

Scientific Libraries User's Guide

004-2151-002

© 1996, 1999 Silicon Graphics, Inc. All Rights Reserved. This manual or parts thereof may not be reproduced in any form unless permitted by contract or by written permission of Silicon Graphics, Inc.

LIMITED AND RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in the Rights in Data clause at FAR 52.227-14 and/or in similar or successor clauses in the FAR, or in the DOD, DOE or NASA FAR Supplements. Unpublished rights reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 1600 Amphitheatre Pkwy., Mountain View, CA 94043-1351.

Autotasking, CF77, Cray, Cray Ada, CraySoft, Cray Y-MP, Cray-1, CRInform, CRI/*TurboKiva*, HSX, LibSci, MPP Apprentice, SSD, SUPERCLUSTER, UNICOS, X-MP EA, and UNICOS/mk are federally registered trademarks and Because no workstation is an island, CCI, CCMT, CF90, CFT, CFT2, CFT77, ConCurrent Maintenance Tools, COS, Cray Animation Theater, Cray APP, Cray C90, Cray C90D, Cray C++ Compiling System, CrayDoc, Cray EL, Cray J90, Cray J90se, CrayLink, Cray NQS, Cray/REELibrarian, Cray S-MP, Cray SSD-T90, Cray SV1, Cray T90, Cray T3D, Cray T3E, CrayTutor, Cray X-MP, Cray XMS, Cray-2, CSIM, CVT, Delivering the power . . ., DGauss, Docview, EMDS, GigaRing, HEXAR, IOS, ND Series Network Disk Array, Network Queuing Environment, Network Queuing Tools, OLNET, RQS, SEGLDR, SMARTE, SUPERLINK, System Maintenance and Remote Testing Environment, Trusted UNICOS, and UNICOS MAX are trademarks of Cray Research, L.L.C., a wholly owned subsidiary of Silicon Graphics, Inc.

SGI is a trademark of Silicon Graphics, Inc. Silicon Graphics, the Silicon Graphics logo, and IRIS are registered trademarks, and CASEVision, IRIS 4D, IRIS Power Series, IRIX, Origin2000, and POWER CHALLENGE are trademarks of Silicon Graphics, Inc. MIPS, R4000, R4400, and R8000 are registered trademarks and MIPSpro and R10000 are trademarks of MIPS Technologies, Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd. VMS and VAX are trademarks of Digital Equipment Corporation.

Portions of this product and document are derived from material copyrighted by Kuck and Associates, Inc.

DynaText and DynaWeb are registered trademarks of Inso Corporation. Silicon Graphics and the Silicon Graphics logo are registered trademarks of Silicon Graphics, Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. X/Open is a trademark of X/Open Company Ltd. The X device is a trademark of the Open Group.

The UNICOS operating system is derived from UNIX® System V. The UNICOS operating system is also based in part on the Fourth Berkeley Software Distribution (BSD) under license from The Regents of the University of California.

St. Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

New Features

Scientific Libraries User's Guide

004-2151-002

This guide has been expanded to include appendixes that describe the implementation of version 2 of the Math library (`libm`), used on UNICOS systems, as well as describing the algorithms used in that library.

Record of Revision

<i>Version</i>	<i>Description</i>
2.0	July 1996. Draft printing to support the Programming Environment 2.0.1 release.
3.0	June 1997. Printing to support the Programming Environment 3.0 release.
3.3	July 1999. Printing to support the Programming Environment 3.3 release.

Contents

	<i>Page</i>
About This Guide	xi
Related publications	xi
Conventions	xiv
Obtaining Publications	xv
Reader Comments	xv
Introduction [1]	1
Parallel Processing Issues [2]	3
Parallel Processing Overview	3
Parallel Processing: Hardware Level	3
Parallel Processing: Operating System Level	4
Parallel Processing: Code Level	5
Costs and Benefits of Parallel Processing	5
Benefits of Parallel Processing	6
Parallel Processing Overhead	6
Parallelism and Vectorization	7
Parallelism and Load Balancing	8
Performance Issues	8
Calculating Speedups in Multitasked Programs	8
Amdahl's Law	9
Determining Efficiency	10
Estimating Percentage of Parallelism	11
Parallel Processing Environments	11
Environment Variables	12
004-2151-002	iii

	<i>Page</i>
The Dedicated Environment	13
The Multiuser Environment	13
Measuring Scientific Library Performance	14
Simple Measurements	14
Determining Multitasking of Routines	15
Parallel Processing Strategies	20
Problem Sizes	20
Strategies for a Dedicated Environment	21
Strategies for a Multiuser Environment	21
LAPACK [3]	25
LAPACK in the Scientific Library	25
Types of Problems Solved by LAPACK	26
Solving Linear Systems	27
Factoring a Matrix	29
Example 1: LU factorization	30
Example 2: Symmetric indefinite matrix factorization	31
Error Codes	33
Example 3: Error conditions	33
Solving from the Factored Form	34
Condition Estimation	36
Example 4: Roundoff errors	37
Use in Error Bounds	37
Equilibration	39
Iterative Refinement	41
Example 5: Hilbert matrix	41
Error Bounds	43
Inverting a Matrix	44
Solving Least Squares Problems	45

	<i>Page</i>
Orthogonal Factorizations	45
Example 6: Orthogonal factorization	46
Multiplying by the Orthogonal Matrix	48
Generating the Orthogonal Matrix	49
Comparing Answers	50
Using Sparse Linear Solvers [4]	53
Sparse Matrices	53
Solution Techniques	54
Direct Methods	55
Iterative Methods	56
Sparse Solvers	57
Data Structures for General Sparse Matrices	57
Direct Solvers	58
Iterative Solvers	59
Other Solvers	60
Choosing a Solver	62
Using Sparse Solvers	62
Tridiagonal Systems	62
General-patterned Sparse Linear Systems	63
Choosing a Method Based on Problem Type	64
Iterative Methods	65
Preconditioning	65
Direct General Sparse Solvers	67
Performance Tuning	67
Parallel Processing	67
Reusing Information	68
Reuse of Structure	68
Multiple Right-hand Sides	68

	<i>Page</i>
Reuse of Values	68
Save/restart	69
SITRSOL Tuning Issues	69
Direct Solver Tuning Issues	70
SITRSOL Quick Reference	72
Usage Examples	75
Example 7: General symmetric positive definite	75
Example 8: General unsymmetric	79
Example 9: Reuse of structure	84
Example 10: Multiple right-hand sides	89
Example 11: Save/restart	93
Out-of-core Linear Algebra Software [5]	101
Out-of-core Routines	101
Virtual Matrices	102
Unit Numbers	102
File Format	103
Leading Virtual Dimension	104
Definition and Redefinition of Elements	104
File Size	104
Packed Storage Mode	105
Page Size	106
Subroutine Types	106
Complex Routines	106
Summary of Routines	107
Initialization and Termination Subroutines	109
Virtual Copy Subroutines	109
Virtual LAPACK Subroutines	110
Virtual BLAS Subroutines	111

	<i>Page</i>
Using Strassen's algorithm	111
Lower-level Routines	112
Examples of Out-of-core Subroutine Use	113
Example 12: Creating a virtual matrix	113
Example 13: Multiplying a virtual matrix	113
Example 14: Example of protocol usage	114
UNICOS Environment Variables	115
Multitasking	116
Error Reporting	116
Performance Measurement and Tuning	117
Page-Buffer Space	118
Memory Usage Guidelines	118
Memory Requirement for VSGETRF and VCGETRF Routines	119
Sample Performance Statistics	119
Appendix A Appendix A: libm Version 2	123
Overview of Math Libraries	123
The 1 ULP Criterion	124
Numerical Methods	125
Side Effects	128
Appendix B Appendix B: Math Algorithms	129
Single-precision Real Logarithm Functions $\ln(x)$ and $\log(x)$	129
Procedure 1: $\ln(x)$ and $\log(x)$	129
Accuracy	131
Single-precision Real Logarithm Functions $\text{ALOG}(x)$ and $\text{ALOG10}(x)$	132
Procedure 2: $\text{ALOG}(x)$ and $\text{ALOG10}(x)$	132
Accuracy	134
Single-precision Real $\text{ASIN}(x)$ and $\text{ACOS}(x)$ Functions	135

	<i>Page</i>
Procedure 3: $\text{ASIN}(x)$	135
Procedure 4: $\text{ACOS}(x)$	136
Accuracy	138
Single-precision Real $\text{ATAN}(x)$ Function	138
Procedure 5: $\text{ATAN}(x)$	138
Accuracy	139
Single-precision $\text{ATAN}(y,x)$ Function	139
Procedure 6: $\text{ATAN}(y,x)$	139
Accuracy	143
Single-precision Real $\text{CBRT}(x)$ Function	143
Procedure 7: $\text{CBRT}(x)$	143
Accuracy	145
Single-precision Real Exponential Function E^x	145
Procedure 8: e^x	145
Accuracy	147
Single-precision Real Power Function x^y	147
Procedure 9: x^y	147
Accuracy	150
Single-precision Real $\text{SIN}(x)$ and $\text{COS}(x)$ Functions	151
Procedure 10: $\text{SIN}(x)$ and $\text{COS}(x)$	151
Accuracy	154
Single-precision Real $\text{COSH}(x)$ and $\text{SINH}(x)$ Functions	155
Procedure 11: $\text{COSH}(x)$ and $\text{SINH}(x)$	155
Accuracy	156
Single-precision Real $\text{SQRT}(x)$ Function	156
Procedure 12: $\text{SQRT}(x)$	156
Accuracy	158
Single-precision $\text{TAN}(x)$ and $\text{COT}(x)$ Functions	158

	<i>Page</i>
Procedure 13: $\text{TAN}(x)$ and $\text{COT}(x)$	158
Accuracy	161
Single-precision Real $\text{TANH}(x)$ Function	162
Procedure 14: $\text{TANH}(x)$	162
Accuracy	163
Glossary	165
Index	171
Figures	
Figure 1. Pipelining in add operation	4
Figure 2. Pipelining and chaining	4
Figure 3. Cost/robustness: general symmetric sparse solvers	63
Figure 4. Cost/robustness: general unsymmetric sparse solvers	64
Figure 5. In-memory to virtual matrix copy	110
Figure 6. Layered software design	112
Tables	
Table 1. Relative cost for <i>SGER</i>	23
Table 2. Factorization forms	30
Table 3. Solves times: LAPACK and solver routines	35
Table 4. Verification tests for LAPACK (all should be $O(1)$)	50
Table 5. Summary of tridiagonal solvers	61
Table 6. <i>SITRSOL</i> argument summary	72
Table 7. <i>iparam</i> summary	73
Table 8. <i>rparam</i> summary	74
Table 9. Summary of out-of-core routines for linear algebra	107

About This Guide

This publication describes the Scientific Libraries (`libsci`) which run on UNICOS systems. The information in this manual supplements the man pages provided with the Scientific Library.

This document is a user's guide for programmers. Readers should have a working knowledge of the UNICOS operating system, have an understanding of the Fortran programming language, and have a working familiarity with scientific and mathematical theories.

Related publications

The following publications provide information related to the Scientific Library:

- *UNICOS User Commands Reference Manual*
- *UNICOS System Libraries Reference Manual*
- *Scientific Library Reference Manual*
- *Optimizing Application Code on UNICOS Systems*
- *CF90 Commands and Directives Reference Manual*
- *Fortran Language Reference Manual, Volume 1*
- *Fortran Language Reference Manual, Volume 2*
- *Fortran Language Reference Manual, Volume 3*
- *LINPACK User's Guide*
- *LAPACK User's Guide*

The following publications provide detailed information about the topics discussed in this manual. In many cases, these documents are referenced specifically in this manual.

- Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. Philadelphia SIAM, 1992.

- Anderson, Edward, Jack Dongarra, and Susan Ostrouchov. Installation guide for LAPACK. LAPACK Working Note 41, Technical Report CS-91-138. University of Tennessee (Feb. 1992).
- Argham, Nicolas J. *Accuracy and Stability of Numeric Algorithms*. Philadelphia SIAM, 1996.
- Arioli, M., J. W. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Anal. Appl.* 10 (1989).
- Ashcraft, Cleve. A vector implementation of the multifrontal method for large sparse, symmetric positive definite linear systems. Technical Report ETA-TR-51. Boeing Computer Services, 1987.
- Duff, I. S., A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Monographs on Numerical Analysis. New York: Oxford University Press, 1986.
- Duff, Iain, Michele Marrone, and Giuseppe Radicati. A proposal for user-level sparse BLAS. Technical Report TR/PA/92/85. CERFACS (Dec. 1992).
- George, Alan and Joseph W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall Series in Computational Mathematics. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.
- Golub, Gene and James M. Ortega. *Scientific Computing: An Introduction with Parallel Computing*. Boston: Academic Press, 1993.
- Golub, Gene H. and Charles F. Van Loan. *Matrix Computations*. 2nd edition. Baltimore, Maryland: Johns Hopkins University Press, 1989.
- Hageman, Louis A. and David M. Young. *Applied Iterative Methods*. Computer Science and Applied Mathematics. New York and London: Academic Press, 1981.
- Heroux, Michael A. A reverse communication interface for “matrix-free” preconditioned iterative solvers. Edited by C.A. Brebbia, D. Howard, and A. Peters *In Applications of Supercomputers in Engineering II*, 207-213. Boston: Computational Mechanics Publications, 1991.
- Heroux, Michael A. A proposal for a sparse BLAS toolkit. Technical Report TR/PA/92/90. CERFACS (Dec. 1992).
- Heroux, Michael A., Phuong Vu, and Chao Wu Yang. A parallel preconditioned conjugate gradient package for solving sparse linear systems on a Cray Y-MP. *Applied Numerical Mathematics*, 8 (1991).

- Hestenes, M. R. and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. National Bureau of Standards* 49 (1952): 409-436.
- Kincaid, David R., Thomas C. Oppe, John R. Respass, and David M. Young. *ITPACKV 2C User's Guide*. Technical Report CNA-191. The University of Texas at Austin: Center for Numerical Analysis, (Nov. 1984).
- Manteuffel, T. A. An incomplete factorization technique for positive definite linear systems. *Math. Comp.* 34 (1980): 473-497.
- Oppe, Thomas C., Wayne D. Joubert, and David R. Kincaid. *NSPCG User's Guide*. The University of Texas at Austin: Center for Numerical Analysis, (Dec. 1988).
- Reid, J. K., editor. On the Method of Conjugate Gradients for the Solution of Large Sparse Systems of Linear Equations. *Large Sparse Sets of Linear Equations*, Academic Press, 1971.
- Saad, Youcef. *SPARSKIT: a basic tool kit for sparse matrix computations*. Preliminary Version.
- Saad, Youcef. Practical use of polynomial preconditionings for the conjugate gradient method., 6(4) (Oct. 1985): 865-881.
- Saad, Youcef and Martin H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal of Scientific and Statistical Computing*, 7(3) (Jul. 1986): 856-869.
- Sonneveld, Peter. CGS, a fast lanczos-type solver for nonsymmetric linear systems. *SIAM Journal of Scientific and Statistical Computing*, 10(1) (Jan. 1989): 36-52.
- Stewart, G. W. *Introduction to Matrix Computations*. Orlando, Florida: Academic Press, 1973.
- Wilkinson, J. H. *The Algebraic Eigenvalue Problem*. Oxford, England: Oxford University Press, 1965.
- Yang, Chao W. A parallel multifrontal method for sparse symmetric definite linear systems on the Cray Y-MP. *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*. Houston, Texas (Apr. 1992).

You can find a good general reference on the solution of sparse linear systems in Golub and Van Loan. You can find a good introduction to direct and iterative methods, as well as methods for special linear systems, in these texts. See the special section of the November 1989 issue of the *SIAM Journal of Scientific and Statistical Computing*, pages 1135-1232 for an updated general reference.

See George and Liu, Duff and Erisman, and Reid for classical references that give a thorough and in-depth treatment of sparse direct solvers. Another common reference is Ashcraft.

The original conjugate gradient algorithm was presented in Hestenes and Stiefel; however, Reid presented the first practical application. A classical text in iterative methods is that of Hageman and Young. You can find good discussions of the biconjugate gradient and biconjugate gradient squared methods in Sonneveld. GMRES is presented by Saad and Schultz.

You can find some references on data structures in SPARSKIT and in the proposals for sparse BLAS.

Three articles that deal directly with the Cray Research *libsci* sparse solvers are Yang on direct solvers, Heroux (1991), and Heroux, Vu, and Yang on *SITRSOL*.

Conventions

The following conventions are used throughout this documentation:

<code>command</code>	This fixed-space font denotes literal items, such as pathnames, man page names, commands, and programming language structures.
<i>variable</i>	Italic typeface denotes variable entries and words or concepts being defined.
[]	Brackets enclose optional portions of a command line.

In addition to these formatting conventions, several naming conventions are used throughout the documentation. “Cray PVP systems” denotes all configurations of Cray parallel vector processing (PVP) systems which run the UNICOS operating system. “Cray MPP systems” denotes all configurations of the Cray T3E series which runs the UNICOS/mk operating system. “IRIX systems” denotes SGI platforms which run the IRIX operating system.

The default shell in the UNICOS and UNICOS/mk operating systems, referred to as the *standard shell*, is a version of the Korn shell that conforms to the following standards:

- Institute of Electrical and Electronics Engineers (IEEE) Portable Operating System Interface (POSIX) Standard 1003.2-1992
- X/Open Portability Guide, Issue 4 (XPG4)

The UNICOS and UNICOS/mk operating systems also support the optional use of the C shell.

Cray UNICOS Version 10.0 is an X/Open Base 95 branded product.

Obtaining Publications

The *User Publications Catalog* describes the availability and content of all Cray Research hardware and software documents that are available to customers. Customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

To order a document, call +1 651 683 5907. SGI employees may send electronic mail to orderdsk@sgi.com (UNIX system users).

Customers who subscribe to the CRInform program can order software release packages electronically by using the `Order Cray Software` option.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and part number of the document with your comments.

You can contact us in any of the following ways:

- Send e-mail to the following address:
techpubs@sgi.com
- Send a fax to the attention of "Technical Publications" at: +1 650 932 0801.
- Use the Feedback option on the Technical Publications Library World Wide Web page:

<http://techpubs.sgi.com>

- Call the Technical Publications Group, through the Technical Assistance Center, using one of the following numbers:

For SGI IRIX based operating systems: 1 800 800 4SGI

For UNICOS or UNICOS/mk based operating systems or Cray Origin 2000 systems: 1 800 950 2729 (toll free from the United States and Canada) or +1 651 683 5600

- Send mail to the following address:

Technical Publications
SGI
1600 Amphitheatre Pkwy.
Mountain View, California 94043-1351

We value your comments and will respond to them promptly.

Introduction [1]

This manual describes the Scientific Libraries which run on UNICOS systems. The information in this manual supplements the man pages provided with the Scientific Library and provides details about the implementation and usage of these library routines on UNICOS systems.

This manual includes the following sections:

- Chapter 2, page 3, discusses parallel processing environments, ways to measure parallel processing performance, and the implementation strategies used in the Scientific Library routines.
- Chapter 3, page 25, discusses dense linear algebra problems, including systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems.
- Chapter 4, page 53, discusses sparse matrices and solution techniques for sparse linear systems.
- Chapter 5, page 101, discusses the out-of-core routines, virtual matrices, and subroutines used with out-of-core routines.
- Appendix A, page 123, discusses `libm` Version 2, the default UNICOS math library.
- Appendix B, page 129, discusses the algorithms used in `libm`.

Parallel Processing Issues [2]

Parallel processing is a method of splitting a computational task into subtasks, and then simultaneously performing the subtasks. This section discusses the different ways parallel processing strategies used on UNICOS systems, and what effects those implementations have on the performance of your code.

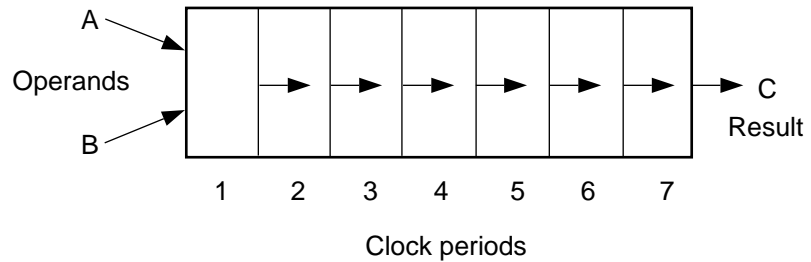
2.1 Parallel Processing Overview

Parallel processing is performed at the hardware level, the operating system level, and the code level. This subsection briefly discusses these different types of parallel processing.

2.1.1 Parallel Processing: Hardware Level

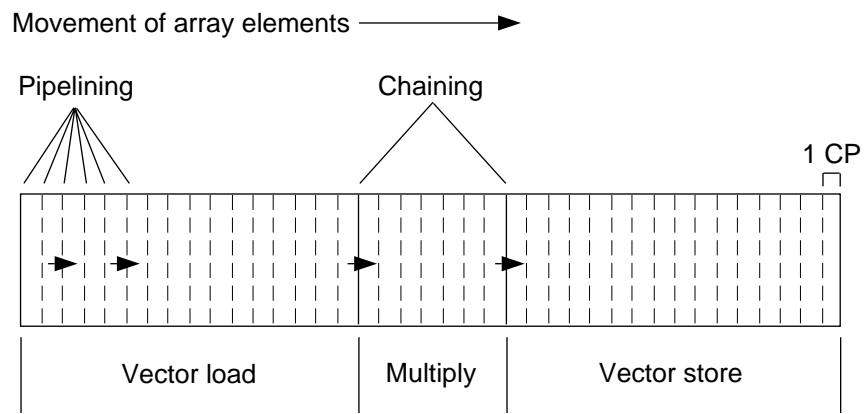
At the hardware level, parallel processing is accomplished by using the following methods:

- *parallel instruction execution*, which is the execution of one instruction per clock period, even those instructions that take several clock periods to complete execution.
- *vectorization*, which is a form of parallel processing that uses instruction segmenting and vector registers.
- *I/O subsystems* or *foreground processors*, which is the execution of operations in parallel with processes running in the main processors that perform I/O.
- *time slicing*, in which the system works on several jobs or processes simultaneously.
- *pipelining*, which allows each step of an operation to pass its result to the next step after only one clock period (see Figure 1).
- *chaining*, which allows the movement of elements to continue from one vector operation to another, so that a process including more than one vector operation is executed as one long vectorized operation (see Figure 2).



a10035

Figure 1. Pipelining in add operation



For illustration only. Functional units are not physically arranged as shown.

a10036

Figure 2. Pipelining and chaining

2.1.2 Parallel Processing: Operating System Level

At the operating system level, parallel processing is accomplished by using the following methods:

- *multiprogramming*, which occurs when a processor switches between the jobs or processes in the system.
- *multiprocessing*, which occurs when processors simultaneously work on as many programs as there are processors.

- *multitasking*, which is a general term describing more than one processor working to complete a single program. This term has become synonymous with parallel processing.

2.1.3 Parallel Processing: Code Level

There are several ways to alter user code to take advantage of parallel processing:

- *macrotasking* lets you take advantage of multitasking at the subroutine level by inserting library calls to express parallelism. This works well on programs that can be explicitly partitioned into tasks and can be simultaneously executed on multiple processors.
- *microtasking* allows you to insert directives at the loop or block level of the code to indicate sections that can be executed on multiple CPUs.

Macrotasking and microtasking are seldom used anymore. *Autotasking* is the preferred method to use for parallel processing at the code level. Like microtasking, Autotasking exploits parallelism at the loop or block level of code. It has lower overhead than microtasking, and it can be made fully automatic. It does not require any direct user intervention, but users can interact with the system via directives and switches.

The Scientific Library routines were designed to be fully optimized to take advantage of parallelism and to automatically use Autotasking during execution.

When using the CF90 compiler, you can use the `f90 -O3` command. This command invokes aggressive optimization and Autotasking. One of the optimization actions the compiler then performs is to substitute Scientific Library routines (where appropriate) in the code. The routines are called at an entry point where they will use less compiling time.

See the *CF90 Commands and Directives Reference Manual* or the `f90(1)` man page for more information about using the `f90` command.

2.2 Costs and Benefits of Parallel Processing

Parallel processing can eliminate idle CPU time because the workload is divided among all CPUs; therefore, the amount of work performed per unit time (the *throughput*) increases. However, parallel processing also introduces some overhead into program execution. In some cases, you may be able to

reduce wall-clock time, but at the cost of extra CPU time which increases because more machine resources are used.

This subsection discusses these benefits and some of the costs of using parallel processing.

2.2.1 Benefits of Parallel Processing

By using parallel processing, you can alleviate some of the following common problems:

- **Maximum-memory jobs:** if the memory is occupied by a few large-memory jobs, one or more of the CPUs might be idle even though there are other jobs to run.
- **Dedicated machine:** if the computer is running a single job, then all other CPUs are idle.
- **Light workload:** if the amount of jobs waiting for a CPU is less than the total number of CPUs, then one or more of the CPUs becomes idle.

With parallel processing, the additional CPUs reduce the wall-clock time instead of sitting idle. Even when very little idle time exists, using additional CPUs can still lead to benefits.

2.2.2 Parallel Processing Overhead

Parallel processing introduces some overhead into program execution. This subsection discusses some of the common types of overhead introduced by parallel processing:

- Multitasked programs require more memory than unitasked programs, and they can contain more code, more temporary variables, and can require additional stack space.
- Multitasked jobs can be swapped more often, and remain swapped longer, on a heavily loaded production system.
- Processors are forced to wait on semaphores during the process of synchronization.
- Overhead is incurred when slave processors are acquired (on entry to a parallel region) and at synchronization points within parallel regions. Tests show that the overhead of executing extra Autotasking code adds a nominal 0% to 5% to the overall execution time.

- If inner-loop Autotasking is used, vector performance can decrease because of shorter vector lengths and more vector loop startups.
- Processors are sometimes held for the next parallel region to improve efficiency. While holding a processor can save time, it also costs time to acquire and hold them.

Because overhead is associated with work distribution, jobs with large granularity have less partitioning than smaller jobs. Large jobs, however, may have problems with load balancing.

2.2.3 Parallelism and Vectorization

A multitasked program usually has the same amount of work for the CPUs to perform as does a similar unitasked program. However, when the work is spread across many processors, the wall-clock time required to complete the work is usually less.

Vectorization is a form of parallel processing in which array elements are processed by groups. Vectorization usually decreases both CPU time and wall-clock time; multitasking decreases only wall-clock time. Thus, vectorization can give you large gains in terms of saving time and have relatively low costs associated with it.

If you make direct calls to Scientific Library routines, your CPU time or wall-clock time could also be affected. Scientific Library routines take advantage of vectorization and multitasking; because of the costs associated with multitasking, you may see an increase in CPU time when using these routines. You can control this increase somewhat by using the `NCPUS` environment variable to control the maximum number of CPUs to be used on a job. See Section 2.4.1, page 12, for details about using environment variables.

When using multitasking, the timing results in a nondedicated environment are not always reproducible. Several factors can affect the timings obtained from multitasked routines, including some of the following:

- System load
- I/O performed by other jobs
- Memory contention

If accurate timing results are desired, it is best to run a job in a dedicated environment where it is the only job being run.

2.2.4 Parallelism and Load Balancing

The parallelism for any region is determined by the number of partitions, or *chunks*, of independent work the region contains. If an autotasked loop has N iterations, N is the extent of parallelism for that loop. This means that the loop can effectively be broken into N independent chunks of work. For autotasked inner loops, the extent of parallelism is the number of iterations divided by the vector length when both Autotasking and vectorization can be used.

Load balancing is the process of dividing work done by each available processor into approximately equal amounts. In a multiuser environment, the number of available processors is constantly changing. When Autotasking is used, it automatically tries to perform dynamic load balancing by creating small-granularity parallelism. For example, if a DO loop is autotasked, work is allocated by default to each task one iteration at a time.

A direct relationship exists between load balancing and the extent of parallelism:

- The higher the extent of parallelism, the easier it is to balance the workload evenly across the processors.
- Small-granularity parallelism is easier to balance across available processors than large granularity parallelism.
- Small-granularity parallelism generates more overhead than large granularity parallelism. Synchronization is required each time a chunk of work is allocated to a processor.

2.3 Performance Issues

This subsection discusses the following different aspects used to define the performance of parallel processing: speedup, efficiency, and percentage of parallelism in a program.

2.3.1 Calculating Speedups in Multitasked Programs

On a dedicated (nonproduction) system, the speedup ratio for a multitasked program can be calculated in the following manner, where T_1 is the wall-clock execution time on a single-processor and T_p is the wall-clock execution time on p processors:

$$\text{Speedup ratio} = \frac{T_1}{T_p}$$

(2.1)

With p CPUs, a speedup ratio as close as possible to p is desired. If the program were completely parallel and no overhead existed, the speedup would be equal to p (for details about the overhead associated with multitasking, see Section 2.2.2, page 6).

For example, suppose a job takes 8.7 seconds to run on a single processor. When the job is rerun using four processors, the execution time decreases to 2.5 seconds; the speedup is the following:

$$s = \frac{8.7}{2.5} = 3.48$$

(2.2)

The speedup ratio of multitasked code has two limitations: Amdahl's Law (representing the speedup related to the sequential portion of the code), and multitasking overhead (discussed in Section 2.2.2, page 6).

2.3.1.1 Amdahl's Law

Some sections of programs, known as *single-threaded code segments*, must use a single processor. Amdahl's Law for parallel processing illustrates the effect of single-threaded code segments on multitasking performance. Amdahl's Law is shown in the following equation:

$$s = \frac{1}{(1 - f) + \frac{f}{p}}$$

(2.3)

The following components appear in this equation:

- s : Maximum expected speedup from multitasking
- p : Number of processors available for parallel execution
- f : Fraction of a program that can execute in parallel (the *parallel fraction*)

For example, suppose that code is 98% parallel, implying that 2% of the code runs in serial mode. Suppose that 64 processors are available. According to Amdahl's Law, the maximum speedup that you can expect is:

$$s = \frac{1}{(1 - 0.98) + \frac{0.98}{64}} = 28.32$$

(2.4)

The speedup from multitasking, s , is in terms of wall-clock time, not CPU time. Speedups equal to the physical number of processors require that the executing program use all processors effectively 100% of the time with no overhead. Because this is not possible, performance is dominated by the fraction of the time spent executing serial code.

Maximum speedup depends on your serial code. If the number of processors were infinite, the parallel term would be 0, but the serial term would remain, giving a maximum possible speedup of the following:

$$s = \frac{1}{.02} = 50$$

(2.5)

This is sometimes interpreted to mean that only 50 processors can be used on a 98% parallel problem. You can use any number of processors, but because the maximum possible speedup is a constant, the efficiency decreases as the number of processors increases.

The `amlaw` command displays the maximum theoretical speedup when you provide the number of CPUs and the percent parallelism. See the `amlaw(1)` man page for information about using the command.

2.3.2 Determining Efficiency

The efficiency of multitasked code is a comparison of the measured speedup with the maximum possible speedup. The overall efficiency of some code may be low because some parts of the code are inefficient; this will affect the overall efficiency.

The efficiency is defined in the following formula, where p is the number of processors, s is the measured speedup, and e is the efficiency:

$$e = \frac{s}{p}$$

(2.6)

If the program were completely parallel, and no overhead existed, the efficiency would be 1.0 (100%). As an example, suppose that the measured speedup running on four processors, compared to one processor, is $s = 3.48$. The efficiency is then defined as follows:

$$e = \frac{3.48}{4} = 0.87 = 87\% \quad (2.7)$$

2.3.3 Estimating Percentage of Parallelism

Another useful way to measure the efficiency of code is to measure the percentage of parallelism (the *parallel fraction* in Amdahl's Law). To estimate this figure, measure the actual time required to run a program on a single processor; also measure the actual time required to run the program in parallel on p processors. The ratio between the two times is the speedup, s . After s and p are known, the equation for Amdahl's Law can be rewritten for f as follows:

$$f = \frac{1 - \frac{1}{s}}{1 - \frac{1}{p}} \quad (2.8)$$

For example, suppose the measured speedup running on four processors compared to one processor is $s = 3.48$. The equivalent parallel fraction is the following:

$$f = \frac{1 - \frac{1}{3.48}}{1 - \frac{1}{4}} = 0.95 \quad (2.9)$$

This program thus achieves 95% parallelism, assuming the idealized model of Amdahl's Law.

2.4 Parallel Processing Environments

UNICOS systems have two basic parallel processing environments: the dedicated environment and the multiuser environment. This subsection provides overview information about these environments, as well as

information about the different environment variables you can set which can help you tune your system's performance.

Note: The settings for the different environment variables can significantly affect results and timings of code. See your system administrator for information about your site's use of these environment variables.

2.4.1 Environment Variables

You can use environment variables to predefine some characteristics of your shell; these environment variables are taken from the execution environment, and the values defined for the variables can affect your parallel processing environment.

The following environment variables allow you to tune the system for parallel processing without rebuilding libraries or other system software:

- **NCPUS:** Specifies the maximum number of CPUs available to work on a program. The default is 4 for machines with more than 4 CPUs. For machines with 4 or fewer CPUs, the default is the number of physical CPUs. The default of 4 CPUs was chosen for multitasked programs because experience has shown that in a nondedicated environment, this number often gives the best performance for a large number of programs.
- **MP_DEDICATED:** Specifies the type of machine environment. If set to 1, it specifies that you are the only user on the system. This allows the multitasking library to schedule differently to take advantage of the dedicated environment. If **MP_DEDICATED** is set to 0 or is not set at all, slave processors return to the operating system after waiting in user space.

If you set **MP_DEDICATED** to 1 in a nondedicated system throughput can be degraded.

- **MP_HOLDTIME:** Specifies the number of clock periods to hold a processor before giving up the CPU when no parallel work is available.

Both **NCPUS** and **MP_DEDICATED** are read by Scientific Library routines, which are greatly affected by the settings for these variables. These routines use different strategies depending on the settings. This can result in different performance numbers.

The following list contains suggested settings for these variables, depending on the system load in a non-dedicated environment:

- **Heavily loaded:** set **MP_DEDICATED** to 0 and **NCPUS** to 1.

- Normal load: set `MP_DEDICATED` to 0 and `NCPUS` to 0.25 of the maximum CPUs available.
- Lightly loaded: set `NCPUS` to the maximum number of CPUs available, essentially treating the machine as a dedicated machine.

2.4.2 The Dedicated Environment

A dedicated work environment is often used in situations in which wall-clock time is as important as CPU time/usage.

To select parallel processing in the dedicated environment, set `MP_DEDICATED` to 1, and change the `NCPUS` environment variable to equal the total number of available processors. This ensures access to a number of processors equal to the value in `NCPUS` for the entire time your code is executing.

A dedicated environment is often used in the following ways:

- When a job is the only one running on the system. In this case, set `NCPUS` equal to the total number of available physical processors.
- When a job requires a large amount of memory; therefore, few, if any, jobs can be active at the same time. In this case, set `NCPUS` equal to the total number of physical processors available if the job requires all of central memory, or to a smaller number if less memory is required.
- When priority schemes are used to favor specific jobs.

The following assumptions are made about dedicated environments:

- The number of processors specified in `NCPUS` are available to users at all times.
- Users want to minimize wall-clock time even if it increases cumulative CPU time.

2.4.3 The Multiuser Environment

In a multiuser work environment, CPU time is often of more importance than wall-clock time. To select parallel processing in a multiuser or production environment, ensure that the `MP_DEDICATED` environment variable is set to the default (0); if `MP_DEDICATED` is not set, a multiuser environment is assumed. `MP_DEDICATED` is actually a tuning parameter that the multiprocessing library uses. Its value determines the strategy that the library uses for attaching and detaching processors to a job.

In the multiuser environment, you cannot control the number of processors that are attached to a job except to specify a maximum number by using the `NCPUS` environment variable. The actual number of processors that will be used cannot be known in advance and may change as the job runs. When working in a lightly loaded, multiuser environment, set `NCPUS` equal to a small number of the available physical processors.

The following assumptions are made about the multiuser environment:

- Users do not know how many processors will be available to a job during run time, except that the number will be less than or equal to `NCPUS`.
- It is probable that fewer processors than the number specified in `NCPUS` will be attached to a job; therefore, you should use a partitioning strategy that gives the best **average** performance.

2.5 Measuring Scientific Library Performance

This subsection contains information that will help you measure the performance of Scientific Library routines in your code. See *Optimizing Application Code on UNICOS Systems*, for additional information about measuring code performance.

2.5.1 Simple Measurements

You can use several simple measurements to check the performance of your code.

A simple way to measure performance is to use dedicated resources to perform your single CPU measurements. This ensures that all of the CPU time accumulated to your job is directly related to the work being done by your code (rather than CPU time that your job accumulates as a result of any swapping that occurs).

However, you can still obtain good measures of performance on a nondedicated machine (especially one that is only slightly busy) by using the `SECOND(3F)` library routine. This routine obtains the elapsed user CPU time since the program started. The CPU time returned by `SECOND` does not include system CPU time.

You must use dedicated resources to obtain accurate wall-clock time measurements when using parallel processing. Running tests on even a minimally loaded machine will show variations in parallel processing

performance because of the unpredictability of obtaining CPU time from the operating system.

You can also use the `RTC(3F)` library routine to measure the performance of routines. `RTC` returns the value of the real-time clock register. This register contains the number of clock ticks. To convert this value to seconds multiply the `RTC` or `IRTC` value by the number of cycles per second. To obtain the cycle time of UNICOS machines, use the `target(1)` command or the `target` system call using the following C function:

```
#include <sys/target.h>
float GET_CTICKS()
/* Return the clock period in picoseconds */
{
    struct target data;
    target(MC_GET_TARGET, &data);
    return data.mc_clk;
}
```

Then, to convert `RTC` to seconds, use the following command:

```
walltime = RTC() * GET_CTICKS() * 1.0e-12
```

2.5.2 Determining Multitasking of Routines

If you are trying to measure performance as a problem size, you should realize that not all Scientific Library routines use parallel processing. For small problem sizes, a routine might not use multiprocessing at all, but it will make use of multiple CPUs for larger problem sizes.

In the following example, the `SGEMV(3S)` routine is used to compare single CPU performance to multitasked performance. The performance measurement is done in loop 1000. The measurement technique used results in inaccurate measurements for small problem sizes because `SGEMV` does not multitask for problem sizes below a certain threshold.

Prior to the measurement loop, the data is initialized in a loop that is parallelized by the call to the `CINIT` subroutine. Because of the cost of acquiring multiple CPUs to apply to a single process, the library routines check the `MP_HOLDTIME` environment variable (this specifies the time that a process should hold on to acquired CPUs after the parallel region that requested those CPUs has completed). Those acquired CPUs may still be attached when measuring the `SGEMV` routine; if the problem size is below the `SGEMV`

multitasking threshold, the CPU time accumulated by these CPUs is included in the timing of the work done by SGEMV.

This same false CPU usage can occur if the problem size in the loop is decremented from large problems to small problems. In this case, the test that measures performance in the problem area around the SGEMV multitasking threshold is affected by the CPUs that are not being applied to the problem but are just waiting for their MP_HOLDTIME to expire. The holding of CPUs between parallel regions benefits those larger problem size test cases because the multitasking library does not need to request additional CPUs for the parallel region if they are already attached and waiting. Setting MP_HOLDTIME to 0 causes idle processes to be immediately released and results in more accurate measurements of the work being done.

TESTCASE

```
#!/bin/sh
maxcpus='4 8'           # max NCPUS
xgf1=sgemv_1.xgf       # results: xgraph input file
xgf2=sgemv_2.xgf       # results: xgraph input file
#
cat > mmstr.f << EOF
PROGRAM MMSTR
PARAMETER (N =1024)
REAL*8 A(N,N),B(N,N),C(N,N)
REAL OPS, MFLOPS
REAL CYCLET, GET_CTICKS
INTEGER LOOPCNT
INTEGER T1, T2, IRTC
EXTERNAL SECOND, IRTC, GET_CTICKS

CYCLET = GET_CTICKS() * 1.0e-12

        NSTART = N
        NEND = 512
        NINK = -128
999     CONTINUE
C
DO 1000 NIND=NSTART,NEND,NINK
  IF (NINK.LT.-8) THEN
    NST1 = MIN(NIND+8,N)
    NEN1 = MAX(NIND-8,8)
  ELSE
    NST1 = NIND
```

```

      NEN1 = NIND
    ENDIF
    DO 1000 NPART=NST1,NEN1,-8
      LDA = NPART
      LDB = NPART
      LDC = NPART
      CALL EX312(A,LDA,NPART)
      CALL EX312(B,LDB,NPART)
    C
      CALL CINIT(C,LDC,NPART)
      LOOPCNT = NPART

    C
      T1 = IRTC()
      S1 = SECOND ()
      DO I=1,LOOPCNT
        CALL SGEMV
      ('N',NPART,NPART,1.,A,LDA,B(1,I),1,0.,C(1,I),1)
      END DO
      S2 = SECOND ()
      T2 = (IRTC() - T1)/NPART
    C
      OPS = 2*(NPART * NPART)
      WRITE(66,'(I5,E12.4)') NPART,S2-S1
      WRITE(77,'(I5,E12.4)') NPART, OPS/(T2 *CYCLET)
    C
1000  CONTINUE
    C
      IF (NINK.EQ.-128) THEN
        NINK = -64
        NSTART = NEND + NINK
        NEND = 256
        GOTO 999
      ELSEIF (NINK.EQ.-64) THEN
        NINK = -32
        NSTART = NEND + NINK
        NEND = 128
        GOTO 999
      ELSEIF (NINK.EQ.-32) THEN
        NINK = -8
        NSTART = NEND + 2 * NINK
        NEND = 8
        GOTO 999
    
```

```

        ENDIF
C
64      FORMAT(3X,I4,2X,'I',5(1X,E12.4,3X,'I'))
2000 CONTINUE
      END
C
      SUBROUTINE EX312(A,LDA,NPART)
      INTEGER NPART,I,J,NP1I,LDA
      REAL*8  A(LDA,NPART)
C
      DO 10 J=1,NPART
        DO 10 I=J,NPART
          NP1I=NPART+1-I
          A(I,J) = DBLE(NP1I)
          A(J,I) = DBLE(NP1I)
10     CONTINUE
      RETURN
      END
C
      SUBROUTINE CINIT(C,LDC,NPART)
      INTEGER NPART,I,J,LDC
      REAL*8  C(LDC,NPART)
C
      DO 10 I=1,NPART
        DO 10 J=1,NPART
          C(I,J) = 0.0
10     CONTINUE
      RETURN
C
      END
EOF
#
# Get clock ticks to compute wall clock seconds
#
cat >get_cticks.c << EOF
#include <sys/target.h>
float GET_CTICKS()
/* Return the clock period in picoseconds */
{
  struct target data;
  target(MC_GET_TARGET, &data);
  return data.mc_clk;
}

```

```
EOF
#
cc -c -g get_cticks.c

cf77 -Zp get_cticks.o mmstr.f

#   get sgemv version

# initialize xgraph file
echo 'Device: Postscript' > $xgf1
echo "TitleText: `date +%d-%h-%y %H:%M` `uname -snmr`SGEMV" >> $xgf1
echo 'XUnitText: matrix dim N' >> $xgf1
echo 'YUnitText: Overhead [%]' >> $xgf1
echo 'YLowLimit: 0' >> $xgf1

# initialize xgraph file
echo 'Device: Postscript' > $xgf2echo "TitleText: `date +%d-%h-%y
%H:%M` `uname -snmr` SGEMV" >> $xgf2
echo 'XUnitText: matrix dim N' >> $xgf2
echo 'YUnitText: MFLOPS' >> $xgf2

NCPUS=1 ./a.out
mv fort.66 x.1
#
# Eliminate Residual CPU usage by setting
# HOLDTIME to 0
#
MP_HOLDTIME=0
export MP_HOLDTIME
for n in $maxcpus
do
    ja
    NCPUS=$n ./a.out
    ja -ct
    echo '\n' '$n CPUs' >> $xgf1
    echo '\n' '$n CPUs' >> $xgf2
    paste x.1 fort.66 |
    awk '{ ov = ($4 - $2)*100/$2; print $1, ov }' >> $xgf1
    cat fort.77 >> $xgf2
done
#
rm -f x.1 fort.66
```

```
test $DISPLAY && xgraph $xgf1 &  
test $DISPLAY && xgraph $xgf2 &
```

2.6 Parallel Processing Strategies

This subsection describes the parallel processing strategies that the Scientific Library routines use in dedicated and multiuser environments. For small or very large problems, the strategies in both environments are the same. For medium-sized problems, however, the strategies differ greatly. The Scientific Library automatically implements these strategies without the need for user intervention.

2.6.1 Problem Sizes

The strategies presented in the following subsections apply to fine-grain data parallel problems. On UNICOS systems, this type of parallelism is effectively used by vectorization on one vector processor and parallelization across several vector processors.

The following is a classification of problem sizes:

- *vector problem* (VP) – Problems large enough for vector processing, but too small for parallel processing.
- *small parallel/vector problem* (SPVP) – Problems large enough for vector and parallel processing, but for which parallel processing degrades vector performance.
- *Medium parallel/vector problem* (MPVP) – Problems large enough for optimal vector and parallel processing, but for which load balancing can be a significant problem if a processor is lost.
- *large parallel/vector problem* (LPVP) – Problems large enough for optimal vector and parallel processing for which load balancing is not a significant problem. In this case, enough work exists to partition the problem into many subproblems so that the effect of losing a processor is minimized.

The boundary between the VP and SPVP classes is independent of the number of processors. However, the other boundaries between classes depend on the number of processors you try to use. A problem size of class LPVP on 2 processors could be of class MPVP on 4 processors and class SPVP on 8 or 16 processors.

2.6.2 Strategies for a Dedicated Environment

In a dedicated environment, the strategies for parallel processing are well-understood, and for every problem size at least one unambiguous optimal partition usually exists. The biggest challenge in this environment is developing a low-cost, straightforward partitioning routine.

Problem sizes of VP class should execute on one processor. To preserve single-processor performance for these small problems, first do a quick, inexpensive check to see if the problem to be solved is small. If so, the problem will be solved on one processor regardless of the number of processors requested.

For SPVP and MPVP problems, partition the problem into exactly p subproblems of approximately equal size. The size defined in NCPUS will define p . Because it is assumed that exactly p processors will be available, partitioning the problem into fewer than p subproblems ensures an increase in elapsed time, t^e . Partitioning the problem into more than p subproblems degrades vector performance (increasing t^v) for SPVP problems and creates load imbalances for MPVP problems, both causing an increase in t^e .

You cannot partition an LPVP problem into kp subproblems, where $k = 2, 3, \dots$ of equal size in which the subproblems have optimal single-processor (vector) performance. For problems of this size, use a self-scheduling or guided-self-scheduling algorithm to process the kp subproblems and to reduce the effect of losing processors. The possibility of losing a processor in a dedicated environment is small; however, if one is lost, it can dramatically impact performance (doubling t^e) for large problems partitioned into only p subproblems.

2.6.3 Strategies for a Multiuser Environment

In a *multiuser environment*, you are not guaranteed a fixed number of processors. Processors often attach to and detach from jobs unpredictably during execution. Therefore, no optimal time-independent partitioning strategy exists for all problem sizes in this environment. Also, the operating system does not give timely information on how many processors are attached, or may be attached, to your job.

The number of processors that will be attached to your job when entering a parallel region is a function of the number of processors requested, the system load, the time since the last parallel region was encountered, and the settings of a variety of multiprocessing tuning parameters.

Despite this unpredictability, some strategies are available. First, the strategies for VP and LPVP problems are the same as in the dedicated environment. You should execute VP problems on one processor regardless of the number of processors requested, because using one processor is the fastest way to solve the problem.

Secondly, by definition, LPVP problems have optimal vector performance regardless of the number of processors actually attached so that CPU time efficiency e^u should always be close to optimum. LPVP problems are also partitioned into kp subproblems, $k = 2, 3, \dots$, where p is the requested number of processors. If p' processors ($p'(<)p$) are attached, the ratio (kp/p') is usually large enough that the remainder $kp \bmod p'$ does not create significant load imbalance.

It can be difficult to develop strategies for SPVP and MPVP problems. As in a dedicated environment, you must partition the problem into exactly p subproblems and execute it on p processors. Because the number of processors attached to a job is unknown, however, the correct value of p is not immediately obvious and, unlike the dedicated environment, you should not always set p to NCPUS. Choosing p incorrectly could lead to a partitioning strategy that is too aggressive (increasing user and elapsed times) or too conservative (reducing parallelism).

To determine the “best” value of $p \leq \text{NCPUS}$ (p specifies the number of subproblems and the number of processors to use) for SPVP and MPVP problems, first consider SPVP problems. Let $m = n = 100$ and consider the performance statistics for SGER(3S) in Table 1. The Scientific Library subroutine SGER is part of Level 2 BLAS (Basic Linear Algebra Subprograms). SGER performs the following rank-one update of a general matrix: $A \leftarrow A + axy^T$. A , x , and y are real-valued and of dimension $m \times n$, $m \times 1$, and $n \times 1$, respectively, and a is a scalar.

To process this equation in parallel, partition A horizontally and/or vertically into submatrices A_{ij} with dimensions $m_i \times n_j$, $i = 1, \dots, n_h$; $j = 1, \dots, n_v$; n_h, n_v are the number of horizontal and vertical partitions, respectively. x and y are partitioned appropriately. This is an SPVP problem because any partition of the original problems produces subproblems that have poorer vector performance.

Table 1. Relative cost for SGER

Number of processors required	Number of processors acquired							
	1	2	3	4	5	6	7	8
1	72.72 (1.00)							
2	84.03 (1.33)	49.79 (0.93)						
3	97.57 (1.80)	70.16 (1.85)	42.63 (1.03)					
4	93.03 (1.63)	54.94 (1.13)	53.21 (1.60)	35.29 (0.94)				
5	126.39 (3.02)	81.47 (2.50)	60.66 (2.08)	58.25 (2.54)	38.23 (1.38)			
6	110.76 (2.32)	64.15 (1.55)	47.88 (1.29)	47.84 (1.70)	45.54 (1.95)	33.56 (1.27)		
7	161.92 (4.95)	98.10 (3.63)	76.81 (3.34)	58.20 (2.54)	57.75 (3.13)	55.70 (3.48)	37.39 (1.84)	
8	127.47 (3.06)	73.02 (2.01)	57.95 (1.90)	45.12 (1.50)	44.01 (1.82)	43.58 (2.15)	41.36 (2.24)	31.72 (1.52)

LAPACK is a public domain library of subroutines for solving dense linear algebra problems, including systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems. It has been designed for efficiency on high-performance computers.

LAPACK is intended to be the successor to LINPACK and EISPACK. It extends the functionality of these packages by including equilibration, iterative refinement, error bounds, and driver routines for linear systems, routines for computing and reordering the Schur factorization, and condition estimation routines for eigenvalue problems.

LAPACK improves on the accuracy of the standard algorithms in EISPACK by including high-accuracy algorithms for finding singular values of bidiagonal matrices and for finding eigenvalues of tridiagonal matrices arising from symmetric eigenvalue problems. Performance issues are addressed by implementing the most computationally-intensive algorithms using the Level 2 and 3 Basic Linear Algebra Subprograms (BLAS).

The *LAPACK User's Guide* documents the original Fortran programs used in this section. Additional details are found in *Accuracy and Stability of Numeric Algorithms*.

Follow-on projects to LAPACK are under development, and new software is being added to the public domain version. In this document, *LAPACK* refers to the collection of Fortran subroutines made available with the 2-29-92 public release, also called LAPACK 1.0, which are documented in the 1992 edition of the *LAPACK Users' Guide*.

This section supplements the information on the LAPACK man pages and in the *LAPACK Users' Guide*. It discusses in more detail how the LAPACK computational routines are used, and uses examples to illustrate the results.

3.1 LAPACK in the Scientific Library

The Scientific Library includes a subset of LAPACK. See the `INTRO_LAPACK(3S)` man page for details about the subset that is available in the current release of the Scientific Library.

Online man pages are available for individual LAPACK subroutines. For example, to view a description of the calling sequence for the subroutine to perform the LU factorization of a real matrix, see the `SGETRF(3S)` man page.

LAPACK routines that operate on 64-bit real and complex data are included in the Scientific Library. The subroutine names in the Scientific Library are the names of the single-precision routines in the standard interfaces; the first letter of the routine name is `S` for real data routines and `C` for complex data routines. When porting applications from other systems that call LAPACK routines in double precision, change the names of the calls (for example, change `CALL DGETRF()` to `CALL SGETRF()`). You also can use a compiler option to disable double-precision constructs in the code and a loader directive to map double-precision names to single-precision names.

Several enhancements improve the performance of the LAPACK routines on UNICOS systems. For example, the solver routines are redesigned for better performance for one or a small number of right-hand sides and to use parallelism when the number of right-hand sides is large.

Tuning parameters for the block algorithms provided in the Scientific Library are set within the `ILAENV` LAPACK routine. `ILAENV` is an integer function subprogram that accepts information about the problem type and problem dimensions and returns a single integer parameter such as the optimal block size, the minimum block size for which a block algorithm should be used, or the crossover point (the problem size at which it becomes more efficient to switch to an unblocked algorithm). Setting tuning parameters occurs without user intervention, but users can call `ILAENV` directly to check the values to be used.

3.2 Types of Problems Solved by LAPACK

This subsection discusses the LAPACK routines for solving the following two basic problems:

- Computing the unique solution to a linear system $AX = B$, where the coefficient matrix A is dense, banded, triangular, or tridiagonal, and the matrix B may contain one or more right-hand sides.
- Computing a least squares solution to an overdetermined system $AX = B$, where A is $m \times n$ with $m \geq n$, or a minimum norm solution to an underdetermined system $AX = B$, where A is $m \times n$ with $m < n$.

See Section 3.3, page 27 and Section 3.5, page 45 for a discussion of the software used to solve these problems. The orthogonal transformation routines described

in Section 3.5, page 45 also have application in eigenvalue and singular value computations.

There are two classes of LAPACK routines: LAPACK *driver routines* solve a complete problem; LAPACK *computational routines* perform one step of the computation. The driver routines generally call the computational routines to do their work, and offer a more convenient interface; therefore, LAPACK users should use the LAPACK driver routines for solving systems of linear equations. The `INTRO_LAPACK(3S)` man page and the man pages for the individual subroutines describe the functions performed by each of the driver routines.

3.3 Solving Linear Systems

A linear system is a set of equations, each of which is linear in its unknowns (for example $2x + y = 4$ or $-x + y = 1$).

You could combine these equations into a linear system, which is written in matrix form, as follows:

$$\begin{bmatrix} 2 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \end{bmatrix}$$

(3.1)

The solution to this system of equations is the set of vectors $[x, y]^T$ that satisfy both equations. The physical interpretation of this system of equations is that it represents two lines in the (x, y) plane, which may intersect in one point, no points (if they are parallel), or an infinite number of points (if the two equations are multiples of each other).

To solve the system, eliminate variables from each successive equation until the system is simple enough to solve directly. To form the simpler system, add $\frac{1}{2}$ times the first equation to the second equation, as follows:

$$\begin{bmatrix} 2 & 1.0 \\ 0 & 1.5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}$$

(3.2)

The second equation can then be solved to get $y = 2$, and this result is substituted into the first equation to get $x = 1$. This process amounts to a factorization of the coefficient matrix:

$$\begin{bmatrix} 2 & 1 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -0.5 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 0 & 1.5 \end{bmatrix} \quad (3.3)$$

followed by two triangular system solutions:

$$\begin{bmatrix} 1 & 0 \\ -0.5 & 1 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \end{bmatrix} \quad (3.4)$$

whose solution is $z_1 = 4, z_2 = 3$ and

$$\begin{bmatrix} 2 & 1.0 \\ 0 & 1.5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \end{bmatrix} \quad (3.5)$$

whose solution is $x = 1, y = 2$. Now consider what happens if the two equations represent parallel lines, with no points in common, as in this example:

$$\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 0 \end{bmatrix} \quad (3.6)$$

The factorization step takes the following form:

$$\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 0 & 0 \end{bmatrix} \quad (3.7)$$

The second factor, which should be upper triangular, has a 0 on the diagonal. This indicates that the triangular system that involves the second factor cannot be solved by back-substitution, and the system does not have a unique solution. The factorization routines in LAPACK detect zeros on the diagonals of the triangular factors and return this information in an error code.

The LAPACK routines for solving linear systems assume the system is already in a matrix form. The data type (real or complex), characteristics of the coefficient matrix (general or symmetric, and positive definite or indefinite if

symmetric), and the storage format of the matrix (dense, band, or packed), determine the routines that should be used.

3.3.1 Factoring a Matrix

Most of the techniques in LAPACK are based on a matrix factorization as the first step. There are two main types of factorization forms:

- explicit: The actual factors are returned. For example, the Cholesky factorization routine `SPOTRF(3S)`, with `UPLO = 'L'`, returns a matrix L such that $A = LL^T$.
- factored form: The factorization is returned as a product of permutation matrices and triangular matrices that are low-rank modifications of the identity. For example, the diagonal pivoting factorization routine `SSYTRF(3S)`, with `UPLO = 'L'`, computes a factorization of the following form:

$$A = (P_1 L_1 P_2 L_2 \dots P_n L_n) D (P_1 L_1 P_2 L_2 \dots P_n L_n)^T \quad (3.8)$$

where each P_i is a rank-1 permutation, each L_i is a rank-1 or rank-2 modification of the identity, and D is a diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks.

Generally, users do not have to know the details of how the factorization is stored, because other LAPACK routines manipulate the factored form.

Regardless of the form of the factorization, it reduces the solution phase to one that requires only permutations and the solution of triangular systems. For example, the LU factorization of a general matrix, $A = PLU$, is used to solve for X in the system of equations $AX = B$ by successively applying the inverses of P , L , and U to the right-hand side:

1. $X \leftarrow PB$
2. $X \leftarrow L^{-1}X$
3. $X \leftarrow U^{-1}X$

In the last two steps, the inverse of the triangular factors is not computed, but triangular systems of the form $LY = Z$ and $UX = Y$ are solved instead.

The following table lists the factorization forms for each of the factorization routines for real matrices. The factorization forms differ for `SGETRF` and

SGBTRF, even though both compute an *LU* factorization with partial pivoting. You can also obtain the same factorizations through the LAPACK driver routines (for instance, *SGESV* or *SGESVX*).

Table 2. Factorization forms

Name	Form	Equation	Notes
<i>SGBTRF</i>	Factored form	$A = LU$	<i>L</i> is a product of permutations and unit lower triangular matrices L_i ; L_i differs from the identity matrix only in column i .
<i>SGTTRF</i>	Factored form	$A = LU$	
<i>SGETRF</i>	Explicit	$A = PLU$	<i>L</i> (or <i>U</i>) is a product of permutations and block unit lower (upper) triangular matrices L_i (U_i); L_i (U_i) differs from the identity matrix only in the one or two columns that correspond to the 1-by-1 or 2-by-2 diagonal block D_i .
<i>SPBTRF</i>	Explicit	$A = LL^T$ or $A = U^T U$	
<i>SPOTRF</i>	Explicit	$A = LL^T$ or $A = U^T U$	
<i>SPPTRF</i>	Explicit	$A = LL^T$ or $A = U^T U$	
<i>SPTTRF</i>	Explicit	$A = LDL^T$ or $A = U^T D U$	
<i>SSPTRF</i>	Factored form	$A = LDL^T$ or $A = U D U^T$	
<i>SSYTRF</i>	Factored form	$A = LDL^T$ or $A = U D U^T$	

Example 1: LU factorization

The *SGETRF* subroutine performs an LU factorization with partial pivoting ($A = PLU$) as the first step in solving a general system of linear equations $AX = B$. If *SGETRF* is called with the following:

$$A = \begin{bmatrix} 4. & 9. & 2. \\ 3. & 5. & 7. \\ 8. & 1. & 6. \end{bmatrix}$$

(3.9)

details of the factorization are returned, as follows:

$$A = \begin{bmatrix} 8. & 1. & 6. \\ 0.5 & 8.5 & -1.0 \\ 0.375 & 0.5441 & 5.294 \end{bmatrix}$$

$$IPIV = [3, 3, 3]$$

(3.10)

Matrices L and U are given explicitly in the lower and upper triangles, respectively, of A :

$$L = \begin{bmatrix} 1. & & \\ 0.5 & 1. & \\ 0.375 & 0.5441 & 1. \end{bmatrix}, U = \begin{bmatrix} 8. & 1. & 6. \\ & 8.5 & -1.0 \\ & & 5.4294 \end{bmatrix}$$

(3.11)

The $IPIV$ vector specifies the row interchanges that were performed. $IPIV(1) = 3$ implies that the first and third rows were interchanged when factoring the first column; $IPIV(2) = 3$ implies that the second and third rows were interchanged when factoring the second column. In this case, $IPIV(3)$ must be 3 because there are only three rows. Thus, the permutation matrix is the following:

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

(3.12)

Generally, the pivot information is used directly from $IPIV$ without constructing matrix P .

Example 2: Symmetric indefinite matrix factorization

`SSYTRF` factors a symmetric indefinite matrix A into one of the forms $A = LDL^T$ or $A = UDU^T$, where L and U are lower triangular and upper triangular matrices, respectively, in factored form, and D is a diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. To illustrate this factorization, choose a symmetric matrix that requires both 1-by-1 and 2-by-2 pivots:

$$A = \begin{bmatrix} 9. & & & \\ 5. & 7. & & \\ -16. & 12. & 3. & \\ 4. & -2. & 10. & 8. \end{bmatrix} \tag{3.13}$$

Only the lower triangle of A is specified because the matrix is symmetric, but you could have specified the upper triangle instead. The output from `SSYTRF` is the following:

$$A = \begin{bmatrix} 9. & & & \\ -16. & 3. & & \\ -0.9039 & -0.8210 & 21.37 & \\ -0.7511 & -0.6725 & 0.4597 & 13.21 \end{bmatrix}$$

$$IPIV = [-3, -3, 3, 4]$$

(3.14)

The signs of the indices in the `IPIV` vector indicate that a 2-by-2 pivot block was used for the first two columns, and 1-by-1 pivots were used for the third and fourth columns. Therefore, D must be the following:

$$D = \begin{bmatrix} 9. & & & \\ -16. & 3. & & \\ & & 21.37 & \\ & & & 13.21 \end{bmatrix} \tag{3.15}$$

Matrix L is supplied in factored form as $L = P_1 L_1 P_2 L_2$, where the parts of each L_i that differ from the identity are stored in A below their corresponding blocks D_i :

$$P_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, L_1 = \begin{bmatrix} 1. & & & \\ 0. & 1. & & \\ -0.9039 & -0.8210 & 1 & \\ -0.7511 & -0.6725 & 0.0 & 1.0 \end{bmatrix} \tag{3.16}$$

$$P_2 = I, L_2 = \begin{bmatrix} 1 & & & \\ 0 & 1 & & \\ 0 & 0 & 1.0 & \\ 0 & 0 & 0.4597 & 1 \end{bmatrix}$$

(3.17)

3.3.2 Error Codes

The LAPACK routines always checks the arguments on entry for incorrect values. If an illegal argument value is detected, the error-handling subroutine XERBLA is called. XERBLA prints a message similar to the following to standard error, and then it aborts:

```
** On entry to SGETRF parameter number 4 had an illegal value
```

All other errors in the LAPACK routines are described by error codes returned in *info*, the last argument. The values returned in *info* are routine-specific, except for *info* = 0, which always means that the requested operation completed successfully.

For example, an error code of *info* > 0 from SGETRF means that one of the diagonal elements of the factor *U* from the factorization $A = PLU$ is exactly 0. This indicates that one of the rows of *A* is a linear combination of the other rows, and the linear system does not have a unique solution.

Example 3: Error conditions

If SGETRF is given the matrix

$$A = \begin{bmatrix} 1. & 4. & 7. \\ 2. & 5. & 8. \\ 3. & 6. & 9. \end{bmatrix}$$

(3.18)

it returns

$$A = \begin{bmatrix} 3. & 6. & 9. \\ 0.3333 & 2. & 4. \\ 0.6667 & 0.5 & 0 \end{bmatrix}$$

$$IPIV = [3, 3, 3]$$

(3.19)

which corresponds to the factorization

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}, L = \begin{bmatrix} 1.0 & & \\ 0.3333 & 1.0 & \\ 0.6667 & 0.5 & 1. \end{bmatrix}, U = \begin{bmatrix} 3. & 6. & 9. \\ & 2. & 4. \\ & & 0. \end{bmatrix}$$

(3.20)

On exit from `SGETRF`, `info = 3`, indicating that $U(3,3)$ is exactly 0. This is not an error condition for the factorization because the factors that were computed satisfy $A = PLU$, but the factorization cannot be used to solve the system.

3.4 Solving from the Factored Form

In LAPACK, the solution step is generally separated from the factorization. This allows the matrix factorization to be reused if the same coefficient matrix appears in several systems of equations with different right-hand sides. If the number of right-hand sides is also large, it is often more efficient to separate the solve from the factorization. The typical usage is found in the driver routine `SGESV`, which solves a general system of equations $AX = B$ by using two subroutine calls, the first to factor the matrix A and the second to solve the system, using the factored form:

```
CALL SGETRF( N, N, A, LDA, IPIV, INFO )
IF( INFO.EQ.0 ) THEN
CALL SGETRS( 'No transpose', N, NRHS, A, LDA,
$           IPIV, B, LDB, INFO )
END IF
```

As shown, you should always check the return code from the factorization to see whether it completed successfully and did not produce any singular factors. To obtain further information about proceeding with the solve, estimate the condition number (see Section 3.4.1, page 36 for details).

Because most of the LAPACK driver routines do their work in the LAPACK computational routines, a call to a driver routine gives the same performance as separate calls to the computational routines. The exceptions are the simple driver routines used for solving tridiagonal systems: `SGTSV`, `SPTSV`, `CGTSV`, and `CPTSV`. These routines compute the solution while performing the factorization for certain numbers of right-hand sides. Because the amount of

work in each loop is small, some reloading of constants and loop overhead is saved by combining the factorization with part of the solve.

Table 3, page 35 shows the times (in microseconds) on one processor of a Cray C90 system for solving a tridiagonal system with one right-hand side, using the LAPACK factor and solve routines separately, compared to the times for the simple driver routines. For comparison, times also are shown for the Scientific Library versions of the equivalent LINPACK routines `SGTSL`, `SPTSL`, `CGTSL`, and `CPTSL`. The LAPACK driver routines are typically about 20% faster than the separate computational routines for one right-hand side, and are faster than LINPACK in all cases except `SPTSL`, which outperforms `SPTSV` by not checking for zeros on the diagonal during the factorization.

This table also shows times for the assembler-coded Scientific Library routine `SDTSOL(3S)` for real general tridiagonal matrices, which does not do pivoting and, like LINPACK, accepts only one right-hand side. This subroutine is much faster than LAPACK on those problems for which it can be used.

Table 3. Solves times: LAPACK and solver routines

Method	Values of n			
	25	50	100	200
<code>SGTTRF + SGTTRS</code>	32.	55.	98.	186.
<code>SGTSV</code>	21.	37.	69.	134.
<code>SGTSL</code>	22.	39.	75.	152.
<code>SPTTRF + SPTTRS</code>	18.	27.	45.	80.
<code>SPTSV</code>	13.	21.	36.	68.
<code>SPTSL</code>	10.	16.	29.	55.
<code>CGTTRF + CGTTRS</code>	54.	98.	188.	365.
<code>CGTSV</code>	41.	78.	152.	300.
<code>CGTSL</code>	51.	103.	197.	403.

Method	Values of n			
	25	50	100	200
CPTTRF + CPTTRS	27.	43.	79.	149.
CPTSV	21.	35.	63.	118.
CPTSL	28.	55.	107.	212.
SDTSOL	7.	10.	14.	18.

3.4.1 Condition Estimation

A return code of *info* = 0 from a factorization routine indicates that the triangular factors have nonzero diagonals. The linear system still may be too ill-conditioned to give a meaningful solution.

One indicator that you can examine before computing the solution is the reciprocal condition number, *RCOND*. The condition number, defined as $\kappa(A) = \|A\| \|A^{-1}\|$, tells how much the relative errors in A and b are magnified in the solution x . *SGECON* and the other condition estimation routines compute $\text{RCOND} = 1/\kappa(A)$ by using the exact 1-norm or infinity-norm of A and an estimate for the norm of A^{-1} , because computing the inverse explicitly would be very expensive, and the inverse may not even exist. By convention, if A^{-1} does not exist, $\kappa(A) = \infty$ and *RCOND* should be computed as 0 or a small number on the order of ϵ , the machine epsilon.

If the condition number is large (that is, if *RCOND* is small), small errors in A and b may lead to large errors in the solution x . The rule of thumb is that the solution loses one digit of accuracy for every power of 10 in the condition number, assuming that the elements of A all have about the same magnitude. For example, a condition number of 100 (*RCOND* = 0.01) implies that the last 2 digits are inaccurate; a condition number of $1/\epsilon$ (*RCOND* < ϵ , the machine epsilon is approximately 1.4×10^{-14} on UNICOS systems) implies that all of the digits have been lost. This value for the machine epsilon assumes a model of rounded floating-point arithmetic with base $\beta = 2$ and $t = 47$ mantissa digits, and $\epsilon = \beta^{(1-t)}$.

The expert driver routine *SGESVX* uses this rule of thumb to decide whether the solution and error bounds should be computed. If *RCOND* is less than the

machine epsilon, `SGESVX` returns `info = N+1`, indicating that the matrix A is singular to working precision, and it does not compute the solution.

Example 4: Roundoff errors

The matrix

$$A = \begin{bmatrix} 1. & 2. & 3. \\ 4. & 5. & 6. \\ 7. & 8. & 9. \end{bmatrix} \quad (3.21)$$

is singular in exact arithmetic, but on UNICOS systems, `SGETRF` returns

$$A = \begin{bmatrix} 7. & 8. & 9. \\ 0.1429 & 0.8571 & 1.714 \\ 0.5714 & 0.5 & -2.478 \times 10^{-14} \end{bmatrix} \quad (3.22)$$

where `IPIV=[3,3,3]` and `info = 0`. In exact arithmetic, $A(3,3)$ would have been 0, but roundoff error has made this entry -2.487×10^{-14} instead. The reciprocal condition number computed by `SGECON` is 3.45×10^{-16} , which is less than the machine epsilon of 1.42×10^{-14} . Therefore, `SGESVX` returns `info= 4` and does not try to solve any systems with this A .

3.4.2 Use in Error Bounds

You can use the condition number to compute a simple bound on the relative error in the computed solution to a system of equations $Ax = b$ (see, *Introduction to Matrix Computations*, by Stewart). If x is the exact solution and \hat{x} is the computed solution, the residual $r = b - Ax = A(x - \hat{x})$. If A is nonsingular:

$$x - \hat{x} = A^{-1}r \quad (3.23)$$

and

$$\|x - \hat{x}\| \leq \|A^{-1}\| \|r\| \leq \|A^{-1}\| \frac{\|A\| \|x\|}{\|b\|} \|r\|$$

(3.24)

because $\|b\| \leq \|A\| \|x\|$. This gives the bound

$$\frac{\|x - \hat{x}\|}{\|x\|} \leq \kappa(A) \frac{\|r\|}{\|b\|}$$

(3.25)

For a description of a more precise bound based on a component-wise error analysis, see Section 3.4.4.1, page 43.

Another application of the condition number is to consider the computed solution $\hat{x} = x + \Delta x$ to be the exact solution of a slightly perturbed linear system:

$$(A + \Delta A)(x + \Delta x) = (b + \Delta b)$$

(3.26)

where ΔA is small in norm with respect to A , and Δb is small in norm with respect to b . For Gaussian elimination with partial pivoting, it has been shown that $\|\Delta A\| \leq \omega \|A\|$ and $\|\Delta b\| \leq \omega \|b\|$, where ω is the product of ε and a slowly growing function of n (see, *The Algebraic Eigenvalue Problem*, by Wilkinson). Proving that an algorithm has this property is the stuff of backward error analysis, and it ensures that, if the problem is well-conditioned, the computed solution is near the exact solution of the original problem. Because $Ax = b$ (Equation 3.25) simplifies to

$$A\Delta x = \Delta b - \Delta A(x + \Delta x)$$

(3.27)

Assuming A is nonsingular,

$$\Delta x = A^{-1}(\Delta b - \Delta A(x + \Delta x))$$

(3.28)

Taking norms and dividing by $\|x\|$,

$$\frac{\|\Delta x\|}{\|x\|} \leq \|A^{-1}\| \left(\frac{\|\Delta b\|}{\|x\|} + \|\Delta A\| + \frac{\|\Delta A\| \|\Delta x\|}{\|x\|} \right)$$

(3.29)

Using the inequality $\|b\| \leq \|A\| \|x\|$,

$$\frac{\|\Delta x\|}{\|x\|} \leq \|A\| \|A^{-1}\| \left(\frac{\|\Delta b\|}{\|b\|} + \frac{\|\Delta A\|}{\|A\|} + \frac{\|\Delta A\| \|\Delta x\|}{\|A\| \|x\|} \right) \quad (3.30)$$

and substituting $\kappa(A) = \|A\| \|A^{-1}\|$,

$$\frac{\|\Delta x\|}{\|x\|} \leq \kappa(A) \left(\frac{\frac{\|\Delta b\|}{\|b\|} + \frac{\|\Delta A\|}{\|A\|}}{1 - \kappa(A) \frac{\|\Delta A\|}{\|A\|}} \right) \quad (3.31)$$

provided $\kappa(A) \|\Delta A\| / \|A\| < 1$. In terms of the relative backward error ω ,

$$\frac{\|\Delta x\|}{\|x\|} \leq \frac{2\omega \kappa(A)}{1 - \omega \kappa(A)} \quad (3.32)$$

In Section 3.4.4.1, page 43, the backward error is defined slightly differently to obtain a component-wise error bound.

3.4.3 Equilibration

The condition number defined in the last section is sensitive to the scaling of A . For example, the matrix

$$A = \begin{bmatrix} 1. & 1 \\ 0 & 1. \times 10^{16} \end{bmatrix} \quad (3.33)$$

has as its inverse

$$A^{-1} = \begin{bmatrix} 1. & -1 \times 10^{-16} \\ 0 & 1. \times 10^{-16} \end{bmatrix} \quad (3.34)$$

and so $RCOND=1/(\|A\| \|A^{-1}\|) = 1. \times 10^{-16}$. Because this value of RCOND is less than the machine epsilon on UNICOS systems, SGEVX (with FACT = 'N') does not try to solve a system with this matrix. However, A has elements that vary widely in magnitude, so the bounds on the relative error in the solution may be pessimistic. For example, if the right-hand side in $Ax = b$ is the following:

$$b = \begin{bmatrix} 2.0 \\ 1.0 \times 10^{16} \end{bmatrix} \tag{3.35}$$

SGETRF followed by SGETRS produces the exact answer, $x = [1., 1.]^T$, on UNICOS systems.

You can improve the condition of this example by a simple row scaling. Scaling a problem before computing its solution is known as *equilibration*, and it is an option to some of the expert driver routines (those for general or positive definite matrices). Enabling equilibration does not necessarily mean it will be done; the driver routine will choose to do row scaling, column scaling, both row and column scaling, or no scaling, depending on the input data. The usage of this option is as follows:

```
CALL SGEVX('E', 'N', N, NRHS, A, LDA, AF,
$      LDAF, IPIV, EQUED, R, C, B, LDB, X, LDX,
$      RCOND, FERR, BERR, WORK, IWORK, INFO)
```

The 'E' in the first argument enables equilibration. For this example, EQUED = 'R' on return, indicating that only row scaling was done, and the vector R contains the scaling constants:

$$R = \begin{bmatrix} 1.0 \\ 1.0 \times 10^{-16} \end{bmatrix} \tag{3.36}$$

The form of the equilibrated problem is:

$$(diag(R) A diag(C)) (diag(C)^{-1} X) = diag(R) B \tag{3.37}$$

or $A_E X_E = B_E$, where A_E is returned in A:

$$A \leftarrow \begin{bmatrix} 1. & 0. \\ 0. & 1.0 \times 10^{-16} \end{bmatrix} \begin{bmatrix} 1. & .1 \\ 0. & 1.0 \times 10^{-16} \end{bmatrix} = \begin{bmatrix} 1. & 1. \\ 0. & 1. \end{bmatrix} \quad (3.38)$$

and B_E is returned in B :

$$B \leftarrow \begin{bmatrix} 1. & 0. \\ 0. & 1.0 \times 10^{-16} \end{bmatrix} \begin{bmatrix} 2. \\ 1.0 \times 10^{16} \end{bmatrix} = \begin{bmatrix} 2. \\ 1. \end{bmatrix} \quad (3.39)$$

The factored form, AF, returns the same matrix as A in this example, because A is upper triangular, and $RCOND = 0.3$, which is the estimated reciprocal condition number for the equilibrated matrix. The only output quantity that pertains to the original problem before equilibration is the solution matrix X . In this example, X is also the solution to the equilibrated problem because no column scaling was done, but if $EQUED$ had returned 'C' or 'B' and the solution to the equilibrated system were desired, it could be computed from $X_E = \text{diag}(C)^{-1} X$.

3.4.4 Iterative Refinement

Iterative refinement in LAPACK uses all same-precision arithmetic, a recent innovation, because it was long believed that a successful algorithm would require the residual to be computed in double precision. The following example illustrates the results of iterative refinement; Section 3.4.4.1, page 43, discusses the error bounds computed in the course of the algorithm.

One possible use of iterative refinement is to smooth out numerical differences between floating-point number representations. For example, a result computed on a UNICOS system, which has about 13 digits of accuracy, may be improved on a UNICOS/mk or IEEE system, which has about 15 digits of accuracy, through the $O(n^2)$ process of iterative refinement, instead of the $O(n^3)$ process of recomputing the solution.

Example 5: Hilbert matrix

The classic example of an ill-conditioned matrix is the Hilbert matrix, defined by $A_{i,j} = 1/(i+j-1)$. For example, the 5-by-5 Hilbert matrix is

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{bmatrix}$$

(3.40)

which has a condition number of 4.77×10^5 . A rule of thumb (Section 3.4.1, page 36) suggests almost 8 digits of accuracy in the solution are possible on UNICOS systems, because $\epsilon = 1.4 \times 10^{-13}$ and $\epsilon\kappa(A) \approx 0.5 \times 10^{-8}$. If the matrix is factored using `SGETRF` and `SGETRS` is used to solve $Ax = b$, where $b = [1, 0, 0, 0, 0]^T$, the following answers are obtained:

	Cray C90 systems	Cray T3D systems
x(1)	25.00000000081	25.000000000442
x(2)	-300.0000000149	-300.00000000880
x(3)	1050.000000064	1050.0000000394
x(4)	-1400.000000097	-1400.0000000609
x(5)	630.0000000477	630.00000003031

Two additional digits are shown for the Cray T3D results, which are closer to the exact solution of $x = [25, -300, 1050, -1400, 630]^T$. The component-wise backward error bounds computed on each system are about 9.1×10^{-16} on Cray C90 systems and about 3.2×10^{-17} on Cray T3D systems.

Iterative refinement does not change either solution on the system that computed it.

But if the Cray C90 factorization and solution are input to `SGERFS` on a Cray T3D system, iterative refinement produces the new solution:

x(1)	25.000000000147
x(2)	-300.000000000274
x(3)	1050.000000120
x(4)	-1400.000000183
x(5)	630.0000000904

which reduces the componentwise backward error bound as computed on Cray T3D systems from 1.7×10^{-15} to 2.3×10^{-17} .

The exact solution is usually not known; therefore, the error bounds are the only measure of how much a solution has improved. A smaller backward error bound does not necessarily mean that the refined solution is more accurate, just that it reflects the structure of the original problem more closely than the unrefined solution (see Arioli, et al., for details).

3.4.4.1 Error Bounds

In addition to performing iterative refinement on each column of the solution matrix, `SGERFS` and the other `xxxRFS` routines also compute error bounds for each column of the solution. These bounds are returned in the real arrays `FERR` (forward error) and `BERR` (backward error), both of length `NRHS`. For a computed solution \hat{x} to the system of equations $Ax = b$, the forward error bound f is the following:

$$\frac{\|x - \hat{x}\|}{\|x\|} \leq f \quad (3.41)$$

and the backward error bound ω bounds the relative errors in each component of A and b in the perturbed equation 2 from Section 3.4.2, page 37.

$$\|\Delta A_{i,j}\| \leq \omega \|A_{i,j}\|, \|\Delta b_i\| \leq \omega \|b_i\| \quad (3.42)$$

In Example 5, page 41, the first column of the inverse of the Hilbert matrix of order 5 is computed by solving $H_5 x = e_1$, and `SGERFS` computed error bounds

of $f \approx 1.4 \times 10^{-8}$ and $\omega \approx 9.1 \times 10^{-16}$ on UNICOS systems. This provides direct information about the solution; its relative error is at most $0(10^{-8})$; therefore, the largest components of the solution should exhibit about 8 digits of accuracy, and the system for which this solution is an exact solution is within a factor of epsilon ($\epsilon \approx 1.4 \times 10^{-14}$) from the system whose solution was attempted, so the solution is as good as the data warrants.

The component-wise relative backward error bound (equation 3) is more restrictive than the classical backward error bound $\|\Delta A\| \leq \omega \|A\|$, because it assumes $\|\Delta A\|$ has the same sparsity structure as $\|A\|$, because if $A_{i,j}$ is 0, so must be $\Delta A_{i,j}$. The backward error for the solution \hat{x} is computed from the equation

$$\omega = \max_i \frac{|r|_i}{(|A| |\hat{x}| + |b|)_i} \tag{3.43}$$

where $r = b - A\hat{x}$, and the forward error bound is computed from the equation

$$f = \frac{\| |A^{-1}| (|r| + (\gamma + 1) \epsilon (|A| |x| + |b|)) \|}{\|x\|} \tag{3.44}$$

where γ is the maximum number of nonzeros in any row of A . To avoid computing A^{-1} an approximation is used for $\| |A^{-1}| g \|$, where g is the nonnegative vector in parentheses (see Arioli, et al., for details).

3.4.5 Inverting a Matrix

Subroutines to compute a matrix inverse are provided in LAPACK, but they are not used in the driver routines. The inverse routines sometimes use extra workspace and always require more operations than the solve routines. For example, if there is one right-hand side in the equation $Ax = b$, where A is a square general matrix, the solves following the factorization require $2n^2$ operations, while inverting the matrix A requires $4/3n^3$ operations, plus another $2n^2$ to multiply b by A^{-1} . The inverse must be computed only once, however, and the cost can be amortized over the number of right-hand sides. Because multiplying by the inverse may be more efficient than doing a triangular solve, the extra cost to compute the inverse may be overcome if the number of right-hand sides is large.

3.5 Solving Least Squares Problems

In some applications, the best solution to a system of equations $AX = B$ that does not have a unique solution is required. If A is overdetermined, that is, A is $m \times n$ with $m \geq n$ and rank at least n , the system does not have a solution, but the linear least squares problem may have to be solved:

$$\text{minimize } \|B - AX\|_2 \tag{3.45}$$

If A is underdetermined, that is, A is $m \times n$ with $m < n$, generally many solutions exist, and you may want to find the solution X with minimum 2-norm. Solving these problems requires that you first obtain a basis for the range of A , and several orthogonal factorization routines are provided in LAPACK for this purpose.

An orthogonal factorization decomposes a general $m \times n$ matrix A into a product of an orthogonal matrix Q and a triangular or trapezoidal matrix. A real matrix Q is orthogonal if $Q^T Q = I$, and a complex matrix Q is unitary if $Q^H Q = I$. The key property of orthogonal matrices for least squares problems is that multiplying a vector by an orthogonal matrix does not change its 2-norm, because

$$\|Qx\|_2 = \sqrt{x^T Q^T Q x} = \sqrt{x^T x} = \|x\|_2 \tag{3.46}$$

3.5.1 Orthogonal Factorizations

LAPACK provides four different orthogonal factorizations for each data type. For real data, they are as follows:

- SGELQF: LQ factorization
- SGEQLF: QL factorization
- SGEQRF: QR factorization
- SGERQF: RQ factorization

Each of these factorizations can be done regardless of whether m or n is larger, but the QR and QL factorizations are most often used when $m \geq n$, and the LQ and RQ factorizations are most often used when $m \leq n$.

The QR factorization of an m -by- n matrix A for $m \geq n$ has the form

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix} \tag{3.47}$$

where Q is an m -by- m orthogonal matrix, and R is an n -by- n upper triangular matrix. If $m > n$, it is convenient to write the factorization as

$$A = \left(Q^{(1)} \quad Q^{(2)} \right) \begin{pmatrix} R \\ 0 \end{pmatrix} \tag{3.48}$$

or simply

$$A = Q^{(1)} R \tag{3.49}$$

where $Q^{(1)}$ consists of the first n columns of Q , and $Q^{(2)}$ consists of the remaining $m-n$ columns. The LAPACK routine `SGEQRF` computes this factorization. See the *LAPACK User's Guide* for details.

Example 6: Orthogonal factorization

The result of calling `SGEQRF` with the matrix A equal to

$$A = \begin{bmatrix} 1. & 2. & 3. \\ -3. & 2. & 1. \\ 2. & 0. & -1. \\ 3. & -1. & 2. \end{bmatrix} \tag{3.50}$$

is a compact representation of Q and R consisting of

$$A = \begin{bmatrix} -4.796 & 1.460 & -0.8341 \\ -0.5176 & -2.621 & -2.754 \\ 0.3451 & -0.03805 & 2.593 \\ 0.5176 & -0.2611 & -0.3223 \end{bmatrix}$$

$$TAU = [1.2085, 1.8698, 1.8118]$$

(3.51)

The matrix R appears in the upper triangle of A explicitly:

$$R = \begin{bmatrix} -4.796 & 1.460 & -0.8341 \\ 0. & -2.621 & -2.754 \\ 0. & 0. & 2.593 \end{bmatrix}$$

(3.52)

while the matrix Q is stored in a factored form $Q = Q_3 Q_2 Q_1$ where each Q_i is an elementary Householder transformation of the following form:
 $Q_i = I - \tau_i v_i v_i^T$. Each vector v_i has $v_i[0 : i - 1] = 0$, $v_i[i] = 1$, and $v[i + 1 : n]$ is stored below the diagonal in the i^{th} column of A . Therefore,

$$Q_1 = I - 1.2085 \begin{bmatrix} 1. \\ -0.5176 \\ 0.3451 \\ 0.5176 \end{bmatrix} [1. \quad -0.5176 \quad 0.3451 \quad 0.5176]$$

(3.53)

$$Q_2 = I - 1.8698 \begin{bmatrix} 0. \\ 1. \\ -0.03805 \\ -0.2611 \end{bmatrix} [0 \quad 1. \quad -0.03805 \quad -0.2611]$$

(3.54)

$$Q_3 = I - 1.8118 \begin{bmatrix} 0. \\ 0. \\ 1. \\ -0.3223 \end{bmatrix} [0 \quad 0 \quad 1 \quad -0.3223]$$

(3.55)

Each transformation is orthogonal (to machine precision) because τ is chosen to have the value $2 / (v^T v)$, so that

$$(I - \tau vv^T) (I - \tau vv^T) = I - 2\tau vv^T + \tau^2 (v^T v) vv^T = I \quad (3.56)$$

3.5.2 Multiplying by the Orthogonal Matrix

In the example of the last subsection, the elementary orthogonal matrices Q_i were expressed in terms of the scalar τ and the vector v that defines them, without multiplying out the expression. This was done to make the point that the elementary transformation is most often used in its factored form. This subsection describes one application in which multiplication by the orthogonal matrix Q is required. An alternative interface for this application is the LAPACK driver routine `SGELS`.

Given the QR factorization of a $m \times n$ matrix A with $m > n$ (Equation 3.47), if R has rank n , the solution to the linear least squares problem (Equation 3.45) is obtained by the following steps:

$$\begin{aligned} \begin{bmatrix} X \\ Y \end{bmatrix} &\leftarrow Q^T B \\ X &\leftarrow R^{-1} X \end{aligned} \quad (3.57)$$

The LAPACK routine `SORMQR` is used in step 1 to multiply the right-hand side matrix by the transpose of the orthogonal matrix Q , using Q in its factored form. The triangular system solution in step 2 can be done using the LAPACK routine `STRTRS`.

Continuing the example of Section 3.5.1, page 45, suppose the right-hand side vector is $b = [1. \ 2. \ 3. \ 4.]^T$. Multiplying b by Q^T by using the LAPACK routine `SORMQR`, you get

$$\begin{bmatrix} x \\ y \end{bmatrix} = Q^T b = \begin{bmatrix} -2.711 \\ -2.273 \\ 0.5712 \\ 4.143 \end{bmatrix} \quad (3.58)$$

and after solving the triangular system with the 3-by-3 matrix R ,

$$x = \begin{bmatrix} 0.7203 \\ 0.6356 \\ 0.2203 \end{bmatrix}$$

(3.59)

The last $m-n$ elements of $Q^T b$ can be used to compute the 2-norm of the residual, because $\|r\|_2 \approx \|y\|_2$. Here, $\|r\|_2 = 4.143$.

3.5.3 Generating the Orthogonal Matrix

The `SORMXX` routines described in the previous subsection are useful for operating with the orthogonal matrix Q in its factored form, but sometimes it is the matrix Q itself that is of interest.

For example, the first step in the computational procedure for the generalized eigenvalue problem $AX = \Lambda BX$ is to find orthogonal matrices U and Z such that $U^T A Z$ is upper Hessenberg and $U^T B Z$ is upper triangular (see Golub and Van Loan for details). First, the matrix B is reduced to upper triangular form by the QR factorization, and A is overwritten by $Q^T A$, using the subroutines of the previous section for multiplying by the orthogonal matrix.

Next, the updated A is reduced to Hessenberg form while preserving the triangular structure of B by applying Givens rotations to A and B , alternately from the left and the right. In order to obtain the orthogonal matrices U and Z that reduce the original problem, it is necessary to keep a copy of the matrix Q from $B = QR$ and continually update it. This requires that Q be generated after the QR factorization.

The `SORGXX` routines generate the orthogonal matrix Q as a full matrix by expanding its factored form. Using the example of Section 3.5.1, page 45, Q can be generated from its factored form by using the following Fortran code:

```
DO J = 1, N
  DO I = J+1, M
    Q(I,J) = A(I,J)
  END DO
END DO
CALL SORGQR(M,M,N,Q,LDQ,TAU,WORK,LWORK,INFO)
```

where the data from the QR factorization of A has been copied into a separate matrix Q because `SORGQR` overwrites its input data with the expanded matrix. The orthogonal matrix is returned in Q :

$$Q = \begin{bmatrix} -0.2085 & -0.8792 & 0.1562 & -0.3989 \\ 0.6255 & -0.4147 & 0.1465 & 0.6444 \\ -0.4170 & -0.2322 & -0.7665 & 0.4296 \\ -0.6255 & 0.03318 & 0.6054 & 0.4910 \end{bmatrix}$$

(3.60)

The matrix $Q^{(1)}$, consisting of only the first n columns of Q , could be generated by specifying N , rather than M , as the second argument in the call to `SORGQR`.

3.6 Comparing Answers

The results obtained by LAPACK routines should be deterministic; that is, if the same input is provided to the same subroutine in the same system environment, the output should be the same. However, because all computers operate in finite-precision arithmetic, a different order of operations may produce a different set of rounding errors, so results of the same operation obtained from different subroutines, or from the same subroutine with different numbers of processors, are not guaranteed to agree to the last decimal place.

In testing LAPACK, the test ratios in the following table were used to verify the correctness of different operations. All of these ratios give a measure of relative error. Residual tests are scaled by $1/\epsilon$, the reciprocal of the machine precision, to make the test ratios $O(1)$, and results that are sensitive to conditioning are scaled by $1/\kappa$, where $\kappa = \|A\| \|A^{-1}\|$ is the condition number of the matrix A , as computed from the norms of A and its computed inverse A^{-1} . If a given result has a test ratio less than 30, it is judged to be as accurate as the machine precision and the conditioning of the problem will allow. See the *Installation Guide for LAPACK* for further details on the testing strategy used for LAPACK.

Table 4. Verification tests for LAPACK (all should be $O(1)$)

Operation or result	Test ratio
Factorization $A = LU$	$\ LU - A\ / (n \ A\ \epsilon)$
Solution \hat{x} to $Ax = b$	$\ b - A\hat{x}\ / (\ A\ \ \hat{x}\ \epsilon)$
Compared to exact soln x	$\ x - \hat{x}\ / (\ \hat{x}\ \kappa \epsilon)$
Reciprocal condition number <code>RCOND</code>	$\max(\text{RCOND}, \kappa) / \min(\text{RCOND}, \kappa)$
Forward error bound f	$\ x - \hat{x}\ / (\ \hat{x}\ f)$

Operation or result	Test ratio
Backward error bound ω	ω/ε
Computation a A^{-1}	$\ I - AA^{-1}\ / (n \ A\ \ A^{-1}\ \varepsilon)$
Orthogonality check for Q	$\ I - Q^H Q\ / (n\varepsilon)$

ε = Machine epsilon
 n = Order of matrices
 $\kappa = \kappa = \|A\| \|A^{-1}\|$

Using Sparse Linear Solvers [4]

Many techniques exist for solving sparse linear systems. The appropriate technique depends on many factors, including the mathematical and structural properties of matrix A , the dimension of A , and the number of right-hand sides b . This section describes some of the properties that are useful in determining a good solution technique, with some common sources of matrices with these properties. Section 4.4, page 62 also describes some techniques you can use to choose the correct solver for a given problem.

4.1 Sparse Matrices

A *linear system* can be described as $Ax = b$, where A is an n -by- n matrix, and x and b are n dimensional vectors. A system of this kind is considered *sparse* if the matrix A has a small percentage of nonzero terms (less than 10%, often less than 1%). Large sparse linear systems occur frequently in engineering and scientific applications, and the solution of these systems (finding x given A and b) is an important and costly step.

If matrix A has a regular pattern, such as a banded or block structure, good performance can easily be obtained when solving the linear system. Good performance is much more difficult to obtain in problems in which A has no discernible pattern. The goal of the routines described in this section is to solve sparse linear systems efficiently, especially those where matrix A has no known regular pattern.

The following list defines the different types of sparse matrices:

- *symmetric positive definite matrix*: A matrix A is *symmetric* if $A = A^T$ (that is, if the coefficients of A are such that $a_{ij} = a_{ji}$ for all i and j).

A matrix A is defined to be *symmetric positive definite (SPD)* if A is symmetric and $y^T A y > 0$ for all vectors $y \neq 0$. It is usually difficult to directly verify that a matrix is SPD; however, it can often be determined by considering the problem source. For example, many finite difference or finite volume approximations of *partial differential equations (PDEs)* with Dirichlet or mixed boundary conditions and other mild assumptions generate SPD matrices. Also, finite element approximations with a symmetric bilinear form and equivalent test and basis functions generate SPD matrices.

SPD matrices occur frequently in applications. Optimal techniques exist to solve the related linear systems. Common sources of SPD matrices are finite

element analysis of structures, the pressure correction phase of a segregated fluid dynamics simulation, and the analysis of electrical networks.

- **Diagonally dominant matrix:** A matrix A is (*strictly*) *diagonally dominant* if $|a_{ii}| > \sum_{i \neq j} |a_{ij}|$ for all i . If A is diagonally dominant, operations that involve A are often numerically stable. Common sources of diagonally-dominant matrices are simple reservoir simulation models and the velocity equations of a segregated fluid dynamics solver.
- **Structurally symmetric matrix:** If the nonzero pattern of A is symmetric, a matrix A is *structurally symmetric*; that is, $a_{ij} \neq 0$ if and only if $a_{ji} \neq 0$. The integer complexity of a solver for a structurally symmetric matrix is greatly reduced compared to a more general solver. If A is diagonally dominant, many of the optimal solution techniques for SPD matrices can be used. Common sources of these matrices are the same as those for diagonally dominant matrices.
- **Banded matrix:** If $a_{ij} = 0$ for $|i - j| > k$, matrix A is *banded*. If k is small in relation to the problem dimension n , special techniques exist for solving the related linear systems. Systems of this form usually occur in special domains with a particular ordering of the grid or node points.
- **Tridiagonal matrix:** If $a_{ij} = 0$ for $|i - j| > 1$, matrix A is *tridiagonal*. Tridiagonal matrices occur frequently in fluid dynamics and reservoir simulation.

During a simulation, an application often generates many sparse linear systems that must be solved. These are usually related to some linear approximation of a nonlinear function or some time-marching scheme for a time-dependent problem. In these cases, the linear systems are often related and information from the previous solution can be used to solve the next linear system.

For example, consider Newton's method for a nonlinear PDE. In this case, the linear system A_n is generated by evaluating the Jacobian at a certain point. A subsequent matrix, A_{n+1} , is generated using the Jacobian at a nearby point. Thus, the two matrices A_n and A_{n+1} are close to each other, and this fact is used in the solver technique. The structure of A_n and A_{n+1} is usually identical and all structural preprocessing can be done once for all related matrices.

4.2 Solution Techniques

Solution techniques for sparse linear systems can usually be divided into two broad classes: *direct methods* and *iterative methods*. The following subsections provide an overview of both classes and provide brief algorithmic descriptions.

4.2.1 Direct Methods

Direct solution methods transform matrix A into a product of several other operators so that each of the resulting operators is easy to invert for a given right-hand side b . For example, the LU factorization of A generates lower and upper triangular matrices, L and U , respectively, such that $A = LU$.

To find $x = A^{-1}b$, compute $y = L^{-1}b$ followed by $x = U^{-1}y$, both of which are straightforward computations. Direct methods are usually popular because they are considered to be very reliable. This is true if the problem dimension and the condition number of A are not too large. See *Matrix Computations*, by Golub and Van Loan for details about error bounds.

Direct methods for dense, tridiagonal, and banded matrices are quite straightforward to implement and use the three basic steps of LU factorization as mentioned previously; they may also solve for a given right-hand side without factorization. However, for general sparse matrices, the situation is considerably more complicated; in particular, the factors L and U can become extremely dense. If pivoting is required, implementing sparse factorization can use a lot of time searching lists of numbers and creating a great deal of computational overhead.

Efficient implementations can be developed, especially for SPD and symmetric pattern, positive definite matrices. See *Computer Solution of Large Sparse Positive Definite Systems*, by George and Liu, for details.

The following algorithm shows the basic steps of the sparse Cholesky solver:

Structural preprocessing phase:

1. Find P so that $\bar{A} = PAP^T$ has factor L with near minimal fill.
2. Compute symbolic factorization, that is, find structure of L .
3. Find optimal memory use and node execution sequence.

Numerical factorization phase:

4. Compute L such that $\bar{A} = LDL^T$.

Solution phase:

5. Given b , compute $y = L^{-1}b$, $z = D^{-1}y$, $x = L^{-T}z$.

In this case, A is SPD, and L and D can be found such that $A = LDL^T$, where L is a lower triangle with unit diagonal, and D is diagonal.

Steps 1 through 3 require only the nonzero structure of A . Therefore, if two matrices A_1 and A_2 have the same structure, these steps can be skipped for A_2 . Furthermore, if the same matrix A is used for more than one right-hand side b , step 5 can be repeated (or if all right-hand sides are available at once, they can be solved for simultaneously). The solvers discussed in Section 4.3, page 57, let you perform each step separately.

4.2.2 Iterative Methods

Iterative solution methods comprise a wide variety of techniques. The solvers presented in this subsection are all in the general class of preconditioned conjugate gradient (CG) methods. These methods attempt to solve $Ax = b$ by solving an equivalent system $M^{-1}Ax = M^{-1}b$, where M is some approximation to A which is inexpensive to construct and can be easily used to compute z such that $z = M^{-1}r$.

This is left preconditioning. It is also possible to apply the preconditioner on the right side of A or on both sides. After the preconditioner is constructed and an initial approximation, x_0 to x is given, the iterative method generates a sequence of vectors $\{x_i\}$ such that x_i converges to x . Each x_i is chosen to satisfy some orthogonality or minimization condition, or both.

Unlike direct methods, iterative methods are more special-purpose. No general, effective iterative algorithms exist for an arbitrary sparse linear system. However, for certain classes of problems, an appropriate iterative method can be used to yield an approximate solution significantly faster than direct methods. Also, iterative methods usually require less memory than direct methods, thus making them the only feasible approach for large problems. See *Matrix Computations*, by Golub and Van Loan, for an introduction to these methods.

The standard preconditioned CG method, shown in the following algorithm, illustrates the basic phases common to all CG type methods used in the solvers:

Preprocessing phase:

1. Compute structure of preconditioner M .
2. Compute values of preconditioner M .
3. Analyze structure of A to optimize performance of sparse matrix vector product, $q = Ap$.

Iterative phase:

4. $r^o = b - Ax^o$
Do $k=0, \dots$
5. $z^k = Mr^k$
6. $\gamma_k = (r^k, z^k)$
7. $\beta_k = \gamma_k / \gamma_{k-1}, \beta_0 = 0$
8. $p^k = z^k + \beta_k p^{k-1}$
9. $q^k = Ap^k$
10. $\delta_k = (q^k, p^k)$
11. $\alpha_k = \gamma_k / \delta_k$
12. $x^{k+1} = x^k + \alpha_k p^k$
13. $r^{k+1} = r^k - \alpha_k p^k$
End Do

Iterative methods are very flexible. Like direct solvers, if two matrices A_1 and A_2 have the same structure, the structural preprocessing needs to be done only once. If there are multiple right-hand sides, Steps 1 through 3 can be skipped after the first right-hand side.

4.3 Sparse Solvers

The remainder of this section reviews the data structures for the Scientific Library routines and provides general guidelines for choosing the appropriate solver.

4.3.1 Data Structures for General Sparse Matrices

A large variety of data structures are used to support sparse matrix computations. Some of the common ones are the compressed sparse column (CSC), compressed sparse row (CSR), and the ELLPACK-ITPACK (ELL) formats. The abbreviations CSC, CSR, and ELL are consistent with those used in SPARSKIT; see *SPARSKIT: a basic tool kit for sparse matrix computations*, by Saad for details. The correct data structure for a given problem depends on many factors. The CSC format is used for all general sparse solvers because it is

relatively easy to use and supports a variety of matrix operations. It is also extremely common and familiar to users.

In the CSC format, the matrix A is represented by three arrays `AMAT`, `ROWIND`, and `COLSTR`. Let `NCOL` be the column dimension of A and `NZA` be the number of nonzero elements in A . `AMAT` is an `NZA`-length scalar array that contains the nonzero elements of A stored column-by-column. `ROWIND` is an `NZA`-length integer array such that each element contains the row index of the corresponding element of `AMAT`. `COLSTR` is an $(\text{NCOL} + 1)$ -length integer array such that the j^{th} element of `COLSTR` points to the start of the j^{th} column of A in `AMAT` for $j = 1, \dots, \text{NCOL}$. The last element of `COLSTR` is defined to be `COLSTR(NCOL + 1) = NZA + 1`.

For example, if A is defined as follows:

$$A = \begin{bmatrix} 11 & 0 & 0 & 14 & 0 \\ 0 & 22 & 23 & 24 & 25 \\ 0 & 32 & 33 & 0 & 0 \\ 41 & 42 & 0 & 44 & 0 \\ 0 & 52 & 0 & 0 & 55 \end{bmatrix}$$

(4.1)

A would be stored in CSC format, as follows:

```
AMAT = ( 11 41 22 32 42 52 23 33 14 24 44 25 55 )
ROWIND = ( 1 4 2 3 4 5 2 3 1 2 4 2 5 )
COLSTR = ( 1 3 7 9 12 14 )
```

If A is symmetric, store only the lower triangle, as in the following example:

```
AMAT = ( 11 41 22 32 42 52 33 44 55 )
ROWIND = ( 1 4 2 3 4 5 3 4 5 )
COLSTR = ( 1 3 7 8 9 10 )
```

If A is tridiagonal or banded, other data structures (and solvers) exist for A which are more efficient; see Section 4.3.4, page 60, for details.

4.3.2 Direct Solvers

The Scientific Library has three general sparse direct solvers, as follows:

- `SSPOTRF(3S)` / `SSPOTRS(3S)`: Factorization and solver routines for a general patterned SPD matrix. `SSPOTRF` computes the Cholesky factorization (LDL^T) of a given matrix stored in symmetric CSC format. `SSPOTRS` computes the solution for a given set of right-hand sides.
- `SSTSTRF(3S)` / `SSTSTRS(3S)`: Factorization and solver routines for a structurally symmetric sparse matrix. No pivoting is performed. `SSTSTRF` computes the LU factorization of a given matrix in CSC format. `SSTSTRS` computes the solution for a given set of right-hand sides.
- `SSGETRF(3S)` / `SSGETRS(3S)`: Factorization and solver routines for a general sparse matrix. Threshold pivoting is performed. `SSGETRF` computes the LDU factorization of a given matrix in CSC format. `SSGETRS` computes the solution for a given set of right-hand sides.

4.3.3 Iterative Solvers

All iterative solvers are used by calling `SITRSOL(3S)`, a core package of optimized preconditioned conjugate gradient (PCG) and PCG-related iterative methods and a selection of preconditioners. A variety of methods and preconditioners is provided in an attempt to compensate for the lack of robustness of any single method and preconditioner. See Heroux, Vu, and Yang, for a full description of `SITRSOL`.

Six iterative methods are provided in `SITRSOL`. Each method is a well-known, proven technique for certain classes of matrices. These six methods are abbreviated as follows:

- BCG: Bi-Conjugate Gradient Method
- CGN: Conjugate Gradient method applied to the normal equations $AA^T y = b, x = A^T y$ (Craig's Method)
- CGS: (Bi)-Conjugate Gradient Squared Method
- GMR: Generalized Minimum Residual (GMRES) Method
- OMN: Orthomin/Generalized Conjugate Residual (GCR) Method
- PCG: Preconditioned Conjugate Gradient Method

Two basic types of preconditioning are available in `SITRSOL`: explicit scaling and implicit preconditioning.

Explicit scaling of the linear system by a diagonal matrix can be directly applied because it does not disturb the structure of the matrix. It is inexpensive

to apply and usually improves the convergence rate. However, when a large variation exists in the magnitude of matrix coefficients, you should avoid explicit scaling. Six types of scaling are available, as follows:

- Symmetric diagonal
- Symmetric row-sum
- Symmetric column-sum
- Left diagonal
- Left row-sum
- Right column-sum

The second type of preconditioning is implicit preconditioning, in which the preconditioned linear system is never explicitly formed but the preconditioner is applied at each iteration. Five types of implicit preconditioning are available, as follows:

- Implicit diagonal scaling
- Incomplete Cholesky factorization
- Incomplete LU factorization
- Neumann polynomial preconditioning
- Least-squares polynomial preconditioning

4.3.4 Other Solvers

Although the focus of this section is on general sparse linear systems, some special solvers for tridiagonal matrices are also available.

For well-conditioned tridiagonal matrices (for example, a diagonally-dominant matrix T), a set of highly optimized solvers is available. If T is not well-conditioned or if there are several hundred right sides to solve simultaneously, LAPACK provides some useful tridiagonal solvers.

Table 5 summarizes the available routines. LAPACK contains other routines for tridiagonal systems which compute condition estimates and perform iterative refinement.

Table 5. Summary of tridiagonal solvers

Name	Type	Description
SDTSOL CDTSOL	Real Complex	Performs a simultaneous factor/solve for a given general tridiagonal matrix T and a single right-hand side. Extremely fast solver if T requires no pivoting and only one solution is needed.
SDTTRF CDTTRF	Real Complex	Computes the factorization for a given general tridiagonal matrix T . Does not use any right-hand side. Extremely fast solver if T requires no pivoting and multiple right-hand sides exist. Solution is computed using SDTTRS (CDTTRS).
SDTTRS CDTTRS	Real Complex	Computes the solution for a given right-hand side by using the factorization computed by SDTTRF (CDTTRF). Extremely fast solver if T requires no pivoting and multiple right-hand sides exist.
SGTSV CGTSV	Real Complex	Performs a simultaneous factor/solve for a given general tridiagonal matrix T and a block of right-hand sides. Performs pivoting.
SGTTRF CGTTRF	Real Complex	Computes the factorization for a given general tridiagonal matrix T . Performs pivoting. Solution is computed using SGTTRS (CGTTRS).
SGTTRS CGTTRS	Real Complex	Computes the solution for a given (block of) right-hand side using the factorization computed by SGTTRF (CGTTRF). Unless many right-hand sides exist, this routine is quite slow.
SPTSV CPTSV	Real Complex	Performs a simultaneous factor/solve for a given symmetric (Hermitian) positive definite tridiagonal matrix T and a block of right-hand sides.
SPTTRF CPTTRF	Real Complex	Computes the Cholesky factorization for a given symmetric (Hermitian) positive definite tridiagonal matrix T . The solution is computed using SPTTRS (CPTTRS).
SPTTRS CPTTRS	Real Complex	Computes the solution for a given (block of) right-hand side by using the factorization computed by SPTTRF (CPTTRF).

You can use these tridiagonal solvers on periodic tridiagonal matrices by using the Sherman-Morrison formulation. See *Scientific Computing: An Introduction to Parallel Computing*, by Golub and Ortega, for details.

4.4 Choosing a Solver

The choice of a good solver for a given problem depends on many factors. This subsection provides some guidelines to help you make a good choice.

4.4.1 Using Sparse Solvers

The sparse solver routines are not intended to address all types of sparse linear systems. In fact, only a small set of important cases (tridiagonal and general pattern sparse matrices) are discussed here. You can consider options other than the Scientific Library routines, some of which are available as libraries from `netlib@ornl.gov`. The following list describes these other options:

- **Banded:** If a matrix is banded, it may be best to apply a direct solver for banded matrices. These exist as a part of LINPACK and LAPACK. See the *LINPACK User's Guide* and the *LAPACK User's Guide* for details. In some cases, iterative solvers work extremely well; however, SITRSOL does not perform as well as solvers such as NSPCG, which exploit the highly regular structure more easily. See the *NSPCG User's Guide* by Oppe, Joubert, and Kincaid, for details.
- **Structured sparse:** If a matrix A is generated by a finite difference or finite element approximation of partial differential equations on a regular mesh, the sparsity pattern of A is structured and solvers that exploit this structure can perform better than those in the Scientific Library. In particular, a skyline solver can work quite well in this case as can iterative solvers in NSPCG, ITPACK, and others that have special-purpose data structures. See the *ITPACKV 2C User's Guide* by Kincaid, Oppe, Respass, and Young, for details.
- **Highly off-diagonally dominant three-dimensional problems:** Problems of this type are very difficult to solve and are the topic of current research. Presently, none of the solvers in the Scientific Library handle these linear systems well.

4.4.2 Tridiagonal Systems

If a tridiagonal linear system is stable and does not require pivoting, use one of the DT solvers even if the matrix is symmetric (SDTSOL, CDTSOL, SDTTRF /

SDTTRS, or CDTTRF / CDTTRS). The DT solvers do not exploit symmetry; however, these solvers usually outperform any other available solver. If pivoting is necessary, use the appropriate solver from LAPACK.

4.4.3 General-patterned Sparse Linear Systems

The Scientific Library contains a variety of general sparse solvers. These solvers make few assumptions about the structure of the sparse matrix A and yet they can be very efficient to use on difficult problems, especially when compared to other solvers. A variety of solvers is necessary to handle the wide range of problems.

The following subsections help you to select the correct solver; Figure 3 and Figure 4 give an approximate relationship between the different solver options.

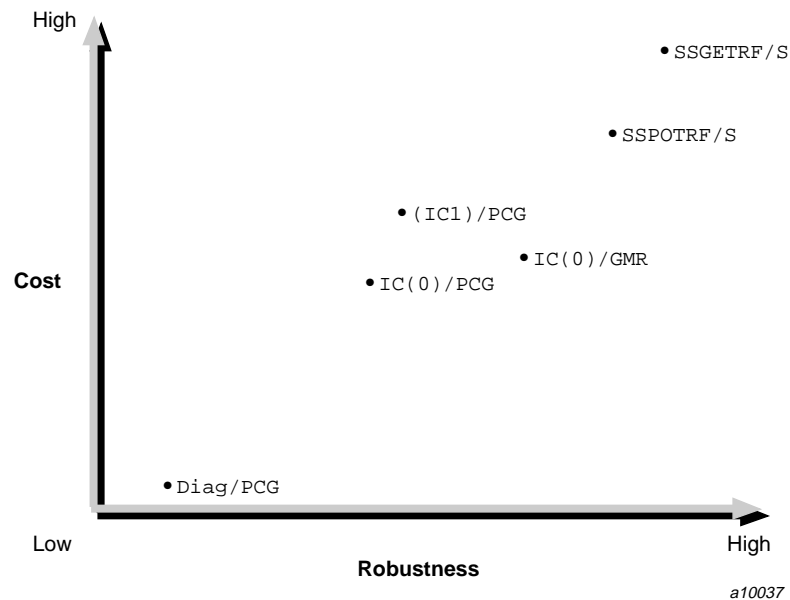


Figure 3. Cost/robustness: general symmetric sparse solvers

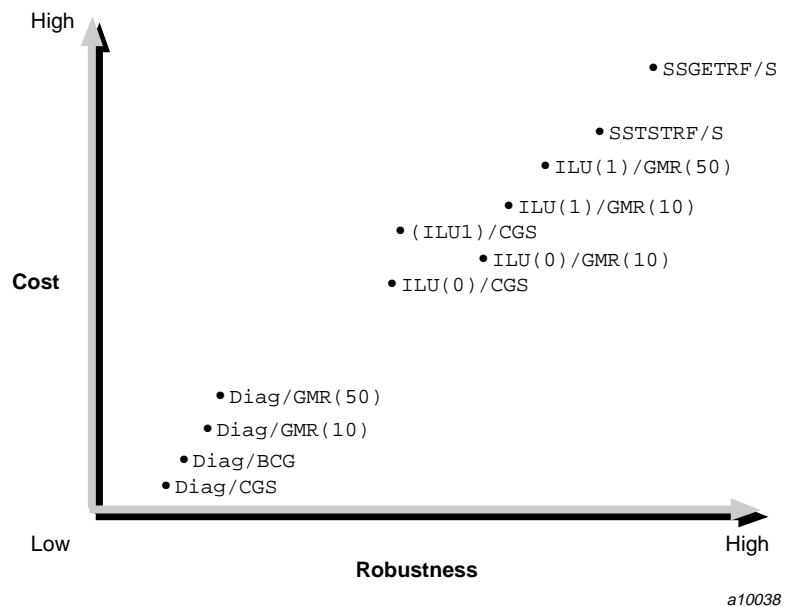


Figure 4. Cost/robustness: general unsymmetric sparse solvers

4.4.4 Choosing a Method Based on Problem Type

Without knowledge about a given problem, it is difficult to recommend a direct or iterative method. However, you can use the following rules of thumb:

- **Well-conditioned problems:** The condition number of a matrix is defined as $\kappa(A) = \|A\| \cdot \|A^{-1}\|$. A *well-conditioned* matrix is one for which $\kappa(A)$ is small. Although small is relative, if $\kappa(A) < 10^3$, A can be considered well-conditioned. In this case, iterative methods usually can perform well by using an inexpensive preconditioner and converging in a few iterations. Direct methods also work. However, the amount of work for a direct method is unaffected by $\kappa(A)$; therefore, direct methods cannot take advantage of an easy problem.
- **Ill-conditioned problems:** Because of the robust, efficient, and elegant algorithms for direct solution of SPD and structurally-symmetric sparse matrices, direct solvers are often better than iterative methods for ill-conditioned problems of these types. However, as the problem size grows, especially for three-dimensional models, a direct solver has a difficult time limiting the memory requirements of the factors L and U . In these

cases, an iterative solver may be appropriate, especially when coupled with a preconditioner that is well-suited to the problem.

4.4.4.1 Iterative Methods

Choosing an iterative method and preconditioner is not an exact science; in fact, many problems exist for which there is no good combination. This problem in method selection and lack of global robustness are primary reasons why iterative methods are not very popular in “black box” form. However, `SITRSOL` does provide a variety of tools so that you can solve most problems if the proper iterative method and preconditioner is chosen.

When choosing an iterative method, you can use the following guidelines. These guidelines depend on whether or not the matrix is symmetric positive or definite or indefinite:

- Symmetric, Positive Definite (SPD): If the matrix is SPD, the standard preconditioned conjugate gradient algorithm is usually the best choice. However, the pure truncated forms of Orthomin and GMRES with a small `iparam(16):ntrunc` may be useful on problems for which the standard conjugate gradient does not work.
- Symmetric, Indefinite: In this case, GMRES is the preferred method.
- Non-Symmetric, Definite: In this case, Orthomin or GMRES is recommended. The value of `iparam(16):ntrunc` should increase with an increase in non-symmetry. If a choice exists between the two methods, select GMRES, because it uses substantially less storage space.
- Non-Symmetric, Indefinite: In this case, no guarantee exists that any method will work. GMRES is probably the first method to try. However, the Bi-Conjugate Gradient Method (BCG) or Conjugate Gradient Squared Method (CGS) work better for many problems. BCG and CGS are desirable because they are not truncated methods and require less storage. In some cases, the Conjugate Gradient Method (CGN) may be sufficient, especially if the preconditioner can reduce the condition number of the preconditioned matrix.

4.4.4.2 Preconditioning

Preconditioned Conjugate Gradient Methods (PCG) are usually not effective unless they are coupled with some preconditioning. The goal of preconditioning is to improve the distribution of eigenvalues of the matrix A (or, in the case of the normal equations, improve the distribution of singular

values) and thus improve the convergence rate of the iterative method. The following preconditioners are listed in order of least expensive per iteration (which also means least robust) to most expensive (and most robust):

- **Scaling:** Scaling is the simplest and cheapest preconditioner. Explicit scaling often improves the performance of the iterative routine and can be used in some form, even if some other implicit preconditioning is done. If nothing is known about the matrix, symmetric diagonal scaling can be used if the matrix is symmetric; left diagonal scaling can be used if the matrix is non-symmetric (unless the matrix has a zero diagonal element). Row or column sum scaling can be more effective in certain cases when there is a large variation in the magnitude of values across the rows or columns of the matrix.
- **Implicit diagonal preconditioning:** It is equivalent to explicit diagonal scaling except in the case of CGN in which explicit diagonal scaling uses the diagonal of A and implicit diagonal scaling uses the diagonal of AA^T . This preconditioning is available if you do not want to scale the matrix explicitly.
- **Polynomial preconditioning:** SITRSOL has the following types of polynomial preconditioning:
 - A truncated Neumann series expansion applies the truncated Neumann series approximation of the inverse of A to the linear system.
 - The second type uses a polynomial s , in which s is chosen so that $s(A)A$ is as close in norm to the identity as possible in a least-squares sense.

These preconditioners are well-suited to vector and parallel architectures and are similar in performance to diagonal preconditioning. However, like diagonal preconditioning, they are usually not very robust, especially if A is not SPD. The least-squares approach is usually the best of the two polynomial preconditioners. See the *SIAM Journal of Scientific Statistical Computing*, by Saad, for details.

- **Incomplete Factorization:** The most robust forms of preconditioners are the Incomplete Cholesky (IC) and Incomplete LU (ILU) factorizations. These methods try to approximate the inverse of A by computing only certain terms of the Cholesky or LU factorizations of A . Although these methods are usually very robust and are often the only techniques to work on difficult problems, they are also very expensive per iteration and do not perform well on vector architectures. They also do not perform very well on parallel architectures, except for machines that have a small number of processors. For nonsymmetric problems, ILU is essential for many problems because other types of preconditioners are not well understood in this case. IC is

often useful, especially for ill-conditioned matrices that come from structural analysis.

4.4.5 Direct General Sparse Solvers

It is usually relatively easy to select the appropriate sparse direct solver, as follows:

- Symmetric, Positive Definite (SPD): If the matrix is SPD, use `SSPOTRF / SSPOTRS`.
- Symmetric, Indefinite: You can often solve these systems by using `SSPOTRF / SSPOTRS` and get a reasonable answer. If this does not work, use `SSGETRF / SSGETRS`.
- Non-Symmetric, Definite: If the matrix is structurally symmetric, or can be made structurally symmetric by adding zero terms to the structure, `SSTSTRF / SSTSTRS` is the best choice. If this does not work, use `SSGETRF / SSGETRS`.
- Non-Symmetric, Indefinite: `SSGETRF / SSGETRS` is often the best choice.

4.5 Performance Tuning

Most of the sparse solvers in the Scientific Library have a variety of tuning parameters that change the characteristics of the solver. These parameters let you tune the performance of the overall computing environment for their problems. This subsection discusses these parameters and presents guidelines for selecting appropriate values.

4.5.1 Parallel Processing

All of the sparse solvers (except the DT tridiagonal solvers) exploit multiple processors if the user allows it. In some cases, the improvement in time to solution is very dramatic. In others, improvement is marginal, and in some cases, because of the structure of the sparse matrix, performance can **decrease**. This happens when there is little or no parallelism to exploit in the problem, yet the overhead of trying to extract parallelism was incurred. In all cases, because of the complexity of sparse computations and the demand on the memory bandwidth, parallel processing is more expensive in **cumulative** CPU time. Thus, parallel speedup is always less than linear speed.

The `NCPUS` environment variable, the number of physical processors, and the number of users on the system controls the number of processors that are available to an application. The best number of processors to use depends on several issues.

4.5.2 Reusing Information

As mentioned earlier, many nonlinear and time-dependent simulations generate sequences of closely related matrices. The sparse solvers in the Scientific Library have several techniques for optimizing performance in this case, as discussed in the following subsections.

4.5.2.1 Reuse of Structure

If two matrices have the same nonzero structure, any preprocessing that involves only the matrix structure can be done once. This leads to significant performance improvements after the first call to the solver. The *ido* argument for all of the direct sparse solvers lets you assert that the structure has already been processed. When using `SSGETRF`, set *iparam(13)* to 1 in this case. For `SITRSOL`, the second argument *ipath* lets you assert that the structure is identical to the previous problem. See Example 9, page 84, for a program that reuses structure.

4.5.2.2 Multiple Right-hand Sides

If more than one right-hand side is solved for a single matrix, the direct solvers can solve for one or all right-hand sides after the factorization is computed. The solve phase of the computation is extremely cheap compared to the factorization. Thus, the cost for solving many right-hand sides is almost the same as solving for one.

`SITRSOL` can also exploit multiple right-hand sides. However, the cost of solving each one is usually greater than a direct solver. If there are many right-hand sides to solve, a high-quality preconditioner should be used in order to reduce the number of iterations and thereby reduce the overall cost of the solution. See Example 10, page 89, for a sample program showing how to use the solvers with multiple right-hand sides.

4.5.2.3 Reuse of Values

If matrices A_1 and A_2 have closely related values, it is possible to exploit this fact with preconditioned iterative methods by keeping the preconditioner generated for A_1 and using it for A_2 . This is especially appropriate if the

preconditioner is expensive to construct. For incomplete Cholesky and incomplete LU preconditioner, use the integer argument *iparam(14):ifcomp*, which lets you assert that the preconditioner was already computed in a previous call and can be reused.

Although this technique is not available to direct solvers, you can couple a direct solver with an iterative scheme so that the direct solver provides the solution for A_1 and can be coupled with an iterative scheme to provide solutions for A_2 .

4.5.2.4 Save/restart

During a long computation, it can be useful to break the computation into several phases. This allows reuse of early phases, minimizes the loss of information in case of system failure, and lets you tune system resources for each phase. Save and restart capabilities are provided to let you take a “snapshot” of the computation between algorithmic phases. All necessary information is stored in a user-defined binary file, which can be used at a later time to resume computation.

All of the direct solvers (SSPOTRF, SSPOTRS, SSTSTRF, SSTSTRS, SSGETRF, and SSGETRS) let you save a binary image after any of the three structural preprocessing steps or after the numerical factorization step. To control this process, use *iparam(6)* and *iparam(7)*. SITRSOL lets you save a binary image after the construction of the preconditioner or after the iterative phase. The *ipath* and *iparam(19)* and *iparam(20)* arguments control the process. See Example 11, page 93, which shows the process for both direct and iterative solvers.

4.5.3 SITRSOL Tuning Issues

The following SITRSOL tuning parameters let you exchange increased memory use for better time to solution. Their use depends on the system resources and the problem being solved.

- Sparse matrix vector product: Computation of the sparse matrix vector product is one of the most time-consuming components of SITRSOL. Its performance can vary greatly depending on available memory. If you set *iparam(21):mvformat=0*, SITRSOL uses the original CSC format matrix to compute the matrix vector product. This requires no extra memory.

However, unless there is a large number of nonzero elements per column, the performance will be very poor. If *iparam(21):mvformat=1*, performance is usually much better. This allows SITRSOL to convert the matrix to an optimal data structure (jagged diagonal format, JAD) for which the sparse matrix vector product performs very well. If the original matrix is stored in

symmetric CSC format, you can improve performance more by setting `iparam(8):isympap=0`, which allows `SITRSOL` to store both the upper and lower triangle for matrix vector multiplication. By default, `SITRSOL` converts to JAD and stores only one of the upper and lower triangle if symmetric.

- Density of incomplete factors: When using incomplete Cholesky or incomplete LU factorization preconditioners, you can reduce the number of iterations by increasing the level of fill in the factors. This is controlled by `iparam(11):lvlfill`. As the value of this parameter increases, the number of nonzero elements in the factors increases. This increases both the time and memory that the preconditioner uses. For difficult problems, however, it can reduce the time to solution. The value of `lvlfill` usually ranges from 0 to 3.
- Size of Krylov Subspace: GMRES (GMR) and Orthomin (OMN) are both truncated minimization techniques (see the *SIAM Journal of Scientific Statistical Computing*, by Saad, for details). `iparam(16):ntrunc` determines the size of the truncated minimization space (Krylov subspace). If `ntrunc` is set to the problem dimension, these methods are direct solvers. As `ntrunc` increases, the robustness of GMR and OMN usually increases, as does memory use, machine performance, and the cost of computation. The correct value for `ntrunc` depends on available memory and the difficulty of the problem being solved. The value of `ntrunc` usually ranges from 10 to 100.
- Diagonal shifting for incomplete Cholesky: For ill-conditioned SPD, the only preconditioner that works is incomplete Cholesky with diagonal shifting, coupled with PCG. In this case, `SITRSOL` tries to compute the incomplete Cholesky factors and, if it detects a negative diagonal, uses a simple iterative technique to shift the diagonal values. After each shift, it restarts the factorization. If shifting is required for a class of related problems, you can usually eliminate restarting by experimenting with `rparam(15):gammin` and setting it to a small positive value to allow the factorization to complete the first time.

4.5.4 Direct Solver Tuning Issues

As with `SITRSOL`, most of the following direct solver tuning parameters let you exchange increased memory use for better time to solution. How to use them depends on the system resources and the problem being solved.

- Supernode augmentation: `iparam(8)` and `iparam(9)` let you specify relaxation factors for growth of supernodes. *supernodes* are a collection of columns that have the same nonzero pattern. Using supernodes, performance improves dramatically because the amount of indirect address is greatly decreased. To

increase the size of supernodes, allow some zero fill in the supernodes. $iparam(8)$ indicates the number of zero elements allowed in a supernode. Although the default value is 0, some zero fill is usually desirable. A value of $iparam(8) = 256$ is usually reasonable; however, if A is very sparse, a value of 512 or 1024 may be appropriate.

The $iparam(9)$ argument indicates the maximum percent of zero-fill in a supernode and is set to 0 by default. A value of 100 is reasonable, which allows the supernode to double in size with zero-fill. If A is very sparse, a value of 200-500 may be appropriate.

- Frontal matrix grouping and parallel execution: When executing on multiple processors, it can be useful to use two types of parallelism in the direct factorization: parallel elimination of supernodes and parallel execution with a frontal matrix. The first type comes from computing with independent supernodes concurrently. In the beginning of the factorization, a large number of supernodes usually can be processed in parallel. As the elimination tree is traversed, however, the number of independent supernodes decreases. At the same time, the size of the frontal matrices grows. Thus, at some point, it is appropriate to switch and allow all processors to work on one frontal matrix.

The $iparam(10)$ and $iparam(11)$ arguments from `SSPOTRF` and `SSTSTRF` (`SSGETRF` does not have these arguments) let you control when the switch occurs. Set both parameters to the same value. When the switch should occur depends on the data. For large problems, however, a value of 10,000 is reasonable. For smaller problems, a smaller value is better. See the *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, by Yang, for a full discussion of this topic.

- Threshold pivoting: `SSGETRF` can perform standard partial pivoting. However, it is often possible to relax the pivoting requirements to improve performance and still obtain good accuracy in the solution. `SSGETRF` uses threshold pivoting to improve performance by letting you specify a parameter δ , $0.0 \leq \delta \leq 1.0$, called a threshold value.

Let a_{*j} denote the maximum, $\max_{i=j,n} |a_{ij}|$, of the possible pivots at the j^{th} step of the factorization. The pivot is then taken to be the first value a_{ij} , $j \leq i \leq n$, such that $|a_{ij}| \geq \delta a_{*j}$ (that is, the first element in the j^{th} column with absolute value greater than or equal to the threshold value times the maximum absolute value in the j^{th} column). If $\delta = 0.0$, no pivoting is done; if $\delta = 1.0$, standard pivoting is done.

As with other tuning parameters, a good value for δ depends on the actual problems. However, experience has shown that you should choose δ

between 0.1 and 0.001. A choice of $\delta = 0.01$ is reasonable. If the matrix is scaled or otherwise well-conditioned, you can choose a much smaller value (such as 0.00001).

4.6 SITRSOL Quick Reference

SITRSOL uses one of six possible preconditioned Conjugate Gradient methods to solve a linear system $Ax = b$. The syntax of the SITRSOL(3S) routine is as follows:

```
CALL SITRSOL(method, ipath, neqns, nvars, x, b, icolptr, irowind,  
value, liwork, iwork, lrwork, rwork, iparam, rparam, ierr)
```

The following tables, Table 6 through Table 8, page 74, provide a summary of the arguments and argument types used with SITRSOL. See the SITRSOL(3S) man page for more details.

Table 6. SITRSOL argument summary

Argument	Type	Description
<i>method</i>	Char*3	Selects iterative method (BCG, CGN, CGS, GMR, OMN, or PCG)
<i>ipath</i>	Integer	Controls execution path
<i>neqns</i>	Integer	Row dimension of matrix
<i>nvars</i>	Integer	Column dimension of matrix
<i>x</i>	Real	Initial approximation and final approximation to the solution vector
<i>b</i>	Real	Right-hand side vector
<i>icolptr</i>	Integer	Column pointer array for matrix
<i>irowind</i>	Integer	Vector that contains the row indices of the elements of the matrix
<i>value</i>	Real	Vector that contains the nonzero elements of the matrix
<i>liwork</i>	Integer	Length of the work vector <i>iwork</i>

Argument	Type	Description
<i>iwork</i>	Integer	Integer work vector
<i>lwork</i>	Integer	Length of the work vector <i>rwork</i>
<i>rwork</i>	Real	Real work vector
<i>iparam</i>	Integer	Vector that contains integer parameters
<i>rparam</i>	Real	Vector that contains real parameters
<i>ierr</i>	Integer	Error flag

Table 7. *iparam* summary

Name	Index	Description
<i>isym</i>	1	Full or symmetric format flag for input matrix
<i>itest</i>	2	Stopping criteria
<i>maxiter</i>	3	Maximum number of iterations
<i>niter</i>	4	Number of iterations performed
<i>msglvl</i>	5	Level of messages output
<i>iunit</i>	6	Unit to which output is written
<i>iscale</i>	7	Diagonal scaling option
<i>isympap</i>	8	Selects full or symmetric jagged diagonal format
<i>iprelrb</i>	9	Selects left or right preconditioning
<i>ipretyp</i>	10	Selects type of preconditioning
<i>lvlfill</i>	11	Selects level of fill for incomplete factorization
<i>maxlfil</i>	12	Maximum storage available for lower triangle factor
<i>maxufil</i>	13	Maximum storage available for upper triangle factor
<i>ifcomp</i>	14	Flag to compute or not compute incomplete factorization
<i>kdegree</i>	15	Degree of polynomial preconditioner
<i>ntrunc</i>	16	Number of vectors retained for truncated GMR and OMN

Name	Index	Description
<i>nvorth</i>	17	Number of vectors to orthogonalize against for GMR
<i>nrstrt</i>	18	Number of iterations to perform before restarting for OMN
<i>irestrt</i>	19	Selects save/restart location
<i>iosave</i>	20	Save/restart I/O unit
<i>mvformat</i>	21	Select format for computation of matrix vector products
<i>nicfmax</i>	22	Set maximum times to try IC factorization
<i>nicfacs</i>	23	Actual number of factorizations tried

Table 8. *rparam* summary

Name	Index	Description
<i>tol</i>	1	Stopping criterion tolerance
<i>err</i>	2	Error estimate
<i>alpha</i>	3	Unused
<i>beta</i>	4	Estimate of the spectral radius
<i>timscal</i>	5	Accumulated time spent scaling user matrix
<i>timsets</i>	6	Accumulated symbolic incomplete factorization time
<i>timsetn</i>	7	Accumulated incomplete numerical factorization time
<i>timset</i>	8	Accumulated total preconditioner setup time
<i>timpre</i>	9	Accumulated time spent applying preconditioning
<i>timmvs</i>	10	Accumulated time to convert from column pointer to jagged diagonal
<i>timmv</i>	11	Accumulated time spent computing matrix vector product
<i>timmtv</i>	12	Accumulated time spent performing matrix vector transpose

Name	Index	Description
<i>timit</i>	13	Time spent in iterative routine (not including preconditioning or matrix vector products)
<i>timtot</i>	14	Total time used
<i>gammin</i>	15	Minimum value for shift factor gamma
<i>gammax</i>	16	Value for shift factor gamma

4.7 Usage Examples

This subsection contains several examples that illustrate how to use the Scientific Library sparse solvers in a variety of circumstances. All matrices solved are very simple. However, they are generated by subroutine MATGEN, which can easily be replaced by more realistic problems.

Note: You should use the sample output from these examples only as approximate values to what you will get by running these programs on your own system. Small differences are likely due to different compilers, operating systems, random numbers, and floating-point representations.

Example 7: General symmetric positive definite

This first example illustrates the simplest use of SITRSOL and SSPOTRF, SSPOTRS for a symmetric positive definite problem.

```

PROGRAM EX1
C
C Purpose:
C Illustrates the use of SITRSOL and SSPOTRF/S to solve a simple
C sparse symmetric linear system by a single call to each subroutine.
C
  PARAMETER (NMAX = 5, NZAMAX = 9)
  PARAMETER (LIWORK = 350, LWORK = LIWORK )
  INTEGER NEQNS, NZA, IPATH, IERR,
&   ROWIND(NZAMAX), COLSTR(NMAX+1), IWORK(LIWORK), IPARAM(40)
  REAL AMAT(NZAMAX), RPARAM(30), X(NMAX), B(NMAX),
&   SOLN(NMAX), WORK(LWORK)
  CHARACTER*3 METHOD
C
C -----
C Define matrix and RHS
C -----

```

```

        CALL MATGEN (NMAX, NEQNS, B, COLSTR, NZAMAX, NZA, ROWIND, AMAT)
C
C
C   -----
C   Solve problem using SITRSOL
C   -----
C
C.....Let the initial guess for x be random numbers between 0 and 1
      DO 20 I = 1, NEQNS
        X(I) = RANF()
    20  CONTINUE
C
C.....Set default parameter values
      CALL DFAULTS ( IPARAM, RPARAM )
C
C.....Select no scaling and left least-squares preconditioning
      IPARAM(7) = 0
      IPARAM(9) = 1
      IPARAM(10) = 5
C
C.....Call SITRSOL to solve the problem using PCG
      IPATH = 2
      METHOD = 'PCG'
      CALL SITRSOL ( METHOD, IPATH, NEQNS, NEQNS, X, B, COLSTR, ROWIND,
&                AMAT, LIWORK, IWORK, LWORK, WORK, IPARAM, RPARAM, IERR )
C
C   -----
C   Solve same problem using SSPOTRF/SSPOTRS
C   -----
C
C.....use all default values
      IPARAM(1) = 0
C.....do all 4 phases of factorization
      IDO = 14
C
C.....compute factorization using SSPOTRF
      CALL SSPOTRF ( IDO, NEQNS, COLSTR, ROWIND, AMAT, LWORK,
&                WORK, IPARAM, IERR )
C
C.....compute solution using SSPOTRS
C
C.....solve standard way
      IDO = 1

```



```

C
C.....Arguments
      INTEGER NMAX, NEQNS, NZA
      INTEGER COLSTR(NMAX+1), ROWIND(NZAMAX)
      REAL B(NMAX), AMAT(NZAMAX)
C
C.....Local variables
      INTEGER NEQNSL, NZAL
      PARAMETER (NEQNSL = 5, NZAL = 9 )
      INTEGER COLSTRL(NEQNSL+1), ROWINDL(NZAL)
      REAL AMATL(NZAL)
C
C.....Define matrix via data statements
C
      DATA (AMATL(I), I=1, NZAL) / 4.0, -1.0, -1.0, 4.0, -1.0, 4.0,
&
      4.0, -1.0, 4.0 /
C
      DATA (ROWINDL(I), I=1, NZAL ) / 1, 2, 4, 2, 3, 3, 4, 5, 5 /
C
      DATA (COLSTRL(I), I=1, NEQNSL + 1) / 1, 4, 6, 7, 9, 10 /
C
C.....Define problem size
      NEQNS = NEQNSL
      NZA = NZAL
C
C.....Check if enough space
      IF (NEQNS .GT. NMAX .OR. NZA .GT. NZAMAX) THEN
          WRITE(*,*)'Not enough space.'
          STOP
      ENDIF
C
C.....Define matrix
      DO 10 I = 1, NZA
          AMAT(I) = AMATL(I)
          ROWIND(I) = ROWINDL(I)
10  CONTINUE
C
      DO 20 I = 1, NEQNS + 1
          COLSTR(I) = COLSTRL(I)
20  CONTINUE
C
C.....Define b to be all 1's
      DO 30 I = 1, NEQNS

```

```

        B(I) = 1.0
30    CONTINUE
C
C.....ALL DONE
        RETURN
        END

```

The following is the output as generated on one processor of a UNICOS system.

```

***** Output from program: EX1 *****
        Difference between SITRSOL and SSPOTRF/S = 0.11093345E-13

```

Example 8: General unsymmetric

This second example illustrates the simplest use of SITRSOL, SSGETRF, SSGETRS, and SSTSTRF, SSTSTRS for an unsymmetric definite problem.

```

PROGRAM EX2
C
C Purpose:
C Illustrates the use of SITRSOL, SSGETRF/S and SSTSTRF/S to
C solve a simple sparse unsymmetric linear system by a single
C call to each subroutine.
C
        PARAMETER (NMAX = 5, NZAMAX = 13)
        PARAMETER (LIWORK = 350, LWORK = LIWORK )
        INTEGER NEQNS, NZA, IPATH, IERR,
&          ROWIND(NZAMAX), COLSTR(NMAX+1), IWORK(LIWORK), IPARAM(40)
        REAL AMAT(NZAMAX), RPARAM(30), X(NMAX), B(NMAX), B1(NMAX),
&          SOLN(NMAX), WORK(LWORK)
        CHARACTER*3 METHOD
C
C -----
C Define matrix and RHS
C -----
        CALL MATGEN ( NMAX, NEQNS, B, COLSTR, NZAMAX, NZA, ROWIND, AMAT)
C
C.....Copy B to B1 for later use
        CALL SCOPY( NEQNS, B, 1, B1, 1 )
C
C -----
C Solve problem using SITRSOL

```

```

C -----
C
C.....Let the initial guess for x be random numbers between 0 and 1
      DO 20 I = 1, NEQNS
          X(I) = RANF()
      20  CONTINUE
C
C.....Set default parameter values
      CALL DFAULTS ( IPARAM, RPARAM )
C
C.....Select nonsymmetric, no explicit scaling, left implicit scaling
C preconditioning
      IPARAM(1) = 0
      IPARAM(7) = 0
      IPARAM(9) = 1
      IPARAM(10) = 1
C
C.....Call SITRSOL to solve the problem using CGS
      IPATH = 2
      METHOD = 'CGS'
      CALL SITRSOL ( METHOD, IPATH, NEQNS, NEQNS, X, B, COLSTR, ROWIND,
&                AMAT, LIWORK, IWORK, LWORK, WORK, IPARAM, RPARAM, IERR )
C
C -----
C Solve same problem using SSGETRF/SSGETRS
C -----
C
C.....use all default values
      IPARAM(1) = 0
C
C.....do all 4 phases of factorization
      IDO = 14
C
C.....threshold for pivoting
      THRESH = 0.01
C
C.....compute factorization using SSGETRF
      CALL SSGETRF ( IDO, NEQNS, COLSTR, ROWIND, AMAT, LWORK,
&                WORK, IPARAM, THRESH, IERR )
C
C.....compute solution using SSGETRS
C
C.....solve standard way

```

```

        IDO = 1
C.....solve for 1 RHS with leading dim = neqns
        NRHS = 1
        LDB = NEQNS
C
        CALL SSGETRS ( IDO, LWORK, WORK, NRHS, B, LDB,
&                    IPARAM, IERR )
C
C
C -----
C Solve same problem using SSTSTRF/SSTSTRS
C -----
C
C.....use all default values
        IPARAM(1) = 0
C
C.....do all 4 phases of factorization
        IDO = 14
C
C.....compute factorization using SSTSTRF
        CALL SSTSTRF ( IDO, NEQNS, COLSTR, ROWIND, AMAT, LWORK,
&                    WORK, IPARAM, IERR )
C
C.....compute solution using SSTSTRS
C
C.....solve standard way
        IDO = 1
C.....solve for 1 RHS with leading dim = neqns
        NRHS = 1
        LDB = NEQNS
C
        CALL SSTSTRS ( IDO, LWORK, WORK, NRHS, B1, LDB,
&                    IPARAM, IERR )
C
C -----
C Compare solutions
C -----
C
C.....Compute 2-norm of the difference between SITRSOL (array X),
C SSTSTRF/S (in array B) and SSTSTRF/S solution (in array B1).
C
C.....compute differences
        CALL SAXPY ( NEQNS, -1., X, 1, B, 1 )

```

```

        CALL SAXPY ( NEQNS, -1., X, 1, B1, 1 )
C
C.....compute norms
        ERR1 = SNRM2( NEQNS, B, 1 )
        ERR2 = SNRM2( NEQNS, B1, 1 )
C
C.....print results
        WRITE(6,11)ERR1, ERR2
11  FORMAT ( '***** Output from program: EX2 *****', /
&          '   Difference between SITSOL and SSGETF/S = ',E15.8, /
&          '   Difference between SITSOL and SSTSTRF/S = ',E15.8, )
C.....all done
        END
C
        SUBROUTINE MATGEN ( NMAX, NEQNS, B, COLSTR, NZAMAX, NZA,
&                          ROWIND, AMAT )
*   The following routine MATGEN defines, in sparse column format,
* the matrix
*
*
*   A = |  4   0   0  -1   0 |
*       | -2   4   0   0   0 |
*       |  0  -2   4   0   0 |
*       | -1   0   0   4  -1 |
*       |  0   0   0  -1   4 |
*
* where A is generated by using a non-standard difference scheme
* for Poisson's Equation with Dirichlet boundary conditions. The
* domain is a unit square with the upper right quarter removed and
* a grid spacing of 0.25. MATGEN also defines b, the right-hand-side.
C
C.....Arguments
        INTEGER NMAX, NEQNS, NZA
        INTEGER COLSTR(NMAX+1), ROWIND(NZAMAX)
        REAL B(NMAX), AMAT(NZAMAX)
C
C.....Local variables
        INTEGER NEQNSL, NZAL
        PARAMETER (NEQNSL = 5, NZAL = 13 )
        INTEGER COLSTRL(NEQNSL+1), ROWINDL(NZAL)
        REAL AMATL(NZAL)
C
C.....Define matrix via data statements

```



```

***** Output from program: EX2 *****
      Difference between SITRSOL and SSGETRF/S = 0.70604095E-12
      Difference between SITRSOL and SSTSTRF/S = 0.70604095E-12

```

Example 9: Reuse of structure

This example solves two linear systems which have the same structure. The first calls to the solvers process both structural and numerical data. The second calls reuse the structural information from the first calls. Note that the timing information reflects that less work was done for the second linear system.

```

PROGRAM EX3
C
C Purpose:
C Illustrates the use of SITRSOL and SSPOTRF/S to solve two sparse
C symmetric linear systems which have the same nonzero structure.
C The first call to the solvers compute both structural and
C numerical data. The second call uses the structural information
C from the first call and only processes the numerical data.
C
      PARAMETER (NMAX = 5, NZAMAX = 9)
      PARAMETER (LIWORK = 350, LWORK = LIWORK )
      INTEGER NEQNS, NZA, IPATH, IERR, JOB,
&          ROWIND(NZAMAX), COLSTR(NMAX+1), IWORK(LIWORK), IPARAM(40)
      REAL AMAT(NZAMAX), RPARAM(30), X(NMAX), B(NMAX),
&          SOLN(NMAX), WORK(LWORK), WORK1(LWORK)
      CHARACTER*3 METHOD
C
C -----
C Define variables for first call to solvers
C -----
      JOB = 1
C JOB is used by MATGEN to generate one of two matrices
C
C.....Do all phases of computation
      IPATH = 2
C
C.....do all 4 phases of factorization
      IDO = 14
C
      WRITE(6,11)
11  FORMAT (' ***** Output from program: EX3 *****', //
&          ' ***** Solve First System *****')
C

```

```

100 CONTINUE
C
C -----
C Define matrix and RHS
C -----
CALL MATGEN ( JOB, NMAX, NEQNS, B, COLSTR, NZAMAX, NZA,
&            ROWIND, AMAT )
C
C -----
C Solve problem using SITSOL
C -----
C
C.....Let the initial guess for x be random numbers between 0 & 1
DO 20 I = 1, NEQNS
    X(I) = RANF()
20 CONTINUE
C
C.....Set default parameter values
CALL DFAULTS ( IPARAM, RPARAM )
C
C.....Select no scaling and left least-squares preconditioning
IPARAM(7) = 0
IPARAM(9) = 1
IPARAM(10) = 5
C
C.....Call SITSOL to solve the problem using PCG
METHOD = 'PCG'
C
C.....Accumulate time spent in solvers
TSTART = SECOND()
CALL SITSOL(METHOD, IPATH, NEQNS, NEQNS, X, B, COLSTR, ROWIND,
&          AMAT, LIWORK, IWORK, LWORK, WORK, IPARAM, RPARAM, IERR)
TTOTAL = SECOND() - TSTART
C
C -----
C Solve same problem using SSPOTRF/SSPOTRS
C -----
C
C.....use all default values
IPARAM(1) = 0
C
C.....compute factorization using SSPOTRF

```

```

        TSTART = SECOND()
        CALL SSPOTRF ( IDO, NEQNS, COLSTR, ROWIND, AMAT, LWORK,
&                   WORK1, IPARAM, IERR )
        TTOTAL = TTOTAL + SECOND() - TSTART
C
C.....compute solution using SSPOTRS
C
C.....solve standard way
        IDO = 1
C.....solve for 1 RHS with leading dim = neqns
        NRHS = 1
        LDB = NEQNS
C
        TSTART = SECOND()
        CALL SSPOTRS ( IDO, LWORK, WORK1, NRHS, B, LDB,
&                   IPARAM, IERR )
        TTOTAL = TTOTAL + SECOND() - TSTART
C
C -----
C Compare solutions
C -----
C
C.....Compute two-norm of the difference between SITRSOL in array X
C and SSPOTRF/S solution in array B.
C
C.....compute differences
        CALL SAXPY ( NEQNS, -1., B, 1, X, 1 )
C
C.....compute norms
        ERR = SNRM2( NEQNS, X, 1 )
C
C.....print results
        WRITE(6,12)ERR, TTOTAL
12  FORMAT ( ' Difference between SITRSOL and SSPOTRF/S = ',E15.8,/
&          ' Total time to compute solutions          = ',E15.8,)
C
C.....Check if JOB = 1. If so, reset to 2 and recall solvers with
C new values and same structure.
        IF ( JOB .EQ. 1 ) THEN
C
C.....Define variables for second call to solvers
        JOB = 2

```



```

        PARAMETER (NEQNSL = 5, NZAL = 9 )
        INTEGER COLSTR(L(NEQNSL+1), ROWINDL(NZAL)
        REAL AMATL(NZAL)
C
C.....Define matrix via data statements
C
        DATA (AMATL(I), I=1, NZAL)/4.0, -1.0, -1.0, 4.0, -1.0, 4.0,
&
        &                4.0, -1.0, 4.0 /
C
        DATA (ROWINDL(I), I=1, NZAL ) / 1, 2, 4, 2, 3, 3, 4, 5, 5 /
C
        DATA (COLSTR(L(I), I=1, NEQNSL + 1) / 1, 4, 6, 7, 9, 10 /
C
C.....Define problem size
        NEQNS = NEQNSL
        NZA = NZAL
C
C.....Check if enough space
        IF (NEQNS .GT. NMAX .OR. NZA .GT. NZAMAX) THEN
            WRITE(*,*)'Not enough space.'
            STOP
        ENDIF
C
C.....Define matrix
        DO 10 I = 1, NZA
            AMAT(I) = AMATL(I)
            ROWIND(I) = ROWINDL(I)
10    CONTINUE
C
C.....If JOB = 2 then double matrix values
        IF (JOB .EQ. 2) THEN
            DO 20 I = 1, NZA
                AMAT(I) = 2.0 * AMAT(I)
20    CONTINUE
        ENDIF
C
        DO 30 I = 1, NEQNS + 1
            COLSTR(I) = COLSTR(L(I)
30    CONTINUE
C
C.....Define b to be all 1's
        DO 40 I = 1, NEQNS
            B(I) = 1.0

```

```

40  CONTINUE
C
C.....ALL DONE
      RETURN
      END

```

The following is the output as generated on one processor of a UNICOS system.

```

***** Output from program: EX3 *****

***** Solve First System *****
      Difference between SITSOL and SSPOTRF/S = 0.11093345E-13
      Total time to compute solutions          = 0.11168460E-02

***** Solve Second System *****
      Difference between SITSOL and SSPOTRF/S = 0.33232593E-14
      Total time to compute solutions          = 0.84099600E-03
      Ratio of times for first and second sol. = 1.328

```

Example 10: Multiple right-hand sides

This example illustrates how to solve a linear system that has multiple right-hand sides.

```

PROGRAM EX4
C
C Purpose:
C Illustrates the use of SITSOL and SSPOTRF/S to solve a simple
C sparse symmetric linear system with multiple right-hand sides.
C
      PARAMETER (NMAX = 5, NZAMAX = 9, NRHS = 3)
      PARAMETER (LIWORK = 350, LWORK = LIWORK )
      INTEGER NEQNS, NZA, IPATH, IERR,
&   ROWIND(NZAMAX), COLSTR(NMAX+1), IWORK(LIWORK), IPARAM(40)
      REAL AMAT(NZAMAX), RPARAM(30), X(NMAX,NRHS), B(NMAX,NRHS),
&   SOLN(NMAX), WORK(LWORK)
      CHARACTER*3 METHOD
C
C -----
C Define matrix and right-hand sides
C -----
      CALL MATGEN ( NMAX, NEQNS, NRHS, B, COLSTR, NZAMAX, NZA,
&   ROWIND, AMAT )

```

```

C
C
C -----
C   Solve problem using SITRSOL
C -----
C
C.....Let the initial guess for x be random numbers between 0 and 1
      DO 10 IRHS = 1, NRHS
        DO 10 I = 1, NEQNS
          X(I,IRHS) = RANF()
10    CONTINUE
C
C.....Set default parameter values
      CALL DFAULTS ( IPARAM, RPARAM )
C
C.....Select no scaling and left Least-squares
preconditioning
      IPARAM(7) = 0
      IPARAM(9) = 1
      IPARAM(10) = 5
C
C.....Call SITRSOL to solve the problem using PCG
      IPATH = 2
      METHOD = 'PCG'
      CALL SITRSOL(METHOD, IPATH, NEQNS, NEQNS, X, B, COLSTR, ROWIND,
&      AMAT, LIWORK, IWORK, LWORK, WORK, IPARAM, RPARAM, IERR)
C
C.....Solve for subsequent RHS
      IPATH = 4
      DO 20 IRHS = 2, NRHS
        CALL SITRSOL(METHOD, IPATH, NEQNS, NEQNS, X(1,IRHS), B(1,IRHS),
&      COLSTR, ROWIND, AMAT, LIWORK, IWORK, LWORK,
&      WORK, IPARAM, RPARAM, IERR )
20    CONTINUE
C
C -----
C   Solve same problem using SSPOTRF/SSPOTRS
C -----
C
C.....use all default values
      IPARAM(1) = 0
C.....do all 4 phases of factorization
      IDO = 14

```

```

C
C.....compute factorization using SSPOTRF
      CALL SSPOTRF ( IDO, NEQNS, COLSTR, ROWIND, AMAT, LWORK,
&                  WORK, IPARAM, IERR )
C
C.....compute solution using SSPOTRS
C
C.....solve standard way
      IDO = 1
C.....solve for all RHS with leading dim = neqns
      LDB = NEQNS
C
      CALL SSPOTRS ( IDO, LWORK, WORK, NRHS, B, LDB,
&                  IPARAM, IERR )
C
C -----
C Compare solutions
C -----
      WRITE(6,11)
11  FORMAT ( '***** Output from program: EX4 *****' )
C
C.....Compute two-norm of the difference between SITRSOL (array X)
C and SSPOTRF/S solution (in array B) for each RHS.
      DO 30 IRHS = 1, NRHS
C
C.....compute differences
      CALL SAXPY ( NEQNS, -1., B(1,IRHS), 1, X(1,IRHS), 1 )
C
C.....compute norms
      ERR = SNRM2( NEQNS, X(1,IRHS), 1 )
C
C.....print results
      WRITE(6,12)IRHS,ERR
12  FORMAT ( 'Difference between SITRSOL & SSPOTRF/S for sol #',
&           i2, ' = ',E15.8, )
30  CONTINUE
C.....all done
      END
C
      SUBROUTINE MATGEN ( NMAX, NEQNS, NRHS, B, COLSTR, NZAMAX, NZA,
&                       ROWIND, AMAT )
* The following routine MATGEN defines, in sparse column format,
* the matrix

```

```

*
*      |  4  -1  0  -1  0  |
*      | -1  4  -1  0  0  |
*  A = |  0  -1  4  0  0  |
*      | -1  0  0  4  -1  |
*      |  0  0  0  -1  4  |

```

* where A is generated by using a five-point difference scheme for
* Poisson's Equation with Dirichlet boundary conditions. The
* domain is a unit square with the upper right quarter removed and
* a grid spacing of 0.25. MATGEN also defines NRHS right-hand-sides.

C

C.....Arguments

```

      INTEGER NMAX, NEQNS, NZA
      INTEGER COLSTR(NMAX+1), ROWIND(NZAMAX)
      REAL B(NMAX,NRHS), AMAT(NZAMAX)

```

C

C.....Local variables

```

      INTEGER NEQNSL, NZAL
      PARAMETER (NEQNSL = 5, NZAL = 9 )
      INTEGER COLSTRL(NEQNSL+1), ROWINDL(NZAL)
      REAL AMATL(NZAL)

```

C

C.....Define matrix via data statements

C

```

      DATA (AMATL(I), I=1, NZAL) / 4.0, -1.0, -1.0, 4.0, -1.0, 4.0,
&
      4.0, -1.0, 4.0 /

```

C

```

      DATA (ROWINDL(I), I=1, NZAL ) / 1, 2, 4, 2, 3, 3, 4, 5, 5 /

```

C

```

      DATA (COLSTRL(I), I=1, NEQNSL + 1) / 1, 4, 6, 7, 9, 10 /

```

C

C.....Define problem size

```

      NEQNS = NEQNSL
      NZA = NZAL

```

C

C.....Check if enough space

```

      IF (NEQNS .GT. NMAX .OR. NZA .GT. NZAMAX) THEN
        WRITE(*,*)'Not enough space.'
        STOP
      ENDIF

```

C

C.....Define matrix

```

DO 10 I = 1, NZA
  AMAT(I) = AMATL(I)
  ROWIND(I) = ROWINDL(I)
10 CONTINUE
C
DO 20 I = 1, NEQNS + 1
  COLSTR(I) = COLSTRL(I)
20 CONTINUE
C
C.....Define b to be all 1's, 2's, 3's, ...
DO 30 IRHS = 1, NRHS
  DO 30 I = 1, NEQNS
    B(I,IRHS) = FLOAT(IRHS)
30 CONTINUE
C
C.....ALL DONE
RETURN
END

```

The following is the output as generated on one processor of a UNICOS system.

```

***** Output from program: EX4 *****
Difference between SITRSOL and SSPOTRF/S for sol # 1 = 0.11093345E-13
Difference between SITRSOL and SSPOTRF/S for sol # 2 = 0.50242959E-14
Difference between SITRSOL and SSPOTRF/S for sol # 3 = 0.22469334E-13

```

Example 11: Save/restart

This example illustrates how to use the save/restart feature of the sparse solvers. The first program, EX5A, computes the incomplete Cholesky preconditioner for SITRSOL and stores it in a binary file SITRSOL.SAV. It also computes the Cholesky factorization by using SSPOTRF and stores it to a binary file SSPOTRF.SAV. The second program, EX5B, opens SITRSOL.SAV and solves the sparse linear system by using the incomplete Cholesky factor computed by EX5A. It also opens SSPOTRF.SAV and calls SSPOTRS to solve the linear system, using the factorization computed by SSPOTRF in EX5A.

Note: Because all preprocessing was done in EX5A, the original copy of the sparse matrix, that is, the arrays AMAT, ROWIND, and COLSTR are not needed in EX5B. The only exception to this is when *iparam(21):mvformat* is set to 0. In this case, the original matrix is used to perform the sparse matrix vector product.

```

PROGRAM EX5A
C
C Purpose:
C Illustrates the use of SITRSOL and SSPOTRF/S to solve a simple
C sparse symmetric linear system with SAVE/RESTART files. This
C program calls SITRSOL to construct the preconditioner and SSPOTRF to
C compute the factorization. Both are saved to binary files for later
C use by the program EX5B.
C
      PARAMETER (NMAX = 5, NZAMAX = 9)
      PARAMETER (LIWORK = 350, LWORK = LIWORK )
      INTEGER NEQNS, NZA, IPATH, IERR,
&          ROWIND(NZAMAX), COLSTR(NMAX+1), IWORK(LIWORK), IPARAM(40)
      REAL AMAT(NZAMAX), RPARAM(30), X(NMAX), B(NMAX),
&          SOLN(NMAX), WORK(LWORK)
      CHARACTER*3 METHOD
C
C -----
C Define matrix and RHS
C -----
      CALL MATGEN (NMAX, NEQNS, B, COLSTR, NZAMAX, NZA, ROWIND, AMAT)
C
C -----
C Preprocess problem using SITRSOL
C -----
C
C.....Let the initial guess for x be random numbers between 0 and 1
      DO 10 I = 1, NEQNS
          X(I) = RANF()
      10  CONTINUE
C
C.....Set default parameter values
      CALL DFAULTS ( IPARAM, RPARAM )
C
C.....Select no scaling and left IC preconditioning
      IPARAM(7) = 0
      IPARAM(9) = 1
      IPARAM(10) = 2
C
C.....Save preconditioner setup for later use
      OPEN(1, FILE='SITRSOL.SAV', FORM='UNFORMATTED',STATUS='NEW')
      IPARAM(19) = 1 ! Save after preconditioner setup

```

```

        IPARAM(20) = 1 ! Put in unit 1
        IPARAM(3)  = 1 ! Perform only 1 iteration
        IPARAM(5)  = 1 ! Fatal error messages only
C
C.....Call SITRSOL to setup the problem using PCG (perform
C  one iteration)
        IPATH = 2
        METHOD = 'PCG'
        CALL SITRSOL (METHOD, IPATH, NEQNS, NEQNS, X, B, COLSTR, ROWIND,
&                   AMAT, LIWORK, IWORK, LWORK, WORK, IPARAM, RPARAM, IERR)
C
C.....Close file
        CLOSE(1)
C
C  -----
C  Solve same problem using SSPOTRF/SSPOTRS
C  -----
C
C.....Save factorization for later use
        OPEN(1, FILE='SSPOTRF.SAV', FORM='UNFORMATTED', STATUS='NEW')
C
C.....Override default values
        IPARAM( 1) = 1
        IPARAM( 2) = 6 ! Output unit for messages
        IPARAM( 3) = 0 ! Report only fatal errors
        IPARAM( 4) = 0 ! Do not save adjacency structure
        IPARAM( 5) = 0 ! This is a fresh start
        IPARAM( 6) = 4 ! Perform save after 4th phase
        IPARAM( 7) = 1 ! Save file to unit 1
        IPARAM( 8) = 0 ! The rest are defaults
        IPARAM( 9) = 0
        IPARAM(10) = 0
        IPARAM(11) = 0
        IPARAM(12) = 0
C.....do all 4 phases of factorization
        IDO = 14
C
C.....compute factorization using SSPOTRF
        CALL SSPOTRF ( IDO, NEQNS, COLSTR, ROWIND, AMAT, LWORK,
&                   WORK, IPARAM, IERR )
C
C.....Close file
        CLOSE(1)
```

```

C
C.....Signal completion
      WRITE(6,11)
11  FORMAT ('***** Output from program: EX5A *****', /
&         '      Preconditioner and Factorization saved')
C
      END
C
      SUBROUTINE MATGEN ( NMAX, NEQNS, B, COLSTR, NZAMAX, NZA,
&                      ROWIND, AMAT )
*   The following routine MATGEN defines, in sparse column format,
*   the matrix
*
*
*   A = |  4  -1   0  -1   0 |
*       | -1   4  -1   0   0 |
*       |  0  -1   4   0   0 |
*       | -1   0   0   4  -1 |
*       |  0   0   0  -1   4 |
* where A is generated by using a five-point difference scheme for
* Poisson's Equation with Dirichlet boundary conditions. The
* domain is a unit square with the upper right quarter removed and
* a grid spacing of 0.25. MATGEN also defines b, the right-hand-side.
C
C.....Arguments
      INTEGER NMAX, NEQNS, NZA
      INTEGER COLSTR(NMAX+1), ROWIND(NZAMAX)
      REAL B(NMAX), AMAT(NZAMAX)
C
C.....Local variables
      INTEGER NEQNSL, NZAL
      PARAMETER (NEQNSL = 5, NZAL = 9 )
      INTEGER COLSTRL(NEQNSL+1), ROWINDL(NZAL)
      REAL AMATL(NZAL)
C
C.....Define matrix via data statements
C
      DATA (AMATL(I), I=1, NZAL) / 4.0, -1.0, -1.0, 4.0, -1.0, 4.0,
&         4.0, -1.0, 4.0 /
C
      DATA (ROWINDL(I), I=1, NZAL ) / 1, 2, 4, 2, 3, 3, 4, 5, 5 /
C
      DATA (COLSTRL(I), I=1, NEQNSL + 1) / 1, 4, 6, 7, 9, 10 /
C

```

```

C.....Define problem size
      NEQNS = NEQNSL
      NZA = NZAL
C
C.....Check if enough space
      IF (NEQNS .GT. NMAX .OR. NZA .GT. NZAMAX) THEN
          WRITE(*,*)'Not enough space.'
          STOP
      ENDIF
C
C.....Define matrix
      DO 10 I = 1, NZA
          AMAT(I) = AMATL(I)
          ROWIND(I) = ROWINDL(I)
10    CONTINUE
C
      DO 20 I = 1, NEQNS + 1
          COLSTR(I) = COLSTRL(I)
20    CONTINUE
C
C.....Define b to be all 1's
      DO 30 I = 1, NEQNS
          B(I) = 1.0
30    CONTINUE
C
C.....ALL DONE
      RETURN
      END

```

The following is the output as generated on one processor of a UNICOS system.

```

***** Output from program: EX5A *****
      Preconditioner and Factorization saved

```

```

PROGRAM EX5B
C
C Purpose:
C Illustrates the use of SITRSOL and SSPOTRF/S to solve a simple
C sparse symmetric linear system with SAVE/RESTART files. This
C program calls SITRSOL to solve a problem using the preconditioner
C computed in EX5A and calls SSPOTRS to solve the problem using
C the factorization computed in EX5A. Note that the original
C sparse matrix is not needed in this program, that is, AMAT,
C ROWIND and COLSTR are not defined.

```

```

C
  PARAMETER (NMAX = 5, NZAMAX = 9)
  PARAMETER (LIWORK = 350, LWORK = LIWORK )
  INTEGER NEQNS, NZA, IPATH, IERR, IWORK(LIWORK), IPARAM(40)
  REAL RPARAM(30), X(NMAX), B(NMAX), WORK(LWORK)
  CHARACTER*3 METHOD

C
C -----
C Solve problem using SITRSOL with Restart
C -----
C
C.....Define problem dimension
  NEQNS = NMAX
C
C.....Let the initial guess for x be random numbers between 0 and 1
C Define B to be all 1's
  DO 10 I = 1, NEQNS
    X(I) = RANF()
    B(I) = 1.0
  10 CONTINUE
C
C.....Set default parameter values
  CALL DFAULTS ( IPARAM, RPARAM )
C
C.....Select no scaling and left IC preconditioning
  IPARAM(7) = 0
  IPARAM(9) = 1
  IPARAM(10) = 2
C
C.....Restore preconditioner setup from earlier call
  OPEN(1, FILE='SITRSOL.SAV', FORM='UNFORMATTED',STATUS='OLD')
  IPARAM(19) = 1 ! Start after preconditioner setup
  IPARAM(20) = 1 ! Read from unit 1
C
C.....Call SITRSOL to solve the problem using PCG and restart data
  IPATH = 5
  METHOD = 'PCG'
  CALL SITRSOL ( METHOD, IPATH, NEQNS, NEQNS, X, B, COLSTR, ROWIND,
&              AMAT, LIWORK, IWORK, LWORK, WORK, IPARAM, RPARAM, IERR )
C
C.....Close file
  CLOSE(1)
C

```

```

C -----
C Solve same problem using SSPOTRS with Restart
C -----
C
C.....Restore factorization from earlier computation
      OPEN(1, FILE='SSPOTRF.SAV', FORM='UNFORMATTED',STATUS='OLD')
C
C.....Override default values
      IPARAM( 1) = 1
      IPARAM( 2) = 6 ! Output unit for messages
      IPARAM( 3) = 0 ! Report only fatal errors
      IPARAM( 4) = 0 ! Do not save adjacency structure
      IPARAM( 5) = 1 ! This is a fresh start
      IPARAM( 7) = 1 ! Read file from unit 1
C
C.....compute solution using SSPOTRS
C
C.....solve standard way
      IDO = 1
C.....solve for 1 RHS with leading dim = neqns
      NRHS = 1
      LDB = NEQNS
C
      CALL SSPOTRS ( IDO, LWORK, WORK, NRHS, B, LDB,
&                  IPARAM, IERR )
C
C -----
C Compare solutions
C -----
C
C.....Compute two-norm of the difference between SITRSOL
C (in array X) and SSPOTRF/S solution (in array B).
C
C.....compute differences
      CALL SAXPY ( NEQNS, -1., B, 1, X, 1 )
C
C.....compute norms
      ERR = SNRM2( NEQNS, X, 1 )
C
C.....print results
      WRITE(6,11)ERR
11  FORMAT ('***** Output from program: EX5B *****',/
&          'Difference between SITRSOL and SSPOTRF/S = ',E15.8, )

```

```
C.....all done
      END
```

The following is the output as generated on one processor of a UNICOS system.

```
***** Output from program: EX5B *****
```

```
      Difference between SITRSOL and SSPOTRF/S = 0.13873790E-13
```

Out-of-core Linear Algebra Software [5]

This section explains the basic use of the Scientific Library routines for out-of-core computations in linear algebra. It gives an overview of the routines, discusses the concept of virtual matrices, describes the types of subroutines used with out-of-core routines, and also provides examples.

5.1 Out-of-core Routines

Some problems are so large that it is not possible or convenient to store all of the data in main memory during program execution. For such problems, you can use an *out-of-core technique*. The term out-of-core is an anachronism (referring to magnetic core memory), but it is still used to refer to algorithms that combine input and output with computation to solve problems in which the data resides on disk or some other secondary random-access storage device.

For example, consider the problem of solving a set of simultaneous linear equations. If n equations are in n unknowns, the amount of data required to represent the problem is n^2 floating-point numbers. Further, the amount of computation required to compute a solution is approximately $(2n^3)/3$ floating-point operations.

Assume that n equals 30,000. The amount of memory required to store the matrix is 900 Mwords (or 7.2 Gbytes, assuming one 64-bit word equals 8 bytes). If the effective computational rate is 2.0 GFLOPS, the amount of time required to solve the problem is 9,000 seconds (2.5 hours). This amount of computation is large compared to the amount of input and output required. Therefore, this problem is computationally intensive and is an excellent candidate for solution by an out-of-core technique.

The Scientific Library contains a unified set of out-of-core routines for solving problems in dense linear algebra. These routines are easy to use and highly efficient. They are similar to, and modeled after, the LAPACK and Level 3 BLAS routines (see Section 5.3.6, page 111).

Rather than ordinary in-memory arrays, the out-of-core routines work with *virtual matrices*. You can visualize the operation of these routines in familiar mathematical concepts. The low-level routines do all of the necessary I/O automatically.

The following are features of the out-of-core routines:

- They are based on state-of-the-art algorithms for numerical linear algebra.
- They contain highly-efficient computational kernels that perform at peak attainable speeds on the hardware.
- They provide highly-efficient, automatic I/O, so users are not involved in the details of the I/O routines.
- They use virtual matrices, which are easy to create and use.
- They contain built-in detailed performance measurement capabilities which can print automatically to give you complete information about software and hardware performance.
- They contain tuning parameters that you can easily change to optimize the software for specific problems and for various computing resources.

5.2 Virtual Matrices

A virtual matrix is an important concept in the out-of-core routines. A virtual matrix is similar to a mathematical matrix, with elements that are accessed using subroutine calls.

A virtual matrix is like a Fortran array, with elements that are real numbers, subscripts that are integers between 1 and a positive number n , and a leading dimension (which you define when the virtual matrix is created).

Unlike a Fortran array, you cannot access a virtual matrix directly from a Fortran (or C) program. Instead, you access a virtual matrix by using calls to subroutines (described in Section 5.3, page 106). These subroutines provide the only mechanism for manipulating a virtual matrix.

Users do not do any explicit input or output to read from or write to a virtual matrix. Although a virtual matrix is actually stored as a file, the library software handles the details of I/O to the matrix automatically and efficiently. This leaves users free to concentrate on the mathematical solution to the problem.

5.2.1 Unit Numbers

The name of a virtual matrix must be an integer number between 1 and 99. The name identifies the Fortran unit number of the file in which the virtual matrix file is stored. By default, unit number 1 is associated with file `fort.1`, unit 2 with file `fort.2`, and so on. Do not use unit 5 or 6, because these unit numbers are associated by default with the special UNICOS files `stdin` and

stdout. Also, do not use any unit number that your program is using for another purpose.

To associate a particular file with a particular unit number, use the assign-by-unit option of the `assign(1)` command. For example, to store a virtual matrix in a file in directory `/tmp/xxx`, name the file `mydata`. To use Fortran unit number 3 for the file, you could issue the following command prior to executing the following command:

```
assign -a /tmp/xxx/mydata u:3
```

Within the out-of-core subroutines, you would use the number 3 as the value of the argument for the virtual matrix name.

You could use the following command to assign the file on unit 1 to SDS (secondary data storage in the SSD solid-state storage device), with a subcode of "scratch" (meaning discard the file at end of processing):

```
assign -F SDS.SCR u:1
```

See the `assign(1)` man page for general information on Fortran unit numbers.

5.2.2 File Format

A virtual matrix is actually stored as a file, in a special format that is useful only for virtual linear algebra routines. But outside of the program, at the operating system level, such a file can be copied, moved, archived, compressed, or treated like any other binary file. The UNICOS `assign` command determines the actual characteristics of the file, including the device to which it is assigned (for example, disk or SSD). See the `assign(1)` man page for more information.

A virtual matrix is a *binary unblocked file*. You do not have to specify the `-su` option on the `assign` command, and you cannot use other formats or conversions in conjunction with the out-of-core routines. Any structure specification other than `-su` is an error. The file is blocked into pages, but this blocking is done by the Scientific Library routines, not by the UNICOS routines, so the `assign` command considers it an **unblocked file**.

The actual input and output is done internally using AQIO (asynchronous queued I/O). This feature allows highly efficient random-access I/O without using any unnecessary intermediate buffering of data.

To use a data file that was created by some means other than using virtual linear algebra routines, you should write a program that reads the file (using the usual Fortran I/O facilities) and copy it, a section at a time, to a virtual

matrix (using the virtual copy routines). If you want to use a virtual matrix as input to some other program, write a program that uses the virtual copy routines to get data from the virtual matrix, then write it out using the usual Fortran I/O facilities. If only the virtual linear algebra routines use the data, it is most convenient to work just with the virtual matrix files themselves, using the subroutines provided.

5.2.3 Leading Virtual Dimension

A virtual matrix has a certain leading dimension, just like a Fortran array. For example, if the virtual matrix is 1000-by-2000 elements, the first (leading) dimension is 1000. This value, 1000, is the value supplied for the leading dimension argument in the subroutines.

When you create a virtual matrix you can use any value for the leading dimension, but after it is defined you cannot change it. You must use the same value in subsequent subroutine calls.

5.2.4 Definition and Redefinition of Elements

When accessing elements of a virtual matrix, the value of the first subscript must be in the range of $1 \leq i \leq LDV$, where i is the subscript, and LDV is the leading dimension defined in the subroutine call. There is no set upper limit to the value of the second subscript, but it must be a positive integer.

When you create a virtual matrix, any element not explicitly defined is undefined and should not be referenced. You should explicitly define every element that you use in a computation. For example, to create an identity matrix that is 2000-by-2000, first set all 4,000,000 elements to 0, then use the virtual copy routines to set the 2000 diagonal elements to 1. Do not just set the diagonal elements to 1 and assume that the off-diagonal elements are 0.

After you define the elements of a virtual matrix, their values remain defined until you explicitly change or remove them.

5.2.5 File Size

The size (in words) of a virtual matrix file is slightly larger than the total number of elements it contains. Thus, a virtual matrix of size 5000-by-5000 would contain slightly more than 25 million words, or 200 Mbytes of data. It is not exactly 25 million words because of the way that the software organizes data internally into pages.

When you define the value of a virtual matrix element, you are implicitly creating file space for all elements up to the one you define. For example, if you declare that a virtual matrix has a leading dimension of 5000, and you define a value for element (1, 1000), the software creates a virtual matrix file large enough to contain elements (i,j) for

$$1 \leq i \leq 5000$$

$$1 \leq j \leq 1000$$

which is 5 Mwords, or 40 Mbytes of file space.

If you are working with a symmetric or a triangular virtual matrix, you can cut the file size roughly in half by using packed storage mode.

5.2.6 Packed Storage Mode

Packed storage of a triangular or symmetric matrix means that only 50% of the matrix is actually stored on disk or SSD. If a real matrix is declared to be lower triangular, only the lower triangle is stored; if declared to be upper triangular, only the upper triangle is stored. If the matrix is symmetric, either the lower or upper triangular part is stored.

Likewise, a complex matrix may be lower or upper triangular, or it may be symmetric, with only the lower or upper triangle being stored. Additionally, a complex matrix may be a *Hermitian matrix* (equal to the conjugate of its transpose), with either the lower or upper triangle being stored.

For the purpose of storing a matrix, the VBLAS routines do not have to distinguish between a triangular, symmetric, or Hermitian matrix; they must know only which part of the matrix is being stored: the full matrix, the lower triangle, or the upper triangle.

In the level 2 BLAS routines, packed storage implies a linearized storage scheme. For the VBLAS routines, packed storage is similar, but more complicated, because it is the page structure of the virtual matrix binary file that is linearized; pages that correspond to the upper (or lower) part of a triangular matrix are omitted.

Three possible storage modes exist:

- Full, meaning that the full matrix is stored
- Lower, meaning that only the lower triangle is stored
- Upper, meaning that only the upper triangle is stored

To define this storage mode, call the `VSTORAGE` routine, which has the calling sequence:

```
CALL VSTORAGE (unit, mode)
```

The *unit* argument is an integer that gives the unit number of the virtual matrix, and *mode* is a character string that specifies the storage mode.

See the `VSTORAGE(3S)` man page for further information.

5.2.7 Page Size

At the internal level, the software organizes virtual matrices into pages. The *page size* (the size of one page) is, by default, 25-by-256 words, or 65,536 words. Internal I/O transfers are done in minimum units of one page. This size gives excellent performance, for both disk and SSD solid-state storage device.

You can redefine the page size that the software uses, although doing so is not advised unless special performance tuning considerations are involved. Internal file structure of a virtual matrix depends on the page size. Thus, a virtual matrix created with a certain page size cannot be read or written later using a different page size; it must be recreated.

5.3 Subroutine Types

The following subroutines are used in out-of-core routines:

- Initialization and termination subroutines (see Section 5.3.3, page 109)
- Virtual copy subroutines (see Section 5.3.4, page 109)
- Virtual LAPACK subroutines (see Section 5.3.5, page 110)
- Virtual BLAS subroutines (see Section 5.3.6, page 111)

Lower-level routines that you cannot call directly also are used (see Section 5.3.7, page 112).

5.3.1 Complex Routines

The out-of-core software described in this subsection deal with matrices of real numbers. These routines also have counterparts that work with matrices of *complex numbers* (numbers that have a real and an imaginary part). For example, the complex two-dimensional counterpart of the virtual copy routine

SCOPY2RV(3S) is routine CCOPY2RV. Likewise, routine VCGETRF(3S) factors a general complex virtual matrix.

5.3.2 Summary of Routines

Table 9 summarizes the out-of-core routines for linear algebra. The real and complex versions of the routines are listed together because their functions are very similar. For example, VSGETRF(3S) and VCGETRF are similar; VSGETRF uses real numbers, and VCGETRF uses complex numbers.

Table 9. Summary of out-of-core routines for linear algebra

Type	Name	Purpose
Initialization and termination	VBEGIN	Initialization
	VEND	Termination
	VSTORAGE	Initialization, declaration of packed matrix storage
Virtual copy routines	SCOPY2RV	Single-precision two-dimensional real-to-virtual copy
	SCOPY2VR	Single-precision two-dimensional virtual-to-real copy
	CCOPY2RV	Complex two-dimensional real-to-virtual copy
	CCOPY2VR	Complex two-dimensional virtual-to-real copy
Virtual LAPACK routines	VSGETRF	Virtual single-precision general triangular factorization (that is, LU decomposition—as part of Gaussian elimination with partial pivoting)
	VCGETRF	Virtual complex general triangular factorization (that is, LU decomposition—as part of Gaussian elimination with partial pivoting)

Type	Name	Purpose
	VSPOTRF	Virtual single-precision positive-definite (symmetric) L triangular factorization (that is, LL' decomposition, as part of Cholesky factorization)
	VSGETRS	Virtual single-precision general triangular solve (that is, backsubstitution, solving multiple right-hand sides as part of Gaussian elimination)
	VCGETRS	Virtual complex general triangular solve (that is, backsubstitution, solving multiple right-hand sides as part of Gaussian elimination)
	VSPOTRS	Virtual single-precision positive-definite (symmetric) triangular solve (that is, backsubstitution, solving multiple right-hand sides as part of Gaussian elimination)
Virtual BLAS routines	VSGEMM	Virtual Single-precision General Matrix Multiplication
	VCGEMM	Virtual Complex General Matrix Multiplication
	VSTRSM	Virtual Single-precision Solve Multiple right-hand sides
	VCTRSM	Virtual Complex Solve Multiple right-hand sides
	VSSYRK	Virtual Single-precision Symmetric Rank- k update

In the virtual copy routine names listed in this table, the letter *s* indicates single precision (real), just as it does in the names for the BLAS routines. The numeral 2 indicates two-dimensional, because these routines copy matrices, as opposed to vectors. The codes *RV* and *VR* indicate real-to-virtual or virtual-to-real (that is, in-memory to virtual memory, or vice versa). The word *real* is used with two different meanings: *not complex* and *in-memory*.

5.3.3 Initialization and Termination Subroutines

To initialize the underlying library routines, you must issue a call to the `VBEGIN(3S)` routine. This routine has several optional arguments that relate to package tuning performance. An important argument is an integer that specifies how many words to use for buffer space. `VBEGIN` automatically allocates the requested amount of memory by using a call to the operating system. A `VEND(3S)` routine also must be called when done with virtual linear algebra. `VEND` closes any open files that are used for virtual matrices and deallocates the memory that was allocated by `VBEGIN`.

5.3.4 Virtual Copy Subroutines

As indicated previously, an important feature of this routine is that users do not do any explicit input or output to a virtual matrix. To create a virtual matrix, copy sections of an in-memory matrix to a section of a virtual matrix, using virtual copy routines. For example, to work with one column of the matrix, call a virtual copy routine to copy the vector, x , to column j of the virtual matrix.

The two virtual copy routines are `SCOPY2RV(3S)` and `SCOPY2VR(3S)`. The numeral 2 indicates two-dimensional (because the routines copy parts of matrices, as opposed to vectors); the letters `RV` and `VR` indicate real-to-virtual and virtual-to-real, respectively.

You can copy any row, column, or rectangular section of the virtual matrix by inputting to the routine the corner subscripts and dimensions of the submatrix. You also can fetch or store one element at a time, although that is less efficient. You never have to do any explicit input or output; the routines do I/O automatically. As far as users are concerned, it is just a matrix copy operation.

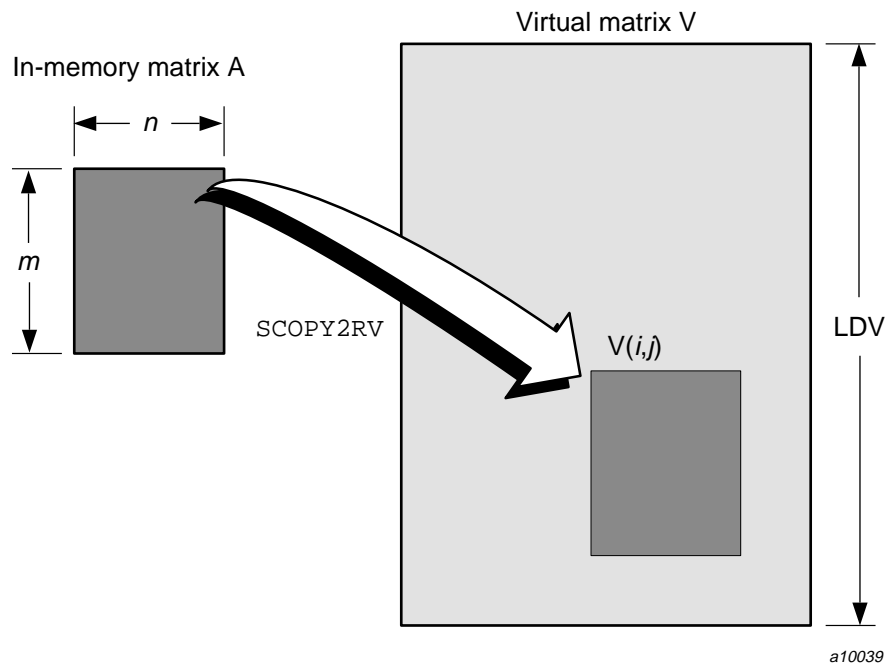


Figure 5. In-memory to virtual matrix copy

5.3.5 Virtual LAPACK Subroutines

The design of the out-of-core routines parallels the design of routines in the library that solve similar problems in memory. The LAPACK routines in the Scientific Library are state-of-the-art routines for solving problems in dense linear algebra. For out-of-core problems in dense linear algebra, the Scientific Library contains a set of routines, called virtual LAPACK routines, that use similar or identical algorithms. See the `INTRO_LAPACK(3S)` man page for more information on the LAPACK library.

The following are the virtual LAPACK routines in the Scientific Library:

- `VSGETRF(3S)` (virtual single-precision general triangular factorization) is the virtual matrix counterpart of `SGETRF`, the LAPACK routine for LU matrix factorization (with partial pivoting). The main difference between the two routines is that, in place of the argument for the matrix name, `VSGETRF` requires the name of a virtual matrix. The name is an integer constant or variable that specifies the unit number of the file in which the matrix resides.

- `VSGETRS(3S)` is the virtual LAPACK routine that performs backsubstitution for solving systems of equations based on the matrix factorization produced by `VSGETRF`. `VSGETRS` is the counterpart of the LAPACK routine `SGETRS`.
- `VCGETRF` and `VCGETRS` are the virtual routines used for applications in which the matrices contain complex numbers, rather than real numbers. Complex-number versions of the virtual copy routines also exist.

5.3.6 Virtual BLAS Subroutines

The LAPACK routines perform much of their computational work through calls to the level 3 BLAS (basic linear algebra subprograms), which are designed to perform very efficiently on parallel-vector computers. Similarly, the virtual LAPACK routines are based on a set of virtual Level 3 BLAS (VBLAS) routines. See the `INTRO_BLAS3(3S)` man page for more information about the Level 3 BLAS routines.

For example, `VSGEMM(3S)` (virtual single-precision general matrix multiplication) is the virtual matrix routine that corresponds to `SGEMM`, the BLAS routine for matrix multiplications.

The calling sequences of the virtual LAPACK and VBLAS routines are similar to those of the corresponding LAPACK and BLAS routines, but where the in-memory routines require an array argument, the virtual routines require an argument that specifies a virtual matrix.

5.3.6.1 Using Strassen's algorithm

Strassen's algorithm for matrix multiplication is a recursive algorithm that is slightly faster than the ordinary inner product algorithm. This additional speed is purchased at the expense of requiring some additional memory for intermediate workspace. Because the virtual linear algebra routines manage their own memory anyway, and perform their work on individual page size blocks, it is easy to use Strassen's algorithm everywhere that a matrix multiplication is required.

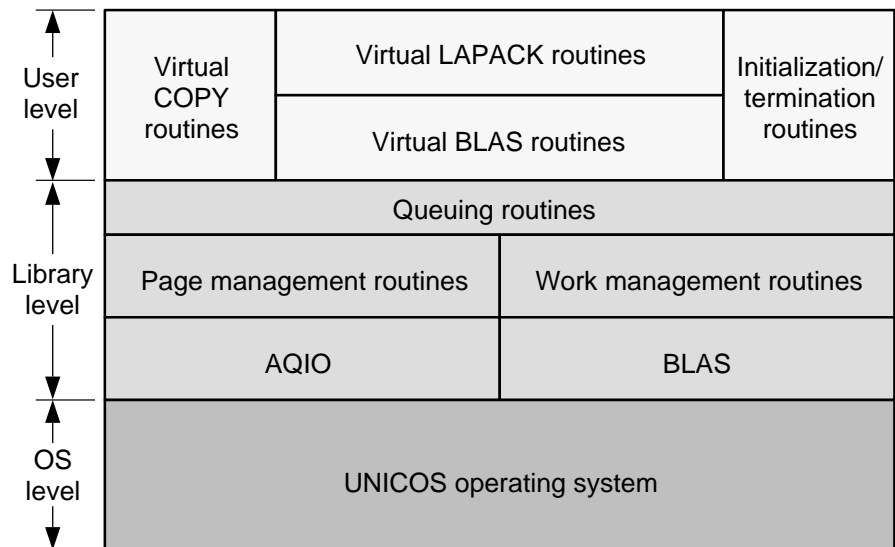
Strassen's algorithm performs the floating-point operations for matrix multiplication in an order that is very different than the usual vector method. In some cases, this could cause round-off problems, possibly leading to numerical differences in the result.

You choose whether to use Strassen's algorithm when calling `VBEGIN`, either by passing an argument to `VBEGIN`, or by setting the `VBLAS_STRASSEN` environment variable. If you use Strassen's algorithm, `VBEGIN` automatically

allocates the necessary workspace. In subsequent virtual matrix computations, Strassen's algorithm is then automatically used for all matrix multiplications, including matrix multiplications done as part of the VSGEMM and VSTRSM routines.

5.3.7 Lower-level Routines

The user-level routines are built on lower-level library routines that manage work request queues, active page queues, and other tasks. These routines, in turn, depend on the AQIO (asynchronous queued I/O) routines and the operating system routines. Figure 6 shows this layered software design.



a10040

Figure 6. Layered software design

Together, these out-of-core routines implement a *paged virtual memory system* that is implemented at the library level. It is highly efficient because the particular structure of problems in linear algebra permits an intelligent paging strategy.

For more details about the I/O routines, see the *Application Programmer's I/O Guide*.

5.4 Examples of Out-of-core Subroutine Use

The following short examples illustrate how you can use out-of-core subroutines to manipulate virtual matrices. For an explanation of the specific arguments for these subroutines, see the man pages for the individual routines.

Example 12: Creating a virtual matrix

The following program shows how you can use the `SCOPY2RV(3S)` routine to create a virtual matrix. A virtual matrix will be created on unit number 1 (which, by default, is on file `fort.1`). Within the program, this matrix is known as `V`, but `V` is just an integer variable that has a value of 1.

1. Call the `VBEGIN` routine to initialize the library routines.
2. Create a vector, `X`, of random numbers, and copy it to one column of the virtual matrix, using the `SCOPY2RV` routine.
3. Repeat this procedure for each column, `J`, of the virtual matrix.

The program for creating a virtual matrix is as follows:

```

INTEGER N
PARAMETER (N = 2000)
INTEGER V, W ! UNIT NUMBER OF THE VIRTUAL MATRICES
PARAMETER (V = 1, W = 2)
REAL X(N) ! VECTOR FOR STORING A COLUMN OF V
CALL VBEGIN
DO, J = 1, N
* CREATE A VECTOR OF RANDOM NUMBERS, AND STORE
* IT IN COLUMN J OF VIRTUAL MATRIX V.
    X = RANF()
    CALL SCOPY2RV(N, 1, X, N, V, 1, J, N)
END DO
*
* STORE THE FIRST COLUMN OF VIRTUAL W IN VECTOR X.
CALL SCOPY2VR(N, 1, W, 1, 1, N, X, N)
* PRINT THE VALUE OF THE FIRST ELEMENT
WRITE(*,*) 'VALUE OF W(1,1) = ', X(1)
CALL VEND
END

```

Example 13: Multiplying a virtual matrix

This example illustrates the virtual BLAS `VSGEMM` routine. It assumes that the virtual matrix `V` has been created on unit 1.

1. Multiply virtual matrix V by itself, creating virtual matrix W on unit number 2.
2. Copy the first column of matrix W into vector X .
3. Print the first element of X , which is the value of virtual matrix element $W(1,1)$.

A program to perform these steps and to multiply a virtual matrix follows.

```

INTEGER N
PARAMETER (N = 2000)
* V, W = UNIT NUMBERS OF THE VIRTUAL MATRICES
INTEGER V, W
PARAMETER (V = 1, W = 2)
* X IS A VECTOR FOR STORING A COLUMN OF V
REAL X(N)
CALL VBEGIN
* MULTIPLY VIRTUAL MATRIX V BY ITSELF,
* CREATING VIRTUAL MATRIX W = V*V
CALL VSGEMM('Notranspose', 'Notranspose', N, N, N,
            1.0, V, 1, 1, N, V, 1, 1, N, 0.0, W, 1, 1, N)
* STORE THE FIRST COLUMN OF VIRTUAL W IN VECTOR X.
CALL SCOPY2VR(N, 1, W, 1, 1, N, X, N)
* PRINT THE VALUE OF THE FIRST ELEMENT
WRITE(*,*) 'VALUE OF W(1,1) = ', X(1)
CALL VEND
END

```

Example 14: Example of protocol usage

A program to solve a large system of equations by using the virtual LAPACK routines might be organized according to the following general outline (it is assumed that you can generate the original matrix one row at a time by computing or reading it):

1. Call VBEGIN to initialize the virtual matrix routines.
2. For each row of the matrix, call the virtual copy routine, SCOPY2RV, to store the row in a virtual matrix.
3. Likewise, create a virtual matrix of right-hand sides.
4. Call the VSGETRF routine to factor the general matrix.
5. Call the VSGETRS routine to solve the right-hand sides.

6. For each column of the solution matrix, call the `SCOPY2VR(3S)` routine to fetch the solution vector and process it.
7. Call the `VEND(3S)` routine to terminate the virtual matrix routines.
8. Close the files.

5.5 UNICOS Environment Variables

The following list summarizes UNICOS environment variables used by Scientific Library routines.

<u>Variable</u>	<u>Definition</u>
NCPUS	Specifies the number of physical processors that are either on your system or which are available. The default value for NCPUS is usually the number of physical processors on the system. If you specify NCPUS to be greater than the number of physical processors available, you can create unnecessary overhead. If you specify NCPUS to be less than the number of available processors, your tasks will not execute as efficiently.
MP_DEDICATED	Determines the type of machine environment. If set to 1, it indicates you are running an application in a dedicated machine environment. Slave processors wait in user space rather than return to the operating system. If MP_DEDICATED is set to 0 or is not set at all, slave processors return to the operating system after waiting in user space. When MP_DEDICATED is set to a value other than 1 or 0, the behavior is undefined. If you set MP_DEDICATED to 1 in a nondedicated machine environment, you can degrade system throughput.
VBLAS_PAGESIZE	The dimension, in real words, of the size of a page. If np represents this value, the total number words on a page is $np \times np$. For example, if np equals 256, the default value, a page is 256×256 (65,536 words).

VBLAS_STATISTICS	If you set this environment variable, the <code>VEND(3S)</code> routine will print automatically, on <code>stdout</code> , a report on performance statistics.
VBLAS_STRASSEN	If you set this environment variable, all matrix multiplications will be done by using Strassen's algorithm for matrix multiplication.
VBLAS_WORKSPACE	Amount of memory, in (real 64-bit) words, used for page buffer space.

5.5.1 Multitasking

Like most routines in the Scientific Library, the virtual linear algebra routines perform multitasking automatically. You can control the use of multitasking by setting the value of the `NCPUS` environment variable to an integer that indicates the number of processors you want to use. For example, the following C shell command sets `NCPUS` equal to 1, indicating single-CPU execution (which effectively inhibits multitasking):

```
setenv NCPUS 1
```

Likewise, the following command means that the software will try to use four CPUs:

```
setenv NCPUS 4
```

The actual number of CPUs used depends on availability of resources.

When running on a dedicated system, you can set the `MP_DEDICATED` environment variable. A C shell command to set the variable follows:

```
setenv MP_DEDICATED
```

See the `INTRO_LIBSCI(3S)` man page for more information on multitasking in the Scientific Library.

5.6 Error Reporting

When the out-of-core software diagnoses an error, it writes an error diagnostic to the standard error file, `stderr`, and terminates. If the out-of-core routines themselves diagnose the error, the error message should be complete and self-explanatory. For instance, a common error is to provide an insufficient amount of memory for workspace. In this case, the error diagnostic will indicate how much memory was needed.

Example:

```
*** Error in routine: VBEGIN
*** Insufficient memory was given;
minimum required (decimal words) = 198144
```

If a lower-level system or library routine diagnosed the error, the diagnostic will include the error code. Usually, you can use the `explain(1)` command to obtain more information about the error by typing one of the following commands:

```
explain sys-xxx
or
```

```
explain lib-xxx
```

The `xxx` argument represents the error code listed in the diagnostic. Use `explain sys` for error status codes numbered less than 100, and `explain lib` for higher-numbered codes.

For example, suppose that unit 1 was assigned to file `/tmp/xxx/yyy/zzz`, by using the following command:

```
assign -a /tmp/xxx/yyy/zzz u:1
```

But suppose that the `/tmp/xxx/yyy` directory was not created. When the VBLAS routine tries to create the file, it cannot, and it aborts after printing the message:

```
*** Error in routine: page_request
*** Error status on AQOPEN for unit number: 1
*** Error status on AQOPEN = -2
```

Because AQIO routines are used internally for input and output, `AQOPEN(3F)`, `AQREAD(3F)`, or `AQWRITE(3F)` usually detects the error. In this case, it was `AQOPEN`. Of more concern, however, is the specific error status. The message indicates that the error occurred on unit number 1, and that the error status code was -2. You can type the `explain sys-2` command, which prints a further description that explains that one of the directories in a path name does not exist. See the `explain(1)` man page for further information.

5.7 Performance Measurement and Tuning

The out-of-core software has a built-in feature for performance measurement, and it automatically collects various VBLAS performance statistics. To print out

these statistics when the `VEND(3S)` routine is called, provide a nonzero argument to the `VEND` routine or set the `VBLAS_STATISTICS` environment variable.

The statistics reported include the following:

- Total elapsed time
- Total CPU time
- Total I/O wait time
- Total workspace used
- Number of words read and written
- A distribution of wait times

You can use this feature in addition to the usual UNICOS performance tools.

See Section 5.8, page 119, for a sample output listing of the statistics report.

5.7.1 Page-Buffer Space

The most important tuning parameter for the VBLAS routines is the value of *nwork*, the amount of page-buffer space. This value is set either as an explicit argument to `VBEGIN(3S)` or by setting the `VBLAS_WORKSPACE` environment variable, prior to calling `VBEGIN`. (See Section 5.5, page 115, for a summary of the UNICOS environment variables that are relevant to performance tuning of the out-of-core routines.)

Note: The *nwork* argument does not usually affect CPU time; only I/O wait time and total elapsed time (wall-clock time) are affected.

5.7.2 Memory Usage Guidelines

As always with out-of-core techniques, a trade-off exists between performance and size. If you use more memory, performance will be better, but the program size increases.

It is difficult to give firm rules for how much memory is needed, but the following are some guidelines:

- The absolute minimum amount of page-buffer space required for the VBLAS routines must be enough to contain three pages.

- If the virtual matrix is on disk, more buffer space is used and I/O performance is increased (at least up to a point).
- If running in a dedicated environment, use as much available memory as possible.
- If running in a production environment, use less memory so that you can schedule and run the job at the same time that your other jobs are running. The turnaround time of a smaller job might be much less than for a large job, even if the I/O wait time for the smaller job is larger.
- A general guideline for optimal performance is to use enough buffer space for one column of pages (that is, $n \times np$ words; np is the number of columns per page, and n is the leading dimension of the matrix (rounded up to a multiple of np)). If you use as much as twice this memory, performance will improve.
- If the virtual matrix is SSD resident, you need much less buffer space to obtain good performance.
- The use of Strassen's algorithm (see Section 5.3.6.1, page 111) usually speeds the computation for a small increase in memory. The amount of memory that Strassen's algorithm uses is reported in the VBLAS statistics (see Section 5.8, page 119).
- Use packed storage mode (see Section 5.2.6, page 105) when appropriate because it saves disk space with no penalty in CPU time.

5.7.3 Memory Requirement for `VSGETRF` and `VCGETRF` Routines

For solution of a general matrix (with `VSGETRF` and `VCGETRF`), a special memory requirement exists. These routines need enough buffer space to contain one column of pages. If the matrix is 5000-by-5000, the buffer space must be 5000-by-256 if the page size is 256. This requirement is necessary because of the nature of Gaussian elimination with partial pivoting. To do the pivots, the performance may be extremely poor if you use less memory.

5.8 Sample Performance Statistics

The following is actual output produced by the VBLAS routines when `VEND` is called with a nonzero argument (or when the `VBLAS_STATISTICS` environment variable is set). All times are in hours (h), minutes (m), and seconds (s).

```

--- Time ---
Began execution (time of call to VBEGIN)           12/29/92 16:17:20
Ended execution (time of call to VEND)             12/29/92 16:39:05

Total wall clock time                             21m 45.635s
Total CPU time                                    15m 39.177s
Total I/O wait time                               1m 58.033s  0.0%
Strassen's algorithm used?                       yes

Total real time in BLAS3 routines                 16m 29.345s  75.8%
Total CPU time in BLAS3 routines                  15m 30.916s  99.1%
Total real time in copy routines                  0.488s  0.0%
Total CPU time in copy routines                   0.450s  0.0%
Total real time in swap routines                  0.000s  0.0%
Total CPU time in swap routines                   0.000s  0.0%

--- Memory Used for Data Structures (words) ---

Workspace for page frames                         1056768
Workspace for Strassen's algorithm                153354
Workspace for other data structures                5578
Total workspace for VBLAS                         1215700

--- Paging ---

Page size (side)                                 256
Number of pages in page queue                     16
Number of page requests                           39512
Number of page hits                               12677      32.1%
Number of page misses                             26835      67.9%
Number of pages read                              26835      67.9%
Number of zero pages allocated                     0          0.0%
Number of pages for which read was unnecessary    0          0.0%
Number of pages written                           7193      18.2%

--- I/O Performance ---

Number of words read                              1.7587E+09
Total time waiting for read                       1m 57.909s  99.9%
Average time per page waiting for read            0.004s
Average time per page in request read queue       0.606s
Ratio (read queue time)/(read wait time)          137.993
Average read request queue throughput (Mw/s)      0.108

```

```

Virtual read rate (Mwords/second)          14.915

Number of words written                    4.7140E+08
Total time waiting for write              0.123s 0.1%
Average time per page waiting for write   0.000s
Average time per page in request write queue 0.074s
Ratio (write queue time)/(write wait time) 4339.977
Average write request queue throughput (Mw/s) 0.882
    
```

```

Virtual write rate (Mwords/second)        3826.032
    
```

AQSTAT Wait Time Distribution (READ)

```

-----
Range (microsecond)    Calls    Pct.    Seconds    Pct.
-----
10** 0 <= t < 10** 1  0      0.0     0.00 E+00  0.0
10** 1 <= t < 10** 2  26625  99.2    4.23E-01  0.4
10** 2 <= t < 10** 3   25  0.1    9.11E-03  0.0
10** 3 <= t < 10** 4   13  0.0    2.85E-02  0.0
10** 4 <= t < 10** 5   44  0.2    2.56E+00  2.2
10** 5 <= t < 10** 6  112  0.4    7.47E+01  63.4
10** 6 <= t < 10** 7   16  0.1    4.02E+01  34.1
10** 7 <= t < 10** 8    0  0.0    0.00E+00  0.0
Total                  26835  1.18E+02
    
```

AQSTAT Wait Time Distribution (WRITE)

```

-----
Range (microseconds)    Calls    Pct.    Seconds    Pct.
-----
10** 0 <= t < 10** 1  0      0.0     0.00E+00  0.0
10** 1 <= t < 10** 2  7178  99.8    1.14E-01  92.4
10** 2 <= t < 10** 3   12     0.2     2.68E-03  2.2
10** 3 <= t < 10** 4    3     0.0     6.62E-03  5.4
10** 4 <= t < 10** 5    0     0.0     0.00E+00  0.0
10** 5 <= t < 10** 6    0     0.0     0.00E+00  0.0
10** 6 <= t < 10** 7    0     0.0     0.00E+00  0.0
10** 7 <= t < 10** 8    0     0.0     0.00E+00  0.0
Total                  7193  1.23E-01
    
```


Appendix A: libm Version 2 [A]

This appendix describes `libm` version 2, the default UNICOS Math Library.

A.1 Overview of Math Libraries

A number of algorithms have been developed to more accurately compute single-precision (64-bit) elementary functions for all UNICOS systems with compressed index and gather/scatter hardware. The goal is improve the quality of results in some user applications, especially those that have, in the past, shown great sensitivity to minor numerical variations in the low-order bits of machine floating-point arithmetic.

The following list shows the math functions that employ these algorithms:

<code>ALOG, ALOG10</code>	<code>ASIN, ACOS</code>
<code>EXP</code>	<code>ATAN, ATAN2</code>
<code>SIN, COS, COSS</code>	<code>SINH, COSH, COSSH</code>
<code>TAN, COT</code>	<code>TANH</code>
<code>SQRT, CBRT</code>	<code>POWER(Y^Y)</code>

`libm` version 2 (`libmv2`) is the default Math Library for UNICOS systems. A link exists between `libm` and `libmv2` so that users can change their accesses as needed. On Cray C90 systems running in C90 mode, only `libmv2` is available.

`libmv2` makes full use of UNICOS system features, such as dual-memory ports, chaining, second vector logical unit, vector recursion, and truncated (chopped) floating addition.

The new functions have very high scalar and vector performance that is only slightly less than that of `libm` version 1. In addition, the ratio of vector to scalar performance remains about 8 to 1. The accuracy of the new functions is such that the computed machine result is always one of the two machine-representable numbers nearest to the infinite-precision result. In all cases, the scalar and vector functions, when given bit-wise identical arguments, will obtain bit-wise identical results. Accurate results are obtained using the existing floating-point hardware. No double-precision arithmetic operations are used.

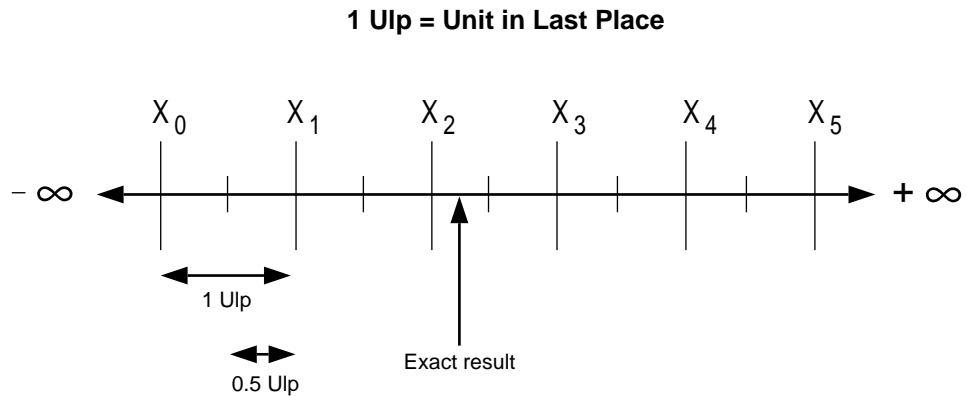
The techniques used in deriving the new algorithms consist principally of extremely careful argument reduction, table lookup algorithms,

pseudo-extended-precision arithmetic, and software rounding to produce a final, correct single-precision result from an intermediate extended-precision quantity.

A.1.1 The 1 ULP Criterion

Because standards (such as IEEE-754) of accuracy are now available for various types of computer hardware, it is only a matter of time before similar standards will be proposed for software, especially elementary functions. `libm` was written in anticipation of this trend.

The concept of the ULP (Unit in Last Place) is useful as full-machine precision is approached. A ULP is the spacing between the adjacent floating-point numbers, as illustrated in the following figure.

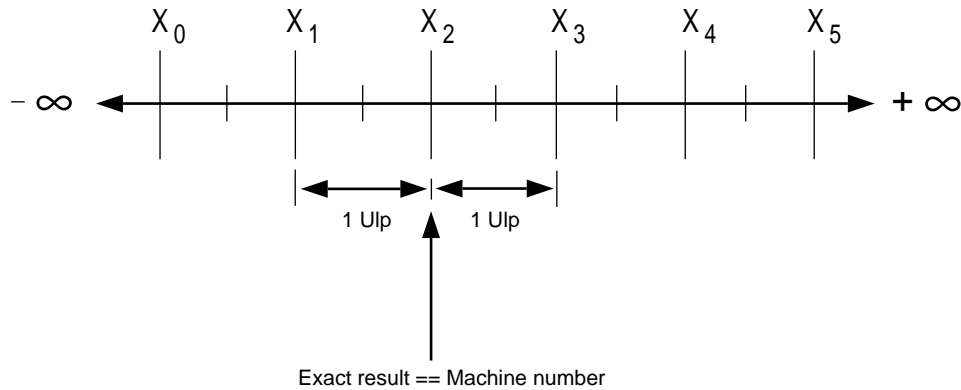


X_2 = Nearest machine number (error ≤ 0.5 Ulp)
 X_3 = Next nearest machine number (error < 1.0 Ulp)

a10532

Here the exact mathematical result, indicated by the vertical arrow, lies between two machine numbers, X_2 and X_3 . If a function algorithm for computing X has an error of $\epsilon \leq \frac{1}{2}$ ULP, X_2 is returned. If the function has the more relaxed tolerance of $\epsilon < 1$ ULP, either X_2 or X_3 can be returned.

The 1 ULP error tolerance has some useful properties, as shown in the following figure. Here the exact result, again shown by the vertical arrow, happens to coincide with X_2 , and the two nearest machine numbers, X_1 and X_3 , are a distance of 1 ULP away.



Exact results are produced when they are machine representable
Zero average bias, no skewed error distribution

a10533

In this case, given an error of $\epsilon < 1$ ULP, only X_2 can be returned. That is, whenever the argument and the result of a function happen to be exactly machine representable, it is the **exact** result that is returned. Also, if almost all results have $\epsilon \leq \frac{1}{2}$ ULP, the results are unbiased (that is, they do not contribute to an accumulation of roundoff errors that would eventually skew the results either high or low).

Here are some examples of the type of exact results that can be expected from a library that has an error $\epsilon < 1$ ULP:

$$\begin{aligned}\sqrt{100.00} &= 10.0 \text{ (exactly, not } 9.9999999) \\ 100.00^{0.5} &= 10.0 \\ 10.0^{2.00} &= 100.0 \\ \log_{10}(100.0) &= 2.0 \\ e^0 &= 1.0 \\ x^{1.0} &= x \text{ (exactly, for any } x)\end{aligned}$$

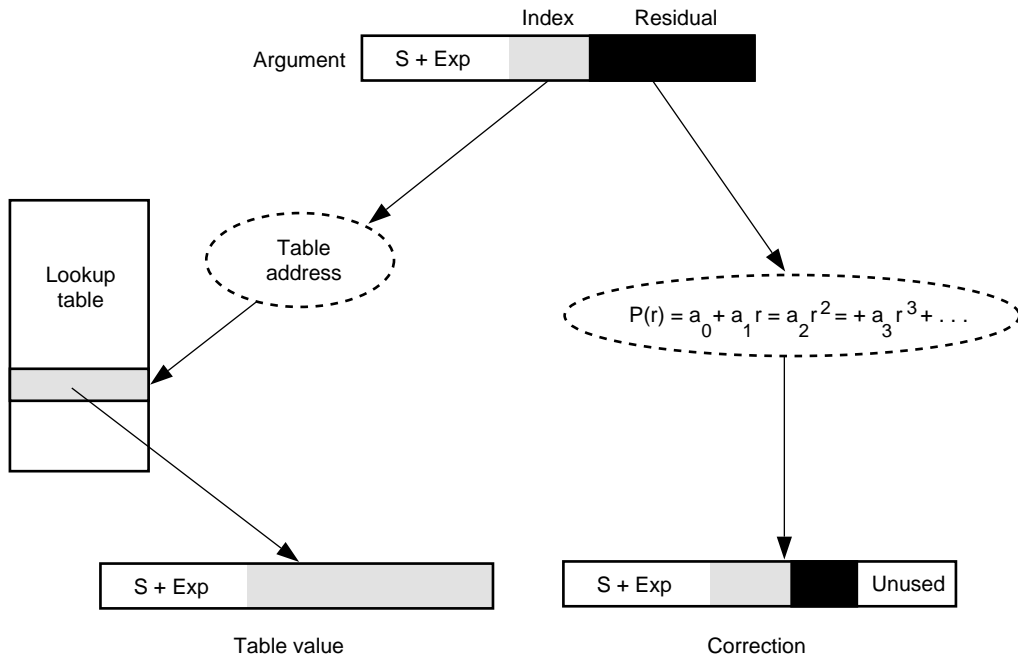
(A.1)

A.1.2 Numerical Methods

Two basic numerical methods were useful in this work. The first consisted of the traditional methods of polynomial approximation as described in Cody, or

Hart and Cheney. The second involved the use of a table lookup followed by a correction, as done in the IBM library for their 370 and 3090 series machines. Although both approaches produced accurate results for most functions, the table lookup method was faster and simpler. Both methods required the use of machine-dependent tricks in order to compute intermediate quantities to more than single precision. Only the table lookup method is discussed here because it was used most often.

Table Lookup Method



Use upper bits of argument as index to lookup table
 Compute correction term using lower bits of argument
 The larger the lookup table, the simpler (and *faster*) the correction
 Table lookup and correction can usually be executed in parallel

a10534

This figure diagrams the table lookup method in its simplest form. The function argument is reduced to a manageable range by using very careful argument reduction techniques. The reduced argument is then split into two pieces, with

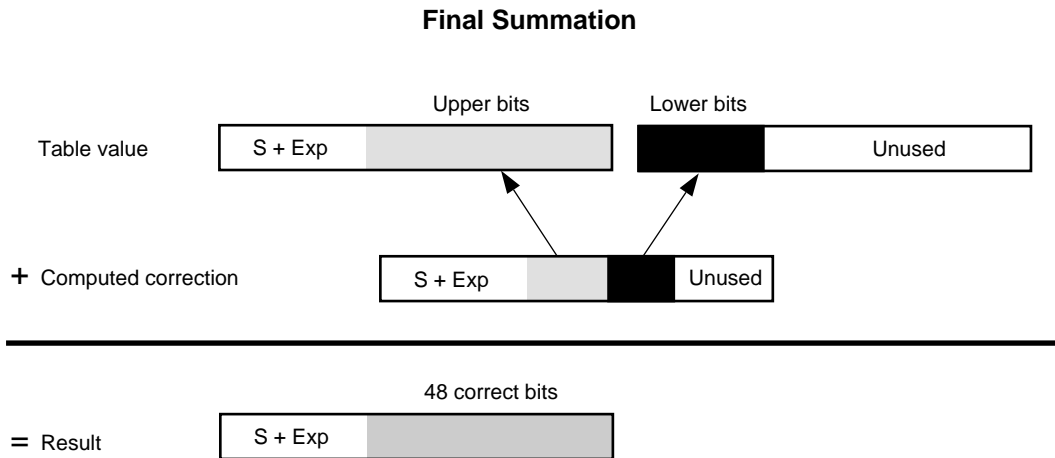
the upper (most significant) bits becoming an index into a lookup table, and the lower bits being used to compute a correction factor.

All of this assumes that a relation of the form,
 $F(x) = F(x_0 + \delta x) = T(x_0) + P(r)$, can be found, where $r = r(x_0, \delta x)$.

The reduced argument r is usually a function of $x_0 - \delta x$. The correction term $P(r)$ is a polynomial that depends on the original argument and, occasionally, on the table lookup value.

This form has several properties that make the method useful. First, in most cases, the correction term $P(r)$ can be computed independently of the table lookup term $T(x_0)$. This parallelism improves performance because the memory access can be overlapped with the CPU computation of $P(r)$.

Second, the correction term $P(r)$ need not be computed to full precision; only the uppermost significant bits are needed, as show below. This feature is crucial to obtaining accurate results because the correction term can be computed using the single-precision hardware, which need not be accurate to full 48-bit precision.



98-99% of results are closest machine number to the correct answer
 Table values are generated once and stored in static common blocks

a10535

The values in the lookup table can be computed in double precision, packed to contain more than 48 bits of precision, and stored in a common block internal to the library. The computation is free because the results are available by using a memory reference at run time.

A.1.3 Side Effects

Potential side effects from the use of `libmv2` consist primarily of small changes in answers due to increased accuracy. Codes that, in the past, have been sensitive to small errors in the least significant bits may be affected. In addition to the new functions listed previously, some of the complex functions are indirectly affected because they call the real functions. Execution timings vary slightly, and they are data-dependent (for vectors) due to possible bank-busy conflicts encountered when accessing the lookup tables.

These changes do not affect double-precision functions (with the notable exception of `DSQRT`, which calls `SQRT`). There are no plans to achieve a double-precision library to 1 ULP accuracy. Double precision is already extremely slow and exists mostly to assist in computation where single precision is not quite accurate enough.

Appendix B: Math Algorithms [B]

This appendix describes the algorithms used by the functions in the Math Library (libm). The “procedures” in this section detail the various steps used in the algorithms.

B.1 Single-precision Real Logarithm Functions $\ln(x)$ and $\log(x)$

Procedure 1: $\ln(x)$ and $\log(x)$

1. The function $\ln(x)$ is the base e logarithm called as the Fortran intrinsic `ALOG(x)`. The function $\log(x)$ (or $\log_{10}(x)$) is the base 10 logarithm called as the Fortran intrinsic `ALOG10(x)`. Both functions are computed using the algorithm for $\ln(x)$ with minor changes required for $\log(x)$ indicated where needed.
2. On entry to $\ln(x)$, if $x \leq 0$, a fatal error occurs. Otherwise the argument x can be any positive normalized floating-point number. For the vectorized versions of $\ln(x)$, if **any** element of the input vector x is 0 or negative, the vector \log function aborts even if the remaining elements of the x vector are legal.
3. Let argument $x = 2^m \times g$, where m is the unbiased exponent, g is the mantissa, and $\frac{1}{2} \leq g < 1$. Then:

$$\begin{aligned}\ln(x) &= \ln(2^m) + \ln(g) \\ &= m \ln(2) + \ln(g) \\ &= L_u + L_l\end{aligned}$$

(B.1)

where L_u is the most significant portion of $\ln(x)$, and L_l is the least significant part of $\ln(x)$, and $L_u + L_l = \ln(x)$ to more than single precision. The evaluation of $\ln(g)$ is done using a table lookup method. The value of g is first split into two pieces, $g = g_0 + dg$, with the uppermost 9 significant bits (g_0) used as an index to an extended-precision lookup table. Then the following mathematical identity is used:

$$\begin{aligned}
 \ln(g) &= \ln(g_0 + dg) \\
 &= \ln(g_0) + \ln\left(1 + \frac{dg}{g_0}\right) \\
 &= \ln(g_0) + \ln(1 + z)
 \end{aligned}
 \tag{B.2}$$

where

$$z \equiv \frac{dg}{g_0} \tag{B.3}$$

and where the resolution of the lookup table was chosen such that $\frac{1}{g_0}$ can be done to sufficient accuracy using the hardware reciprocal iteration. Then, using the series expansion for $\ln(1 + z)$, if entry $\ln(x)$

$$\ln(1 + z) = z + p_2z^2 + p_3z^3 + p_4z^4 + p_5z^5 + p_6z^6 \tag{B.4}$$

If entry $\log(x)$

$$\log_{10}(1 + z) = q_1z + q_2z^2 + q_3z^3 + q_4z^4 + q_5z^5 + q_6z^6 \tag{B.5}$$

The coefficients P_i and Q_i are computed from a minimax method especially for this algorithm. Thus, Equation B.2 becomes

$$\begin{aligned}
 \ln(g) &= T_u + T_l + [z + z^2P(z)] \\
 &= S_u + S_l
 \end{aligned}
 \tag{B.6}$$

where $T_u + T_l = \ln(g_0)$ to more than single precision, and it is taken from the lookup table. For the entry $\log(x)$ the value is $T_u + T_l = \log_{10}(g_0)$. The extra bits from T_l are packed into an unused portion of the exponent field of T_u so that each element of the lookup table will fit into one Cray 64-bit machine word. The terms are combined carefully to avoid loss of precision due to truncation and roundoff errors. The result, $S_u + S_l = \ln(g)$ to more

than single precision. The values of S_u and S_l are now substituted for $\ln(g)$ into Equation B.1 to obtain the following:

$$\begin{aligned}\ln(x) &= m \ln(2) + \ln(g) = m \ln(2) + \ln(g_0 + dg) \\ &= m \ln(2) + S_u + S_l \\ &= L_u + L_l\end{aligned}\tag{B.7}$$

The product $m \ln(2)$ (or $m \log_{10}(2)$ if entry is $\log(x)$) is computed to more than single precision. The result equals $\ln(x)$ to within 1 ULP (Unit in Last Place), that is, almost full single precision.

If the original value of x is close to 1, that is $|x - 1| \leq 2^{-8}$, then the preceding method is not used. Instead $z \equiv x - 1$ is set and Equation B.4 and Equation B.5 are used to obtain the following:

$$\begin{aligned}\ln(x) &\equiv \ln(1 + z) \\ &= z + p_2 z^2 + p_3 z^3 + p_4 z^4 + p_5 z^5 + p_6 z^6 \quad \text{for } \ln \\ &= q_1 z + q_2 z^2 + q_3 z^3 + q_4 z^4 + q_5 z^5 + q_6 z^6 \quad \text{for } \log \\ &= L_u + L_l\end{aligned}\tag{B.8}$$

The coefficients P_i and Q_i are the same as in Equation B.4, and Equation B.5, respectively. The limits on $|x - 1|$ were chosen such that, for either method, $\ln(x)$ is accurate to within 1 ULP to obtain almost full single-precision accuracy. In all cases, the final addition is done using software rounding to obtain a correct single-precision result.

B.1.1 Accuracy

Extensive testing with various sets of 10^5 random arguments shows that, in the range $0 < x \leq 25$, the function $\ln(x)$ is 99.9% exact with a maximum error of .66 ULPs. On the range $25 < x < 10^{2466}$ the function is (apparently) 100% exact with a maximum error of 0.50 ULPs. For the function $\log(x)$, on the range $0 < x < 10^{2466}$, the result is 99.9% exact with a maximum error of 0.63 ULPs.

B.2 Single-precision Real Logarithm Functions $\text{ALOG}(x)$ and $\text{ALOG10}(x)$

Procedure 2: $\text{ALOG}(x)$ and $\text{ALOG10}(x)$

1. The function $\ln(x)$ is the base e logarithm called as the Fortran intrinsic $\text{ALOG}(x)$. The function $\log(x)$ (or $\log_{10}(x)$) is the base 10 logarithm called as the Fortran intrinsic $\text{ALOG10}(x)$. Both functions are computed using the algorithm for $\ln(x)$ with minor changes required for $\log(x)$ indicated where needed.
2. If $x \leq 0$, a fatal error occurs. Otherwise the argument x can be any positive floating-point number and the function $\ln(x)$ is computed by one of the following methods.
3. If the value of x is close to 1, for example $e^{-\frac{1}{32}} < x < e^{\frac{1}{32}}$, then a power series method is used. Let $f \equiv x - 1$ which can be done exactly, and also define:

$$\alpha \equiv \frac{2f}{2+f}$$

$$= \frac{f^2}{2+f}$$

$$\alpha \equiv \alpha_1 + \alpha_2$$

(B.9)

Then the result is:

$$\begin{aligned} \ln(x) &\equiv \ln(1+f) = \ln \left[\frac{1 + \frac{\alpha}{2}}{1 - \frac{\alpha}{2}} \right] \\ &\equiv \alpha_1 + (\alpha_2 + p_0\alpha^3 + p_1\alpha^5 + p_2\alpha^7) \\ &\equiv S_u + S_l \end{aligned}$$

(B.10)

The coefficients p_i are chosen by a minimax approximation. The result is $S_u + S_l = \ln(x)$ to more than working precision.

4. If the argument x is outside the range $e^{-\frac{1}{32}} < x < e^{\frac{1}{32}}$ then we use a table lookup method. Let the argument $x = 2^m \times g$ where m is the unbiased exponent, g is the mantissa, and $1 \leq g < 2$. Split g further as $g = F + f$ where $F \equiv 1 + i2^{-7}$, $i = 0, 1, \dots, 2^7$, and is taken from the leading 7 bits of g . Then compute the following:

$$\begin{aligned}\ln(x) &= \ln(2^m) + \ln(F + f) \\ &= m \ln(2) + \ln(F) + \ln\left(1 + \frac{f}{F}\right)\end{aligned}$$

(B.11)

5. The evaluation of $m \ln(2)$ in Equation B.11 is done to more than working precision by storing the constant $\ln(2)$ in two words, $\ln_2u + \ln_2l = \ln(2)$. The upper constant \ln_2u has enough trailing zeros that the product $m \cdot \ln_2u$ can be computed exactly in working precision for all values of m .
6. The evaluation of $\ln(F)$ in Equation B.11 is done using a table lookup method:

$$\ln(F) = \ln(1 + i2^{-7}) = T_u + T_l$$

(B.12)

where $T_u + T_l = \ln(F)$ to more than working precision. Each upper word of the table T_u has enough trailing zeroes in its mantissa so that the sum $m \cdot \ln_2u + T_u$ can be computed exactly in working precision for all values of i and m .

7. The final term in Equation B.11, $\ln\left(1 + \frac{f}{F}\right)$, is computed by the same power series method as in Procedure 2, step 3, page 132. Define

$$\beta \equiv \frac{2f}{2F + f}$$

(B.13)

then the term $\ln\left(1 + \frac{f}{F}\right)$ can be evaluated as:

$$\begin{aligned}\ln\left(1 + \frac{f}{F}\right) &\equiv \beta + p_0\beta^3 + p_1\beta^5 + p_2\beta^7 \\ &= P_u + P_l\end{aligned}$$

(B.14)

The coefficients p_i are the same as in Procedure 2, step 3, page 132. Note that in this case it is not necessary to split the leading β term as was done for α in Equation B.9.

8. The terms in Equation B.11 must be combined carefully to compute the final value of $\ln(x)$. Define the following:

$$\begin{aligned} L_u &\equiv m \ln_2 u + T_u \text{ exact} \\ L_l &\equiv m \ln_2 l + T_l \end{aligned} \tag{B.15}$$

Combining Equation B.11 with Equation B.12, Equation B.14, and Equation B.15 gives the following:

$$\begin{aligned} \ln(x) &= m \ln(2) + \ln(F) + \ln\left(1 + \frac{f}{F}\right) \\ &= L_u + [P_u + (P_l + L_l)] \end{aligned} \tag{B.16}$$

$$\equiv S_u + S_l \tag{B.17}$$

where $S_u + S_l = \ln(x)$ to more than working precision.

9. For the entry `ALOG10(x)`, the result is found by computing

$$\begin{aligned} \log(x) &\equiv \frac{\ln(x)}{\ln(10)} \\ &= (S_u + S_l) \times (\ln_{10_invu} + \ln_{10_invl}) \end{aligned} \tag{B.18}$$

using the results S_u and S_l from either Equation B.10, or Equation B.17. The multiplication is done by splitting each working precision number into two pieces, a “head” and a “tail,” and carefully forming the product.

B.2.1 Accuracy

Extensive testing with various sets of 10^5 random arguments shows that, in the range $0 < x \leq \infty$, the function $\ln(x)$ is around 99% exact with a maximum error of 0.7 ULPs. For the function $\log(x)$ on the range $0 < x \leq \infty$, the result is around 99% exact with a maximum error of 0.8 ULPs. This algorithm is a variation of the method described in P.T.P. Tang, “Table-driven implementation

of the Logarithm Function in IEEE Floating-point Arithmetic,” *ACM Transactions on Mathematical Software*, Vol. 16, No. 4, Dec. 1990, pp. 378–400.

B.3 Single-precision Real ASIN(x) and ACOS(x) Functions

Procedure 3: ASIN(x)

1. The computation of $\text{asin}(x)$ uses the absolute value of the argument. Let $f = |x|$ and use the identity:

$$\text{asin}(-x) = -\text{asin}(x) \tag{B.19}$$

If $f > 1$, a fatal error occurs. Otherwise, evaluate $\text{asin}(x)$ according to one of the following methods, depending on the magnitude of f .

2. On the range $f = [0, \frac{1}{2}]$, evaluate the following:

$$\text{asin}(f) = f + fP(t) \tag{B.20}$$

where $t = f^2$, $P(t)$ is a minimax polynomial of the form

$$P(t) = p_1t + p_2t^2 + \dots + p_{12}t^{12} \tag{B.21}$$

and the coefficients $p_1 \dots p_{12}$ are derived especially for this algorithm.

3. On the range $f = [\frac{1}{2}, 1]$, use the following identity:

$$\text{asin}(f) = \frac{\pi}{2} - 2 \text{asin} \left[\sqrt{\frac{1-f}{2}} \right] \tag{B.22}$$

Perform argument reduction by letting $t = \frac{1-f}{2}$, which is evaluated carefully to avoid roundoff problems near $f=1$ as:

$$t = \frac{\left(\frac{1}{2} - f + \frac{1}{2}\right)}{2} \tag{B.23}$$

Even though the value of t is exact, the square root must be done in extended precision as:

$$Y = 2\sqrt{t} = Y_u + Y_l \tag{B.24}$$

where $Y_u + Y_l$ represents $2\sqrt{t}$ to more than single precision. Likewise, the constant $\frac{\pi}{2}$ is needed to more than single precision as:

$$\frac{\pi}{2} = \left(\frac{\pi}{2}\right)_u + \left(\frac{\pi}{2}\right)_l \tag{B.25}$$

Equation B.22 for the $\text{asin}(f)$ can then be evaluated using the same polynomial as in Procedure 3, step 2, page 135,

$$\begin{aligned} \text{asin}(f) &= \frac{\pi}{2} - 2 \text{asin}(t) \\ &= \left(\frac{\pi}{2}\right)_u - Y_u - Y_u P(t) - Y_l + \left(\frac{\pi}{2}\right)_l \end{aligned} \tag{B.26}$$

where the terms are summed carefully to obtain full precision. The $Y_l P(t)$ can be neglected.

4. The sign of the result has the sign of argument x . The principal value of $\text{asin}(x)$ is the following on the range $-1 \leq x \leq +1$:

$$-\frac{\pi}{2} < \text{asin}(x) < \frac{\pi}{2} \tag{B.27}$$

Procedure 4: ACOS(x)

1. The arccosine is reduced to the computation of arcsine by the identity:

$$\operatorname{acos}(x) = \frac{\pi}{2} - \operatorname{asin}(x) \quad (\text{B.28})$$

The argument range $x = [-1, 1]$ is divided into four regions in order to achieve the required 1 ULP accuracy.

2. On the range $x = [0, \frac{1}{2}]$, let $f = |x|$ as in Procedure 3, step 1, page 135,

$$\operatorname{acos}(x) = \frac{\pi}{2} - \operatorname{asin}(f) \quad (\text{B.29})$$

where $\operatorname{asin}(f)$ is computed using Equation B.20.

3. On the range $x = [-\frac{1}{2}, 0]$, use

$$\operatorname{acos}(x) = \frac{\pi}{2} + \operatorname{asin}(f) \quad (\text{B.30})$$

where $\operatorname{asin}(f)$ is computed using Equation B.20.

4. On the range $x = [\frac{1}{2}, 1]$, use the identity

$$\operatorname{acos}(x) = 2 \operatorname{asin} \left[\sqrt{\frac{1-f}{2}} \right] \quad (\text{B.31})$$

with the argument reduction and square root done as in Procedure 3, step 3, page 135.

5. On the range $x = [-1, -\frac{1}{2}]$, use the following identity:

$$\operatorname{acos}(x) = \pi - 2 \operatorname{asin} \left[\sqrt{\frac{1-f}{2}} \right] \quad (\text{B.32})$$

with the argument reduction and square root done as in Procedure 3, step 3, page 135. The constant π is needed to extended precision as $\pi = \pi_u + \pi_l$.

6. The sign of the result is always positive. The principal value of $\text{acos}(x)$ is $0 \leq \text{acos}(x) < \pi$ on the argument range $-1 \leq x \leq +1$.

B.3.1 Accuracy

Extensive testing shows that the algorithms for $\text{ASIN}(x)$ and $\text{ACOS}(x)$ obtain the correct result approximately 99.2% of the time, with the largest error less than 0.63 ULP. Testing was done using various sets of up to 250,000 random numbers distributed linearly over the entire range of legal arguments from $-1 \leq x \leq 1$.

B.4 Single-precision Real $\text{ATAN}(x)$ Function

Procedure 5: $\text{ATAN}(x)$

1. The argument for $\text{atan}(x)$ can be any valid floating-point number in the range $x = [-\infty, +\infty]$. Let $f = |x|$ and use the identity $\text{atan}(-x) = -\text{atan}(x)$ to put the correct sign in the result. Three algorithms are used, depending on the magnitude of f .
2. If $f < \frac{1}{16}$, compute $\text{atan}(f)$ using a minimax power series in $t = f^2$, $\text{atan}(f) = f + fP(t)$ where

$$P(t) = p_1t + p_2t^2 + \dots p_5t^5 \tag{B.33}$$

3. If $\frac{1}{16} \leq f < 16$, compute $\text{atan}(f)$ using a table lookup method as follows. Let f_0 be obtained from the exponent and uppermost 5 bits from the mantissa of f . Note that $\text{atan}(f_0)$ can be obtained from a lookup table,

$$T = \text{atan}(f_0) = T_u + T_l \tag{B.34}$$

where T_u and T_l represent $\text{atan}(f_0)$ to more than single precision. Now use the standard trigonometric identity

$$\text{atan}(f) = \text{atan}(f_0) + \text{atan}(\delta f) \tag{B.35}$$

where

$$\delta f = \frac{f - f_0}{1 + f_0 f} \quad |\delta f| < \frac{1}{32} \quad (\text{B.36})$$

and where δf and $\text{atan}(\delta f)$ need not be computed to full precision. The size of the lookup table was chosen so that the hardware division has sufficient accuracy for Equation B.36.

$\text{atan}(\delta f)$ is computed using the first three terms of the same power series as in Procedure 5, step 2, page 138, namely $\text{atan}(\delta f) = \delta f + \delta f Q(f)$ where $t = (\delta f)^2$ and $Q(t) = p_1 t + p_2 t^2 + p_3 t^3$ using the same p_1 , p_2 , and p_3 as in Equation B.33. Equation B.35 then becomes

$$\begin{aligned} \text{atan}(f) &= T_u + [T_l + \text{atan}(\delta f)] \\ &= T_2 + [T_l + \delta f + \delta f Q(t)] \end{aligned} \quad (\text{B.37})$$

The terms are summed carefully to preserve accuracy.

4. If $f \geq 16$, then the identity $\text{atan}(f) = \frac{\pi}{2} - \text{atan}\left(\frac{1}{f}\right)$ is used where the inverse $\left(\frac{1}{f}\right)$ can again be done to sufficient accuracy using the hardware reciprocal approximation unit. The $\text{atan}\left(\frac{1}{f}\right)$ term is computed by replacing f with $\frac{1}{f}$ and using the power series in Procedure 5, step 2, page 138.
5. A software rounded addition is done for the final addition in all the preceding steps to produce a correctly rounded single-precision result. The sign of the result is the same as the sign of the argument x .

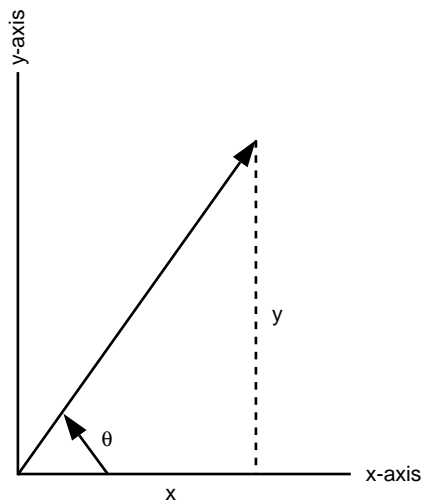
B.4.1 Accuracy

This routine was tested on various sets of 250,000 random arguments in all three ranges. It gives correctly rounded results approximately 99.9% of the time, with no error greater than 1 ULP. The largest observed error was about 0.55 ULP.

B.5 Single-precision ATAN(y,x) Function

Procedure 6: ATAN(y,x)

1. This function computes the angle that the point (x,y) makes with the positive x axis, and it is used to translate between polar and cartesian coordinates.



$$\tan(\theta) = y/x$$

$$\theta = \text{atan2}(y, x)$$

a10536

The arguments for $\text{ATAN2}(x,y)$ can be any valid floating-point numbers in the range $x = [-\infty, +\infty], y = [-\infty, +\infty]$. However, $\text{ATAN2}(0,0)$ is illegal and causes a fatal error. Let $X = |x|, Y = |y|$, and use the identities:

$$\text{atan2}(Y, X) = -\text{atan2}(-Y, X) \tag{B.38}$$

$$\text{atan2}(Y, -X) = \pi - \text{atan2}(Y, X) \tag{B.39}$$

Equation B.38, is used to put the correct sign in the result in the third and fourth quadrants. Equation B.39 is used to express the result in the second quadrant in terms of $\text{atan}(\frac{y}{x})$. The division of $\frac{Y}{X}$ needs to be done to more than single precision, so let $f = \frac{Y}{X}$ (upper) and $f_l = \frac{Y}{X}$ (lower) where $f + f_l = \frac{Y}{X}$ to more than single precision. Six different algorithms are used, depending on the magnitude of f and depending on the sign of x .

2. If $f \leq \frac{1}{16}$, compute $\text{atan}(f)$ using a minimax power series in $t = f^2$. If $x \geq 0$, then $\text{atan2}(y, x) = \text{atan}(f) = f + f_l + fP(t)$ and the $f_l P(t)$ term can be neglected.

If $x < 0$, then $\text{atan2}(y, x) = \pi - \text{atan}(f)$ where

$$P(t) = p_1 t + p_2 t^2 + \dots + p_5 t^5 \quad (\text{B.40})$$

where $\pi = \pi_u + \pi_l$ and where the subtraction from π is done carefully to avoid loss of precision.

3. If $\frac{1}{16} \leq f < 16$, compute $\text{atan}(f)$ using a table lookup method as follows. Let f_0 be obtained from the exponent and uppermost 5 bits from the mantissa of f . Note that $\text{atan}(f_0)$ can be obtained from a lookup table,

$$T = \text{atan}(f_0) = T_u + T_l \quad (x \geq 0) \quad (\text{B.41})$$

where T_u and T_l represent $\text{atan}(f_0)$ to more than single precision. If x is negative, then a similar lookup table is used:

$$T = \pi - \text{atan}(f_0) = T_u + T_l \quad (x < 0) \quad (\text{B.42})$$

Now use the standard trigonometric identity

$$\text{atan}(f) = \text{atan}(f_0) + \text{atan}(\delta f) \quad (\text{B.43})$$

where

$$\delta f = \frac{f - f_0 + f_l}{1 + f_0 f} \quad (\text{B.44})$$

$|\delta f| < \frac{1}{32}$ and where δf and $\text{atan}(\delta f)$ need not be computed to full precision. The size of the lookup table was chosen so that the hardware division has sufficient accuracy for Equation B.44.

$\text{atan}(\delta f)$ is computed using the first three terms of the same power series as in Procedure 6, step 2, page 140, namely $\text{atan}(\delta f) = \delta f + \delta f Q(t)$ where $t = (\delta f)^2$ and $Q(t) = p_1 t + p_2 t^2 + p_3 t^3$ using the same p_1 , p_2 , and p_3 as in Equation B.40. Equation B.43, then becomes

$$\begin{aligned}\text{atan}(f) &= T_u + [T_l + \text{atan}(\delta f)] \\ &= T_u + [T_l + \delta f + \delta f Q(t)]\end{aligned}\tag{B.45}$$

The terms are summed carefully to preserve accuracy.

If $x \geq 0$, the result is $\text{atan2}(y, x) = \text{atan}(f)$. If $x < 0$, the result is $\text{atan2}(y, x) = \pi - \text{atan}(f)$ where the subtraction is done carefully to avoid loss of precision. Table lookup values are obtained using Equation B.41, or Equation B.42.

4. If $f > 16$ and $x \geq 0$, then the identity $\text{atan2}(y, x) = \frac{\pi}{2} - \text{atan}\left(\frac{1}{f}\right)$ is used where the inverse $\left(\frac{1}{f}\right)$ can again be done to sufficient accuracy using the hardware reciprocal approximation unit.

If $f > 16$ and $x < 0$, then the identity

$$\begin{aligned}\text{atan2}(y, x) &= \pi - \left[\frac{\pi}{2} - \text{atan}\left(\frac{1}{f}\right) \right] \\ &= \frac{\pi}{2} + \text{atan}\left(\frac{1}{f}\right)\end{aligned}\tag{B.46}$$

is used where the inverse $\left(\frac{1}{f}\right)$ can again be done to sufficient accuracy using the hardware reciprocal approximation unit.

The $\text{atan}\left(\frac{1}{f}\right)$ term is computed by replacing f with $\frac{1}{f}$ and using the power series in Procedure 6, step 2, page 140.

If $x \geq 0$, as in $y \rightarrow \infty$, the result returned is the truncated value of $\frac{\pi}{2}$ so that the result stays in the first quadrant. If $x < 0$, then as $y \rightarrow \infty$, the result is the rounded value of $\frac{\pi}{2}$ so that the answer stays in the second quadrant. Likewise, as $y \rightarrow 0$ where $x < 0$, the result approaches the truncated value of π , and it stays in the second quadrant.

5. A software rounded addition is done for the final addition in all the preceding steps to produce a correctly rounded single-precision result. The sign of the result is the same as the sign of the argument y .

B.5.1 Accuracy

This routine was tested on various sets of 250,000 random arguments in all three ranges. It gives correctly rounded results approximately 99.6% of the time, with no error greater than 1 ULP. The largest observed error was approximately 0.54 ULP.

B.6 Single-precision Real CBRT(x) Function

Procedure 7: CBRT(x)

1. Let $f = \text{abs}(x)$ and choose the sign of the result by using $\text{cbrt}(-x) = -\text{cbrt}(x)$.
2. Express the argument f as $f = 2^n \cdot m$, where $\frac{1}{2} \leq m < 1$.

Let $k = \text{int}(\frac{n}{3})$ and $i = n - 3k$, $i = -2, -1, 0, 1$, or 2 . The result can be computed as the following:

$$\text{cbrt}(f) = \text{cbrt}(m) \cdot 2^{i/3} \cdot 2^k \tag{B.47}$$

$$\text{cbrt}(f_0) = \text{cbrt}(m) \cdot 2^{i/3} \tag{B.48}$$

The values of $2^{i/3}$ can be generated as precomputed constants, and the final multiplication by 2^k can be done by adjusting the exponent.

3. To find $\text{cbrt}(m)$, in the range $0.5 \leq m < 1$, obtain an initial approximation by the power series $\text{cbrt}(m) \approx p_0 + p_1 \cdot m + p_2 \cdot m^2 + \dots + p_5 \cdot m^5$ where the coefficients $p_{(0-5)}$ are found by a minimax method. This gives an approximation good to about six digits.

Now set $y_0 = \text{cbrt}(f_0)$ using Equation B.48. Next perform one Newton iteration in single precision using:

$$y_1 = y_0 + \frac{1}{3} \cdot \left(\frac{f_0}{y_0 \cdot y_0} - y_0 \right) \tag{B.49}$$

4. Then perform an additional Newton iteration in pseudo-extended precision by rewriting Equation B.49 slightly as:

$$y_2 = y_1 + \frac{1}{3} \cdot del \tag{B.50}$$

where

$$del = \frac{f_0 - y_1 \cdot y_1 \cdot y_1}{y_1 \cdot y_1} \tag{B.51}$$

The numerator must be computed carefully:

- a. The product $y_1 \cdot y_1$ must be done with pseudo-precision, in 24-bit pieces, and keeping the upper 48+24 bits of the result, such as $y_1 = A + B$ where A is the upper 24 bits, and B the lower, and $y_1 \cdot y_1 = C + D + E$ where C is the upper 24, D the middle 24, and E the lower 48 bits of the product.
 - b. The cube term, $y_1 \cdot y_1 \cdot y_1$ is approximated by the product $y_1^3 = (A + B) \cdot (C + D + E)$ so that $f_0 - y_1^3 = f_0 - AC - (AD + BC) - (AE + BD)$ and the $B \cdot E$ term is neglected. All the previous operations are done using single-precision hardware only.
 - c. The denominator, $\frac{1}{y_1 \cdot y_1}$, can be done using the ordinary single-precision product $y_1 \cdot y_1$, followed by a half-precision reciprocal approximation to compute del .
5. The final result is obtained by using the value of del in Equation B.50, as

$$y_2 = y_1 + \left[\frac{1}{3} \cdot del + ROUND \right] \tag{B.52}$$

where the *ROUND* factor is chosen to perform a rounded floating addition. Finally, the corrected sign is added to the result.

6. Next it is necessary to scale back by adding k to the exponent of y_2 .

B.6.1 Accuracy

This algorithm gives the correctly rounded result 100% of the time for all numbers tested, throughout the entire range of floating-point numbers.

B.7 Single-precision Real Exponential Function E^x

Procedure 8: e^x

1. If $x > 5.676875408785942 \times 10^3$, print an error message and abort. If $x < -5.678954850238224 \times 10^3$, quietly return 0. Otherwise, compute e^x as follows.
2. Let the argument be decomposed as

$$x_n = \frac{x}{\ln(2)} \tag{B.53}$$

where x_n is a real number consisting of an integer part n and 7 leading fraction bits $i = [0,127]$, plus $\frac{1}{256}$. Compute a reduced argument r as the following:

$$r = x - \left[n + \frac{i + \frac{1}{2}}{128} \right] \cdot \ln(2) \tag{B.54}$$

where $|r| < \frac{\ln(2)}{256}$. Then the result will be

$$\begin{aligned} e^x &= 2^n T e^r \\ &= 2^n \cdot [T + T \cdot (e^r - 1)] \end{aligned} \tag{B.55}$$

where $T = 2^{(i+1/2)/128}$

3. The argument reduction in Equation B.54 must be done in pseudo-double precision to preserve accuracy. Let

$c_1 = \ln(2)$ most significant 27 bits

$c_2 = \ln(2)$ next most significant 48 bits, and

$c_1 + c_2 = \ln(2)$ to more than single precision.

Equation B.54 is evaluated as:

$$r = x - \frac{x_n c_1}{2} - \frac{x_n c_1}{2} - x_n c_2 \tag{B.56}$$

The constant c_1 is chosen so that the product $x_n c_1$ can be computed exactly (with no hardware rounding) in a single word. The term $x_n c_1$ is subtracted in two steps to avoid loss of a bit when x is slightly less than a power of 2. The lower term $x_n c_2$ makes up for bits lost from cancellation from the first subtraction.

4. In Equation B.55 the factor of $2^{(i+1/2)/128}$ can be stored in a 128-word lookup table, indexed by i

$$T = 2^{\frac{(i+\frac{1}{2})}{128}} = T_u + T_l \tag{B.57}$$

where T_u has the uppermost 48 bits, T_l has 13 more bits, and $T_u + T_l = T$ to more than single precision. The T array is computed in double precision and stored in a static table.

It happens that all the exponents of T_u are 400001₈. Therefore, the extra 13 bits of T_l can be stored in the (unnecessary) bits in the exponent field. This allows the table lookup to be performed with only one memory reference for scalar e^x and with a single gather for vector e^x .

5. Computation of $e^r - 1$ is done using a minimax polynomial especially derived for this algorithm. Let $f(r) = e^r - 1 = r + r^2 Q(r)$ where $Q(r) = q_2 + q_3 r + q_4 r^2 + q_5 r^3$. The final summation to obtain $[T + T \times f(r)]$ is

$$W = T e^r = T_u + \{[T_u \times f(r) + T_l] + round\}$$

(B.58)

where *round* is a software correction term that depends on the exponent of T_u and the sign of $T_u \times f(r) + T_l$. The lowest-order term $T_l \times f(r)$ is omitted because it is negligible.

6. Final scaling is to multiply by 2^n by adding n to the exponent in Equation B.58.

$$e^x = 2^n W$$

(B.59)

B.7.1 Accuracy

Extensive testing shows that this algorithm obtains the correct result approximately 99.8% of the time, with the largest error less than 0.51 ULP. Testing was done using various sets of 10^4 random numbers distributed linearly and logarithmically over the entire range of legal arguments.

B.8 Single-precision Real Power Function x^y

Procedure 9: x^y

1. The function is evaluated according to the identity $z = x^y = e^{y \ln(x)}$. To achieve an accurate value of z for all possible ranges of x and y , it is necessary to evaluate the intermediate quantity $y \ln(x)$ to at least 13 bits more than single precision.
2. Before the computation, the routine checks for the following special cases:
 - if $x < 0$, a fatal error occurs
 - if $x = 0$ and $y < 0$, a fatal error occurs
 - if $x = 0$ and $y = 0$, a fatal error occurs
 - if $x = 0$ and $y > 0$, a result of 0.0 is returned
 - if $x > 0$ and $y = 0$, a result of 1.0 is returned

Routine x^y gives a fatal overflow error whenever the correct result would overflow. Routine x^y returns 0.0 with no warning message whenever the correct would underflow. For the vectorized versions of x^y , if **any** element of the inputs x or y would cause one of the preceding fatal errors, the vector

x^y function aborts, even if the remaining elements of the x and y vectors are legal.

3. Let $x = 2^m \times g$, where m is the unbiased exponent, g is the mantissa, and $\frac{1}{2} \leq g < 1$. Then:

$$\begin{aligned} \ln(x) &= \ln(2^m) + \ln(g) \\ &= m \ln(2) + \ln(g) \\ &= L_u + L_l \end{aligned}$$

(B.60)

where L_u is the most significant portion of $\ln(x)$, and L_l is the least significant part of $\ln(x)$ and $L_u + L_l = \ln(x)$ to 13 bits more than single precision. The evaluation of $\ln(g)$ is done using a table lookup method.

The value of g is first split into two pieces, $g = g_0 + dg$, with the uppermost 13 significant bits (g_0) used as an index to a double-precision lookup table. Then the following mathematical identity is used:

$$\begin{aligned} \ln(g) &= \ln(g_0 + dg) \\ &= \ln(g_0) + \ln\left(1 + \frac{dg}{g_0}\right) \\ &= \ln(g_0) + \ln(1 + z) \end{aligned}$$

(B.61)

where

$$\begin{aligned} z &\equiv \frac{dg}{g_0} \\ &= dg \times \left(\frac{1}{g_0}\right)_u + dg \times \left(\frac{1}{g_0}\right)_l \\ &= z_u + z_l \end{aligned}$$

(B.62)

and where the reciprocal $\frac{1}{g_0}$ is found from a lookup table computed to 15 bits more than single precision. Then, using the series expansion for $\ln(1 + z)$:

$$\ln(1+z) = [z_u + z_l + p_2 z^2 + p_3 z^3 + p_4 z^4 + p_5 z^5] \quad (\text{B.63})$$

Equation B.61 then becomes:

$$\begin{aligned} \ln(g) &= T_u + T_l + [z_u + z_l + z^2 P(z)] \\ &= S_u + S_l \end{aligned} \quad (\text{B.64})$$

$T_u + T_l = \ln(g_0)$ to more than single precision, and it is taken from the lookup table. The terms are combined carefully to avoid loss of precision due to truncation and roundoff errors. The result, $S_u + S_l = \ln(g)$ to more than single precision. The values of S_u and S_l are now substituted for $\ln(g)$ into Equation B.60 to obtain the following:

$$\begin{aligned} \ln(x) &= m \ln(2) + \ln(g) \\ &= m \ln(2) + \ln(g_0 + dg) \\ &= m \ln(2) + S_u + S_l \\ &= L_u + L_l \end{aligned} \quad (\text{B.65})$$

The product $m \ln(2)$ is computed to more than single precision. The result equals $\ln(x)$ to 13 bits more than single precision.

If the original value of x is close to 1, that is:

$$|x - 1| \leq 0.75 \times 2^{-14} \quad (\text{B.66})$$

then the preceding method is not used. Instead $z \equiv x - 1$ is set and Equation B.63 is used to obtain the following:

$$\begin{aligned} \ln(x) &\equiv \ln(1+z) \\ &= [z + p_2 z^2 + p_3 z^3 + p_4 z^4 + p_5 z^5] \\ &= L_u + L_l \end{aligned} \quad (\text{B.67})$$

The limits on $|x - 1|$ were chosen such that, for either method, $\ln(x)$ is accurate to within 13 bits more than single precision.

4. Now, split y (the power in x^y) into two pieces, $y = y_u + y_l$ and compute $P = y \times \ln(x)$ in pseudo-double precision as

$$\begin{aligned} P &= (y_u + y_l) \times (L_u + L_l) \\ &= P_u + P_l \end{aligned}$$

(B.68)

The product P is kept to more than single precision.

5. Compute the final result $e^{y \ln(x)}$ as:

$$\begin{aligned} x^y &\equiv e^{y \ln(x)} \\ &\approx e^P = e^{P_u + P_l} \end{aligned}$$

(B.69)

The function $e^{P_u + P_l}$ is done by a method similar to computing `EXP`, and it can make sure of the same lookup table as the `EXP` library function.

The final expression for $e^{P_u + P_l}$ becomes the following:

$$e^{P_u + P_l} = T_u + [T_l + T_u \times f(r)]$$

(B.70)

where r is the reduced argument obtained within the `EXP` function, T is the `EXP` table lookup and $f(r) \equiv e^r - 1$. The final addition is done using software rounding to obtain a correct single-precision result.

B.8.1 Accuracy

Extensive testing with combinations of arguments x and y varied such that $z = x^y$ returns values throughout the full range of floating-point numbers $[\approx 10^{-2466} - 10^{+2466}]$ shows that this function returns the correctly rounded result about 99% of the time, with the remainder of results being less than 1 ULP in error.

B.9 Single-precision Real SIN(x) and COS(x) Functions

Note: `COSS` is a combined sine/cosine function. It returns `COS(x)` as its real part and `SIN(x)` as its imaginary part. `COSS(x)` returns the same results as calling `SIN` and `COS` but executes faster than two separate calls to `SIN` and `COS`.

Procedure 10: SIN(x) and COS(x)

1. Let `a=1` if computing `cos(x)`. Otherwise, let `a=0` if computing `sin(x)` and `e` equal to the sign bit of the argument `x`.
2. If $|x| > 2^{25}$, print an error message and abort. Otherwise, compute `sin(x)` or `cos(x)` as follows.
3. Find

$$N = \text{INT} \left[\frac{|x|}{\pi/4} \right]$$

(B.71)

which is the number of multiples of $\frac{\pi}{4}$ in `x`. Find the octant of the circle (2π) in which `N` lies. Let $m = N \pmod{8} = d \cdot 2^2 + c \cdot 2^1 + b$ where:

`b` = low-order bit of `m` (2^0)

`c` = middle bit of `m` (2^1)

`d` = high-order bit of `m` (2^2)

4. Add 1 to `N` if `N` is odd — let $N = N + b$ and let $x_n = \text{FLOAT}(N)$, to convert `N` to a real. The reduced argument is $f = |x| - x_n \frac{\pi}{4}$, which is evaluated in extended precision as

$$f_1 = |x| - \frac{x_n c_1}{2} - \frac{x_n c_1}{2} - x_n c_2$$

(B.72)

$$f = f_1 - x_n c_3$$

(B.73)

where $c_1 + c_2 + c_3 = \frac{\pi}{4}$ to more than single precision, and where c_1 and c_2 are chosen such that $x_n c_1$ and $x_n c_2$ can be computed exactly within a single 64-bit word (that is, without any rounding by the multiply hardware).

5. Now compute $\Delta = \text{dble}(f) - \text{sngl}(f)$. This is the residual error in f . Δ is the difference between a full double-precision evaluation of $|x| - x_n \frac{\pi}{4}$ and the value of f from Equation B.73. Δ is adequately estimated using only single precision as $\Delta = (f_1 - f) - x_n c_3$ and is the part of $x_n c_3$ that did not contribute to f in Equation B.73.
6. If $f < 0$,

$$\begin{aligned} f &= -f = |f| \\ \Delta &= -\Delta \end{aligned}$$

(B.74)

Let $tflag=a\b\c$ where a , b , and c are the bit flags defined in Procedure 10, step 1, page 151 and Procedure 10, step 3, page 151. If $tflag=0$, compute $\sin(f)$ using either Method a or Method b in Procedure 10, step 7, page 152. If $tflag=1$, compute $\cos(f)$ using either Method a or Method b in Procedure 10, step 7, page 152. If the entry point was `COSS`, then compute $\sin(f)$ and $\cos(f)$ in parallel.

7. Compute $\sin(f)$ and $\cos(f)$ by one of two methods, depending on the magnitude of f .
 - a. If $f \leq 16$, compute $\sin(f)$ and $\cos(f)$ by a minimax polynomial:

$$\sin(f) = f + fP(t)$$

(B.75)

$$\cos(f) = 1 + Q(t)$$

(B.76)

where $t = f^2$ and

$$P(t) = p_1 t + p_2 t^2 + p_3 t^3$$

(B.77)

$$Q(t) = q_1 t + q_2 t^2 + q_3 t^3 \quad (\text{B.78})$$

Equation B.75 is evaluated by first replacing f with $(f + \Delta)$ and keeping the first order Δ term, giving the result as $\sin(f) = f + [fP(t) + \Delta]$, where the summation is done carefully to preserve full accuracy. The Δ term is not needed for computing $\cos(f)$.

- b. If $\frac{1}{16} < f \leq \frac{\pi}{4}$, compute $\sin(f)$ or $\cos(f)$ using a table lookup method. First split f into an upper and lower part by letting $f + \Delta = f_0 + \delta f$ or $\delta f = (f - f_0) + \Delta$ where f_0 is taken from the uppermost 6 bits of f . Using the leading bits of f_0 as an index, look up the values of $\sin(f_0) = S_u + S_l$ and $\cos(f_0) = C_u + C_l$ where (S_u, S_l) represent $\sin(f_0)$ to more than single precision, and likewise for (C_u, C_l) . Compute a correction to the table value using standard trigonometric identities as:

$$\begin{aligned} \Delta S &= \sin(f_0 + \delta f) - \sin(f_0) \\ &= \sin(f_0) [\cos(\delta f) - 1] + \cos(f_0) \sin(\delta f) \end{aligned} \quad (\text{B.79})$$

Similarly, a correction term for computing cosine is given by

$$\begin{aligned} \Delta C &= \cos(f_0 + \delta f) - \cos(f_0) \\ &= \cos(f_0) [\cos(\delta f) - 1] - \sin(f_0) \sin(\delta f) \end{aligned} \quad (\text{B.80})$$

In both expressions for ΔS and ΔC , the quantities $\sin(\delta f)$ and $[\cos(\delta f) - 1]$ are approximated by the same power series as in Equation B.75 and Equation B.76 but with f replaced by δf

$$S = \sin(\delta f) = \delta f + \delta f P[(\delta f)^2] \quad (\text{B.81})$$

$$C = \cos(\delta f) - 1 = Q[(\delta f)^2] \quad (\text{B.82})$$

B.10 Single-precision Real COSH(x) and SINH(x) Functions

Procedure 11: COSH(x) and SINH(x)

1. If $|x| > 5677.5686$ (approximately), a fatal error occurs. Otherwise, the computation continues. Let $f = |x|$ and use the following equations to get the sign of the result.

$$\cosh(-f) = \cosh(f) \tag{B.86}$$

$$\sinh(-f) = -\sinh(f) \tag{B.87}$$

2. On the range $f = [0, \frac{113}{128}]$: if a COSH entry evaluate $\cosh(f) = 1 + Q(t)$; if a SINH entry, evaluate $\sinh(f) = f + fP(t)$ where $t = f^2$ and $Q(t)$ and $P(t)$ are minimax polynomials of the following form:

$$Q(t) = q_1t + q_2t^2 + q_3t^3 + q_4t^4 + q_5t^5 + q_6t^6 + q_7t^7 \tag{B.88}$$

$$P(t) = p_1t + p_2t^2 + p_3t^3 + p_4t^4 + p_5t^5 + p_6t^6 + p_7t^7 \tag{B.89}$$

where the coefficients q_i and p_i are derived especially for Cray PVP floating-point hardware.

3. On the argument range $f = [\frac{113}{128}, 5677.5686]$, evaluate $\cosh(f)$ and $\sinh(f)$ using the e^f function. If a COSH entry, let

$$\begin{aligned} \cosh(f) &= \frac{1}{2} \left(e^f + \frac{1}{e^f} \right) \\ &= \frac{1}{2} \left(E_u + E_l + \frac{1}{E_u} \right) \end{aligned} \tag{B.90}$$

If a SINH entry, let

$$\begin{aligned} \sinh(f) &= \frac{1}{2} \left(e^f - \frac{1}{e^f} \right) \\ &= \frac{1}{2} \left(E_u + E_l - \frac{1}{E_u} \right) \end{aligned}$$

(B.91)

where $e^f = E_u + E_l$ to more than single precision by using a special version of the EXP function, which provides the lower bits of the EXP result. Argument ranges were chosen such that the single-precision reciprocal approximation is sufficiently accurate for computing E_u^{-1} .

4. Add the sign to the result. If COSH entry, $\cosh(x) = \cosh(f)$. If SINH entry, $\sinh(x) = \text{SIGN}(\sinh(f), x)$.

B.10.1 Accuracy

The SINH(x) and COSH(x) routines were tested on various sets of 250,000 random arguments over the entire range of legal arguments. They give correctly rounded results approximately 97.0% of the time, with no error greater than 1 ULP. The largest observed error was about 0.78 ULP.

B.11 Single-precision Real SQRT(x) Function

Procedure 12: SQRT(x)

1. The function \sqrt{x} is computable for all positive real numbers and 0. The range of the function is approximately $[0, 0.5221944407 \times 10^{1233}]$, where the argument x is in the range $[0, 0.2726870339 \times 10^{2466}]$.

Let $x = 2^{2n} 2^b g$ where n is integral, b is 0 or 1, and $\frac{1}{2} \leq g < 1$. Then:

$$\sqrt{x} = \sqrt{2^{2n} 2^b g}$$

(B.92)

$$= 2^n \sqrt{g} \quad (b = 0)$$

(B.93)

$$= 2^n \sqrt{2} \sqrt{g} \quad (b = 1)$$

(B.94)

2. Find an initial estimate to \sqrt{g} using the King and Phillips approximation formula (see King and Phillips, *The Logarithmic Error and Newton's Method for the Square Root*):

$$f_0 = \sqrt{g} \approx \frac{\sqrt{\frac{1}{2} + g}}{2^{\frac{3}{8}} \sqrt{\sqrt{\frac{1}{2}} + 1}}$$

(B.95)

If $b = 1$, multiply the f_0 from Equation B.95, by $\sqrt{2}$.

3. Find the second estimate, f_1 , by using the Newton-Raphson iteration,

$$f_1 = \frac{1}{2} \left(f_0 + \frac{f}{f_0} \right)$$

(B.96)

where only the hardware half-precision reciprocal is needed. Find the third estimate, f_2 , by another Newton-Raphson iteration:

$$f_2 = \frac{1}{2} \left(f_1 + \frac{f}{f_1} \right)$$

(B.97)

using only the hardware half-precision reciprocal.

4. Finish \sqrt{f} by using a third Newton-Raphson iteration, rewritten in a different form:

$$f_3 = f_2 + \frac{f - f_2^2}{2f_2}$$

(B.98)

The term $f - f_2^2$ is done in pseudo-double precision, although the $\frac{1}{2}$ precision hardware reciprocal can still be used. The final addition is done using software rounding.

5. The final result is obtained by substituting f_3 in Equation B.93: $\sqrt{x} = 2^n f_3$. This is done easily by adding n to the exponent of f_3 .

B.11.1 Accuracy

This routine was tested on various sets of 250,000 random arguments over the entire range of floating-point numbers. Although testing was not exhaustive, the algorithm appears to give correctly rounded results 100% of the time, with no error greater than 0.50 ULP.

B.12 Single-precision TAN(x) and COT(x) Functions

Procedure 13: TAN(x) and COT(x)

1. Let $a=1$ if computing $\cot(x)$. Otherwise, let $a=0$ if computing $\tan(x)$ and d equal to the sign bit of the argument x .
2. If $|x| > 2^{25}$, print an error message and abort. Otherwise, compute $\tan(x)$ or $\cot(x)$ as follows.
3. Find

$$N = \text{INT} \left[\frac{|x|}{\pi/4} \right]$$

(B.99)

which is the number of multiples of $\frac{\pi}{4}$ in x . Find the quadrant of the circle (2π) in which N lies. Let $m = N \pmod{4} = c \cdot 2^1 + b$ where b is equal to the low-order bit of m (2^0) and c is equal to the next bit of m (2^1).

4. Add 1 to N if N is odd — let $N = N + b$ and let $x_n = \text{FLOAT}(N)$, to convert N to a real. The reduced argument is $f = |x| - x_n \frac{\pi}{4}$, which is evaluated in extended precision as

$$f_1 = |x| - \frac{x_n c_1}{2} - \frac{x_n c_1}{2} - x_n c_2$$

(B.100)

$$f = f_1 - x_n c_3$$

(B.101)

where $c_1 + c_2 + c_3 = \frac{\pi}{4}$ to more than single precision, and where c_1 and c_2 are chosen such that $x_n c_1$ and $x_n c_2$ can be computed exactly within a single 64-bit word (that is, without any rounding by the multiply hardware).

5. Now compute $\Delta = \text{dble}(f) - \text{sngl}(f)$. This is the residual error in f . Δ is the difference between a full double-precision evaluation of $|x| - x_n \frac{\pi}{4}$ and the value of f from Equation B.101. Δ is adequately estimated using only single precision as $\Delta = (f_1 - f) - x_n c_3$ and is the part of $x_n c_3$ that did not contribute to f in Equation B.101.

6. If $f < 0$,

$$\begin{aligned} f &= -f = |f| \\ \Delta &= -\Delta \end{aligned}$$

(B.102)

Let $tflag = a \setminus b \setminus c$ where a , b , and c are the bit flags defined in Procedure 13, step 1, page 158 and Procedure 13, step 3, page 158. The \setminus symbol denotes the Boolean XOR operation. If $tflag = 0$, compute $\tan(f)$ using either Method A or Method B in Procedure 13, step 7, page 159. If $tflag = 1$, compute $\cot(f)$ using either Method A or Method B in Procedure 13, step 7, page 159.

7. Compute $\tan(f)$ and $\cot(f)$ by one of two methods, depending on the magnitude of f .

- a. If $f \leq \frac{1}{16}$, compute $\tan(f)$ and $\cot(f)$ by a minimax polynomial:

$$\tan(f) = f + fP(t)$$

(B.103)

$$\cot(f) = \frac{1}{f} + fQ(t)$$

(B.104)

where $t = f^2$ and

$$P(t) = p_1t + p_2t^2 + p_3t^3 + p_4t^4 \tag{B.105}$$

$$Q(t) = q_1 + q_2t + q_3t^2 + q_4t^3 \tag{B.106}$$

Equation B.103 is evaluated by first replacing f with $(f + \Delta)$ and keeping the first order Δ term, giving the result as $\tan(f) = f + [fP(t) + \Delta]$, where the summation is done carefully to preserve full accuracy.

When computing $\cot(f)$ for Equation B.104 replace f with $(f + \Delta)$ and keep terms to first order in Δ :

$$\begin{aligned} \frac{1}{f} \rightarrow \frac{1}{f + \Delta} &\approx \frac{1}{f} - \frac{\Delta}{f^2} \\ &= \left. \frac{1}{f} \right]_u + \left. \frac{1}{f} \right]_l - \frac{\Delta}{f^2} \end{aligned} \tag{B.107}$$

where the reciprocal $\frac{1}{f}$ is done to extended precision. This gives $\cot(f)$ as:

$$\cot(f) = \left. \frac{1}{f} \right]_u + fQ(t) + \left. \frac{1}{f} \right]_l - \frac{\Delta}{f^2} \tag{B.108}$$

where the terms are added carefully to avoid loss of precision.

- b. If $\frac{1}{16} < f \leq \frac{\pi}{4}$, compute $\tan(f)$ or $\cot(f)$ using a table lookup method. First split f into an upper and lower part by letting $f + \Delta = f_0 + \delta f$ or $\delta f = (f - f_0) + \Delta$ where f_0 is taken from the uppermost 10 bits of f . Using the leading bits of f_0 as an index, look up the values of $\tan(f_0) = T_u + T_l$ and $\cot(f_0) = C_u + C_l$ where (T_u, T_l) represent $\tan(f_0)$ to more than single precision, and likewise for (C_u, C_l) . Compute a correction to the table value using standard trigonometric identities as:

$$\begin{aligned}\Delta T &= \tan(f_0 + \delta f) - \tan(f_0) \\ &\approx \frac{[1 + T_u^2] \tan(\delta f)}{1 - T_u \tan(\delta f)}\end{aligned}\tag{B.109}$$

where terms in T_l have been neglected. Similarly, a correction term for computing cotangent is given by:

$$\begin{aligned}\Delta C &= \cot(f_0 + \delta f) - \cot(f_0) \\ &\approx \frac{[1 + C_u^2] \tan(\delta f)}{1 + C_u \tan(\delta f)}\end{aligned}\tag{B.110}$$

where terms in C_l have been neglected. The size of the lookup tables was chosen such that, in Equation B.109 and Equation B.110, the hardware division has sufficient accuracy. In both expressions for ΔT and ΔC , the quantity $\tan(\delta f)$ is approximated by a shorter version of the same power series as in Equation B.103, but with f replaced by δf

$$T = \tan(\delta f) = \delta f + p_1 (df)^3\tag{B.111}$$

and where the coefficient p_1 is the same as in Equation B.105. So, to compute $\tan(f)$, evaluate $\tan(f) = T_u + [\Delta T + T_l]$. To compute $\cot(f)$, evaluate $\cot(f) = C_u + [\Delta C + C_l]$. The final summations are done carefully in order to preserve accuracy, and the final addition is done using a software rounded add.

8. The sign of the result is $\text{SIGN} = (c \setminus d)$ where \setminus is the Boolean XOR operation. The values c and d are the bit flags as defined in Procedure 13, step 1, page 158 and Procedure 13, step 3, page 158. When $\text{SIGN}=1$, the result is negative. When $\text{SIGN}=0$, the result is 0 or positive.

B.12.1 Accuracy

For various sets of 250,000 random arguments in the range $[-\pi, \pi]$, approximately 99.4% were correct. The largest error was around 0.75 ULP. Similar accuracy holds throughout the legal range of arguments $|x| < 2^{25}$ with no errors greater than 1 ULP found.

B.13 Single-precision Real $\text{TANH}(x)$ Function

Procedure 14: $\text{TANH}(x)$

1. If $|x| > 0.2726870339^{+2466}$ (approximately), a floating exception error occurs. Otherwise, the computation continues. Let $f = |x|$ and use

$$\tanh(-f) = -\tanh(f) \tag{B.112}$$

to obtain the sign of the result.

2. On the range $x = [0, \frac{1}{16}]$, evaluate $\tanh(f) = f + fP(t)$ where $t = f^2$ and $P(t)$ is a minimax polynomial of the following form:

$$P(t) = p_1t + p_2t^2 + p_3t^3 + p_4t^4 \tag{B.113}$$

where the coefficients p_i are derived especially for Cray PVP floating-point hardware.

3. On the argument range $f = [\frac{1}{16}, 17.5]$, use a table lookup method based on the following mathematical identity:

$$\tanh(a+b) = \frac{\tanh(a) + \tanh(b)}{1 + \tanh(a)\tanh(b)} \tag{B.114}$$

$$= \tanh(a) + \frac{[1 - \tanh^2(a)] \tanh(b)}{1 + \tanh(a)\tanh(b)} \tag{B.115}$$

Let f_0 be obtained from the exponent and uppermost 9 bits of f , and define the difference $\delta f \equiv f - f_0$. Substituting f_0 and δf for a and b in Equation B.115 results in the following:

$$\tanh(f) \equiv \tanh(f_0 + \delta f) \tag{B.116}$$

$$= \tanh(f_0) + \frac{[1 - \tanh^2(f_0)] \tanh(\delta f)}{1 + \tanh(f_0) \tanh(\delta f)} \quad (\text{B.117})$$

$$= T_u + T_l + \frac{(1 - T_u^2) \tanh(\delta f)}{1 + T_u \tanh(\delta f)} \quad (\text{B.118})$$

where $\tanh(\delta f) \equiv T_u + T_l$ is from the table lookup and is to more than single precision. And the term $\tanh(\delta f) = \delta f + \delta f (p_1 t + p_2 t^2)$ where $t = \delta f^2$ and p_1 and p_2 come from the polynomial $P(t)$ defined in Equation B.113. The hardware single-precision reciprocal is of sufficient accuracy to use in the division in Equation B.118.

4. On the argument range $f = [17.5, 0.2726870339^{+2466}]$, set $\tanh(f) = 1$.
5. Add the sign to the result: $\tanh(x) = \text{SIGN}(\tanh(f), x)$.

B.13.1 Accuracy

For various sets of 250,000 random arguments in the range $[-17.5, 17.5]$ approximately 98.5% were correct. Similar accuracy holds throughout the legal range of arguments $|x| < \infty$, with the largest error around 0.97 ULP.

Basic Linear Algebra Subprogram

A set of commonly used algebraic equations defined by C. L. Lawson, and J. J. Dongarra, in a series of papers (see bibliography of *LAPACK User's Guide*, publication TPD-0003, pp. 112–115, entries [16], [17], [18], [19], and [38]).

BCG

See Bi-Conjugate Gradient Method.

Bi-Conjugate Gradient Method

One of the iterative methods provided through the `SITRSOL` package of optimized preconditioned iterative methods.

Bi-Conjugate Gradient Squared Method

One of the iterative methods provided through the `SITRSOL` package of optimized preconditioned iterative methods.

BLAS

See Basic Linear Algebra Subprogram.

CGN

See Conjugate Gradient Method.

CGS

See Bi-Conjugate Gradient Squared Method.

computational routines

Term used to define LAPACK routines that perform a distinct computational task.

Conjugate Gradient Method

One of the iterative methods provided through the `SITRSOL` package of optimized preconditioned iterative methods.

dedicated environment

A parallel processing environment in which the `NCPUS` environment variable is equal to the number of available processors. This ensures that the number of processors specified by `NCPUS` is available at all times.

direct solution methods

Direct solution methods for sparse linear systems transform the matrix A into a product of several other operators so that each of the resulting operators is easy to invert for a given right-hand side b .

driver routines

Term used to define LAPACK routines used for solving standard types of problems.

equilibration

The process of scaling a problem before computing its solution.

Fourier analysis

The mathematical process of resolving a given function, $f(x)$, into its frequency components, which means finding the sequence of constant amplitudes to plug into a Fourier series to reconstruct the original function.

GCR

See Orthomin/Generalized Conjugate Residual Method.

Generalized Minimum Residual Method

One of the methods provided through the `SITRSOL` package of optimized preconditioned iterative methods.

GMR/GMRES

See Generalized Minimum Residual Method.

Hermitian matrix

A complex matrix which is equal to the conjugate of its transpose, with either the lower or upper triangle being stored.

iterative solution methods

Iterative solution methods for sparse linear systems attempt to solve $Ax = b$ by solving an equivalent system $M^{-1}Ax = M^{-1}b$, where M is some approximation to A which is inexpensive to construct and can be easily used to compute z . Unlike direct methods, iterative methods are more special-purpose. There are no general, effective iterative algorithms for an arbitrary sparse linear system.

LAPACK

A public domain library of subroutines for solving dense linear algebra problems, including systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems. It has been designed for efficiency on high-performance computers.

large parallel/vector problem

A class of problem size in which problems are large enough for optimal vector and parallel processing and for which load balancing is not a significant problem. In this case, enough work exists to partition the problem into many subproblems so that the effect an unavailable processor is minimized.

linear system

A set of simultaneous linear algebraic equations.

load balancing

The process of dividing work done by each available processor into approximately equal amounts.

LPVP

See Large Parallel/Vector Problem.

medium parallel/vector problem

A class of problem size in which problems are large enough for optimal vector and parallel processing, but for which load balancing can be a significant problem if a processor is unavailable.

multiuser environment

A parallel processing environment in which users do not know how many processors will be available to a job during run time, except that the number will be less than or equal to NCPUS.

OMN

See Orthomin/Generalized Conjugate Residual Method.

Orthomin/Generalized Conjugate Residual Method

One of the methods provided through the SITRSOL package of optimized preconditioned iterative methods.

out-of-core technique

A term that refers to algorithms that combine input and output with computation to solve problems in which the data resides on disk or some other secondary random-access storage device.

packed storage

A type of matrix in which half of the matrix (triangular or symmetric) is stored on disk or SSD.

PCG

See Preconditioned Conjugate Gradient Method.

parallel instruction execution

The execution of one instruction per clock period, even those instructions that take several clock periods to complete execution.

pipelining

A method of execution which allows each step of an operation to pass its result to the next step after only one clock period.

Preconditioned Conjugate Gradient Method

One of the iterative methods provided through the SITRSOL package of optimized preconditioned iterative methods.

single-threaded code segments

A section of a program that must use a single processor.

small parallel/vector problem

A class of problem size in which problems are large enough for vector and parallel processing, but for which parallel processing degrades vector performance.

sparse matrix

A linear system which can be described as $Ax = b$, where A is an n -by- n matrix, and x and b are n dimensional vectors. A system of this kind is considered *sparse* if the matrix A has a small percentage of nonzero terms (less than 10%, often less than 1%).

SPD

See Symmetric Positive Definite Matrix.

SPVP

See Small Paralell/Vector Problem.

Strassen's algorithm

A recursive algorithm that is slightly faster than the ordinary inner product algorithm. Strassen's algorithm performs the floating-point operations for matrix multiplication in an order differently from the vector method; this can cause round-off problems.

supernodes

A collection of columns that have the same nonzero pattern.

Symmetric Positive Definite matrix

If a matrix is SPD, the standard preconditioned conjugate gradient algorithm is usually the best choice. However, the pure truncated forms of Orthomin and

GMRES with a small *iparam(16):ntrunc* value may be useful on problems for which the standard conjugate gradient does not work.

time slicing

A method of execution in which the system works on several jobs or processes simultaneously.

vector problem

A class of problem size in which problems are large enough for vector processing, but too small for parallel processing.

vectorization

A form of parallel processing that uses instruction segmenting and vector registers.

virtual matrices

A virtual matrix is similar to a Fortran array, but it cannot be accessed directly from a program. It can only be accessed with calls to specific subroutines. Users do not do any explicit input or output to read from or write to a virtual matrix.

VP

See Vector Problem.

well-conditioned matrix

The condition number of a matrix is defined as $\kappa(A) = \|A\| \cdot \|A^{-1}\|$. A *well-conditioned* matrix is one for which $\kappa(A)$ is small. Although small is relative, if $\kappa(A) < 10^3$, A can be considered well-conditioned.

1 ULP
 examples, 125

A

acos(x), 135
 accuracy, 138
aggressive optimization
 with CF90 compiler, 5
alog(x), 132
 accuracy, 134
Amdahl's Law for multitasking, 9
AQIO routines, 103
asin(x), 135
 accuracy, 138
asynchronous queued I/O routines, 103
atan(x), 138
 accuracy, 139
atan(y,x), 139
 accuracy, 143
Autotasking, 5

B

banded matrix, 54
Basic Linear Algebra Subprograms, 25
BCG, 59
Bi-Conjugate Gradient Method, 59
Bi-Conjugate Gradient Squared Method, 59
binary unblocked file, 103
BLAS, 25

C

cbrt(x), 143
 accuracy, 145

CGN, 59
CGS, 59
chaining, 3
computational routines, 27
computing a simple bound, 37
condition estimation, 36
condition number, 36
Conjugate Gradient Method, 59
cos(x), 151
 accuracy, 154
cosh(x), 155
 accuracy, 156
cot(x), 158
 accuracy, 161

D

data structures
 and sparse matrices, 57
dedicated environment
 parallel processing strategies, 21
dedicated parallel processing environment
 characteristics, 13
dedicated work environment, 13
diagonally dominant matrix, 54
direct general purpose sparse solvers, 67
direct solver tuning issues, 70
direct solver tuning parameters
 frontal matrix grouping, 71
 supernode augmentation, 70
 threshold pivoting, 71
direct solvers, 58
driver routines, 27, 34

E

- e^x, 145
 - accuracy, 147
- EISPACK, 25
- environment variables, 12
 - MP_DEDICATED, 12
 - MP_HOLDTIME, 12
 - NCPUS, 12
 - suggested settings, 12
 - tuning, 12
 - UNICOS, 115
- environments, 11
- equilibration, 39
- error bounds, 37
- error bounds computations, 43
- error codes, 33
- error conditions, 33
- error reporting, 116
- examples
 - creating a virtual matrix, 113
 - error conditions, 33
 - LU factorization, 30
 - multiplying a virtual matrix, 113
 - orthogonal factorization, 46
 - out-of-core technique, 101
 - protocol usage, 114
 - roundoff errors, 37
 - single/multitasked performance, 15
 - sparse solvers
 - general symmetric positive definite, 75
 - general unsymmetric, 79
 - multiple right-hand side, 89
 - reuse of structure, 84
 - save/restart, 93
 - symmetric indefinite matrix factorization, 31
- explicit form, 29

F

- factored form, 29
- factoring a matrix, 29

- factorization forms, 29

G

- GCR, 59
- general patterned sparse systems, 63
- Generalized Minimum Residual Method, 59
- GMR, 59
- GMRES, 59
- guidelines
 - choosing a solver
 - based on problem type, 64
 - general patterned sparse linear systems, 63
 - iterative methods, 65
 - preconditioning, 65
 - tridiagonal systems, 62

H

- Hermitian matrix, 105
- highly off-diagonally-dominant 3D problems, 62
- Hilbert matrix, 41
- Householder transformation, 47

I

- I/O subsystems, 3
- ILAENV, 26
- ill-conditioned problems, 64
- implicit diagonal preconditioning, 66
- inverse of dense matrix, 44
- iterative methods, 59
- iterative refinement, 41
- iterative solvers, 59

L

- LAPACK

- and tuning parameters, 26
 - data types supported, 25
 - error codes, 33
 - factoring a matrix, 29
 - iterative refinement, 41
 - naming scheme, 26
 - orthogonal factorizations, 45
 - overview, 25
 - result comparisons, 50
 - simple driver routines, 34
 - solving from the factored form, 34
 - solving linear systems, 28
 - types of problems solved, 26
 - types of routines, 27
 - large parallel/vector problem (LPVP) problem
 - size, 20
 - leading virtual dimensions, 104
 - least squares problem, 26
 - least squares problems
 - solving, 45
 - libm
 - 1 ULP criterion, 124
 - acos(x), 135
 - accuracy, 138
 - algorithms, 129
 - alog(x), 132
 - accuracy, 134
 - asin(x), 135
 - accuracy, 138
 - atan(x), 138
 - accuracy, 139
 - atan(y,x), 139
 - accuracy, 143
 - cbirt(x), 143
 - accuracy, 145
 - cos(x), 151
 - accuracy, 154
 - cosh(x), 155
 - accuracy, 156
 - cot(x), 158
 - accuracy, 161
 - e^x, 145
 - accuracy, 147
 - features, 123
 - functions, 123
 - ln(x), 129
 - accuracy, 131
 - numerical methods, 125
 - overview, 123
 - side effects, 128
 - sin(x), 151
 - accuracy, 154
 - sinh(x), 155
 - accuracy, 156
 - sqrt(x), 156
 - accuracy, 158
 - tan(x), 158
 - accuracy, 161
 - tanh(x), 162
 - accuracy, 163
 - x^y, 147
 - accuracy, 150
 - linear system, 53
 - linear systems
 - solving, 27
 - LINPACK, 25
 - ln(x), 129
 - accuracy, 131
 - Load balancing, 8
 - load balancing
 - and parallelism, 8
 - definition, 8
 - logarithm functions
 - single-precision, 129
 - accuracy, 131
 - single-precision real, 132
 - accuracy, 134
 - LPVP
 - definition, 20
 - LU factorization, 30
- M**
- macrotasking, 5

- matrix characteristics
 - Non-Symmetric Definite, 65
 - Non-Symmetric Indefinite, 65
 - Symmetric Indefinite, 65
 - Symmetric Positive Definite (SPD), 65
- matrix inversion, 44
- medium parallel/vector problem (MPVP)
 - problem size, 20
- memory usage guidelines, 118
- microtasking, 5
- MP_DEDICATED environment variable, 12
- MP_HOLDTIME environment variable, 12
- MPVP
 - definition, 20
 - in multiuser environments, 22
- multiprocessing, 4
- multiprogramming, 4
- multitasking, 5
 - Amdahl's Law, 9
 - and vectorization, 7
 - efficiency, 10
 - overview, 3
 - speedup ratio, 7
 - user code, 5
 - when used, 15
- multitasking variables, 116
- multiuser environment, 14
 - parallel processing strategies, 21
- multiuser environments
 - and SPVP and MPVP problems, 22
- multiuser parallel processing environment
 - characteristics, 14

N

- NCPUS environment variable, 7, 12
- Non-Symmetric Definite matrix, 65
- Non-Symmetric Indefinite matrix, 65
- numerical methods, 125

O

- OMN, 59
- orthogonal factorizations, 45
- orthogonal matrix
 - generating, 49
 - multiplying by, 48
- Orthomin/Generalized Conjugate Residual Method, 59
- out-of-core routines, 101
 - features, 101
 - subroutines, 106
 - complex routines, 106
 - initialization and termination, 109
 - lower-level routines, 112
 - routine summary, 107
 - virtual BLAS routines, 111
 - virtual copy, 109
 - virtual LAPACK routines, 110
- out-of-core technique, 101
- overdetermined linear system, 26

P

- packed storage mode, 105
 - full type, 105
 - lower type, 105
 - upper type, 105
- page size, 106
- page-buffer space, 118
- parallel instruction execution, 3
- parallel processing
 - and vectorization, 7
 - benefits, 6
 - calculating program speedup, 8
 - costs/benefits discussion, 5
 - efficiency, 10
 - overhead, 6
 - overview, 3
 - speedup ratio, 7
 - user code, 5

- parallel processing environments
 - dedicated environment, 13
 - multiuser environment, 14
 - overview, 11
- parallel processing strategies, 20
 - dedicated environment, 21
 - multiuser environment, 21
 - problem sizes, 20
- partitions of work, 8
- PCG, 59
- performance issues
 - determining efficiency, 10
 - in parallel processing, 8
 - percentage of parallelism, 11
 - speedup, 8
- performance measurements, 14, 117
 - RTC command, 15
 - SECOND command, 14
 - simple measurements, 14
 - use of dedicated resources, 14
- performance tuning, 67
 - direct solvers, 70
 - memory usage guidelines, 118
 - page-buffer space, 118
 - parallel processing, 67
 - reusing information, 68
 - SITRSOL tuning, 69
- pipelining, 3
- polynomial preconditioning, 66
- Preconditioned Conjugate Gradient Method, 59
- preconditioning, 65
 - incomplete factorization, 66
 - polynomial, 66
 - scaling, 66
- problem sizes, 20
 - large parallel/vector problem, 20
 - medium parallel/vector problem, 20
 - small parallel/vector problem, 20
 - vector problem, 20

Q

- QR factorization, 46

R

- reciprocal condition number, 36
- reusing information
 - multiple right-hand sides, 68
 - save/restart, 69
 - structure, 68
 - values, 68
- roundoff errors, 37

S

- sample performance statistics, 119
- SGEMV command, 15
- sin(x), 151
 - accuracy, 154
- single-threaded code segments, 9
- sinh(x), 155
 - accuracy, 156
- SITRSOL preconditioning
 - explicit scaling, 59
 - implicit preconditioning, 60
- SITRSOL summary, 72
- SITRSOL tuning issues, 69
- SITRSOL tuning parameters
 - Density of Incomplete Factors, 70
 - diagonal shifting for incomplete Cholesky, 70
 - size of Krylov subspace, 70
 - Sparse Matrix Vector Product, 69
- skyline solver, 62
- small parallel/vector problem (SPVP) problem
 - size, 20
- solution techniques
 - direct methods, 55, 166
 - iterative methods, 56
 - sparse linear systems, 54

- solvers
 - choosing correct solvers, 62
 - solving dense linear systems, 34
 - sparse linear solvers, 53
 - sparse linear systems
 - solution techniques, 54
 - direct methods, 55, 166
 - iterative methods, 56
 - sparse matrices
 - banded matrix, 54
 - data structures, 57
 - diagonally dominant matrix, 54
 - direct solvers, 58
 - iterative solvers, 59
 - other types of solvers, 60
 - overview, 53
 - structurally symmetric matrix, 54
 - Symmetric Positive Definite matrix, 53
 - tridiagonal matrix, 54
 - types of, 53
 - sparse solvers
 - and banded matrices, 62
 - direct general, 67
 - implemented by Cray Research, 57
 - parallel processing, 67
 - problem type, 64
 - reusing information, 68
 - skyline solver, 62
 - using, 62
 - SPD matrix, 65
 - speedup ratio, 7
 - speedups
 - in multitasked programs, 8
 - limitations, 9
 - SPVP
 - definition, 20
 - in multiuser environments, 22
 - sqrt(x), 156
 - accuracy, 158
 - Strassen's algorithm, 111
 - structurally symmetric matrix, 54
 - subroutines
 - out-of-core, 106
 - Symmetric Indefinite matrix, 65
 - symmetric indefinite matrix factorization, 31
 - Symmetric Positive Definite matrix, 53, 65
- T**
- tan(x), 158
 - accuracy, 161
 - tanh(x), 162
 - accuracy, 163
 - throughput, 5
 - timeslicing, 3
 - timing results
 - dedicated environment, 7
 - factors affecting, 7
 - timings
 - LAPACK and solver routines, 35
 - tridiagonal matrix, 54
 - tridiagonal solvers
 - summary, , 61
 - tridiagonal systems, 62
 - Tuning parameters, 26
- U**
- ULP
 - definition, 124
 - underdetermined linear system, 26, 45
 - UNICOS environment variables, 115
 - multitasking, 116
 - unit numbers, 102
- V**
- vector problem (VP) problem size, 20
 - vectorization
 - and multitasking, 7
 - and parallel processing, 7
 - definition, 3

virtual matrices, 102
 binary unblocked file, 103
 defining elements, 104
 definition, 101
 file format, 103
 file sizes, 104
 leading virtual dimensions, 104
 packed storage mode, 105
 page size, 106
 unit numbers, 102
VP
 definition, 20

W

well-conditioned matrix, 64
well-conditioned problems, 64

X

x^y , 147
 accuracy, 150
XERBLA, 33