative errors. His model lead him to a pair of seven digit integers whose ratio 4195835/3145727 appeared to be an instance of the worst case error. He posted this example to `comp.sys.intel` on November 14.

Now Moler had first learned of the FDIV bug a few days prior to Coe's post, from another electronic mailing list about floating-point arithmetic maintained by David Hough. At that point Moler began to follow `comp.sys.intel`. At first, he was curious but not alarmed. But Coe's discovery, together with a couple of customer calls to MathWorks tech support, raised his level of interest considerably. On November 15, Moler posted to the newsgroups `comp.soft-sys.matlab` and `comp.sys.intel`, summarizing what was known up to then, including Nicely's example and Coe's example, and pointing out that the divisor in both cases was a little less than 3 times a power of 2.

On November 22, two engineers at the Jet Propulsion Laboratory suggested to their purchasing department that the laboratory stop ordering computers with Pentium chips. Steve Young, a reporter with CNN, heard about JPL's decision, found Moler's posting on the newsgroup, gave him a quick phone call, then sent a video crew to MathWorks in Massachusetts and interviewed him over the phone from California. That evening, CNN's *Moneyline* show used Young's news about JPL, and his interview with Moler, to make the Pentium Division Bug mainstream news. Two days later—it happened to be Thanksgiving—stories appeared in the *New York Times*, the *Boston Globe* [Zitner 1994], the San Jose *Mercury News* [Takahashi 1994], and others. Hundreds of articles appeared in the next several weeks.

In the meantime, Moler had begun collaborating with Coe, Mathisen, Peter Tang of Argonne National Laboratory, and several hardware and software engineers from Intel to develop, implement, test, and prove the correctness of software to work around the FDIV bug (and related bugs in the Pentium's on-chip tangent, arctangent, and remainder instructions). [NA Moler 2013d] By December 5, they had developed a clever workaround: examine the four high-order bits of the divisor's fraction (the part of the significand that is represented explicitly); if they are `0001`, `0100`, `0111`, `1010`, or `1101`, multiply both the dividend and divisor by 15/16 before performing the division operation. In all five cases, the effect of this scaling by 15/16 is to move the divisor away from a "risky" pattern to a "safe" pattern. In all cases the quotient of the scaled operands will be the same as the correct quotient of the original operands. [NA Moler 2016, file moler_5.txt: email message dated Mon Dec 5 6:20:44 EST 1994] A few days later they refined this to scale the operands by 15/16 only if the eight high-order bits of the divisor's fraction are `00011111`, `01001111`, `01111111`, `10101111`, or `11011111`, thus greatly reducing the number of cases that would be scaled. This technique was published to the newsgroups for all to use freely.

In the fall of 1994, MathWorks was still a small company with fewer than 250 employees. The product name, MATLAB, was known to customers, but the company name, MathWorks, was not widely known. On November 23, the company announced a version of MATLAB that could detect and correct the division bug. The public relations firm issued a press release with the headline

The MathWorks Develops Fix for the Intel Pentium(TM) Floating Point Error

On the next day—the day the story appeared in the New York Times and other major newspapers—this PR message showed up in the fax machines of media outlets all over the country. It turned out to be a very successful press release. [NA Moler 2013c]

The "Pentium Aware" release of MATLAB, release 4.2c.2, was made available at no charge to users of MATLAB for Windows on December 27. This version of MATLAB not only compensated for the Pentium Division Bug but also by default printed a short report each time the bug was encountered. It also offered options to suppress the messages, count the number of occurrences, or suppress the corrections altogether. [NA Moler 2016, file moler_7.txt: email message dated 27 Dec 1994 18:52:06 -0500]

4.6 Cell Arrays

By the mid-1990s, MATLAB users were going way beyond numerical computation and needed more general data structures than the rectangular, homogeneous, numeric matrix. Cell arrays were introduced in 1996. A cell array is an indexed, possibly inhomogeneous collection of MATLAB quantities—arrays of various sizes and types, strings, other cell arrays. Cell arrays are created by curly braces '{ }'.

```
c = {magic(3); uint8(1:10); 'hello world'}

c =
  3×1 cell array
    {3×3  double  }
    {1×10 uint8   }
    {'hello world'}
```

Cell arrays can be indexed by both curly braces and smooth parentheses. With braces, c{k} is the contents of the k-th cell. With parentheses, c(k) is a cell array containing the specified cells. Think of a cell array as a collection of mailboxes. With parentheses, box(k) is the k-th mailbox. With braces, box{k} is the mail in the k-th box.

```
M = c{1}
c2 = c(1)

M =
     8     1     6
     3     5     7
     4     9     2

c2 =
  1×1 cell array
    {3×3 double}
```

4.7 Structures

Structures, and associated "dot notation", as seen in C and Pascal, were also introduced in 1996. The graphics system now uses this notation. For another example, let's create a grade book for a small class.

```
Math101.name = {'Alice Jones';'Bob Smith';'Charlie Brown'};
Math101.grade = {'A-'; 'B+'; 'C'};
Math101.year = [4; 2; 3]
```

```
Math101 =
  struct with fields:
     name: {3×1 cell}
    grade: {3×1 cell}
     year: [3×1 double]
```

To call the roll, we need the list of names.

```
disp(Math101.name)

    'Alice Jones'
    'Bob Smith'
    'Charlie Brown'
```

Changing Charlie's grade to a 'W' involves both structure and array notation.

```
Math101.grade(3) = {'W'};

disp(Math101.grade)
    'A-'
    'B+'
    'W'
```

4.8 Numerical Methods

The textbook by Forsythe, Malcolm, and Moler [Forsythe et al. 1977] contains Fortran programs for common numerical methods, including interpolation, zero and minimum finding, quadrature and random number generation. Translating these programs into MATLAB provided the start of a numerical methods library.

4.9 ODEs

The numerical solution of ordinary differential equations has been a vital part of MATLAB since the first MathWorks MATLAB. [Shampine and Reichelt 1997]. ODEs are the heart of Simulink®, MATLAB's companion product.

The Van der Pol oscillator is a classical ODE example.

$$\frac{d^2y}{dx^2} = \mu(1-y^2)\frac{dy}{dt} - y$$

The parameter μ is the strength of the damping term. When $\mu = 0$, the harmonic oscillator results.

This MATLAB function expresses the oscillator as a pair of first order equations.

```
function dydx = vanderpol(t,y)
   mu = 5;
   dydx = [y(2); mu*(1-y(1)^2)*y(2)-y(1)];
end
```

In the following code, a string containing the name of this function is passed as the first argument to a numerical ODE solver, `ode23s`. The code produces the plot shown in Figure 16:

```
tspan = [0 30];
y0 = [0 0.01]';
[t,y] = ode23s('vanderpol',tspan,y0);
plot(t,y(:,1),'ko-',t,y(:,2),'k-')
xlabel('t')
```

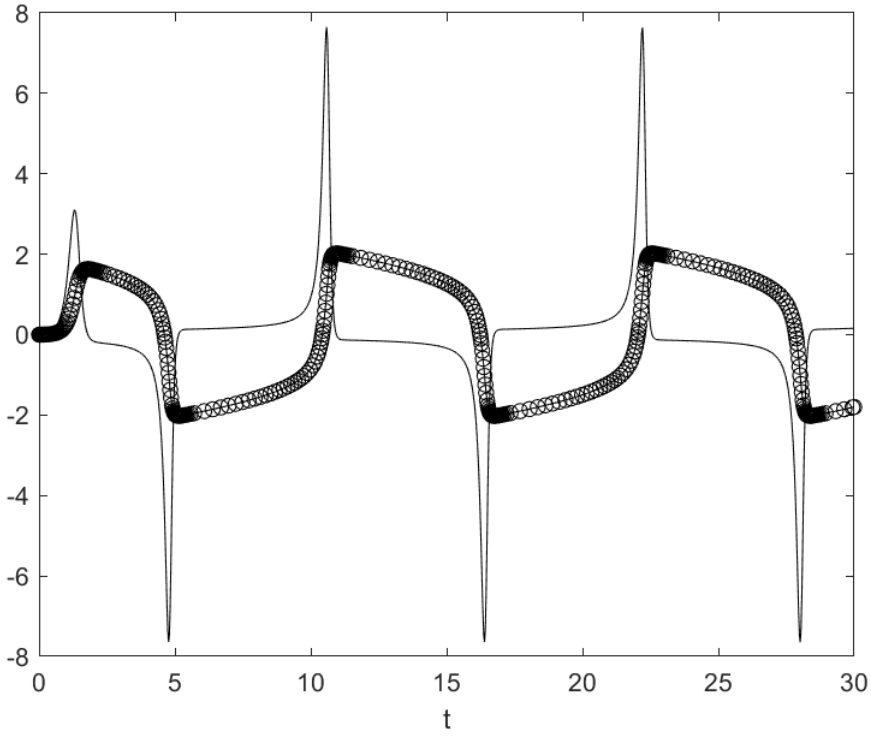


Fig. 16. The Van der Pol oscillator is a classical ODE example.

The Van der Pol oscillator, with the parameter μ set to 5, is a mildly stiff differential equation. In anticipation, the code uses the `ode23s` solver; the '`s`' in the name indicates it is for stiff equations. In the plot you can see some clustering of steps where the solution is varying rapidly. A nonstiff solver would have taken many more steps.

A stiff ODE solver uses an implicit method requiring the solution of a set of simultaneous linear equations at each step. The components of the iconic MATLAB backslash operator for solving a linear system are quietly at work here.

In all, there are seven routines in the MATLAB ODE suite. [NA Moler 2014f] Their names all begin with "ode"; this is followed by two or three digits, and then perhaps one or two letters. The digits indicate the order; roughly speaking, higher order routines work harder and deliver higher accuracy per step. A suffix '`s`' or '`t`' or '`tb`' designates a method for stiff equations. Here is the list of the functions in the suite:

`ode45`	The first choice for most nonstiff problems
`ode23`	Less stringent accuracy requirements than `ode45`
`ode113`	More stringent accuracy requirements than `ode45`
`ode15s`	The first choice for most stiff problems
`ode23s`	Less stringent accuracy requirements than `ode15s`
`ode23t`	Moderately stiff problems without numerical damping
`ode23tb`	Less stringent accuracy requirements than `ode15s`

The functions ode23 and ode45 are the principal MATLAB (and Simulink) tools for solving nonstiff ordinary differential equations. Both are single-step ODE solvers; they are also known as Runge-Kutta methods. Each step is *almost* independent of the previous steps, but two important pieces of information are passed from one step to the next. The step size expected to achieve a desired accuracy is passed from step to step. And, in a strategy known as FSAL ("First Same As Last"), the final function value at the end of a successful step is used as the initial function value at the following step.

The BS23 algorithm used by ode23 is due to Larry Shampine and Przemyslaw Bogacki [Bogacki and Shampine 1989]. The "23" in the name indicates that two simultaneous single-step formulas, one of second order and one of third order, are involved; the difference between the two formulas provides an error estimate that is used to adjust the step size if necessary.

Before today's version of ode45, there was an earlier one. In a 1969 NASA report, Erwin Fehlberg introduced a so-called six-stage Runge-Kutta method that requires six function evaluations per step [NA Fehlberg 1969]. These function values can be combined with one set of coefficients to produce a fifth-order accurate approximation and with another set of coefficients to produce an independent fourth-order accurate approximation. Comparing these two approximations provides an error estimate and resulting step size control.

In the early 1970's, Shampine and his colleague H. A. (Buddy) Watts at Sandia Laboratories published a Fortran code, RKF45 (Runge-Kutta-Fehlberg method with 4th and 5th order approximations), based on Fehlberg's algorithm [NA Shampine and Watts 1976]. In 1977, Forsythe, Malcolm, and Moler used RKF45 as the ODE solver in their textbook *Computer Methods for Mathematical Computations* [Forsythe et al. 1977]. Fortran source code for RKF45 is still available from netlib.

RKF45 became the basis for the first version of ode45 in MATLAB in the early 1980s and for early versions of Simulink. The Felhberg (4,5) pair was used for almost fifteen years until Shampine and Mark Reichelt modernized the suite [Shampine and Reichelt 1997]. Today's implementation of ode45 is based on an algorithm of Dormand and Prince [Dormand and Prince 1980; Shampine 1986]. It uses six stages, employs the FSAL strategy, provides fourth- and fifth-order formulas, and has local extrapolation and a companion interpolant.

The anchor of the MATLAB differential equation suite is ode45. The MATLAB documentation recommends ode45 as the first choice, and Simulink blocks set ode45 as the default solver. But ode23 has a certain simplicity, and frequently it is especially suitable for graphics. A quick comparison of ode23 and ode45: ode23 is a three-stage, third-order, Runge-Kutta method. whereas ode45 is a six-stage, fifth-order, Runge-Kutta method; ode45 does more work per step than ode23, but can take much larger steps.

A key feature of the design of MATLAB's ODE syntax is that it is possible to use all of the ODE solvers in exactly the same way; despite that fact that different solvers may need additional information as arguments, all seven functions have the same API, so it is just a matter of providing appropriate values for the arguments (some of which may be ignored by some solvers). In particular, there are flexible options for specifying where (within the domain) solution points are desired, and interpolation is automatically performed if necessary to produce those solution points; this can be especially important for producing good (smooth-looking) plots.

4.10 Text

MATLAB relies on the Unicode character set. Character vectors are delineated by single quotes. The statement

```
h = 'hello world'
```

produces

```
h =
    'hello world'

disp(h)

'hello world'

d = uint8(h)

d =
  1×11 uint8 row vector
  104  101  108  108  111   32  119  111  114  108  100
```

Short character strings are often used as optional parameters to functions.

```
[U,S,V] = svd(A,'econ');  % Economy size, U is the same shape as A.

plot(x,y,'o-')   % Plot lines with circles at the data points.
```

Multiple lines of text, or many words in an array, must be padded with blanks so that the character array is rectangular. The char function provides this service. For example, here is a 3-by-7 array.

```
class = char('Alice','Bob','Charlie');

class =
  3×7 char array
    'Alice  '
    'Bob    '
    'Charlie'
```

Or, you could use a cell array.

```
class = {'Alice', 'Bob', 'Charlie'}'

class =
  3×1 cell array
    {'Alice'  }
    {'Bob'    }
    {'Charlie'}
```

Recently, a comprehensive `string` data type was introduced.(see Section 5.10).

4.11 Evolution of the MathWorks Logo

MathWorks is perhaps the only company in the world that has the solution to a partial differential equation as its logo. The graphic has evolved over the years [NA Moler 2014c; NA Moler 2014e; NA Moler 2014d; NA Moler 2014b; NA Moler 2014a] from a two-dimensional contour plot—a versionFFa of which appeared in Cleve Moler's PhD dissertation [Moler 1965, Figure (8.15)]—to a black-and-white three-dimensional surface plot, to the same three-dimensional plot with various color schemes and lighting models (see Figure 17).

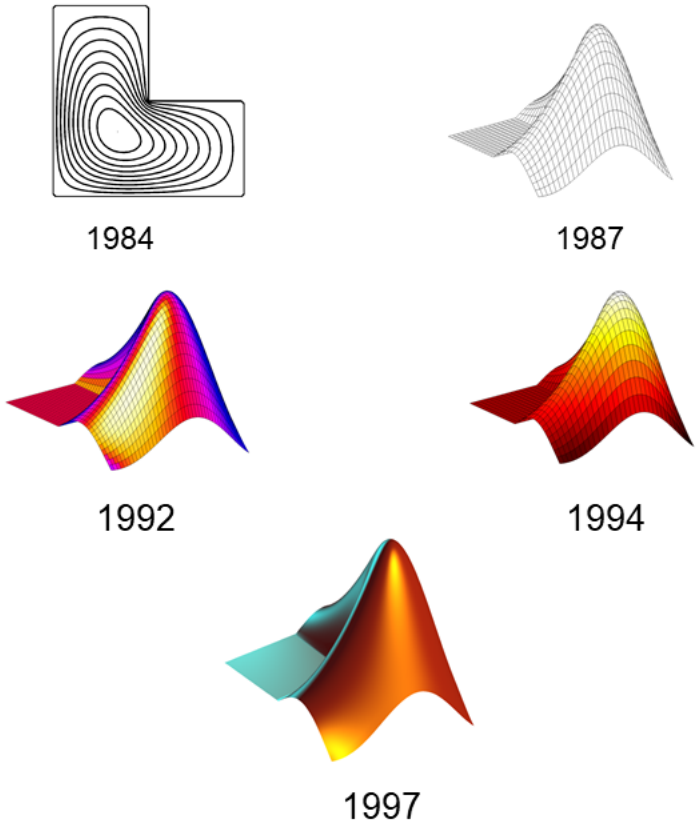


Fig. 17. Evolution of the MathWorks logo from a 2-D contour plot to a 3-D surface plot with color and lighting. (Source: MathWorks.)

5 RECENT DEVELOPOMENTS

5.1 LAPACK

In 2000, MathWorks returned to the Fortran world for numerical linear algebra library by adopting LAPACK [Anderson et al. 1999] in MATLAB 6. This replaced the old collection of LINPACK and EISPACK subroutines rewritten in C. Here is part of the commentary [NA Moler 2000] on this major transition:

> MATLAB started its life in the late 1970s as an interactive calculator built on top of LINPACK and EISPACK, which were then state-of-the-art Fortran subroutine libraries for matrix computation. The mathematical core for all versions of MATLAB, up to version 5.3, has used translations to C of about a dozen of the Fortran subroutines from LINPACK and EISPACK.
>
> LAPACK is the modern replacement for LINPACK and EISPACK. It is a large, multi-author, Fortran library for numerical linear algebra. A new version was released in July and is available from NETLIB (www.netlib.org/lapack). LAPACK was originally intended for use on supercomputers and other high-end machines. It uses block algorithms, which operate on several columns of a matrix at a time. On machines with high-speed cache memory, these

block operations can provide a significant speed advantage. LAPACK also provides a more extensive set of capabilities than its predecessors do.

The speed of all these packages is closely related to the speed of the Basic Linear Algebra Subroutines, or BLAS. EISPACK did not use any BLAS. LINPACK used Level 1 BLAS, which operate on only one or two vectors, or columns of a matrix, at a time. Until now, MATLAB has used carefully coded C and assembly language versions of these Level 1 BLAS. LAPACK's block algorithms also make use of Level 2 and Level 3 BLAS, which operate on larger portions of entire matrices. The NETLIB distribution of LAPACK includes Reference BLAS written in Fortran. But the authors intended that various hardware and operating system manufacturers provide highly optimized, machine-specific, versions of the BLAS for their systems.

It is finally time to incorporate LAPACK into MATLAB. Almost all modern machines have enough cache memory to profit from LAPACK's design. Several key chip and computer vendors now offer optimized Level 1, 2, and 3 BLAS. A new alternative to the vendor BLAS is available from ATLAS, a research project at the University of Tennessee, where routines optimized for any particular machine can be generated automatically from parameterized code fragments.

The first MATLABs ran in the one-half megabyte memory available on the first PC, so it was necessary to keep code size at a minimum. One general-purpose eigenvalue routine, a single-shift complex QZ algorithm not in LINPACK or EISPACK, was developed for all complex and generalized eigenvalue problems. The extensive list of functions now available with LAPACK means that MATLAB's space saving general-purpose codes can be replaced by faster, more focused routines. There are now 16 different code paths underlying the eig function, depending on whether there are one or two arguments, whether the arguments are real or complex, whether the problem is symmetric and whether the eigenvectors are requested.

...

Regrettably, one popular MATLAB feature must be a casualty with the introduction of LAPACK. The `flops` function, which keeps a running count of the number of floating-point operations, is no longer feasible. Most of the floating point operations are now done in optimized BLAS that do not keep flop counts. However, with modern computer architectures, floating-point operations are no longer the dominant factor in execution speed. Memory references and cache usage are most important.

5.2 FFTW

With MATLAB 5.3 and a 266 MHz Pentium laptop, a one-million-point real FFT took about 6 seconds. In 2000, with new FFT code incorporated into MATLAB 6.0, the same computation took about 1.2 seconds. The new code was based on FFTW, "the Fastest Fourier Transform in the West," developed by Matteo Frigo and Steven G. Johnson at the Massachusetts Institute of Technology [Frigo 1999; Frigo and Johnson 1998]. Frigo and Johnson won the 1999 J. H. Wilkinson Numerical Software Prize for their work.

The `fft` function in MATLAB 5 could use fast versions of FFT only when the length was a product of small primes; otherwise a slow FFT (with running time quadratic in the length) had to be used. The FFTW improvements allowed MATLAB 6 to use fast FFT algorithms for any length, even a large prime number.

Measurements showed that when the length n was a power of 2 less than or equal to $2^{16} = 65536$, the `fft` times for MATLAB 5.3 and MATLAB 6.0 were nearly the same; but when n was a power of 2 greater than 2^{16}, MATLAB 6.0 was about twice as fast because it made more effective use of

cache memory. When n was not a prime and not a power of 2, MATLAB 6.0 was two to four times faster than MATLAB 5.3. [NA Moler and Eddins 2001]

5.3 Desktop

The MATLAB desktop was introduced in 2000 [NA Moler 2018f]. Figure 18 shows the default layout of the desktop as it looks today. Four panels are visible, the current folder viewer on the left, the workspace viewer on the right, the editor/debugger in the top center, and the traditional command window in the bottom center. A file viewer and a command history window are not in the default layout but can be included in personalized layouts.

Any of the panels can be closed or undocked into a standalone window. When you have two screens available, you can put the command window on one screen and the editor on the other.

5.4 Function Handles

MATLAB has several functions that take other functions as arguments. These are known as "function functions". For many years function arguments were specified by a string. The numerical solution of the Van der Pol oscillator is an example mentioned eatlier. The function `vanderpol.m` defines the differential equation as a 2-by-1 first order system.

```
function dydt = vanderpol(t,y)
   dydt = [y(2); 5*(1-y(1)^2)*y(2)-y(1)];
end
```

The function name is then passed as a character string to the ODE solver ode45.

```
tspan = [0 150];
y0 = [1 0]';
[t,y] = ode45('vanderpol',tspan,y0);
```

It turns out that over half the time required by this computation is spent in repeatedly decoding the string argument. Performance is improved with the introduction the function handle, which is the function name preceded by the "at sign", @vanderpol.

The "at sign" is also used to create a function handle that defines an anonymous function, which is the MATLAB instantiation of Church's lambda calculus. Ironically, anonymous functions are often assigned to variables in practice, thereby negating the anonymity.

```
vdp = @(t,y) [y(2); 5*(1-y(1)^2)*y(2)-y(1)];
```

Let's compare the times required in the Van der pol integration with character strings, function handles and anonymous functions.

```
tic, [t,y] = ode45('vanderpol',tspan,y0); toc
Elapsed time is 0.200928 seconds.

tic, [t,y] = ode45(@vanderpol,tspan,y0); toc
Elapsed time is 0.076443 seconds.

tic, [t,y] = ode45(vdp,tspan,y0); toc
Elapsed time is 0.051992 seconds.
```

The times required for the latter two are comparable and are significantly faster than the first.

An anonymous function can be returned as the value of another anonymous function; what is nowadays commonly called "lexical scoping" is properly observed so that the inner function can refer to variables bound by the outer function. However, MATLAB syntax does not provide a way

to compute a function and immediately apply it to arguments, so the following sort of expression (which a LISP programmer might expect to work in MATLAB):

```
((@(x)(@(y)x+y))(3))(5)
```

is rejected as a syntax error. However, if the function handles are assigned to variables along the way:

```
fx = @(x)(@(y)x+y); fy = fx(3); fy(5)
```

then the expected answer 8 is indeed produced; the function handle in `fy` "remembers" that the variable `x` had the value 3.

An alternative is to use the `feval` function (for "function evaluation"):

```
fx = @(x)(@(y)x+y);
feval(feval(fx,3),5)
```

5.5 Objects

MATLAB has users with a broad spectrum of programming skills. At one end of the spectrum are scientists and engineers who are casual programmers but who use MATLAB extensively for their technical computing needs. For them, ease of programming is of primary importance. At the other end of the spectrum are the professional programmers, including those who work at MathWorks. They desire a rich, full featured, powerful programming language.

Responding primarily to internal demand, major enhancements to the MATLAB object-oriented programming capabilities, with behaviors found in C++ and Java, were made in 2008. Creating classes can simplify programming tasks that involve specialized data structures or large numbers of functions. MATLAB classes support function and operator overloading, controlled access to properties and methods, reference and value semantics, and events and listeners.

The MATLAB graphics system is one large, complex example of the object-oriented approach.

5.6 Symbolic Math Toolbox™

In the mathematical world, MATLAB is often compared to symbolic algebra systems, including MACSYMA [Martin and Fateman 1971] and its descendants. The primary data structure in these languages is some sort of symbolic expression. At the most basic level, these systems can do calculus. They know that the derivative of $\cos x$ is $-\sin x$. It is possible to do numeric computation like MATLAB, but this is not their strength.

With MATLAB, symbolic manipulation is done by the optional add-on Symbolic Math Toolbox. This toolbox is now at version 8.4. The command

```
methods sym
```

lists over 400 functions.

The toolbox provides functions for solving, plotting, and manipulating symbolic math equations. The operations include differentiation, integration, simplification, transforms, and equation solving. Computations can be performed either analytically or using variable-precision arithmetic.

5.7 Making MATLAB More Accessible

The first versions of MATLAB were simple terminal applications. Over time, separate windows for graphics, editing, and other tools were added. These gradually made MATLAB easier to use, especially for users without prior programming experience. Two features that have had the biggest impact are the MATLAB Desktop described above and the Live Editor.

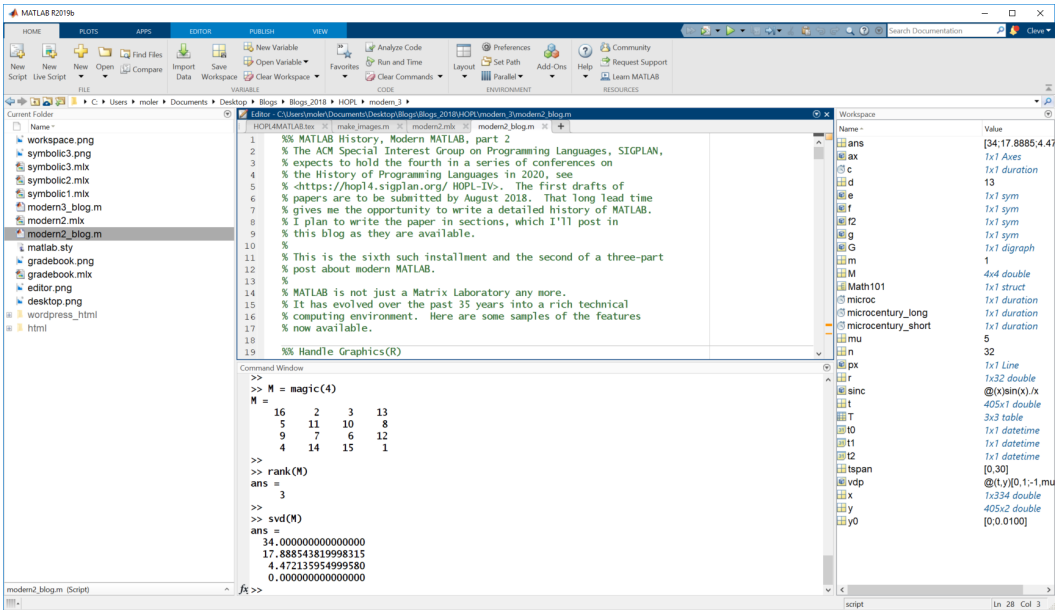


Fig. 18. The appearance of the MATLAB desktop in 2019.

The MATLAB Live Editor notebook interface was introduced in 2016. MATLAB input, output and graphics are combined in a single interactive document. The document may be exported to HTML, PDF, or LaTeX.

Figures 19–21 show an an extended example illustrating how the Live Editor typesets results from Symbolic Math Toolbox.

The "magic squares" from recreational mathematics (n-by-n matrices containing the integers 1 through n^2 for which every column, every row, and the two major diagonals have the same sum) have some interesting properties from a linear algebra point of view. It is for this reason that even the very earliest design for MATLAB included a function to generate magic squares—though the earliest documentation remarked only that the function `magic` produces "interesting test matrices" [Moler 1980]. The 1981 manual further says, "Magic square. `MAGIC(N)` is an `N` by `N` matrix constructed from the integers 1 through `N**2` with equal row and column sums."

Figures 22 and 23 show a demonstration using Classic MATLAB of various properties of a 4-by-4 magic square, including the fact that the sum of every row, column, and major diagonal is 34 and the fact that it is a singular matrix (because its rank is 3, not 4). Figures 24 and 25 show a comparable demonstration using today's MATLAB Live Editor™.

5.8 Parallel Computing

Parallel Computing Toolbox™ (PCT) was introduced at the 2004 Supercomputing Conference. The next year, at Supercomputing 2005, Bill Gates gave the keynote talk [Gates 2005], using MATLAB to demonstrate Microsoft's entry into High Performance Computing.

PCT supports coarse grained, distributed memory parallelism by running many MATLAB workers on many machines in a cluster, or on many cores in a single machine. [Luszczek 2009]

Here is an extended example illustrating how the Live Editor typesets results from the Symbolic Toolbox.

First, let's reset the figure size.

```
figure_size(360,270)
```

The statement

```
x = sym('x')
```

```
x = x
```

creates a symbolic variable that just represents itself.

The variable does not have a numeric value.

The statement

```
f = 1/(4*cos(x) + 5)
```

```
f =
```

$$\frac{1}{4\cos(x)+5}$$

creates a symbolic function of that variable.

Let's differentiate `f` twice.

```
f2 = diff(f,2)
```

```
f2 =
```

$$\frac{4\cos(x)}{(4\cos(x)+5)^2}+\frac{32\sin(x)^2}{(4\cos(x)+5)^3}$$

And then integrate the result twice. Do we get back to where we started?

```
g = int(int(f2))
```

```
g =
```

$$-\frac{8}{\tan\left(\frac{x}{2}\right)^2+9}$$

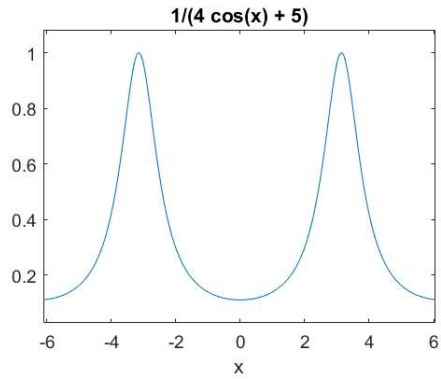
No, `f` and `g` are not the same expression.

1

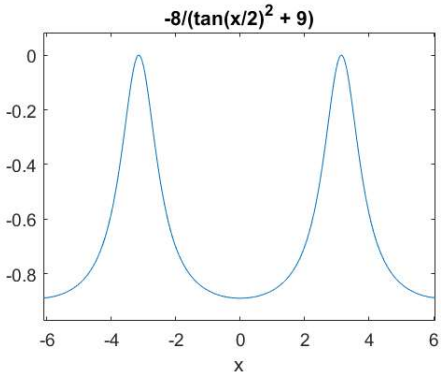
Fig. 19. Use of Symbolic Math Toolbox within MATLAB Live Editor (1 of 3).

Let's plot them both.

```
ezplot(f)
```




```
ezplot(g)
```



The two plots have the same shape, but the y axes are not the same.

Let's compute the difference; this is the "error" made by differentiating twice and then integrating twice.

```
e = f - g
```

```
e =
```

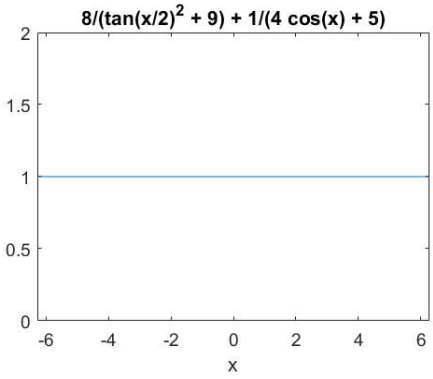
$$\frac{8}{\tan\left(\frac{x}{2}\right)^2 + 9} + \frac{1}{4\cos(x) + 5}$$

The system reorders the terms to `e = -g + f`.

The error isn't zero. What is it? First a plot.

Fig. 20. Use of Symbolic Math Toolbox within MATLAB Live Editor (2 of 3).

```
ezplot(e,[-2*pi 2*pi 0 2])
```



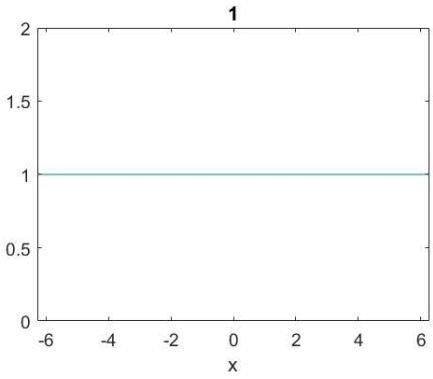
Now let's do the most difficult computation of this entire example.

```
s = simplify(e)
```

```
s = 1
```

It's easy to miss the result, s = 1.

```
ezplot(s)
```



So, e is an elaborate way of writing the constant function, 1.

That's the "constant of integration", the infamous +C that we always forget to include in indefinite integration.

3

Fig. 21. Use of Symbolic Math Toolbox within MATLAB Live Editor (3 of 3).

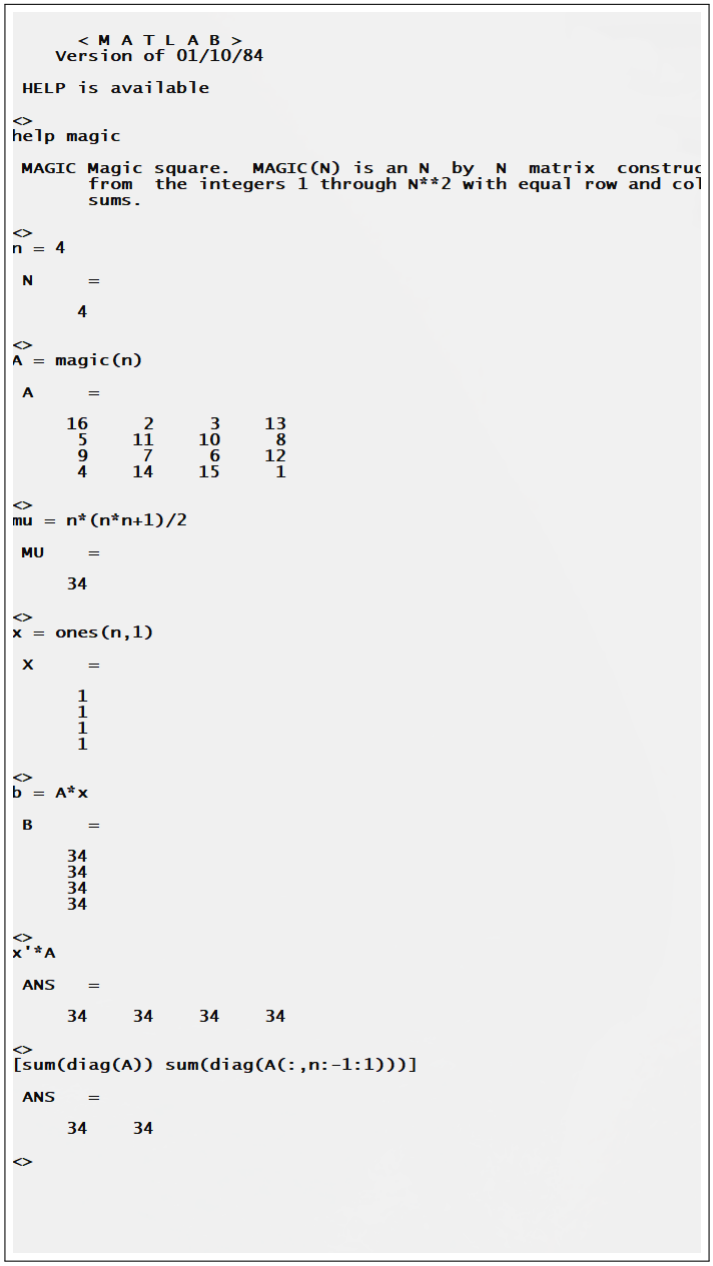


Fig. 22. Exploring simple properties of magic squares in Classic MATLAB (1 of 2).

By far the most popular feature of the PCT is the parallel `for` loop command, `parfor`. This allows the creation of "embarrassingly parallel" jobs with many variations of a single program and no communication between processes.

Parallel Computing Toolbox rests upon the implementation of the MPI (Message Passing Interface) library [Gropp et al. 1998; Snir et al. 1998]. MATLAB provides message-passing functions that are

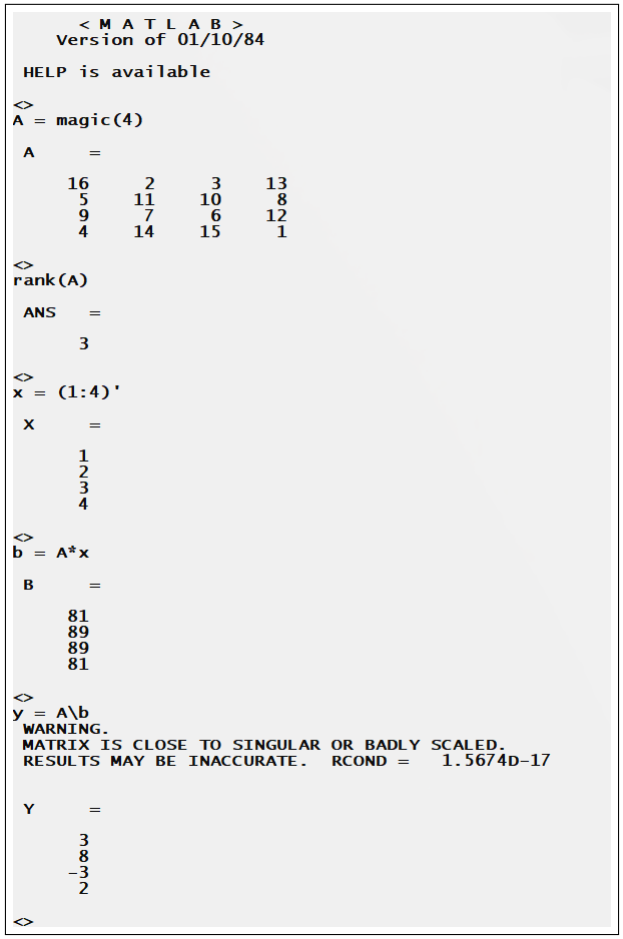


Fig. 23. Exploring simple properties of magic squares in Classic MATLAB (2 of 2).

high-level abstractions of functions described in the MPI standard; these include point-to-point communication, broadcast, barrier, and reduction operations. An important part of the design of the Parallel Computing Toolbox is that messages may be used to transmit or exchange arbitrary MATLAB data types, including numerical arrays of any precision, structure arrays, and cell arrays. In the general case, a single MATLAB message becomes two MPI messages: a short header message of known size that indicates the MATLAB data type and associated size information, followed by a payload message. The size information in the first message allows the receiver to prepare a buffer adequate to receive the payload. For certain small data sizes, the payload can be included in the header message, eliminating the need for a separate payload message. For complicated data types, the data array is serialized by the sender into a byte stream and then deserialized by the receiver, but for MATLAB data types that can be mapped directly onto an MPI data type, the serialization/deserialization process is skipped and the contents of the data array are sent directly. [Sharma and Martin 2009]

```
help magic
```

```
  magic  Magic square.
     magic(N) is an N-by-N matrix constructed from the integers
     1 through N^2 with equal row, column, and diagonal sums.
     Produces valid magic squares for all N > 0 except N = 2.

     Documentation for magic
```

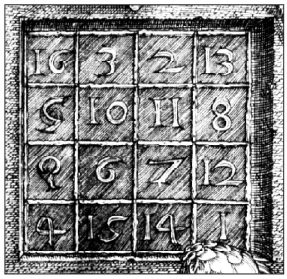
Here is a magic square of order 4.

```
n = 4;
A = magic(n)
```

```
A = 4×4
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Detail from Albrecht Dürer's Melencolia I.

```
load detail X
gray_image(X)
```



For a magic square of order n the expected sum is

```
mu = n*(n*n+1)/2
```

```
mu = 34
```

Here is a column vector of 1's.

```
x = ones(n,1)
```

```
x = 4×1
     1
     1
     1
     1
```

1

Fig. 24. Exploring simple properties of magic squares in today's MATLAB Live Editor (1 of 2).

The matrix-vector product sums the rows.

```
b = A*x
```

```
b = 4×1
    34
    34
    34
    34
```

Now the column sums.

```
x'*A
```

```
ans = 1×4
    34    34    34    34
```

And the two diagonals.

```
[sum(diag(A)) sum(diag(A(:,n:-1:1)))]
```

```
ans = 1×2
    34    34
```

Even-ordered magic squares are singular.

```
rank(A)
```

```
ans = 3
```

Consequently, solving a linear system may not produce the expected solution.

```
x = (1:n)'
```

```
x = 4×1
     1
     2
     3
     4
```

```
b = A*x
```

```
b = 4×1
    81
    89
    89
    81
```

Does this get back to x? No.

```
y = A\b
```

```
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =  4.625929e-18.
y = 4×1
    2.0000
    5.0000
    0.0000
    3.0000
```

2

Fig. 25. Exploring simple properties of magic squares in today's MATLAB Live Editor (2 of 2).

5.9 GPUs

Support for Graphics Processing Units was added to the Parallel Computing Toolbox in 2010. Eight years later, in release R2018a, the gpuarray has grown to have 385 associated methods, including all the familiar matrix computations, including the functions `lu`, `eig`, `svd` and `mldivide` (the backslash operator).

5.10 Strings

In 2016, MathWorks began the process of providing more comprehensive support for text by introducing the double quote character and the string array data structure. The statement

```
class = ["Alice", "Bob", "Charlie"]'
```

produces

```
class =
  3×1 string array
    "Alice"
    "Bob"
    "Charlie"
```

There are some very convenient functions for the new string data type.

```
proverb = "A rolling stone gathers momentum"
words = split(proverb)

proverb =
    "A rolling stone gathers momentum"
words =
  5×1 string array
    "A"
    "rolling"
    "stone"
    "gathers"
    "momentum"
```

Addition of strings is concatenation.

```
and = " and ";
merge = class(1);
for k = 2:length(class)
    merge = merge + and + class(k);
end
merge

merge =
    "Alice and Bob and Charlie"
```

The `regexp` function provides the regular expression pattern matching seen in Unix and many other programming languages.

5.11 Execution Engine

Recent years have seen changes in how MATLAB code gets executed and corresponding improvements in performance. The changes have been based on JIT, *just-in-time*, compilation of MATLAB code into machine code.

The first MATLAB JIT compiler, in the early 2000s, compiled a complete function at a time, but only when strict constraints were met, and only for a subset of the MATLAB language. As a result, this first JIT compiler had a relatively small impact on overall MATLAB performance.

A new, trace-based JIT compiler was introduced in 2015. This compiler has completely replaced the original interpreter. It is based on compiling *traces*, or particular code paths, with run-time knowledge of the dynamic types of program variables. This new MATLAB *execution engine* caches compiled traces for reuse and records fast-lookup links between them.

The new JIT compiler was built with evolution in mind. Several layers of compilation and optimization have been created to support different classes of code:

- Long, straight-line code scripts. These are often executed just once, or only a few times per session, and so compilation cost is not amortized by run-time savings.
- Fortran-like code, with tight loops containing simple matrix indexing and arithmetic.
- Library-rich technical computing, with vectorized code, complex array indexing, and many calls to specialized functions.

The JIT compiler can now execute long code scripts quickly, using a lightweight analysis and compilation strategy. It can link traces across loop boundaries, enabling a variety of loop optimizations for Fortran-like code. And, for the most rich and complex code, it can perform deep analysis and optimizations, using multiple CPU cores and background threads so that code execution does not have to wait for the analysis to be completed. Typical MATLAB workflows now execute twice as fast, on average, as they did four years ago, with no code changes needed.

This multilayer JIT compilation strategy continues to be an area of development at MathWorks.

5.12 Development Process

In the early days of MathWorks, only a handful of people worked on the source code for MATLAB. In principle any one could make a change to any part of the code. In practice people stuck to their individual areas of expertise.

Today the development process is much more structured and professional. Formal proposals for any changes or additions are reviewed by committees. Peer code reviews are required. Extensive automated testing is done before any changes are accepted.

The development process is facilitated by two powerful, in-house systems. The bug tracking system is named “Gecko”, after the bug-eating lizard. Gecko has evolved into a system for tracking almost all the activities affecting software. The testing system is “Build and Test” or “BAT”. Each developer has a personal copy of MATLAB, together with the software they working on, in a “sandbox”. Any kind of change can be made in an individual sandbox. When something is ready to be part of MATLAB, a job is submitted to BAT. It must pass a multi-stage series of test suites before it is accepted into the next release.

MathWorks was one of the earliest innovators for continuous integration, even before that term was widely adopted. MATLAB 4.0 took almost two years to port and release on all platforms. With the new, multiplatform, continuous-integration system, MATLAB 5.0 was released on the PC, the Mac, and a number of Unix platforms simultaneously. The entire product suite is relased twice a year. Some large organizations may choose to install releases less often than twice a year.

5.13 Toolboxes

Much of the power of today's MATLAB derives from the toolboxes available for specialized applications. In release 2018a there are 63 of them. Here are the categories.

Parallel Computing (2)
Math, Statistics, and Optimization (9)
Control Systems (8)
Signal Processing and Wireless Communications (11)
Image Processing and Computer Vision (6)
Test and Measurement (5)
Computational Finance (8)
Computational Biology (2)
Code Generation (7)
Application Deployment (3)
Database Access and Reporting (2)

5.14 Today's why Command

The `why` command from Classic MATLAB has survived over 40 years of software refactoring, upgrades, and releases, and is still popular. In addition to 10 fixed responses, it is now capable of generating random, arbitrarily long, syntactically correct English sentences with nouns, verbs, adjectives, and adverbs. One possibility:

```
why

Some old and nearly bald mathematician suggested it.
```

6 SUCCESS

MathWorks now employs over 5000 people (30% located outside the United States) and in 2018 had over $1 billion in revenue (60% from outside the United States), with customers in 185 countries and software installed at over 100,000 business, government, and university sites [NA MathWorks, Inc. 2019]. More than 6,500 colleges and universities around the world use MATLAB and Simulink for teaching and research in a broad range of technical disciplines. There are more than 4 million users of MATLAB worldwide, and more than 2000 MATLAB-based books in 27 languages.

Why has MATLAB been successful? There are several reasons.

Mathematics. Years ago, Georgia Tech's Professor Jim McClellan said, "The reason MATLAB is so good at signal processing is that it was not designed for signal processing. It was designed to do mathematics." The same could be said for dozens of fields other than signal processing. Matrices and ordinary differential equations are the basis for much of today's technical computing.

Quality. MATLAB results are known for their precision and correctness.

Language. Fundamentally, MATLAB is a computer programming language for manipulating arrays. The simplicity and power of array operations, subscript notation, and `for` loops have proved to be useful far beyond numerical linear algebra.

Toolboxes. Much as MATLAB has done for linear algebra and mathematics, a wide range of specialized libraries extend MATLAB to provide the fundamental tools in disciplines that rely on mathematical algorithms, especially when linear algebra or array based. This includes areas such as signal processing, control systems, image processing, statistics, optimization, and machine learning.

Graphics. Many people use MATLAB just for graphics. Line plots, bar plots, scatter plots, pie charts, three dimensional graphs, movies—the list expands with every release. Production of publication quality graphics has always been one of MATLAB's strengths.

Interactivity. MATLAB was originally intended to give students a means of doing matrix computation without the delays of batch processing and edit-compile-link-load-execute work flow of Fortran or C. This was attractive in the early days of mainframe and time-sharing and is also attractive today when everybody has their own personal computer.

Education. Students first encounter MATLAB in college and find it useful in industry after graduation.

Documentation. Professionally written, carefully maintained, and backed by a powerful doc information architecture and extensive usability testing.

Support. MathWorks provides professional support including regular semi-annual releases, fast e-mail and telephone support, on-line and in-house training, consulting and a strong user community.

The Web. The web site mathworks.com was among the first registered commercial internet sites and the company provides many MATLAB workflows through the website. This includes MATLAB Online, a code exchange, online training, and extensive video "how to" examples.

Perhaps one additional measure of the popular success of MATLAB is the recent publication of the book *MATLAB for Dummies* [Sizemore and Mueller 2014].

EPILOGUE

Moler was never able to teach the matrix analysis course that first motivated all this. By the time Classic MATLAB was ready for anybody else to use it, he was in a computer science department, with other responsibilities.

ACKNOWLEDGMENTS

Thanks to Steve Eddins and Jonathan Foot from MathWorks and to Guy Steele and the HOPL shepherding process for significant improvements to this paper.

A SYNTAX DIAGRAMS FOR CLASSIC MATLAB

This appendix presents the syntax diagrams from the 1981 *MATLAB Users' Guide* [NA Moler 1981; NA Moler 2018g], with commentary on the syntax of the MATLAB programming language and some remarks on changes introduced in 1981 relative to the 1980 design paper [Moler 1980]. (In some cases, the commentary and remarks use today's terminology rather than the precise terminology that might have been used in 1981; the intent is to explain the meaning of the syntax diagrams of 1981 to today's audience.) The syntax diagrams are of the same form used by Niklaus Wirth in documenting Pascal [Jensen and Wirth 1974] and PL/0 [Wirth 1976], as well as in the specification of Fortran 77 [Engel 1976, Appendix F], but rendered as what we now call "ASCII art" figures.

```
line

       |-----> statement >----|
       |                      |
       |-----> clause >-------|
       |                      |
-------|-----> expr >---------|------>
     | |                      | |
     | |-----> command >------| |
     | |                      | |
     | |-> > >-> expr >-> < >-| |
     | |                      | |
     | |----------------------| |
     |                          |
     |        |-< ; <-|         |
     |--------|       |---------|
              |-< , <-|
```

A command line can be any sequence of items, where each item is a statement, clause, expression, command, or macro replacement (interpretation of an expression to produce a character vector, followed by parsing and interpretation of that character vector as an item), and where adjacent items are separated by either a comma or a semicolon. MATLAB normally prints the value computed by each statement or top-level expression; a semicolon indicates that the value computed by the preceding statement or expression should *not* be printed.

The 1980 design paper [Moler 1980] does not have the >*expr*< syntax for macro replacement.

```
statement

     |-> name >--------------------------------|
     |          |                              |
     |          |         |--> : >---|         |
     |          |         |          |         |
     |          |-> ( >---|-> expr >-|---> ) >-|
     |                    |              |      |
-----|                    |-----< , <----|      |--> = >--> expr >--->
     |                                          |
     |       |--< , <---|                      |
     |       |          |                      |
     |-> < >---> name >---> > >----------------|
```

A statement (what we would nowadays call an “assignment statement” consists of a left-hand side, an equals sign, and a right-hand-side expression. The left-hand side may be a name, a subscripted name, or a list of names within angle brackets. If it is a subscripted name, the subscripts are separated by commas, and each subscript may be either an expression or a single colon symbol ‘`:`’.

```
clause

     |---> FOR  >---> name >---> = >---> expr >---------------|
     |                                                        |
     | |-> WHILE >-|                                          |
     |-|           |-> expr >----------------------          |
     | |-> IF    >-|          |   |   |   |   |   |          |
-----|                        <   <=  =   <>  >=  >          |---->
     |                        |   |   |   |   |   |          |
     |                        ----------------------> expr >--|
     |                                                        |
     |---> ELSE >-----------------------------------------------|
     |                                                        |
     |---> END  >-----------------------------------------------|
```

A clause is either a `for` loop header, a `while` or `if` header (which must compare the values of two expressions), an `else` delimiter for use with `if`, or an `end` delimiter that terminates a `for`, `while`, or `if` control structure.

Today’s MATLAB uses ‘`~=`’ rather than ‘`<>`’ for the “not equal to” operator, and all six of the comparison operators produce results of type “logical array” rather than numeric arrays.

Today’s MATLAB allows the use of any expression (possibly involving logical operators) rather than just a single comparison operation in a `while` or `if` clause.

Today’s MATLAB has a `return` statement that returns control from a function in a manner that would be familiar to any user of languages such as C or Java.

Today’s MATLAB allows the use of `break` and `continue` statements within `for` and `while` loops in a manner that would be familiar to any user of languages such as C or Java.

Today’s MATLAB has a `try`/`catch` statement that can react to errors in a manner that would be familiar to Java programmers.

Today’s MATLAB has a parallel `for` loop clause indicated by `parfor` (see Section 5.8).

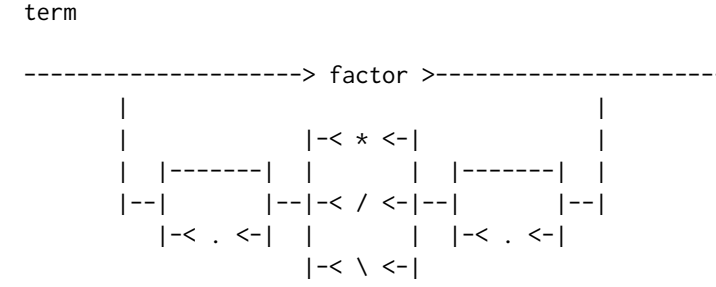
```
expr

       |-> + >-|
       |       |
-------|-------|-------> term >----------->
       |       |    |             |
       |-> - >-|    |  |-< + <-|  |
                    |  |       |  |
                    |--|-< - <-|--|
                       |       |
                       |-< : <-|
```

An expression is a sequence of one or more terms, where the first term may be preceded by a (unary) ‘`+`’ or ‘`-`’ operator and adjacent terms are separated by a ‘`+`’ or ‘`-`’ or ‘`:`’ operator. Note

that an expression containing three terms separated by two ‘:’ operators is interpreted as a single range-construction operation.

Today’s MATLAB also has logical operators that that produce logical arrays as values: ‘&’ represents logical AND, ‘|’ represents logical OR, ‘&&’ represents logical AND with short-circuiting (the expression to its right is not evaluated if the expression to its left produces a `false` value), ‘||’ represents logical OR with short-circuiting (the expression to its right is not evaluated if the expression to its left produces a `true` value), and ‘~’ represents logical NOT. The four binary logical operators have lower precedence than comparison operators. (Today’s MATLAB also supports bitwise logical operations on two’s-complement integers, but they are provided by functions such as `bitand` and `bitor` and `bitxor` rather than as operators.)

This syntax diagram for `expr` is equivalent to (but more concise than) the syntax diagram for `Expression` in PL/0 [Wirth 1976, page 310].



A term is a sequence of one or more factors, where adjacent terms are separated by a multiplication or division operator. The syntax supports four multiplication operators:

‘*’ represents matrix multiplication
‘.*’ represents elementwise multiplication
‘.*.’ represents the Kronecker tensor product

(It was only during preparation of this paper that we discovered that Classic MATLAB would also accept the fourth case ‘*.’ and treat it as if it were ‘.*’).

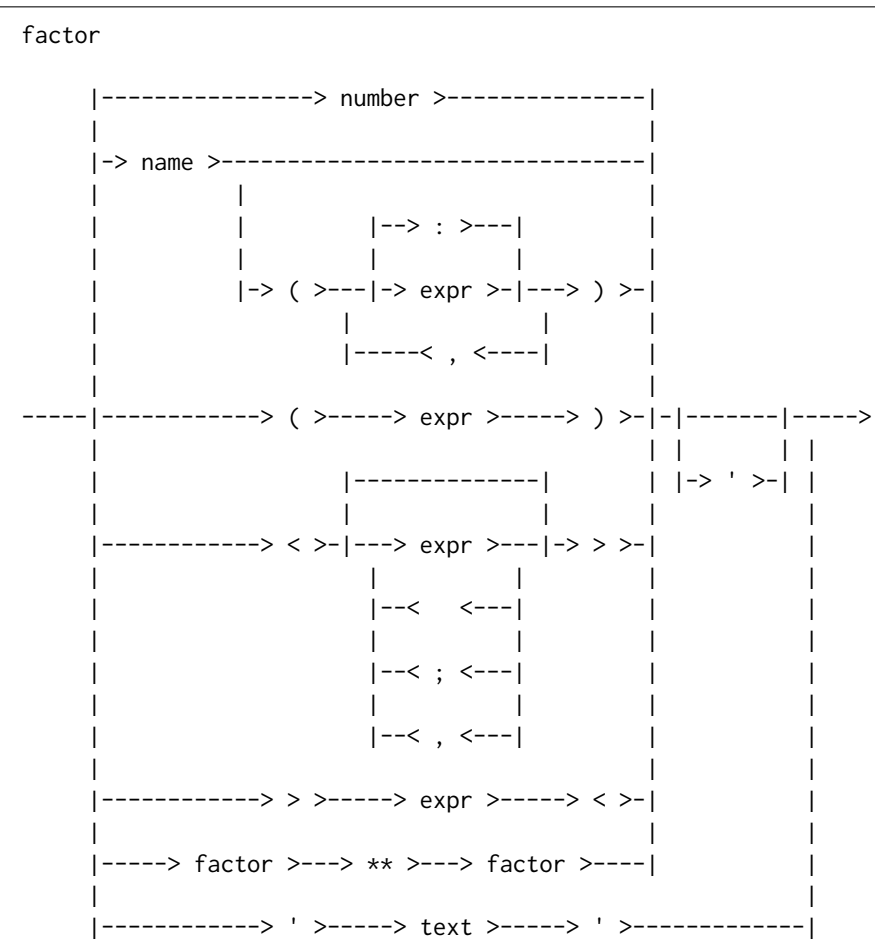
The syntax similarly supports eight division operations, but only six were intentionally defined:

‘\’ represents matrix left division
‘/’ represents matrix right division
‘.\’ represents elementwise left division
‘./’ represents elementwise right division
‘.\.’ represents Kronecker tensor left division
‘./.’ represents Kronecker tensor right division

The 1980 design paper [Moler 1980] does not provide for using periods before and/or after the multiplication and division operators; it had only matrix multiplication and not elementwise multiplication.

The Kronecker tensor operator symbols were eventually removed from the MATLAB syntax, though the corresponding function `kron` remains for computing the Kronecker tensor product (and Kronecker division operations are easily expressed using `kron` after taking the elementwise reciprocal of the divisor).

This syntax diagram for `term` is equivalent to (but more concise than) the syntax diagram for `Term` in PL/0 [Wirth 1976, page 310], except that it permits more kinds of multiplication and division operators.



A factor may be a number; a name; a subscripted name; an expression in parentheses; an empty list '<>'; a list of one or more expressions within angle brackets '< >' where each pair of adjacent expressions is separated by whitespace, a semicolon, or a comma; a macro replacement (interpretation of an expression to produce a character vector, followed by parsing and interpretation of that character vector as an expression); a pair of factors separated by the matrix exponentiation operator '**'; or a text string (which produces a character vector as its value). If the factor is a subscripted name, the subscripts after the name are surrounded by parentheses '( )' and separated by commas, and each subscript may be either an expression or a single colon symbol ':'. If the factor is anything except a text string, it may be followed by an apostrophe ''' to indicate the operation of (complex conjugate) matrix transposition.

The 1980 design paper [Moler 1980] does not have the >*expr*< syntax. for macro replacement.

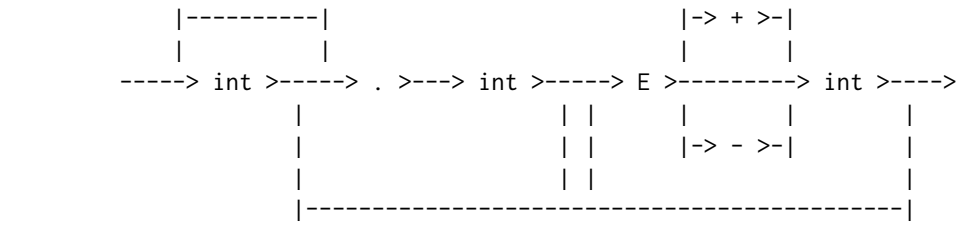
This syntax diagram for `factor` contains as a subdiagram the syntax diagram for `Term` in PL/0 [Wirth 1976, page 310], which allows a factor to be a number, an identifier, or a parenthesized expression.

Today's MATLAB has a second form of subscripting, used with cell arrays (see Section 4.6, that is indicated by using curly braces '{ }' after a name rather than parentheses.

The character translation table in Classic MATLAB allowed the use of square brackets '`[ ]`' rather than angle brackets '`< >`' to surround a list. In today's MATLAB, square brackets are used in preference to angle brackets. In addition, curly braces '`{ }`' may be used to surround a list to indicate construction of a cell array rather than a matrix (see Section 4.6).

Today's MATLAB uses '`^`' rather than '`**`' to represent the exponentiation (matrix power) operator, and furthermore uses '`.^`' to represent elementwise exponentiation. A following apostrophe '`'`' still indicates complex conjugate matrix transposition, as it always has ever since Classic MATLAB, but today's MATLAB also allows use of '`.'`' to indicate the operation of transposition without taking complex conjugates.

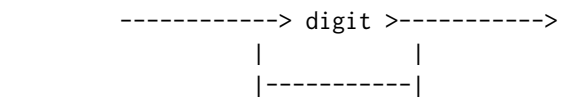
`number`



The syntax for numbers is reasonably conventional by either the standards of 1980 or today's standards. A number consists of a significand optionally followed by a exponent. The significand may be either an int, an `int` followed by a decimal point (period) followed by an int, or a decimal point (period) followed by an int. An exponent consists of the letter '`E`', then optionally a sign ('`+`' or '`-`'), then an `int`.

This syntax diagram for `int` is a subdiagram of the syntax diagram for `Unsigned number` in Pascal [Wirth 1976, page 353; Jensen and Wirth 1974, page 116]; the one difference is that MATLAB allows a number to begin with a decimal point, whereas Pascal requires at least one digit to precede a decimal point.

`int`

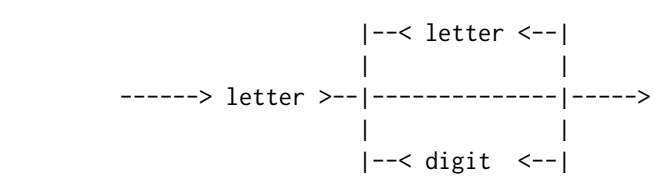


The syntax for numbers is quite conventional by either 1980 standards or today's standards: it is simply a sequence of digits, always interpreted as a decimal numeral, even if the first digit is '`0`'.

Today's MATLAB supports the use of hexadecimal notation (indicated by a leading '`0x`') and of binary notation (indicated by a leading '`0b`') for integer values.

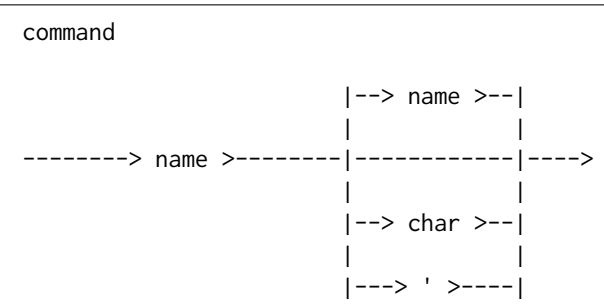
This syntax diagram for `int` is equivalent to the syntax diagram for `Unsigned integer` in Pascal [Wirth 1976, page 353] [Jensen and Wirth 1974, page 116].

`name`

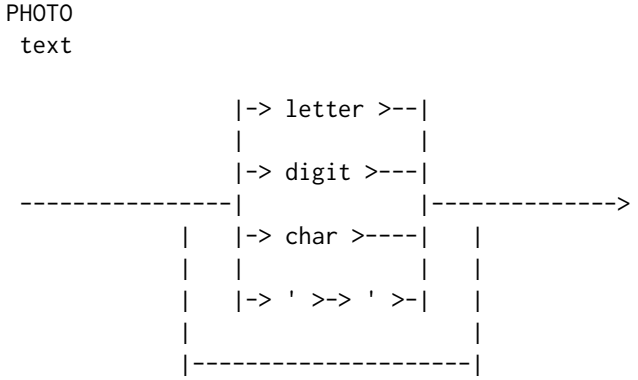


The syntax for names is quite conventional by either 1980 standards or today's standards: it is simply a letter followed by any number of letters and digits. (Note, however, that in Classic MATLAB—both the 1980 design and the 1981 implementation—-only the first four letters of a name are significant.)

This syntax diagram for `name` is equivalent to the syntax diagram for `Identifier` in Pascal [Wirth 1976, page 353] [Jensen and Wirth 1974, page 116].



The 1980 design paper [Moler 1980] simply defines a command to be a "reserved name"; in the 1981 Users' Guide an additional argument is provided for, which may be a name or a single character. (Note that `char` refers to any character in the character set *except* the apostrophe, so the use of an apostrophe as a command argument is provided for separately in the syntax diagram.)



The text in a text string may consist of any sequence of characters, except that a single apostrophe in the desired text is represented as two consecutive apostrophes. The 1980 design paper [Moler 1980] does not contain a syntax diagram for *text*, but simply describes "text" as meaning "any sequence of letters, digits, and characters" where "character" *does* include the apostrophe as a possibility; this may have been an error, corrected in the 1981 Users' Guide with the introduction of the syntax diagram for `text`.

This same possible error occurs near the bottom of the syntax diagram for `Constant` in Pascal [Wirth 1976, page 353] [Jensen and Wirth 1974, page 116]. The *Pascal User Manual* does separately say in the text, "Sequences of characters enclosed by single quote marks are called strings. To include a quote mark in a string, one writes the quote mark twice" [Jensen and Wirth 1974, page 11]. However, the BNF in Appendix D of the *Pascal User Manual* does not address this point [Jensen and Wirth 1974, page 111]. Similar observations apply to the discussion of strings in the *Pascal Report* [Jensen and Wirth 1974, page 138].

B PHOTO GALLERY

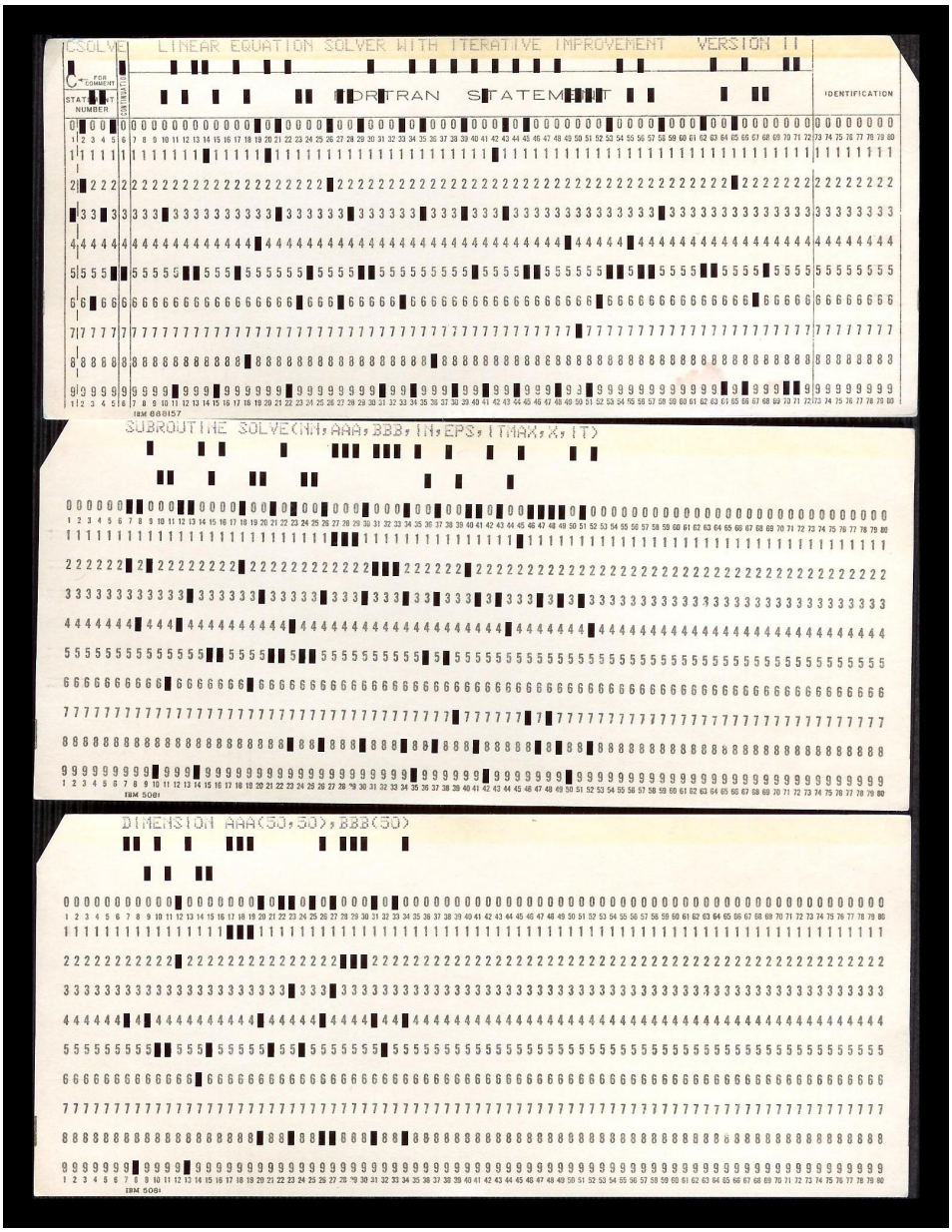


Fig. 26. In 1962, after Forsythe's numerical analysis course and a visit to Stanford by Wilkinson, Moler wrote a Fortran program to solve systems of simultaneous linear equations, 15 years before the introduction of the MATLAB backslash operator. Decks of punched cards for the program were distributed fairly widely at the time, including via SHARE, the IBM User's Group. Pictured are three punched cards from that program. (Source: Cleve Moler.)

Fig. 27. The organizing committee for the 1964 Gatlinburg/Householder meeting on Numerical Algebra. All six members of the committee—J. H. Wilkinson, Wallace Givens, George Forsythe, Alston Householder, Peter Henrici, and F. L. Bauer—have influenced MATLAB. (Source: Oak Ridge National Laboratory.)

Fig. 28. The first "personal computer" that Cleve Moler ever used was the Tektronix 4081, which Argonne National Laboratory acquired in 1978. The machine was the size of a desk and consisted of a Tektronix graphics display attached to an Interdata 7/32, the first 32-bit minicomputer. There was only 64 *kilobytes* of memory—but there was a Fortran compiler, and so, by using memory overlays, it was possible to run MATLAB. (Source: Ned Thanhouser.)

Fig. 29. Participants at the Householder Symposium, 1981, Oxford University. Among them are Leslie Fox, center in brown suit, chairman of the conference; J. H. Wilkinson, front row, fifth from left; Cleve Moler, second row, second from left; Jack Dongarra, second row, rightmost. Classic MATLAB, running at the Oxford Computing Laboratory, was demonstrated informally during coffee breaks. (Source: Oxford University.)

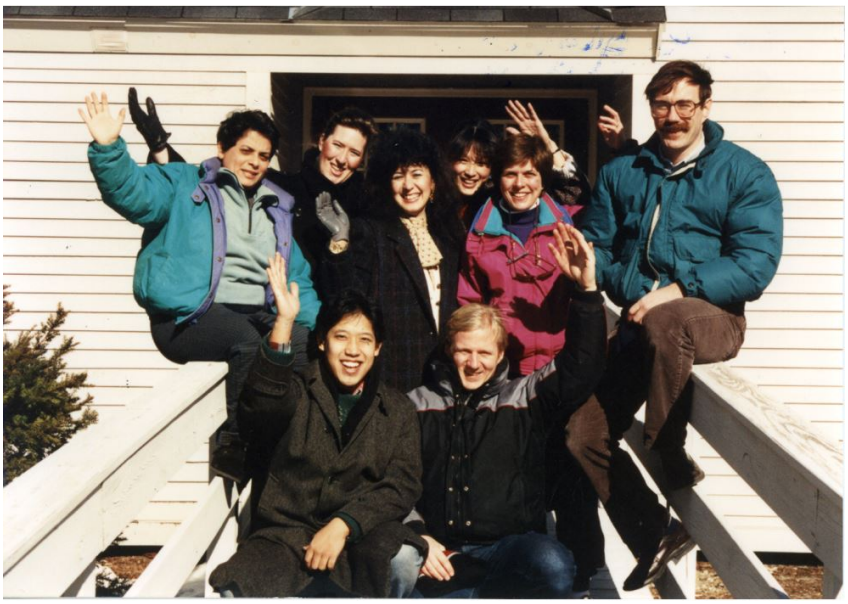


Fig. 30. MathWorks company photo 1988, Sherborn, Massachusetts; 8 employees. (Source: MathWorks.)

Fig. 31. MathWorks company photo 1991, South Natick, Massachusetts; 44 employees. Jack Little is at the upper right and Cleve Moler is second from the far left. (Source: MathWorks.)

Fig. 32. MathWorks company photo 2004, over 1100 employees in 11 offices worldwide. Loren Shure holds a sign commemorating the Boston Red Sox remarkable win of the 2004 World Series. (Source: MathWorks.)

Fig. 33. MathWorks company photo 2007, Boston Convention Center; over 1800 employees worldwide. (Source: MathWorks.)

Fig. 34. MathWorks company photo 2019, 35th anniversary; nearly 5000 employees worldwide. (Source: MathWorks.)

REFERENCES

E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, USA. https://doi.org/10.1137/1.9780898719604

P. Bogacki and L. F. Shampine. 1989. A 3(2) pair of Runge - Kutta formulas. *Applied Mathematics Letters* 2, 4, 321–325. https://doi.org/10.1016/0893-9659(89)90079-7

James M. Boyle, William J. Cody, Wayne R. Cowell, Wayne R. Cowell, Burton S. Garbow, Yasuhiko Ikebe, Cleve B. Moler, and Brian T. Smith. 1972. NATS: A Collaborative Effort to Certify and Disseminate Mathematical Software. In *Proceedings of the ACM Annual Conference—Volume 2* (Boston, Massachusetts, USA) (*ACM '72*). ACM, New York, New York, USA, 630–635. https://doi.org/10.1145/800194.805838

Carl De Boor. 1978. *A Practical Guide to Splines.* Applied Mathematical Sciences, Vol. 27. Springer-Verlag New York, New York, New York, USA. 348 pages.

Carl de Boor. 1990. An Empty Exercise. *SIGNUM Newsletter* 25, 2 (April), 3–7. https://doi.org/10.1145/101070.101072 Also available at http://ftp.cs.wisc.edu/Approx/empty.pdf

Carl de Boor. 2004. *Spline Toolbox: For Use with MATLAB®: User's Guide, Version 3.* The MathWorks, Inc., 3 Apple Hill Drive, Natick, Massachusetts, USA. The copyright dates indicate that the first edition was published in 1990.

J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. 1979. *LINPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia. https://doi.org/10.1137/1.9781611971811

J. R. Dormand and P. J. Prince. 1980. A Family of Embedded Runge-Kutta Formulae. *J. Comput. Appl. Math.* 6, 1, 19–26. https://doi.org/10.1016/0771-050X(80)90013-3

Frank Engel. 1976. Draft Proposed ANS FORTRAN BSR X3.9 X3J3/76. *SIGPLAN Not.* 11, 3 (March), 1–212. https://doi.org/10.1145/956013.1070892

George Elmer Forsythe, Michael A. Malcolm, and Cleve B. Moler. 1977. *Computer Methods for Mathematical Computations.* Prentice-Hall, Englewood Cliffs, New Jersey. 259 pages.

George E. Forsythe and Cleve B. Moler. 1967. *Computer Solution of Linear Algebraic Systems.* Prentice-Hall, Englewood Cliffs, New Jersey. 148 pages.

Matteo Frigo. 1999. A Fast Fourier Transform Compiler. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) (*PLDI '99*). Association for Computing Machinery, New York, New York, USA, 169–180. https://doi.org/10.1145/301618.301661

Matteo Frigo and Steven G. Johnson. 1998. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, Vol. 3. IEEE (May), 1381–1384. https://doi.org/10.1109/ICASSP.1998.681704

B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler. 1977. *Matrix Eigensystem Routines—EISPACK Guide Extension.* Lecture Notes in Computer Science, Vol. 51. Springer-Verlag, Berlin. 343 pages. https://doi.org/10.1007/3-540-08254-9

Bill Gates. 2005. Keynote Speech. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing* (*SC '05*). IEEE Computer Society, USA, 0. Video is available in the ACM Digital Library at https://dl.acm.org/doi/abs/10.5555/1105760.1116037 (Bill Gates keynote begins at 0:30:10; the portion that mentions MATLAB begins at 1:04:10).

John R. Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse Matrices in MATLAB: Design and Implementation. *SIAM J. Matrix Anal. Appl.* 13, 1, 333–356. https://doi.org/10.1137/0613024 Also available at NON-ARCHIVAL https://www.mathworks.com/help/pdf_doc/otherdocs/simax.pdf

William Gropp, Steven Huss-Lederman, and Marc Snir. 1998. *MPI—The Complete Reference: The MPI-2 Extensions.* Vol. 2. MIT Press, Cambridge, Massachusetts, USA.

Kathleen Jensen and Niklaus Wirth. 1974. *Pascal User Manual and Report* (second ed.). Springer-Verlag, New York, New York, USA. 167 pages.

S. C. Johnson. 1978. A Portable Compiler: Theory and Practice. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Tucson, Arizona, USA) (*POPL '78*). Association for Computing Machinery, New York, New York, USA, 97–104. https://doi.org/10.1145/512760.512771

Stephen C. Johnson. 2020. Personal communication by email with Cleve Moler and Guy Steele. 1 Jan. 2020. Received 2:37 pm EST.

Stephen C. Johnson and Cleve Moler. 1994. Compiling MATLAB. In *Proceedings of the USENIX Symposium on Very High Level Languages (VHLL).* USENIX Association, Berkeley, California, USA (Oct.), 119–127. https://www.usenix.org/legacy/publications/library/proceedings/vhll/index.html

C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Software* 5, 3 (Sept.), 308–323. https://doi.org/10.1145/355841.355847

Lennart Ljung. 1987. *System Identification: Theory for the User.* Prentice-Hall, Englewood Cliffs, New Jersey, USA. 519 pages.

Lennart Ljung. 2014. *System Identification Toolbox™: User's Guide, revision R2014b.* The MathWorks, Inc., 3 Apple Hill Drive, Natick, Massachusetts, USA (Oct.). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.461.1702&rep=

rep1&type=pdf The printing history indicates that the first edition was published in April 1988.

Piotr Luszczek. 2009. Parallel Programming in MATLAB. *The International Journal of High Performance Computing Applications* 23, 3, 277–283. https://doi.org/10.1177/1094342009106194

Marvin Marcus. 1993. *Matrices and MATLAB™: A Tutorial.* Prentice-Hall, Englewood Cliffs, New Jersey, USA. 736 pages. The manuscript for this book was prepared on a Macintosh® IIci using the word processing program WriteNow™ v2.2 and the equation processing program Expressionist® v2.07. Line drawings were prepared with MacDraw II® v1.1. Camera ready copy was produced on a LaserWriter® II modified with a 600 d.p.i Accel-a-Writer™ controller.

John Markoff. 1994. Flaw Undermines Accuracy of Pentium Chips. *New York Times* (24 Nov.). https://www.nytimes.com/1994/11/24/business/company-news-flaw-undermines-accuracy-of-pentium-chips.html (also at Internet Archive 26 May 2015 11:01:56). Section D, page 1 of print edition, datelined San Francisco, November 23; continued on page 5. Includes quote from Cleve Moler. Includes a photograph of Cleve Moler sitting in front of a desk computer running MATLAB. Includes a sidebar with information credited to Cleve Moler.

W. A. Martin and R. J. Fateman. 1971. The MACSYMA System. In *Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation* (Los Angeles, California, USA) (*SYMSAC '71*). Association for Computing Machinery, New York, New York, USA, 59–75. https://doi.org/10.1145/800204.806267

Cleve Moler. 1980. Design of an Interactive Matrix Calculator. In *Proceedings of the May 19–22, 1980, National Computer Conference* (Anaheim, California) (*AFIPS '80*). ACM, New York, New York, USA, 363–368. https://doi.org/10.1145/1500518.1500576

Cleve Moler. 1982. Demonstration of a Matrix Laboratory. In *Numerical Analysis*, J. P. Hennart (Ed.). Springer Berlin Heidelberg, Berlin, Germany, 84–98. https://doi.org/10.1007/BFb0092962

Cleve Moler. 1988. MATLAB—A Mathematical Visualization Laboratory. In *Thirty-Third IEEE Computer Society International Conference: Digest of Papers* (*COMPCON Spring 88*). IEEE Computer Society, IEEE, Washington, DC, USA (March), 480–481. https://doi.org/10.1109/CMPCON.1988.4915

Cleve Moler. 2015. Technical Perspective: Not Just a Matrix Laboratory Anymore. *Commun. ACM* 58, 10 (Sept.), 90. https://doi.org/10.1145/2814849 This article is a Technical Perspective on [Trefethen 2015].

Cleve B. Moler. 1965. *Finite Difference Methods for the Eigenvalues of Laplace's Operator.* Ph.D. Dissertation. Stanford University, Stanford, California, USA (May). https://www.worldcat.org/title/finite-difference-methods-for-the-eigenvalues-of-laplaces-operator/oclc/25608368

Cleve B. Moler. 1967. Accurate Solution of Linear Algebraic Systems: A Survey. In *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference* (Atlantic City, New Jersey, USA) (*AFIPS '67 (Spring)*). Association for Computing Machinery, New York, New York, USA (April), 321–324. https://doi.org/10.1145/1465482.1465533

Cleve B. Moler. 1969. State of the Art in Matrix Computations. *SIGNUM Newsletter* 4, 1 (Jan.), 22–28. https://doi.org/10.1145/1198450.1198454

Cleve B. Moler. 1972. Matrix Computations with Fortran and Paging. *Commun. ACM* 15, 4 (April), 268–270. https://doi.org/10.1145/361284.361297

Lawrence F. Shampine. 1986. Some Practical Runge-Kutta Formulas. *Math. Comp.* 46, 173, 135–150. https://doi.org/10.1090/S0025-5718-1986-0815836-3

Lawrence F. Shampine and Mark W. Reichelt. 1997. The MATLAB ODE Suite. *SIAM Journal on Scientific Computing* 18, 1, 1–22. https://doi.org/10.1137/S1064827594276424

Gaurav Sharma and Jos Martin. 2009. MATLAB®: A Language for Parallel Computing. *International Journal of Parallel Programming* 37, 1, 3–36. https://doi.org/10.1007/s10766-008-0082-5 https://link.springer.com/content/pdf/10.1007/s10766-008-0082-5.pdf

Jim Sizemore and John Paul Mueller. 2014. *MATLAB for Dummies.* John Wiley & Sons, 111 River Street, Hoboken, New Jersey, USA. 432 pages. https://www.wiley.com/en-us/MATLAB+For+Dummies-p-9781118820100

B. T. Smith, J. M. Boyle, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. 1974. *Matrix Eigensystem Routines—EISPACK Guide.* Lecture Notes in Computer Science, Vol. 6. Springer-Verlag, Berlin. 551 pages. https://doi.org/10.1007/978-3-540-38004-7

Marc Snir, William Gropp, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. 1998. *MPI—The Complete Reference: The MPI Core* (second ed.). Vol. 1. MIT Press, Cambridge, Massachusetts, USA.

Dean Takahashi. 1994. Flaw Undermines Accuracy of Pentium Chips. *Mercury News* (19 Dec.). https://www.mercurynews.com/2014/07/24/1994-the-pentium-principle/ Republished online on July 24, 2014; accessed January 4, 2020. Includes quote from Cleve Moler.

Lloyd N. Trefethen. 2000. *Spectral Methods in MATLAB.* Society for Industrial and Applied Mathematics, Philadelphia. https://doi.org/10.1137/1.9781611971811

Lloyd N. Trefethen. 2007. Computing Numerically with Functions Instead of Numbers. In *Proceedings of the 2007 International Workshop on Symbolic-Numeric Computation* (London, Ontario, Canada) (*SNC '07*). Association for Computing Machinery, New York, New York, USA, 28. https://doi.org/10.1145/1277500.1277505

Lloyd N. Trefethen. 2015. Computing Numerically with Functions Instead of Numbers. *Commun. ACM* 58, 10 (Sept.), 91–97. https://doi.org/10.1145/2814847 For the Technical Perspective on this article, see [Moler 2015].

J. H. Wilkinson and C. Reinsch. 1971. *Handbook for Automatic Computation, Volume II: Linear Algebra.* Die Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen, Vol. 186. Springer-Verlag, Berlin. 440 pages. https://doi.org/10.1007/978-3-642-86940-2

Niklaus Wirth. 1976. *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliffs, New Jersey, USA. 366 pages.

Aaron Zitner. 1994. Sorry, Wrong Number: Results are in: Intel computerchip sometimes makes inaccurate math. *Boston Globe* (24 Nov.). https://bostonglobe.newspapers.com/image/440628213/ Page 1 of print edition; continued on page 78. Includes quote from Cleve Moler.

NON-ARCHIVAL REFERENCES

Bob Bemer. 2000. How ASCII Got Its Backslash. Aug. 2000. Archived at https://web.archive.org/web/20130119163809/http://www.bobbemer.com/BACSLASH.HTM

Bob Bemer's brief account of his pivotal role in introducing the backslash character into the ASCII character set.

Stanley Cohen. 1973. *The SPEAKEASY-3 Reference Manual.* Technical Report ANL-8000. Argonne National Laboratory, Argonne, Illinois, USA (1 May). https://doi.org/10.2172/4334589 https://www.osti.gov/biblio/4334589

Part One provides an introduction to the SPEAKEASY language, intended to help scientists quickly formulate problems for computer processing. Its form is intended to be similar to that of mathematics; in particular, vector and matrix operations are supported. Part Two describes the SPEAKEASY system: operational features, library facilities, and the interactive mode and its editor. Part Three covers "linkules" (compiled Fortran packages that can be invoked from SPEAKEASY programs) and interfaces. Part Four describes the set of SPEAKEASY HELP documents available to the user using the interactive HELP command.

Available from U. S. Department of Energy Office of Scientific and Technical Information.

Erwin Fehlberg. 1969. *Low-order Classical Runge-Kutta Formulas with Stepsize Control and Their Application to Some Heat Transfer Problems.* Technical Report NASA TR R-315. National Aeronautics and Space Administration, Washington, DC, USA (July). https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19690021375.pdf (also at Internet Archive 17 Feb. 2015 08:03:45).

Stephen C. Johnson. 1975. Yacc: Yet Another Compiler-Compiler. https://pdfs.semanticscholar.org/76d1/ae5b5190a563418d333c061043b416b8668a.pdf Archived at https://web.archive.org//web/20200203211143/https://pdfs.semanticscholar.org/76d1/ae5b5190a563418d333c061043b416b8668a.pdf 33 pages.

S. C. Johnson. 1978. Lint, a C Program Checker. 26 July 1978. https://pdfs.semanticscholar.org/7461/7cffa3c6438d04aa99bef1cca415de47d0d3.pdf (also at Internet Archive 3 Feb. 2020 21:10:52). 12 pages.

Stephen C. Johnson. 1979. A Tour through the Portable C Compiler. https://pdfs.semanticscholar.org/454e/0ca1aa19db18737a25abe2e54dc0f7014e01.pdf (also at Internet Archive 27 May 2019 09:02:49). Part of the Unix Programmer's Manual. 23. pages.

Lennart Ljung. 2012a. Introduction to System Identification (video recording, 46 minutes). 3 April 2012. https://www.mathworks.com/videos/introduction-to-system-identification-81796.html Also on YouTube: https://youtu.be/u7hJ1aF-JrU (retrieved 20 May 2017)

"In this webinar, you will have a unique chance to learn about system identification from a world-renowned subject expert, Professor Lennart Ljung. Professor Ljung will explain the basic concepts of system identification and will show you how to get started with System Identification Toolbox™."

Lennart Ljung. 2012b. System Identification Toolbox: History and Development (video recording, 4 minutes). https://www.mathworks.com/videos/lennart-ljung-on-system-identification-toolbox-history-and-development-96989.html Also on YouTube: https://youtu.be/JFQnSX5dsaE (retrieved 4 Dec. 2014)

"Professor Lennart Ljung describes how he developed System Identification Toolbox™ and why he chose to write it in MATLAB®."

MathWorks, Inc. 2019. Company Overview. 2 pages. https://www.mathworks.com/content/dam/mathworks/fact-sheet/company-fact-sheet-8282v19.pdf (retrieved 30 Dec. 2019; also at Internet Archive 3 Feb. 2020 20:43:44).

Cleve Moler. 1981. MATLAB User's Guide. May 1981. ASCII file; approximately 52 pages (roughly 3100 lines of text); see also [NA Moler 2018g].

Cleve Moler. 2006. Cleve's Corner: The Growth of MATLAB and The MathWorks over Two Decades. *The MathWorks News&Notes* (Jan.), 22–24. https://www.mathworks.com/company/newsletters/articles/the-growth-of-matlab-and-the-mathworks-over-two-decades.html (also at Internet Archive 14 July 2014 16:31:11).

Cleve Moler. 2013a. Blog post: Cleve's Corner: Backslash. 19 Aug. 2013. https://blogs.mathworks.com/cleve/2013/08/19/backslash (also at Internet Archive 20 Aug. 2013 05:28:53).

Cleve Moler. 2013b. Blog post: Cleve's Corner: George Forsythe. 7 Jan. 2013. https://blogs.mathworks.com/cleve/2013/01/07/george-forsythe (also at Internet Archive 8 June 2015 09:56:09).

Cleve Moler. 2013c. Blog post: Cleve's Corner: Pentium Division Bug Affair. 13 May 2013. https://blogs.mathworks.com/cleve/2013/05/13/pentium-division-bug-affair (also at Internet Archive 24 Sept. 2015 14:03:33).

Cleve Moler. 2013d. Blog post: Cleve's Corner: Pentium Division Bug Revisited. 29 April 2013. https://blogs.mathworks.com/cleve/2013/04/29/pentium-division-bug (also at Internet Archive 11 June 2013 10:36:59).

Cleve Moler. 2014a. Blog post: Cleve's Corner: MathWorks Logo, Part Five. Evolution of the Logo. 1 Dec. 2014. https://blogs.mathworks.com/cleve/2014/12/01/mathworks-logo-part-five-evolution-of-the-logo (also at Internet Archive 7 Jan. 2015 06:53:42).

Cleve Moler. 2014b. Blog post: Cleve's Corner: MathWorks Logo, Part Four. Method of Particular Solutions Generates the Logo. 17 Nov. 2014. https://blogs.mathworks.com/cleve/2014/11/17/mathworks-logo-part-four-method-of-particular-solutions-generates-the-logo (also at Internet Archive 31 Dec. 2014 22:09:26).

Cleve Moler. 2014c. Blog post: Cleve's Corner: MathWorks Logo, Part One. Why Is It L Shaped? 13 Oct. 2014. https://blogs.mathworks.com/cleve/2014/10/13/mathworks-logo-part-one-why-is-it-l-shaped (also at Internet Archive 17 Oct. 2014 11:01:40).

Cleve Moler. 2014d. Blog post: Cleve's Corner: MathWorks Logo, Part Three. PDE Toolbox. 3 Nov. 2014. https://blogs.mathworks.com/cleve/2014/11/03/mathworks-logo-part-three-pde-toolbox (also at Internet Archive 8 Nov. 2014 19:44:44).

Cleve Moler. 2014e. Blog post: Cleve's Corner: MathWorks Logo, Part Two. Finite Differences. 22 Oct. 2014. https://blogs.mathworks.com/cleve/2014/10/22/mathworks-logo-part-two-finite-differences (also at Internet Archive 30 Oct. 2014 09:41:43).

Cleve Moler. 2015. Blog post: Cleve's Corner: ALGOL 60, PL/0 and MATLAB. 15 June 2015. https://blogs.mathworks.com/cleve/2015/06/15/algol-60-pl0-and-matlab (also at Internet Archive 24 Sept. 2015 18:41:56).

Cleve Moler. 2016. Blog post: Cleve's Corner: The Pentium Papers — My First MATLAB Central Contribution. 5 Sept. 2016. https://blogs.mathworks.com/cleve/2016/09/05/the-pentium-papers-my-first-matlab-central-contribution (also at Internet Archive 7 Sept. 2016 21:36:23).

Cleve Moler and Steve Eddins. 2001. Technical Article: Faster Finite Fourier Transforms MATLAB. https://www.mathworks.com/company/newsletters/articles/faster-finite-fourier-transforms-matlab.html (also at Internet Archive 24 March 2014 12:50:52).

Cleve B. Moler. 2000. Blog post: Cleve's Corner: MATLAB Incorporates LAPACK. http://www.mathworks.com/company/newsletters/articles/matlab-incorporates-lapack.html (also at Internet Archive 30 June 2012 04:25:47).

Cleve B. Moler. 2013e. Blog post: Cleve's Corner: LINPACK Benchmark. 24 June 2013. http://blogs.mathworks.com/cleve/2013/06/24/the-linpack-benchmark (also at Internet Archive 5 Dec. 2013 20:15:12).

Cleve B. Moler. 2014f. Blog post: Cleve's Corner: Ordinary Differential Equation Suite. 12 May 2014. https://blogs.mathworks.com/cleve/2014/05/12/ordinary-differential-equation-suite (also at Internet Archive 18 May 2014 09:29:38).

Cleve B. Moler. 2017. Blog post: Cleve's Corner: Wilkinson & Reinsch Handbook. 4 Dec. 2017. http://blogs.mathworks.com/cleve/2017/12/04/wilkinson-and-reinsch-handbook-on-linear-algebra (also at Internet Archive 27 Dec. 2019 13:56:32).

Cleve B. Moler. 2018a. Blog post: Cleve's Corner: EISPACK. 2 Jan. 2018. http://blogs.mathworks.com/cleve/2018/01/02/eispack-matrix-eigensystem-routines (also at Internet Archive 3 Feb. 2020 18:46:03).

Cleve B. Moler. 2018b. Blog post: Cleve's Corner: LINPACK. 23 Jan. 2018. http://blogs.mathworks.com/cleve/2018/01/23/linpack-linear-equation-package (also at Internet Archive 3 Feb. 2020 18:50:10).

Cleve B. Moler. 2018c. Blog post: Cleve's Corner: Modern MATLAB, Part 1. 21 March 2018. http://blogs.mathworks.com/cleve/2018/03/21/matlab-history-modern-matlab-part-1 (also at Internet Archive 3 Feb. 2020 19:01:50).

Cleve B. Moler. 2018d. Blog post: Cleve's Corner: Modern MATLAB, Part 2. 30 April 2018. http://blogs.mathworks.com/cleve/2018/04/30/matlab-history-modern-matlab-part-2 (also at Internet Archive 3 Feb. 2020 19:03:08).

Cleve B. Moler. 2018e. Blog post: Cleve's Corner: PC MATLAB. 9 March 2018. http://blogs.mathworks.com/cleve/2018/03/09/matlab-history-pc-matlab-version-1-0 (also at Internet Archive 3 Feb. 2020 19:00:14).

Cleve B. Moler. 2018f. Blog post: Cleve's Corner: Technical Computing Environment. 14 May 2018. http://blogs.mathworks.com/cleve/2018/05/14/the-matlab-technical-computing-environment (also at Internet Archive 3 Feb. 2020 19:04:26).

Cleve B. Moler. 2018g. Blog post: Cleve's Corner: The Historic MATLAB Users' Guide. 5 Feb. 2018. http://blogs.mathworks.com/cleve/2018/02/05/the-historic-matlab-users-guide (also at Internet Archive 3 Feb. 2020 18:58:06).

Associated Press. 1994. No title (news article). 24 Nov. 1994. Dateline Boston, AP 24 Nov 94 0:10 EST V0204.

L. F. Shampine and H. A. Watts. 1976. *Practical Solution of Ordinary Differential Equations by Runge-Kutta Methods. [Writing a High-Quality Code; Description of RKF45, in FORTRAN for CDC 6600 Computer].* Technical Report SAND76-0585. Sandia National Laboratory, Albuquerque, New Mexico, USA (Dec.). https://www.osti.gov/biblio/7318812 (OSTI bibliographic record only; document itself is not available at this URL).

Abstract: All those factors bearing on the creation of a high-quality Runge–Kutta code for the solution of ordinary differential equations are studied. These include the selection of a formula based on many theoretical and experimental measures of quality; the various aspects of constructing efficient, reliable algorithms; and the design of convenient but flexible software. A code, RKF45, is presented as a concrete realization of the conclusions of the study. 29 tables.

Stanford University. 1979. Stanford University Bulletin: Courses and Degrees 1979–1980. Sept. 1979. 646 pages. https://purl.stanford.edu/ry567fw5185